Exploiting Independent State For Network Intrusion Detection

Robin Sommer
TU München
sommer@in.tum.de

Vern Paxson
ICSI/LBNL
vern@icir.org

Abstract

Network intrusion detection systems (NIDSs) critically rely on processing a great deal of state. Often much of this state resides solely in the volatile processor memory accessible to a single user-level process on a single machine. In this work we highlight the power of independent state*, i.e., internal fine-grained state that can be propagated from one instance of a NIDS to others running either concurrently or subsequently. Independent state provides us with a wealth of possible applications that hold promise for enhancing the capabilities of NIDSs. We discuss an implementation of independent state for the Bro NIDS and examine how we can then leverage independent state for distributed processing, load parallelization, selective preservation of state across restarts and crashes, dynamic reconfiguration, high-level policy maintenance, and support for profiling and debugging. We have experimented with each of these applications in several large environments and are now working to integrate them into the sites' operational monitoring. A performance evaluation shows that our implementation is suitable for use even in large-scale environments.*

1 Introduction

Network intrusion detection systems (NIDSs) of any sophistication rely on managing a significant amount of state. The state reflects the NIDS's model of the communications currently active in the network and also the NIDS's analysis over time, both in the past (previous activity by hosts or users, suspicion levels, relationships between connections) and in the future (timers used to model protocol interactions and to drive detection algorithms). Managing this state raises significant issues, among which are its sheer volume [7]. Another issue that to date has received less attention, concerns the degree to which the state is often tied to a single executing process. That is, often much of a NIDS's state resides solely in the volatile processor memory accessible to a single user-level process on a single machine. Generally, any state that exists more broadly than in the context of a single process is a minor subset of the NIDS process's full state: either higher-level results (often just alerts) sent between processes to facilitate correlation or aggregation, or log files written to disk for processing in the future. The much richer (and bulkier) internal state of the NIDS remains exactly that, internal. It cannot be accessed by other processes unless a special means is provided for doing so, and it is permanently lost upon termination of the NIDS (which, due to a crash, may happen unexpectedly).

In this work we argue for the great utility of incorporating *independent state* into intrusion detection systems. The goal is to enable much of the semantically rich, detailed state that hitherto could exist only within a single executing process to become independent of that process. We consider two basic types of independent state. *Spatially independent* state can be propagated from one instance of a NIDS to other, concurrently executing, instances. *Temporally independent* state continues to exist after an instance (or all instances) of a NIDS has exited. For both types of independence, the state essentially exists "outside" of any particular process.

Our contribution is not the fundamental notion of state that can be shared between processes or accessed over time—that already appears in numerous existing systems—but rather the benefits of doing so within a framework that *(i)* is *unified*, i.e., it covers all of the systems' state in the same way, and *(ii)* encompasses *fine-grained* state. This second is particularly important: by keeping fine-grained state, rather than only aggregated state such as alerts or activity summaries, we can continue to process the independent state using the full set of mechanisms provided by the system. We explore such a framework by implementing independent state for the Bro intrusion detection system [16].

Bro is a highly stateful NIDS. Its basic model has two main layers: event generation and policy script execution. Events are generated by an *event engine* which performs policy-neutral analysis of network traffic at different semantic levels. For example, there are events for attempted/established/terminated/rejected connections, the requests and replies for a number of applications, and successful and unsuccessful user authentication. The user

writes *policy scripts* using a specialized, richly-typed high-level language. These scripts execute on the events generated by the event engine and codify the actions the NIDS should take: updating data structures describing the activity seen on the network, sending out real-time alerts, recording activity transcripts to files, and executing programs as a means of reactive response. Thus, both the event engine layer and the policy script layer generate and manage a great deal of state. In this work, we strive to convert all of this state into independent state.

Independent, fine-grained state provides us with a wealth of possible applications that hold great promise for enhancing the power of a NIDS. These include coordinating distributed monitoring; increasing NIDS performance by splitting the analysis load across multiple CPUs in a variety of ways; selectively preserving key state across restarts and crashes; dynamically reconfiguring the operation of the NIDS on-the-fly; tracking the usage over time of the elements of a NIDS's scripts to support high-level policy maintenance; and enabling detailed profiling and debugging. We have implemented all of these and will discuss them in depth.

As a first example, consider a set of NIDSs at different locations of a network, each able to identify suspicious activity in its segment. Traditionally, either each NIDS works independently of its peers, or there is an explicit mechanism to send, receive and incorporate alerts. With independent state, it is possible to *transparently* leverage the others' results. We simply tell the systems what state should be synchronized among them. This state can span the range of individual analysis variables, low-level (e.g., packet signature match) or high-level (e.g., successful SSL negotiation) events, large tables storing accumulated context, or operator alerts. We further emphasize that this is only one of many applications for independent state, as we will develop subsequently.

In the next section, we give an overview of previous work related to our efforts. In §3 we then discuss the design and implementation of independent state within our architecture. We examine in §4 the powerful features and applications mentioned above that fine-grained independent state enables. In §5 we evaluate the communication performance of the architecture, and we summarize in §6.

2 Related Work

While the unifying concept of independent state has not been previously formulated in network intrusion detection research, some of its aspects can be found in earlier NIDSs. A number of NIDSs facilitate distributing the detection processing across multiple locations in a network. They employ different approaches to do so, but distribution implicitly requires the exchange of state.

NetSTAT [26] describes attack scenarios using state transition diagrams. If, due to the characteristics of an attack scenario, a single NetSTAT probe is unable to detect an attack solely by itself, it is configured with a partial scenario and communicates its analysis to other probes, thereby transferring state. MetaSTAT [27] adds dynamic reconfiguration capabilities to the STAT framework.

Emerald [17] hierarchically organizes monitors which exchange messages to propagate results and subscribe to services. GrIDS [23] models large-scale attacks by activity graphs. Its components monitor traffic at multiple locations and communicate by sending or requesting information. AAFID [22] builds on autonomous agents which communicate their results to hierarchically organized monitors. AAFID's design specifically addresses dynamic reconfiguration and acknowledges the utility of persistent state, although the prototype does not implement it.

The "Intrusion Detection Message Exchange Format" (IDMEF [9]) aims at defining a standard format to exchange alerts between different NIDSs. It differs from our work by its focus on interoperability and its restriction to the exchange of high-level state.

By setting up a network of communicating NIDSs, we are building a distributed system. The mechanisms that we employ (e.g., serialization, persistence, and synchronization) are well-established in other areas (see, e.g., [24, 21]). Applying them to network intrusion detection provides us with a wide variety of new applications.

3 Design and Implementation

We first discuss our main tool for implementing independent-state: a serialization framework. We next turn to how the user interacts with the framework by discussing its script-level interface, and finish with a discussion of addressing the need for secure and robust communication between concurrent NIDS processes.

3.1 Serialization Framework

For Bro, there are two main layers of operation, each of which stores a significant amount of state. The C++ event engine layer analyzes network traffic in a policy-neutral fashion, producing a stream of events reflecting the activity present in the traffic stream. The activity encompasses different semantic levels: individual packets, byte-stream signatures, connections, applications, and interrelationships between connections (e.g., stepping stones [28]). While the event engine's operation is tunable by redefining user-visible parameters, its algorithms—and therefore the types of state it stores—are fixed. On the other hand, the policy script layer, which executes scripts written in a custom

language over the stream of events, allows the user to arbitrarily change and extend the standard set of scripts (and in fact the user is expected to do so, to express site-specific policy). Since this layer equips the user with a full scripting language providing a rich set of control constructs and compound data types, the corresponding types of state are only determined when the scripts are loaded at run-time.

There are four main types of internal, event-engine state in Bro: connection state, analyzer state, timers, and control state. The policy script layer includes six types of state: the scripts themselves, data stored by the scripts, operations on this data (see below for why we term these a form of "state"), event generation, function calls, and byte-level signatures.[1]

The main mechanism for making the state independent is a *serialization* framework that enables us to convert all of Bro's main data structures into a self-contained binary representation and back. Once we have this, we can, for example, make state temporally independent by serializing it into a file at the termination of a Bro instance. A new instance can then read it back upon start-up. Similarly, to make state spatially independent, we can send it over the network to some remote instance.

Making object-oriented data structures serializable is, by itself, fairly straightforward and well-established practice [21]. However, adding full serializability to a complex system like Bro, which was not designed with this in mind, raises numerous subtle issues we must address. One of the basic problems that arise is the *time* needed to serialize state. Bro is a realtime system that must keep up with a high-volume stream of packets. If it spends too much time on other things than processing packets, it risks dropping packets. Therefore, we implemented *incremental* serialization: serialization proceeds in steps intermixed with packet processing. In this way, it takes more time to finish the serialization, but our ability to keep pace with the packet stream improves.

Due to limited space we must omit further discussion of a number of other issues (e.g., incremental serialization, restoring references to shallow-copied objects, using process-independent object names to synchronize multiple processes, locally rescheduling timers instantiated by other instances of the NIDS). See [19] for more details.

3.2 Using Independent State

The framework presented in the previous section is internal to Bro's event engine and hidden from the user, while the interface to the framework is defined via new semantics expressed at the policy script level. The development of the elements of the interface has been mainly driven by the needs of particular applications, and thus will continue to be extended as we gain more experience with using it. We note that having the general serialization framework in place, the semantic interface was quite easy to add, and we expect this to hold for future extensions, too.

First, we illustrate how the user can create temporally-independent state, which essentially means writing different elements of the NIDS's state into files and reading them back again later, possibly after having first modified them using other instances of the NIDS. We then discuss controlling spatially-independent state, which is done in the context of communication between multiple instances of the NIDS. All the language constructs and functions are accessible at the script-level. To ease their use, we have also developed standard scripts to accomplish a number of common tasks.

3.2.1 Temporally Independent State

To make state temporally independent, we store it in files. These files can then be read by another instance at a later point of time.

The most obvious use of temporally independent state is to make data *persistent*. The data is stored into a set of files just before a Bro process terminates, and re-read when a new instance starts up. Instead of storing all global data per default, we let the user selectively define which script-level data to save by adding an attribute `&persistent` to its type declaration. For example,

```
global saw_Blaster: set[addr] &persistent;
```

declares a set of addresses for which any changes to the set will be propagated to future invocations of Bro. Such a set is useful, for example, in tracking which addresses have already generated alerts in the past in order to reduce the volume of future alerts. Furthermore, because temporally-independent state includes its associated timestamps and timers, we could also use:

```
global saw_Blaster: set[addr]
    &persistent &create_expire=30days;
```

and Bro will delete each set element 30 days after it was added, so we will be reminded of all still-active Blasters once a month.

The reason we structure the interface so that the user explicitly marks which state to keep persistent, with all other state by default remaining non-persistent, is both that the volume of the entire set of state can be very large, and also that we find that policy scripts are often written in a style that presumes that state exists only during the execution of a single instance of Bro. We return to this point when discussing checkpointing in §4.1.

Along with `&persistent`, we also added a function `make_connection_persistent`, which tells Bro to

[1] We do not discuss signatures further due to limited space.

store the associated state of a particular connection. In addition to automatically writing all persistent state at termination, the new script function `checkpoint` can be called anytime during operation. It uses incremental serialization to avoid packet drops and can be called by another standard policy script to save Bro's state at regular time intervals. Similarly, the new function `rescan_state` reads state back from disk. One application here is to transfer data between two Bro instances. Another is more powerful: By copying a state file into that of a running instance, we can *change* its configuration on-the-fly—both the values of its global variables and also the values of its functions and event handlers, i.e., we can dynamically change the code it executes.

Along with script variables and function definitions, we also developed a way to make event generation temporally-independent. By calling the function `capture_events`, our policy script can tell Bro to write all events raised during run-time into a file. One use is to later replay these events in another instance of Bro for debugging and exploring alternate analyses.

Alternatively, we can simply print the data, either in a "pretty-printed" human-readable form, or encoded as XML (although this latter is not fully implemented at this point). This provides us with a more abstract view of network activity than raw packets, and we expect this to be highly useful for traffic analysis.

3.2.2 Spatially Independent State

For spatially-independent state, we need to transfer state from one NIDS instance to another running concurrently. We do so by establishing network connections between the instances. One of the instances calls the new function `listen`, which opens a port on the local host waiting for connections from other instances. Once a connection is established, there are several ways to exchange state. Figure 1 shows how we integrated them into Bro's architecture. Using the script-function `request_remote_events` one side can subscribe to a set of events, meaning that whenever the other side generates one of the events, it automatically forwards the event to the other side:

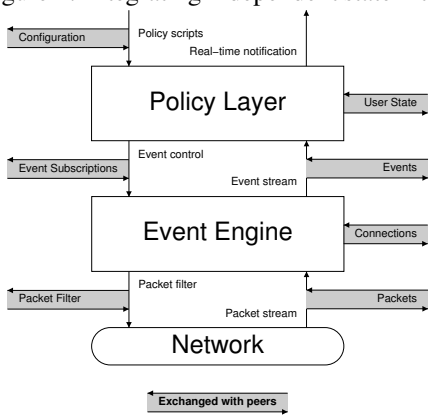
```
# Request all HTTP events from peer.
request_remote_events(10.0.0.1,
    47756/tcp, /http_.*/);
```

At the receiving end the event looks the same as one generated locally (though it's possible to test whether a particular event did indeed originate locally, or remotely).

In addition to sharing events, multiple Bro instances may share data, too. When a global script-level identifier is declared as `&synchronized`, modifications to its value will be propagated to all peers for which the identifier is also declared `&synchronized`. For example:

```
global saw_Blaster: set[addr] &synchronized;
```

Figure 1: Integrating independent state into Bro.



will cause the script variable `saw_Blaster` to be synchronized across each active Bro process. Any change made by one of them to the set will be transparently reflected in the value of the set as seen by the others.

We implemented synchronized tables by propagating changes to the data in terms of *descriptions of the operations to perform on the data* rather than the full (and probably mostly unmodified) data itself. For example, when we insert an element into a large set, we propagate "insert element 'foo' into set 'bar' ". This can in some circumstances however lead to race conditions. Avoiding them would require mutually-exclusive data operations, for example by using a token-based reservation system [24], but this would violate Bro's realtime processing constraints due to having to *wait* for access before performing an operation. For further details, see [19].

3.3 Robust and Secure Communication

Clearly, inter-NIDS communication requires robust and secure operation. Regarding robustness, a key point is that, from the perspective of the NIDS process's main functionality, inter-NIDS communication should be unobtrusive. In particular, inevitable networking difficulties such as timeouts or unexpected termination should not perturb the main operation. Therefore, rather than adding a network communication component directly into the current event engine / script interpreter structure, we instead spawn a second process exclusively dedicated to handling the communication with peers. The two processes communicate by means of a Unix pipe. (We did not use threads in order to keep their address spaces separate.) On multi-processor systems, using two processes has the additional advantage of making use of more than one CPU. Subsequently, we refer to the two processes as *main process* and *communication process*, respectively.

One key element of our design was to base it on semantically unidirectional communication. This means that

Bro's processing never expects one side to reply to something the other side sent. While doing so restricts error detection and handling somewhat, it also significantly eases implementation by avoiding having to deal with unreceived replies (which would require timeouts and a failure-recovery scheme). We believe that the decrease in complexity wins more in terms of robustness than we lose in terms of error processing. Yet, we note that the unidirectionality only affects the core-level communication. It is still quite possible to build a *script-level* handshake mechanism by passing a sequence of events between two peers.

In §3.2.2 we discussed how the NIDS's realtime constraints leads us to abide impure synchronization semantics, i.e., the possibility of race conditions. Similarly, our communication design does not make any timing guarantees for the communication. For example, transferring large amounts of data may delay the reception of an event. Also, while all state from one endpoint will always arrive in the order in which it was sent, state from multiple endpoints may be received intermixed.

Along with designing for robust communication, we also need to secure the communication, i.e., provide for confidentiality and authentication. We do so via SSL (implemented using OpenSSL [15]). See [19] for a discussion of such security aspects.

3.4 Integrating External State

While we performed our initial implementation of independent state in the context of the Bro NIDS, the concept applies more generally to other applications as well. In principle, *any* system, not just other NIDS instances, may choose to make its fine-grained internal state accessible to peers.

As a major step in this direction, the lightweight, highly portable *Broccoli*[2] library [4] enables arbitrary applications to partake in the exchange of Bro's state. Broccoli is an independent implementation of (parts of) the serialization/communication protocol that we have developed for the Bro system. Broccoli nodes can request, send, and receive events just like Bro instances can. In [8], we demonstrate some of the power of such NIDS-external independent state by supplying the Bro system with host-based application context.

4 Applications

We now describe several powerful applications of independent state in network intrusion detection. We first show how we can use independent state to greatly enhance Bro's traditional model of regular checkpointing, including support for robust crash recovery. Then we discuss distributed intrusion detection, concentrating on the utility of spatially independent state. Finally, we show how independent state can be used for dynamic reconfiguration, profiling, and debugging.

We implemented each of these applications. Given independent state, combined with the NIDS's flexibility, we found all rather easy to achieve. Although at first blush each might seem to be yet-another-extension of Bro's generally-extensible functionality, the *ease* of implementation proves the power of the approach.

Our experiences with these applications come from monitoring the access links in several large environments: the Münchner Wissenschaftsnetz (*MWN*; research network including two universities and other institutions; Gbps, heavily loaded), the University of California, Berkeley (*UCB*; Gbps, heavily loaded), and the Lawrence Berkeley National Laboratory (*LBNL*; Gbps, medium load).

4.1 Checkpointing

IDS's face fundamental state management problems. Either the system uses a static allocation of state for its analysis, in which case it becomes vulnerable to easy forms of attacker evasion; or it allocates different types of state dynamically, in which case managing and reclaiming that state becomes a major burden. While Bro provides a variety of timers for use in state management, from operational experience we have found that state still inexorably accrues, in part due to our reluctance to assign timers to every data item because it's hard to determine good *a priori* settings for these, or even identify all of them (there are hundreds of script-level variables).

To date, Bro's only support for large-scale state reclamation has been the brute force approach of simply starting over from scratch. That is, to run Bro 24x7 we (and other Bro users) resort to *checkpointing*, which in this context means periodically starting up a new instance of Bro and killing off the old one. The frequency with which this is done ranges from daily (LBNL) to every few hours (MWN, UCB).

For Bro, the two main types of state lost when checkpointing are internal connection state (including analyzer-specific state and attached timers) and script-level data. However, the concept of persistence described in §3.2.1 now enables us to individually choose connections (by calling `make_connection_persistence`) and script-level data (via `&persistent` declarations) to transform into independent state, thus enabling a new Bro instance to use them as part of its initial state. Doing so allows us to continue longer-running forms of analysis uninterrupted, such as tracking slow scans, long-lived interactive connec-

[2] *Broccoli* is the healthy acronym for "Bro Client Communications Library."

tions, usernames, inferred software versions, alerts already generated, and addresses that Bro has blocked in the past using its dynamic blocking facility.

While temporally-independent state thus enables us to keep key state across restarts, implementing it soundly also requires a dynamic *handover* mechanism. The problem here is that the currently executing instance of Bro has to save its persistent state at some specific point in time, *after* which the new instance can begin executing. If we have to wait for the new instance to start up, we will incur a monitoring outage. We solve this problem by using not temporally independent state, but spatially independent. We implement dynamic handover by starting up the new instance and having it connect via a (local) network connection to the old instance, requesting its current set of persistent state. After this has been successfully transmitted, the old instance terminates itself, and the new one starts processing.

As already discussed, we do not simply make *all* state persistent. Doing so would defeat the purpose of checkpointing. But having the tools now to selectively make state persistent, the next step is to identify the state for which this makes sense. For our operational environments, we keep internal connection state for interactive services that tend to have long-lived connections: FTP, SSH, *telnet*, and *rlogin* connections. For script-level data, we took Bro's default policy scripts (as of version 0.8a57) as representative for the usage of state in Bro scripts. Our first observation is that nearly all of the scripts store their relevant data in tables or sets. We found five basic usages: (1) remembering messages already logged to avoid duplication, (2) remembering hosts which have done "something" (e.g., propagating a worm), (3) associating additional state with connections (e.g., which FTP data connections have been negotiated by a control channel), (4) holding configuration data, such as particular hosts allowed to do "something", (e.g., connect to a certain host; this data is more or less fixed), (5) remembering additional data derived from the script's analysis (e.g., software installed on a host).

Taking the MWN environment as a test case, we made all tables belonging to the first group persistent. Most of these tables are low in volume,[3] and suppressing unnecessary log messages is a vital NIDS capability [2, 10]. For the second group, we differentiate between short-term (minutes or less) and longer-term data. The former is often quite large in volume and often not worth keeping. For example, the script recognizing the Blaster worm [3] by its scanning activity keeps two tables: one tracking hosts that have communicated over TCP port 135 within the last five minutes, and the second remembering all already-identified worm sources. We decided to make only the latter persistent.

[3] With the notable exception of the table `weird_ignore` recording all the "crud" [16]. In large networks, we see tons of crud.

The third group (associating additional state) is more problematic. Ideally, we would like to keep information for all *persistent* connections, but discard all the rest. But to do so the scripts would need significant restructuring, as their semantics vary too much to automatically deduce which information is associated with persistent connections. There are some tables, though, which we know always correspond to state for persistent connections (e.g., the FTP analyzer script remembers FTP connections). We made these persistent, but left all other tables unmodified (i.e., ephemeral).

We also left the fourth group untouched, as configurations are mostly static and better changed manually if the need arises. Finally, for the last group we found we needed to make case-by-case decisions. For example, to keep vulnerability profiles [20] one of the scripts detects the software used by different hosts, an excellent example of information we declare `&persistent` so we do not lose it.

4.2 Crash Recovery

A related application of independent state is better recovery from crashes. Three main reasons for the crash of a NIDS are resource exhaustion, attacks, and programming errors [16]. In most systems, including Bro, in each case we lose all the state so far collected by the system. By using the `checkpoint` function (see §3.2.1) regularly, however, we can significantly mitigate the effects of crashes, so that we only lose data accumulated since the last checkpoint.

Our experience is that crash recovery is invaluable. This is not only the case when actively developing the IDS, but also in a production environment, where crashes are still a fact of life, particularly due to resource exhaustion. Not only does crash recovery allow us to continue operating with only a minor loss of state (in terms of the importance of the state), but the checkpoint also allows us to analyze the particularly significant state post mortem (cf. §4.4).

4.3 Distributed Analysis

Once we've provided a means for a NIDS to communicate its state, we can then use that mechanism to distribute its analysis. To date, distributed NIDS have generally imposed a specific model on the form of distribution. For example, DIDS [18] pioneered the *sensor model*, gathering low-level data remotely while performing higher level semantic analysis centrally. On the other hand, Emerald [17] constructs a hierarchical structure to propagate information up to the root level.

Independent, fine-grained state opens up new degrees of flexibility for distributed analysis. In this section we look at three different models, all of which we have been able to implement and experiment with by having added independent state to Bro. The first model supports load-balancing

for monitoring high-volume links. The second supports the well-known "distributed sensor" model. The third looks at propagating information between decoupled systems.

4.3.1 Load-balancing

On today's high-volume links,[4] it is exceedingly difficult to analyze the full packet stream with a single NIDS (unless one utilizes custom hardware [14]). One strategy for coping with such a load is to distribute the analysis across several machines, each doing only a part of the work. A key question then is how to coordinate their operation. Currently, using Bro operationally we do this by running several independent instances on different slices of the network traffic. But without any state sharing, this loses important information. Thus, our goal is to retain the depth of analysis a single Bro could in principle achieve if it could cope with the load.

To this end, we first need to decide how to divide the traffic between the multiple systems. We can either do so statically (each system gets all packets matching some fixed criteria) or dynamically (e.g., for each connection we decide individually which system will analyze it). Our initial efforts have focused on static approaches due to their simplicity, distributing based on: *(i)* the local IP space, or *(ii)* application protocol.

Dividing by IP space: Fruitfully spliting up the local IP space requires knowledge of the network to find a division so that individual NIDS receive comparable loads. From our operational experience, measuring the volume and leveraging the expertise of the network's administrators to do so is not hard. The main advantage of distribution based on dividing the IP space is the ease of further distributing the load by introducing additional systems. The main disadvantage is that, without any communication, we cannot correlate traffic between different subnets anymore, such as detecting scans.

To assess this approach, we examined the Bro 0.8a53 policy scripts to determine the degree of communication they would require. We found that there is one dominant case where without communication we would lose information: several scripts store information about individual hosts, usually of the form "host a.b.c.d did something [n times]". For example, the worm detection script keeps a table storing all already-known worm infectees. Not propagating this state among the concurrent Bro's would have two effects: *(i)* each of the instances would alert individually if it recognizes the worm, and *(ii)* more importantly, if an instance cannot identify the worm by itself, it obviously cannot use this information in other contexts (e.g., treat signatures matching a known worm infectee different from other matches).

[4] E.g., the traffic level in the MWN (UCB) environment sustains more than 250 (400) Mbps averaged over an entire day.

With spatially independent state, however, we can easily solve these problems by declaring the tables `&synchronized` (per §3.2.2). Now each instance propagates its state to the peers.

Dividing by application: To divide the load by application, we delegate applications that make up a significant share of the load to dedicated systems. If, for example, there is a large fraction of HTTP traffic, we could exclude HTTP processing from the main system and move its analysis to another machine. This is in fact what we do operationally at LBNL. But this approach lacks general scalability: the load is only significantly reduced if the NIDS does indeed spend quite some time processing the particular application. For Bro, this is true for HTTP (due to Bro's detailed analysis of the HTTP sessions), and also for a few other applications, but these total only a handful.

Again we examined the scripts to assess where division by application would require inter-Bro communication. While usually for application-specific analysis no communication is needed, one exception is the FTP analyzer because it parses the PORT negotiation of FTP data connections. A more general problem concerns analyzers that need to see traffic from all applications, such as Bro's scan detector. To detect vertical port scans, it counts connection attempts to different ports (applications) per host. Other examples include the ICMP analyzer correlating ICMP "unreachable" messages with corresponding connections, and the analyzer that derives vulnerability profiles [20].

It appears clear that the communication for these analyses can be addressed using spatially independent state, and we expect to gain operational experience in doing so at LBNL, where it has long been desired to coordinate the separate HTTP Bro.

4.3.2 Sensor Model

A well-established architecture for distributed network intrusion detection is the sensor model [1], in which we place *sensors* at different points in the network, usually performing low-level analysis like protocol-decoding or byte-signature matching. The sensors then send their results to an *analyzer* which correlates the data from all of its input sources.

Bro is conceptually well-suited for this kind of deployment. Its architecture already clearly separates between low-level and high-level analysis by means of its division into event engine and policy script interpreter. The main interface between these two layers are the events. So, the most obvious way to apply the distributed sensor model to Bro is to spatially separate the event engine from the script layer (i.e., run them as separate processes). This becomes easy to achieve using spatially independent state.

However, as we will discuss in Section 5, propagating large volumes of data comes at a non-negligible cost.

Therefore, while propagating all of the event layer's information will work well for smaller setups, it does not scale to high-volume networks. Hence, in such environments it is more promising to partition the processing a layer up. That is, the sensors would perform the usual script-level analysis in addition to their event engine processing, with those scripts synchronized as discussed in §4.3.1, and then we would dedicate an additional CPU to correlating their combined output at a meta-level, for example by using correlation methods such as presented in [6, 13, 25].

As a first step in this direction we implemented a simple but operationally very fruitful extension: combining all log messages coherently in a single place. In the MWN setup, a central server receives all output in realtime. We note that while this is available in other distributed NIDSs, their communication is often *restricted* to the exchange of log-like messages. With our architecture, centralized logging is an almost trivial application: each script-level log statement generates an event which gets propagated to the central server.

4.3.3 Propagating Information

Another potentially valuable application of spatially independent state is using it to tell other systems some facts about our analysis. We discuss two examples here, the first (intensifying analysis for suspicious hosts) of which we have already experimented with, and the second (propagate IPs that we have chosen to dynamically block) of which we plan to set in place in the near future.

Suspicious hosts: As mentioned above, due to the large load on a high-volume link, a single system cannot run detailed analysis on the full traffic. One solution is to run only coarse-grained analysis on all of the traffic, but to intensify the inspection for hosts found to be conspicuous. For example, often administrators observe that attackers first perform scans of the network before actually targeting some hosts. Large scans are easily detectable using coarse-grained analysis. After identifying a scanner, we can then look at the packets coming from the same source in more detail.

We implemented this by running two instances of Bro. The first instance watches a large fraction of the traffic but only runs a modest set of policy scripts (most notably the scan detector). When it generates an alert for some host, it also sends an event containing the host's IP address to the second Bro instance. By default, the second instance does not see any traffic at all. But if it receives such a suspicious address, it modifies its analysis to include all packets coming from that source. In addition to using more scripts and a large set of signatures, it saves the complete set of packets to disk.

Propagating blocked hosts: Our LBNL environment currently runs several Bro's at different entry points into the network. As discussed in [11], LBNL's security policy includes dynamically blocking scanners detected by Bro by modifying the border router's access control list. Because not infrequently a scanner first probes a set of addresses corresponding to one entry point and then later another set corresponding to a different entry point, there is considerable operational interest in enabling the different Bro's to communicate their blocking decisions to one another.

4.4 Reconfiguration, Profiling and Debugging

A final set of applications for independent state leverage the broader notion of independent state encompassing not only data values but also functions and policy script event handlers. Such independent state allows us, first, to both tune and retarget the system without having to restart it; and, second, to inspect the system's state during run-time in several different ways.

Dynamic Reconfiguration: We can use the independence of broader forms of state (functions and event handlers) to dynamically reconfigure a running Bro. Doing so supports both operational flexibility and tuning. In terms of operational flexibility, frequently during daily operations the need arises to change the configuration of the NIDS in response to a newly perceived threat or problem. For example, we have detected a break-in and now want to alert on any return by the attacker; or, we have learned a new attack signature and want to immediately start using it; or, a new source of benign traffic has appeared which is overwhelming the NIDS and we want to skip processing it for now. These all can occur in a *fire-fighting* mode, i.e., we really need to deploy the change immediately. With independent state, we can introduce such changes—including modified function and event handler definitions—directly, without incurring the loss of fine-grained state that would occur using our enhanced checkpointing.

Another use of dynamic reconfiguration is to support *tuning*, i.e., optimizing the NIDS's configuration for the local environment. From our experience, one of the most common problems with making configuration changes for tuning is that the effects of the changes often do not show up immediately. Until now, making such changes has required restarting the NIDS, with the consequent loss of the system's state. In addition, the effects of many changes are only visible when the system has built up a significant amount of state, which can take a long time after a conventional restart. This is particularly true for configuration parameters like timeouts and thresholds. We can ameliorate this problem by collecting traffic traces and testing against them off-line, but such traces can be huge and unwieldy to work with.

While our enhanced checkpointing can help, it does not fully solve the problem. Often when making many small

changes in a short time, we do not actually want the controlled loss of state which checkpointing achieves, but prefer to keep *all* state. We want ideally to have the system just pick up the changes and keep running, similar to the fire-fighting changes discussed above.

The way we do such on-the-fly changes in practice is as follows. Consider an already-running Bro whose configuration we would like to change in some respect. We first make the modification to the configuration, i.e., edit the scripts. We then convert the full configuration into persistent state, stored in a file, as described in §3.2.1. Finally, we copy the file into a directory regularly checked by the running instance, which notices the update, loads it, and switches to using it. No other state is lost.

Profiling and Debugging: Another significant problem when operating a NIDS is understanding its behavior during operation. When developing policy scripts, we find they can work in unexpected ways, due to either programming errors, or to encountering network traffic with different characteristics than we expected. These kinds of problems are very hard to track down, as often they only manifest themselves after a considerable amount of run-time.

We find it is a great help if we can take a look at Bro's current state. With independent state, this becomes easy to achieve, since the files generated by `checkpoint` contain all the necessary information. In Figure 2(a), for example, we see the table containing all currently known port scanners at a given point of time. Included in the output are timestamps when the entries were last accessed. In Figure 2(b), we see the same table from about 1.5 hours later. For larger tables, the differences may be hard to see, but the ASCII output formats are suitable for processing with Unix utilities such as *sort* and *diff*, as illustrated in Figure 2(c). Now the differences become obvious; we can actually *see* the scan detector working.

Along with data values, our implementation of independent state provides timestamping for script functions, too. Figure 2(d) shows a checkpoint of the `check_hot` function from the default scan detection script. The different basic blocks in the code are annotated with timestamps indicating the last time they were executed, and with counts of how many times they have been executed. These annotations can be invaluable for profiling, assessing code coverage, and detecting stale script elements. We can again use tools like *diff* to dynamically track which portions of the code are being executed, and how frequently.

5 Performance Evaluation

To examine the performance of our architecture in more detail, we assessed the *communication system*, as this encompasses most of the other components (e.g., the serialization framework). First, we used synthetic stress-tests to gauge the maximal throughput. Then we turned to real-world traffic captured in the MWN environment to assess the system's performance on realistic data.

For all of our experiments, we used two machines, acting as sender and receiver. For most of the measurements, the systems ran a Linux 2.6.x kernel. For the experiments involving live-capture of network traffic, we used hosts running FreeBSD 5.2.1. In both cases, the sending systems were dual-Xeons, 3Ghz, with 2GB of RAM; the receivers were dual-Opterons, 1.8Ghz, with 2GB of RAM. Senders and receivers were connected by a 100Mb/s switch.

5.1 Benchmarks

First, we instrumented the sender to emit events at configurable rates. Starting with no output at all, we increased the rate by 1000 events every 5 seconds until either the sender or the receiver could not handle the load anymore. Simultaneously, the sender sent out *ping* messages every second to which the receiver responded with *pongs*, measuring the lapsed interval as a *ping-time*. At the time the sender's main process sees the pong, the ping/pong combo has traversed four different processes (sender's main, sender's communication, receiver's communication, receiver's main, and back). Thus, the ping-time is a measure for the lag which the communication introduces. The smaller the ping-times, the faster events (and other information) are propagated. If the ping-times start to increase, it is a sign that some queue on the path is filling up, i.e. a limit has been reached.

Figure 3(a) shows the rate of emitted events as well as the ping-times. We see that with increasing output the ping-times were roughly constant (with a median of 2ms) until the sender's rate got to about 44,000 events per second (which comprise a volume of 8.3MB per second). At this point, the ping-times exceeded 0.1s for the first time and continued to increase. It turned out that it was the receiver that became overloaded. For every received event, its main process had to call a (empty) script-level handler, which became too much of a burden eventually. In our synthetic benchmark, the sender itself did not raise the events but only sent them out. Therefore, event handling was not a problem on its side.

Events have different types of arguments. The events used for Figure 3(a) carried two parameters: a string and a compound connection type, which is rather complex, containing more compound objects itself. If we reduce the complexity of the arguments we are able to achieve higher rates, up to more than 100,000 events per second when sending events without any parameter at all.

We conducted similar benchmarks with state-propagating operations (as triggered by `&synchronzized` script variables; see §3.2.2) and

Figure 2: Visualizing state (addresses randomized)

```
ID reported_port_scans = {
 [165.11.184.36, 148.126.197.84, 100]  @01/21-12:11
 [138.112.68.194, 108.45.144.114, 1000]  @01/21-11:00
 [138.112.68.194, 108.45.144.114, 100]  @01/21-10:59
 [138.112.68.194, 108.45.144.114, 10000]  @01/21-11:01
 [138.112.68.194, 108.45.144.114, 50]  @01/21-10:59
 [165.11.184.36, 148.126.197.84, 50]  @01/21-12:11
}
```

(a) Subset of reported port scans at time T

```
ID reported_port_scans = {
 [138.112.68.194, 108.45.144.114, 50]  @01/21-10:59
 [138.112.68.194, 108.45.144.114, 10000]  @01/21-11:01
 [138.112.68.194, 108.45.144.114, 1000]  @01/21-11:00
 [163.184.146.140, 146.74.170.189, 50]  @01/21-13:16
 [165.11.184.36, 148.126.197.84, 100]  @01/21-12:11
 [165.11.184.36, 148.126.197.84, 50]  @01/21-12:11
 [138.112.68.194, 108.45.144.114, 100]  @01/21-10:59
 [163.184.146.140, 146.74.170.189, 100]  @01/21-13:16
}
```

(b) Subset of reported port scans at time T + 1.5 hr

```
  [138.112.68.194, 108.45.144.114, 1000]  @01/21-11:00
  [138.112.68.194, 108.45.144.114, 100]  @01/21-10:59
  [138.112.68.194, 108.45.144.114, 50]  @01/21-10:59
+ [163.184.146.140, 146.74.170.189, 100]  @01/21-13:16
+ [163.184.146.140, 146.74.170.189, 50]  @01/21-13:16
  [165.11.184.36, 148.126.197.84, 100]  @01/21-12:11
  [165.11.184.36, 148.126.197.84, 50]  @01/21-12:11
```

(c) `diff` of (a) and (b)

```
ID check_hot = check_hot
(@01/21-12:23 #4715580)
{
local id = c$id;
local service = id$resp_p;
if (service in allow_services ||
    c$service == "ftp-data")
  (@01/21-12:23 #2932175)
  return (F);

if (state == CONN_ATTEMPTED)
  (@01/21-12:23 #1138955)
  check_spoof(c);
[...]}
```

(d) Script function from `scan.bro` with timestamps.

full network packets. We omit the corresponding plots here as they look similar to Figure 3(a). For state operations, we sent table assignments of the form `t[index]="string"`. We could send up to 58,000 such operations before the ping-times exceeded 0.1s. Again it was the receiving main process which was not able to keep up. To measure sending raw packets, we transfered a trace captured in the MWN environment. The ping-times began exceeding 0.1s when the transmission rate hit 16,000 packets per second. The data rate corresponded to more than 11 MB/s, approaching the maximum bandwidth of the link.

We repeated the benchmarks with SSL encrypted sessions, finding that the amount of objects that could transmit decreased noticeably. Figure 3(b) shows a fall-off to 11,000 events per second for when the ping-times crossed the 0.1s limit.

To summarize, our architecture can transfer tens of thousands of objects per second, and, therefore, seems suitable for use in high-performance environments. However, we note that these benchmarks represent a best-case: the Bro system is concentrating solely on communication. Thus, we now evaluate performance with additional packet input.

5.2 Performance on Realistic Data

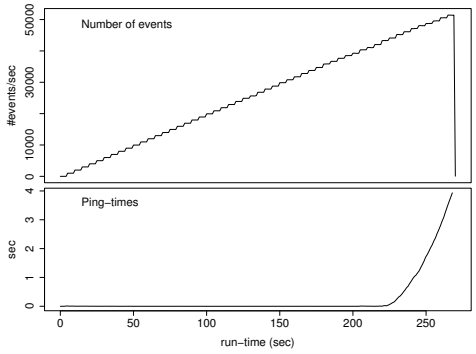
The benchmarks presented in §5.1 suggest that the communication framework is not going to be a bottleneck. However, in a high-volume environment, the packet processing itself is already a very demanding task. Thus, we will now examine how well the communication blends in.

Pseudo-realtime: First, we need a methodology to perform *reproducible* measurements. A common approach to evaluate the performance of a NIDS is to capture a packet trace and run the NIDS offline on it with different configurations. However, to evaluate communication performance, this approach does not work well: the NIDS can process a trace more quickly than the corresponding realtime. This leads to its analysis time being "compressed" (we term this *trace time*). Yet, the communication, located in a separate process, is decoupled from trace time; it is performed in *realtime*.

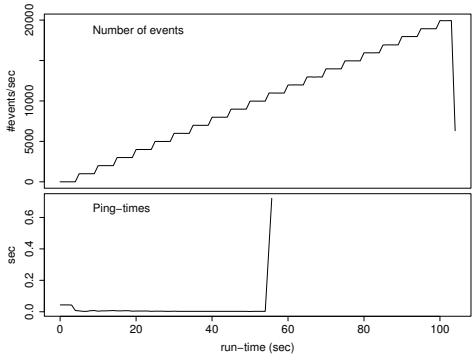
Nevertheless, we wanted to keep the trace-based evaluation approach for its reproducibility. Hence, we needed to synchronize network time and realtime. To this end, we added a *pseudo-realtime* mode to Bro. If activated, the main process reads packet input from a trace but deliberately inserts delays into its processing. These delays resemble the inter-packet gaps observed when capturing the trace. That is, the processing of a new packet is deferred until the corresponding amount of realtime has passed.

Given the same input, a pseudo-realtime Bro performs the same operations as a live Bro, i.e., the two do not differ in terms of detection. However, the times at which operations are performed could slightly vary, leading to different system and network loads. To ensure that the pseudo-realtime mode does indeed provide similar results in terms of load as running on live traffic, we performed an experiment. We started a Bro process on live MWN input and simultaneously captured the traffic on the same machine, using the same BPF filter as the Bro process. Then we reran Bro offline on the trace in pseudo-realtime, configuring it to use its default set of analyzers (plus some reduced timeouts and including UDP packets). To avoid losing packets, we excluded two high-volume networks from the analysis. The resulting 30-minute trace had a volume of 832MB, consisting of 4.9M packets (61.0% TCP, 39.0% UDP). Based on ports, HTTP and DNS were the most prevalent protocols

Figure 3: Propagating increasing number of events.

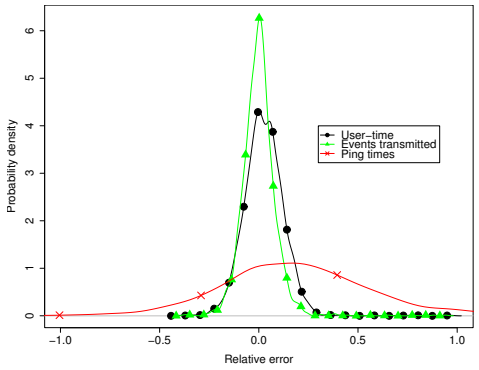


(a) Plain connection.

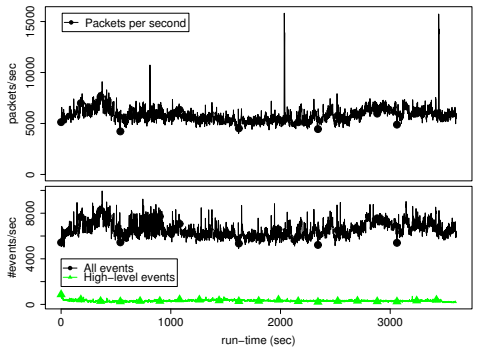


(b) SSL connection.

Figure 4: Pseudo-realtime



(a) Bro running live vs. pseudo-realtime



(b) Number of (filtered) packets and events on trace.

(29.1% and 8.1% of the packets, respectively). The filtered trace contained 1.4M connections/flows.

During the runs we logged the user-level CPU utilization of the main process and the number of transmitted events, both per one-second intervals. Additionally, we measured ping-times once per second. Then we calculated the relative errors of the pseudo-realtime figures compared to the live figures. Figure 4(a) shows the corresponding densities. For user-time and number of transmitted events, we see a nearly perfect match. The errors were larger for the ping-times. The median of the relative errors was 0.16, the standard deviation was 0.47. That is, in the live run, on average the pings needed longer than in pseudo-realtime. However, the median of the *absolute* errors was 2ms (standard deviation 5ms), i.e. the differences were in fact rather small in absolute terms. We assume that these discrepancies stem from the increased system load when running live. In fact, examining the *system* times of the main processes, the median of the relative errors was much larger than for the user times (0.64 vs. 0.02). This is due to the live packet capturing and filtering which takes place inside the kernel.

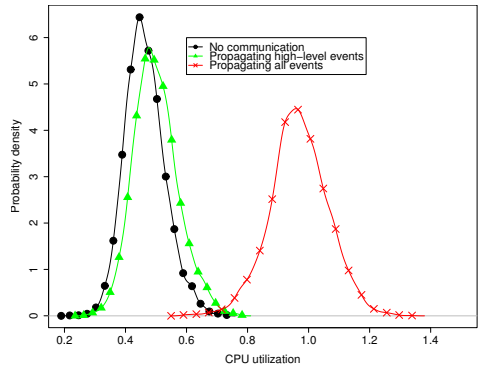
All in all, we believe that pseudo-realtime provides us with reproducible yet realistic measurements. Therefore, we used it to examine the communication framework in more detail.

Measurements: Having the pseudo-realtime mode in place, we examined three different Bro configurations to understand the impact of communication. We captured a one-hour packet trace in the MWN environment. We included *all* packets except those of one high-volume subnet. To ensure that we do not lose packets, we captured the traffic with a high-performance Endace DAG card [5]. The trace spanned 1 hour with a volume of 88GB and a mean rate of 40K pps. It consisted of 5.2M flows, 144M packets, 92.2% TCP and 6.7% UDP, with HTTP and SSH the most common protocols (50% and 6% respectively).

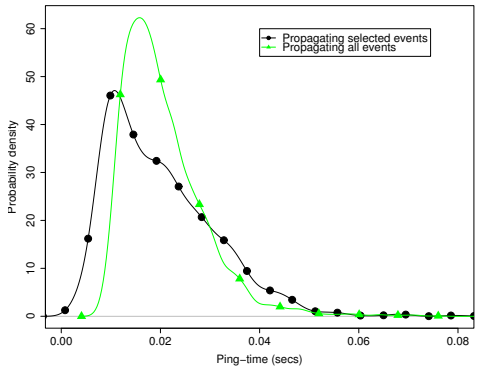
We first run Bro without any communication at all (using the same setup as before). Then we configured it to propagate *all* events to its peer. Finally, we changed the configuration to emit only a subset of events, which consisted of all but the connection setup and tear-down events (and their UDP equivalents raised for basic UDP requests and replies). These events are semantically rather low-level but comprise 95% of all events in our trace. Figure 4(b) shows the differences in the number of events over time.

For the three different configurations, in Figure 5(a) we see the densities of CPU utilization per second. The measured utilizations are the total sum of user and system time of both main and communication processes (which is why on the two-CPU machine the values can exceed 1). We

Figure 5: Different configurations on trace.



(a) Total CPU utilization.



(b) Ping-times

see that when emitting all events the average utilization increased noticably: the median shifts from 0.46 without any communication to 0.97. Yet, with the subset of events the performance impact is only marginal (median 0.49). Looking at the ping times (Figure 5(b)), we hardly see a difference between the two runs which involve communication: with all events, the median of the ping-times was 19ms while with the subset it was 18ms. Thus, the communication system was working well within its capacity limits. In general, the ping times are pleasantly low, considering the ping/pong path across four processes which also had to handle a significant packet and event load.

Based on these results we draw two main conclusions. First, our architecture works well enough to support propagating thousands of events even when having to handle a high packet load (which may either represent normal activity or be due to an attack). However, in such situations it does incur a noticable overhead: the increased CPU utilization is non-negligible. This implies that simply forwarding *all* events will not scale very well in larger installations. Giving the amount of traffic, this is hardly surprising. However, we also see that with smaller amounts of events, the performance overhead is not significant. Thus, distribution schemes which focus on propagating higher-level events are a very promising approach for large-scale installations.

6 Summary

In this work we demonstrated the power of exploiting *independent state* in network intrusion detection. While traditionally much of a NIDS's state resides solely in volatile memory, we instead argue for making all of a NIDS's state exist (potentially) "outside" of any particular process. To this end, we developed the notions of *spatially independent state* (state that can be propagated from one instance of the NIDS to another concurrently running process) and *temporally independent state* (state that continues to exists after the termination of all instances, available to future processes). The architecture we implemented facilitates independent state for the Bro intrusion detection system [16]. It is *unified* in that it encompasses all of the internal, fine-grained state of the NIDS. Thereby, we can continue to process independent state using the full set of mechanisms provided by the system.

The main internal mechanism of our architecture is a *serialization framework*. While its implementation was straight-forward in general, the system's internal complexity gave us a number of subtle issues to solve. Having the serialization in place, we added a user-level interface driven by our operational applications. It enables the user to selectively declare state to be independent. To achieve temporal independence, we serialize state into files, either when an instance exits, or incrementally as it executes. A subsequent process can then read it back. To achieve spatial independence, we added secure network communication to the NIDS, allowing instances to share state across different locations.

The architecture provides us with a wealth of possible applications. We enhanced Bro's traditional model of regular checkpointing by allowing a *controlled* loss of state, added crash-recovery, examined different approaches for distributing the monitoring and analysis, enabled run-time policy management, and greatly extended the system's profiling and debugging facilities. These applications were driven by our operational experiences, and we experimented with all of them in several large-scale environments. A performance evaluation of the communication component shows that our implementation is suitable for deployment even in large-scale installations. Our architecture has been included into the latest Bro development version, and we are in the process of setting up our monitoring environments to use independent state operationally. We expect that in regular operational use, the power of independent state will soon prove invaluable. Moreover, a client library has been developed to interface trusted applications to the Bro system [4]. With its help, we have already successfully integrated Apache web servers [8] and SSH servers into Bro's analysis. In addition, we are working on extending Bro's event model to more directly support scalable distributed event analysis [12]. Furthermore, for users who are less fa-

miliar with the details of Bro's operation, we are planning to provide predefined sets of related script-level objects. This will enable the user to share coherent chunks of state among instances easily, rather than needing to identify individual relevant script-level variables and events himself.

7 Acknowledgments

We would like to thank the Lawrence Berkeley National Laboratory (LBNL), Berkeley, USA; the Leibniz-Rechenzentrum, München, Germany; and the University of California, Berkeley, USA. We would also like to thank Anja Feldmann for supporting our work and providing feedback, and Mark Allman and Scott Campbell for their helpful comments. Finally, we would like to thank the anonymous reviewers for their valuable suggestions.

This work was made possible by the U.S. National Science Foundation grant STI-0334088, for which we are grateful.

References

[1] E. G. Amoroso. *Intrusion Detection: An Introduction to Internet Surveillance, Correlation, Trace Back and Response*. Intrusion.Net Books, New Jersey, 1999.

[2] S. Axelsson. The Base-Rate Fallacy and Its Implications for the Difficulty of Intrusion Detection. In *ACM Conference on Computer and Communications Security*, pages 1–7, 1999.

[3] CERT Advisory CA-2003-20 W32/Blaster worm. `http://www.cert.org/advisories/CA-2003-20.html`.

[4] Broccoli: The Bro Client Communications Library. `http://www.cl.cam.ac.uk/~cpk25/broccoli/`.

[5] ENDACE Measurement Systems. `http://www.endace.com/`.

[6] H. Debar and A. Wespi. Aggregation and Correlation of Intrusion-Detection Alerts. In *Proc. Recent Advances in Intrusion Detection*, number 2212 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[7] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational Experiences with High-Volume Network Intrusion Detection. In *Proc. 11th ACM Conference on Computer and Communications Security*, 2004.

[8] H. Dreger, C. Kreibich, V. Paxson, and R. Sommer. Enhancing the Accuracy of Network-based Intrusion Detection with Host-based Context. In *Proc. Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, 2005.

[9] Intrusion Detection Message Exchange Format. `http://www.ietf.org/html.charters/idwg-charter.html`.

[10] K. Julisch. Clustering Intrusion Detection Alarms to Support Root Cause Analysis. *ACM Transactions on Information and System Security*, 6(4):443–471, 2003.

[11] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *Proc. IEEE Symposium on Security and Privacy*, 2004.

[12] C. Kreibich and R. Sommer. Policy-controlled Event Management for Distributed Intrusion Detection. In *Proc. 4th International Workshop on Distributed Event-Based Systems*, 2005.

[13] C. Krügel, T. Toth, and C. Kerer. Decentralized Event Correlation for Intrusion Detection . In *Proc. Information Security and Cryptology*, volume 2288 of *Lecture Notes in Computer Science*, 2001.

[14] C. Krügel, F. Valeur, G. Vigna, and R. A. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proc. IEEE Symposium on Security and Privacy*, 2002.

[15] `OpenSSL. http://www.openssl.org`.

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

[17] P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *National Information Systems Security Conference*, Baltimore, MD, October 1997.

[18] Snapp, S., et al. DIDS (Distributed Intrusion Detection System) – Motivation, Architecture, and an Early Prototype. In *Proc. 14th NIST-NCSC National Computer Security Conference*, 1991.

[19] R. Sommer. *Viable Network Intrusion Detection in High-Performance Environments*. PhD thesis, TU München, 2005.

[20] R. Sommer and V. Paxson. Enhancing Byte-Level Network Intrusion Detection Signatures with Context. In *Proc. 10th ACM Conference on Computer and Communications Security*, 2003.

[21] J. Soukup. *Taming C++ – Pattern Classes and Persistence for Large Projects*. Addison-Wesley, 1994.

[22] E. H. Spafford and D. Zamboni. Intrusion Detection Using Autonomous Agents. *Computer Networks*, 34(4):547–570, 2000.

[23] Staniford-Chen, S., et al. GrIDS – A Graph-based Intrusion Detection System for Large Networks. In *Proc. 19th NIST-NCSC National Information Systems Security Conference*, 1996.

[24] A. S. Tanenbaum and M. V. Steen. *Distributed Systems – Principles and Paradigms*. Prentice Hall, 2002.

[25] A. Valdes and K. Skinner. Probabilistic Alert Correlation. In *Proc. Recent Advances in Intrusion Detection*, number 2212 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[26] G. Vigna and R. A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 7(1):37–71, 1999.

[27] G. Vigna, R. A. Kemmerer, and P. Blix. Designing a Web of Highly-Configurable Intrusion Detection Sensors. In *Proc. Recent Advances in Intrusion Detection*, number 2212 in Lecture Notes in Computer Science, 2001.

[28] Y. Zhang and V. Paxson. Detecting Stepping Stones. In *Proc. 9th USENIX Security Symposium*, pages 171–184. The USENIX Association, 2000.