T U M

I N S T I T U T F Ü R I N F O R M A T I K

Exploiting Independent State For Network Intrusion Detection

Robin Sommer, Vern Paxson

TUM-I0420
November 08

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-11-I0420-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Exploiting Independent State For Network Intrusion Detection

Robin Sommer
TU München
Germany
sommer@in.tum.de

Vern Paxson
ICSI / LBNL
Berkeley, CA, USA
vern@icir.org

Technische Universität München
Technical Report #TUM-I0420

November 8, 2004

Abstract

Network intrusion detection systems (NIDSs) rely on managing a significant amount 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 develop an architecture that facilitates *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.

Our unified architecture provides us with a wealth of possible applications that hold promise for enhancing the power of a NIDS. We examine how we can 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.

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 and how the NIDS can efficiently retrieve elements from it. A third issue, however, and one that to date has received less attention than the first two, 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. Usually, 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 develop an architecture that facilitates *independent state* for the Bro intrusion detection system [Pax99]. The goal of the architecture 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 (such as a Bro process) 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 in a sense exists "outside" of any particular process.

Our contribution concerns not the fundamental notion of state that can be shared between processes or accessed over time—that already appears in existing systems—but rather an architecture for doing so 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 to the architecture's power: because we keep 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 believe no existing NIDS incorporates such a general mechanism.

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 after presenting the architecture.

As a first example, consider a set of NIDSs at different locations of a network, each able to identify suspicious activity in its network 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 general, we can distinguish between several types of state. We will refer to state that does not change over the course of the execution of an instance of a NIDS as *static state*. Such state often includes the NIDS's configuration (e.g., the signature set it uses) and perhaps a database of information about the network it is protecting, such as the types of operating systems installed on the monitored hosts or their "active mapping" profiles [SP03a]. We note that the latter might in fact change over the course of execution, but if the NIDS does not have a means to incorporate such changes, the state is effectively static.

We refer to NIDS state that does change, on the other hand, as *dynamic state*. Here, we make a further distinction: we will refer to state that effectively exists at only a single slice (quantum) of time as *volatile state*, and state that exists over an interval of time as *non-volatile state*. For Bro, slices of time are quantized in terms of the arrival and processing of individual network packets. Examples of volatile state include events generated by Bro's event engine, or the values of local variables when Bro invokes policy script functions (since Bro specifies the execution of such functions as atomic, i.e., they run to completion before Bro considers any further input).

The ultimate goal for our architecture is to support making all of Bro's state both temporally and spatially independent. Our first observation is that this should be easy for static state. Since, by definition, static state cannot change over time, there must already be a means to specify it upon startup, which we can use again to recreate the state in other instances. However, we will also develop the theme that converting static state to dynamic state extends the flexibility of a NIDS, so our architecture aims to accommodate doing so. Our second observation is that volatile state, by definition, cannot be independent. Hence, we also aim to find ways to convert volatile state to non-volatile state to enable making it independent. In summary, our goal is to convert as much state as possible to being dynamic, non-volatile, and thus temporally- and spatially-independent.

While our architecture for independent state is implemented for Bro, we believe that other systems would likewise significantly benefit from independent state. However, depending on the flexibility of the particular system, some applications might be harder to realize than others. In particular, much power is lost if a NIDS does not provide a user-level scripting language. This is why we chose Bro as our target platform: its flexible, policy-neutral approach is ideal for taking full advantage of independent state. Nevertheless, independent state is a new general concept for network intrusion detection. This is important to keep in mind, in particular when in the following we necessarily have to delve into details of Bro.

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

2 Related work

To our knowledge, the unifying concept of independent state has not been previously formulated in network intrusion detection research. Some of its aspects, however, 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 [VK99] 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 [VKB01] adds dynamic reconfiguration capabilities to the STAT framework.

Emerald [PN97] hierarchically organizes monitors which exchange messages to propagate results and subscribe to services. GrIDS [SCCC+96] models large-scale attacks by activity graphs. Its components monitor traffic at multiple locations and communicate by sending or requesting information. AAFID [SZ00] builds on autonomous agents which communicate their results to hierachically organized monitors. AAFID's design specifically addresses dynamic reconfiguration and acknowledges the utility of persistent state, although the prototype does not implement it.

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

By setting up a network of communicating NIDSs, we are building a distributed system. Principles of such systems are discussed for example in [TS02]. To make a system's state independent, our main tool is a serialization framework, for which [Sou94] discusses different approaches.

3 Bro's State

To develop independent state, our first task is to identify the different types of state present in a NIDS. While there is a common subset of state held by most NIDSs (e.g., connection state), we delve into Bro's particulars here as they provide a variety of ways to explore the architectural notion.

For Bro, there are two main layers of operation, each of which stores a significant amount of state. The "event engine" layer, implemented in C++, 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 [ZP00]). 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.

3.1 Event engine state

There are four main types of internal, event-engine state in Bro: connection state, analyzer state, timers, and control state. Using our earlier terminology, they are all "dynamic" and "non-volatile."

Connection state: Bro's main unit of organization is a connection. By definition, each packet belongs to exactly one connection.[1] Bro keeps a map of all currently active connections.

With each connection it associates a variety of information such as its start time, the hosts (IP addresses) involved, the amount of data transferred so far, transport protocol state, and so on. In terms of volume, the per-connection state is by far the most dominant internal state. Therefore, it is particularly important to implement a sophisticated expiration policy to avoid resource exhaustion. Bro uses an extensive set of timeouts for this, as well as transport-protocol analysis. (By default, state for TCP connections is kept indefinitely, until the connection terminates due to a FIN exchange or a RST.)

Analyzer state: Bro contains an extensive set of protocol-specific analyzers, e.g., decoders for TCP and HTTP, which maintain their own set of internal state. For example, the TCP analyzer buffers out-of-sequence data, and the HTTP analyzer accumulates client and server headers. Since most analyzers work on a per-connection basis, they attach their state to Bro's generic connection state. Some analyzers also

[1] The notion of a connection is obvious for TCP. For UDP and ICMP, Bro uses a flow-like definition.

store global data, however, which is therefore not linked to a particular connection. One example is the stepping stone analyzer [ZP00], which keeps a set of candidate stepping stone connections.

Timers: Bro's main mechanism for expiring state is via timeouts. The number of concurrently active timers can easily reach tens of thousands on a high-volume link.

Control state: There are several parameters that control the operation of the event engine. The user's scripts can dynamically change them. The most significant of these are timeout values and the current packet filter (in `tcpdump` [TCP] syntax), which implicitly controls which analyzers execute. While these parameters are small in terms of volume, they have a major effect on the performance of the system.

3.2 Policy script 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 (not discussed further due to limited space).

Scripts: The scripts are static, non-volatile state. They define the behavior of the system by defining types, event handlers and functions.

Data: Event handlers and functions are able to store global and local data by defining variables. Both are dynamic, but while global data is non-volatile, local data is volatile since Bro fully executes function calls within a single (uninterrupted) slice of time. A number of different data types exist, with tables indexed by a set of types and yielding an arbitrary type being particularly common. For example, Bro's scan detector—which detects horizontal and vertical scanning, as well as password guessing—keeps *(i)* a large script-level table containing pairs of communicating hosts, *(ii)* another counting to how many different hosts a particular host has attempted to initiate connections, and *(iii)* in fact 21 additional tables and sets. For automatic expiration, Bro keeps timestamps for the entries in the various tables (and sets) indicating either their oment of creation or last access (depending on the table's declaration in the script) which then drives timeouts used to delete the entries.

Data operations: Depending on the data type, different types of operations exists. For example, for tables we can insert or remove an entry. Ordinarily, operations would be viewed not as state but as transformations to state. However, with our definition of "volatile" state, we can view specific instances of operations as momentarily existing in and of themselves, and thus constituting dynamic, volatile state. Designating them as such then gives us an opportunity to consider transforming them into dynamic, non-volatile state, which we return to in §4.1.2.

Event generation: Similar to data operations, we view the generation of an event as a form of dynamic, volatile state.[2]

Function calls: Function calls in Bro are quite similar to the invocation of an event handler, and so we likewise view calls as dynamic, volatile state. The principle difference is that function calls return values, while event handler invocations do not.

Signatures: Bro actually has two scripting languages: one for specifying event handlers, and a second for writing efficient, byte-stream intrusion detection signatures [SP03b].[3] These signatures are static, non-volatile state. For some NIDSs (e.g.,

[2] Most events are generated inside Bro's core, so one could argue whether they are indeed *script-level* state. We decided to put them here because their role is as triggers for script-level actions. The event engine does not itself process events, it only generates them.

[3] Throughout the text, when using the term "script" we always refer to the former.

Snort [Roe99]), signatures comprise the main type of script-level state.

4 Architecture Design and Implementation

After identifying the types of states that Bro stores, we proceed to investigate how to make each one dynamic and, particularly, independent. The main mechanism for this 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 data structures serializable is, by itself, fairly straightforward. But adding full serializability to a complex system like Bro, which was not designed with this in mind, raises numerous subtle issues we must address. Therefore, we first describe the implementation in more detail. We then present the corresponding interface available to the user, and finally we discuss some of the issues involved in *securely* sending state over the network. Our implementation has been integrated into the latest development version of Bro, and we are now using it operationally.

4.1 Serialization

Given an object-oriented design, the methodology of adding serializability to a class hierarchy is well-established [Sou94]. Each class gets two new methods, one for serializing and one for de-serializing. When called, each of them first calls the corresponding method of the class's ancestor, and then (de-)serializes all attributes unique to the class itself. We simply followed this approach, using a generic `SerializationFormat` class as the interface to convert between C++ data types and an external representation. Doing so keeps the serialization process independent of the underlying external format, so we are, for example, able to create both a (portable) binary version and an XML version.

For memory management, Bro uses reference counting extensively. To recreate the reference structure when de-serializing, objects may be assigned a set of unique identifiers. Such an object is fully serialized only once, when encountered for the first time. Upon subsequent serialization requests, e.g., due to being referenced by some other object, we only store its ID.

A basic problem that arises is the *time* needed to serialize state. Bro is a real-time 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 (see [DFPS04]). 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.

Incremental serialization leads to another problem, though. By serializing chunks of the state at different points of time, we may incur inconsistencies. Consider, for example, two script-level tables that contain related data derived from the same connection. It could happen that after the first table has been serialized, we process some packets that remove the connection data from the second table. Thus, serializing it in the next step would leave the two tables out of synchronization.

We have not yet addressed this problem—it has not arisen in our operational use so far—but our strategy for doing so is to use a transaction model [TS02], for which we have implemented partial support by funneling all state-changing data operations through a common location during serialization. By converting them into non-volatile state (as discussed below) we would produce a transaction log which could then be replayed upon reinstantiation.

We now turn to looking at the different types of state, discussing some of the particulars involved in their serialization, along with some points not yet fully implemented. As we will discuss, for some of these latter, it is not in fact clear whether supporting them makes sense. For others, adding support is straightforward, but to date we have not found a need for them in operational use.

4.1.1 Serialization of event engine state

Connection state: Connection state (and the attached *analyzer state*) is rather easy to serialize and to restore. The most important problem is the potentially very large number of concurrent active connections. For example, on one of the high-volume links we monitor operationally, we regularly find more than 30,000 concurrent connections even when using aggressively small timeouts.

This leads to two major problems. First, the volume of the data gets high: a single connection entry can exceed 1 KB in-memory, and the external representation—which we have not optimized for space—is even larger.

Second, and more importantly, it simply does not make sense to store *all* connections: the vast majority are quite short (100s of msec or a few seconds). Considering that it takes some time to serialize them (particularly given incremental serialization as described above), most of them will already be finished before they are read back. Therefore, we only selectively store connections specifically requested by the user. For example, the user can restrict state-independence to types of connections expected to be long-lived, like FTP or SSH sessions.

As a consequence, we have not yet implemented serialization for all analyzer-specific connection state. While all transport-layer protocols are fully serializable, so far the only supported application-layer protocol is FTP (we do not need it for SSH as, due to the encrypted nature of the payload, Bro presently only decodes the initial handshake and then confines itself to the generic TCP analysis). But adding serializability to other protocols as needed should be straightforward.

Timers: The main problem when de-serializing timers is deciding for temporally-independent state when to schedule them for expiration. Two approaches come to mind: first, keep the original expiration time and execute it immediately if that has already passed; second, adjust the time by subtracting the difference between termination of the old instance and start of the new one. Observing that most timers are set to some absolute time plus some timeout (for example, the initiation of a connection plus the time interval after which a response should have been seen), we took the former option. Additionally, when deserializing multiple timers, we make sure to expire them in their original order to preserve their semantic causalities. So far, we only implemented serialization for timers related to Bro's connection management, but supporting the other types will be simple as the need arises.

Control state: Unfortunately, Bro's control state is not located at some well-defined point of the class hierarchy, but distributed in several places. For user-redefinable timeouts, we can simply leverage the serialization of script-level data described below. For the packet filter, we added a method to store a string containing the filter specification and then read it back and change the filter to reflect it. We note that the system may need some time to compile and install a new specification; usually a couple of milliseconds. Additionally, at least on FreeBSD, it clears the current packet buffers.[4] As therefore we will likely miss some packets, there is a tradeoff involved when installing a new filter.

4.1.2 Serialization of policy script state

Scripts: The main components of scripts are type definitions, global variables, functions, and event handlers. All of them are fully serializable and deserializable.

In addition, we added support for *changing* the definition of a function and adding new event handlers during run-time. Thus, this type of state is no longer static but dynamic.[5]

Data: The most obvious state to serialize is the global data amassed by the scripts. We added serialization to all of Bro's different kinds of values, with full type-checking during reinstantiation. While not articulated as such in Bro's design, there are two basic types of values, static and mutable. Static values (of which all are atomic values that do not contain other values) are not themselves changeable at the script-level, since they are deep-copied upon assignment, similar to an `int` or `double` in C++. Mutable values are container values (e.g., tables and records) into which other elements can be inserted, thereby changing their values. Mutable values are shallow-copied, similar to C++ objects manipulated via pointers. Serialization of static values is straightforward, as there is no *aliasing* involved. For mutable values, we recreate the reference structure when deserializing (see §4.1).

We do not serialize data local to functions, as this volatile state is of no use outside of a particular function execution. (Bro does not presently support permanent, local function variables analogous to `static` variables in C++.)

A final performance issue arises with serializing tables: they can get huge (many thousands of entries), and thus serializing them all at once can take too long and incur packet drops. The natural solution here is to make the current incremental serialization mechanism more fine-grained, with the incremental unit being not an entire value, but an element of a value. In this fashion, we could serialize a large table a few elements at a time. Of course, this again raises the issue of consistency already discussed above, again requiring transaction logging.

Data operations: As mentioned earlier, we include data operations as part of a broader notion of "state," in particular as a form of volatile state. Our goal in doing so is to expose them as amenable to a form of "independence" similar to that which we strive to provide for data objects. In particular, if we have fully independent data, then we can propagate 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 can simply propagate "insert element 'foo' into set 'bar' ". (Propagating operations may introduce synchronization problems, though, as discussed in §4.2.2.)

Implementing this independence for operators turned out to be quite difficult. We first identified all the atomic operations that can be performed upon Bro script values, for example: binding a value to a global identifier, changing the value of an element in a container, or adding/deleting elements to/from containers. We then implemented a serialization for an abstraction of the operator. The main problem here rests in the need to *name* a value: when we want to change a value, we need to say which value we mean. Obviously, this is not a problem for static values—they cannot be modified, but only be created and bound to some global identifier (which, by

[4] We devised a FreeBSD kernel patch that avoids flushing the buffers.

[5] We explicitly do not allow changing type definitions during run-time as this would circumvent Bro's static type checking.

definition, has a name). For mutable values, however, we had to introduce a naming mechanism, so one Bro instance can communicate to another which value (e.g., which element of a table) it has modified.

We solved this problem by introducing a new, non-user-visible global namespace. Each mutable value, on which operations are to be tracked, is bound to a hidden identifier unique among multiple instances of Bro (by incorporating hostname and process ID into its name). If multiple instances share independent state, they agree on these names first. Subsequent operations are then expressed in terms of these names. For container values, each included mutable element gets its own unique identifier. This ensures that shallow-copied values are treated correctly.

Event generation: Given serialization of Bro types, it now becomes easy to also serialize Bro events, since an event is simply a name plus a set of typed values. In addition, we keep the time when the event was generated. The ability to serialize events—transforming event generation from a form of volatile state to non-volatile state—is very powerful. It means we can now send events between multiple concurrent Bro instances (leveraging spatial independence) and record events to disk and later replay them (temporal independence).

Function calls: Unlike event handlers, function calls return values, and hence are inherently synchronous rather than asynchronous. In Bro, function calls are very similar to event handler invocations. The only difference is that function calls return a value, while event handler invocations do not, the latter being asynchronous while function calls are inherently synchronous.

This difference has major implications for state independence. Because of the synchronous semantics of function calls, we cannot change them from volatile state to non-volatile state without violating either those semantics, or incurring potentially lethal blocking delays waiting for calls to complete and return. We chose the former; see §4.3 for further discussion.

Signatures: A final type of state in Bro, currently static in nature, is the set of byte-stream signatures used by its signature engine [SP03b]. Because we have not yet tackled making this type of state independent, we do not discuss it further (the optimized data structures used for signatures are not amenable to incremental updates).

We would like to convert these to dynamic state, enabling us to change signatures on-the-fly and send them from one Bro instance to another. Implementing this change, however, is quite challenging because the optimized data structures Bro uses internally to match signatures with high performance are not amenable to incremental updates. Changing a single signature currently requires recomputing the entire decision tree used to determine (prior to regular-expression matching) which rules are candidates for matching a given byte stream. Because we have not yet tackled making this type of state independent, we do not discuss it further.

4.2 Interface

We now turn to how the user interacts with the serialization framework presented in the previous section. The framework itself is internal to Bro's event engine and hidden from the user, while the interface 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 Bro's state into files and reading them back again later, possibly after having first modified them using other instances of Bro. We then discuss controlling spatially-independent state, which is done in the context of communication between multiple instances of Bro. 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.

4.2.1 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. Since our policy may be that once we've detected a Blaster infection, we don't need to hear about it again. 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 will return to this point when we discuss checkpointing in §5.1.

Along with `&persistent`, we also provide a function `make_connection_persistent`, which tells Bro to store the associated state of a particular connection. There is also an associated standard policy script that uses this function to automatically save state for all connections belonging to a user-definable set of services (like FTP and SSH). In addition to automatically writing all persistent state at termination, the 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 function `rescan_state` reads state back from disk. While by itself this is not of much use during operation, we can also then *copy* state files from one Bro instance to another and incorporate them directly into the second instance *while it runs*. One application here is to transfer data between two Bro instances. Another is more powerful: we have added a new command-line option that tells Bro to write all state contained in one of its scripts into files. That means we get access to all global identifiers, types, functions, and event handlers. By copying one of these files into another running instance, we can *change* its configuration on-the-fly—both the values of its global variables but also the values of its functions and event handlers, i.e., we can change the code it executes.

Here we see some of the broader potential of the move from static state to dynamic state. Of course, such flexibility also requires discipline: otherwise it can lead to confusion and disarray if we lose track of just what code and data a given instance of Bro is executing. But we were motivated to add this functionality by the observation of a corresponding operational need for rapid reconfiguration. Often when running Bro operationally, we will encounter a new traffic pattern for which the current Bro configuration is deficient (e.g., it fails to generate alerts for a newly discovered attack, or it generates a flood of alerts for a new type of traffic that is in fact benign). To date, accommodating such changes in the configuration has required terminating the existing instance of Bro—losing all of its state in the process—and starting up a new instance. But by using `rescan_state`, we can make the changes to the configuration, test it using a separate instance of Bro, and, once satisfied that the changes are correct, incorporate them into the running, operational instance without the need to restart it.

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.

This can be very helpful for debugging, as we do not need real network packets to reproduce a situation. Although this is not suitable for all cases—if we need access to event engine state, replaying events is not sufficient—it suffices in many situations. There is one other means for manipulating persistent state, which is to print it. We can do so either in a "pretty-printed" human-readable form, or encoded as XML, although this latter is not fully implemented at this point. Once we have finished implementing the XML output, we expect this to be highly useful for traffic analysis independent of the task of intrusion detection. For example, when combined with event capturing, it gives us a more abstract view of network activity than raw packets, but remains machine parsable. In addition, when coupled with an XML reader, this facility will enable us to directly manipulate Bro's state using whatever tools we have available for editing XML. While such "outside the system" editing has the potential for introducing nasty, hard-to-find bugs, they can also prove to be life-savers during emergencies that sometimes arise operationally.

4.2.2 Spatially Independent State

For spatially-independent state, we need to transfer state from one Bro instance to another running concurrently. While one way to do this would be via the already-mentioned `checkpoint`/`rescan_state` functions (coupled with manually copying the files), doing so would be crude and quite limited in power because it would hide the presence of multiple Bro's from one another.

A more direct way is to establish network connections between the instances. To do so, one of the instances calls the new function `listen`, which opens a port on the local host waiting for connections from other instances:

```
# Listen on interface 10.0.0.1:47756
# for SSL-authenticated connections.
listen_ssl(10.0.0.1, 47756/tcp);
```

These in turn initiate connections by means of the new `connect` function:

```
# Connect to 10.0.0.1:47756, using SSL.
connect_ssl(10.0.0.1, 47756/tcp);
```

Once a connection is established, there are several ways to exchange state. Using `request_remote_events` one side can request 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 (although by calling the function `is_remote_event` the script can distinguish between local and remote events, and then, additionally, calling `event_source` to retrieve more information about the originator).

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`:

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

In addition, we can explicitly request the full set of persistent state (i.e., all data declared `&persistent`, and all connections marked by `make_connection_persistent`) from another host, reinstantiating it locally.

Finally, for the event engine's control state, the new function `send_capture_filter` sends a `tcpdump` filter to the other side, which then decides (as discussed in the next section) whether to install it. Because the filter fundamentally determines the type of traffic available for analysis, it effectively controls which analyzers are activated, and thus the remote Bro's processing load vs. degree of detailed monitoring.

We implement synchronized tables by propagating data operations as discussed in §4.1.2. We have to be aware, though, that this may lead to synchronization problems. For example, consider a synchronized table that counts alerts generated by a particular source address. If we have parallelized our NIDS processing by having multiple processes performing different types of analysis, then each of them might determine that the same source address has generated an alert. If the processes share a common, independent table, then one of them modifying the table might experience a race condition with another process modifying the table at the same time; the winner of the race will overwrite the new value provided by the loser, and the net effect is that the source address may be charged with only one new alert against it rather than two.

To avoid these race conditions, we would have to ensure mutually-exclusive data operations, for example by using a token-based reservation system [TS02]. But this would violate Bro's real-time processing constraints: before performing an operation, an instance would have to *wait* until access is granted. Since this in untenable, we explicitly use *loose synchronization*, which incurs the race conditions described above.

On the other hand, if in the above example the action being performed is incrementing an alert counter, the operation is in fact not "set it to the value $n+1$" but rather "increment it". If we propagate this operation rather than the resulting value ($n+1$), then the increment will be performed twice and we obtain the correct value in the table of $n+2$. Indeed, "increment" and "decrement" are two of the types of operations which our implementation propagates.

Thus, there are not any synchronization problems when using Bro's `++`/`--` operators on independent data (with the exception of a potential lag until all instances received the operation). For operations we cannot treat in this fashion, we include the old value when propagating an event. (For example, the operation "assign 7 to the global `alert_level`" is propagated as "assign 7 to the global `alert_level`, its previous value was 12".) By doing this, we are able to at least detect and report desynchronizations (in this case, if when changing `alert_level` to 7, we notice that its value before the change is not in fact 12).

4.3 Robust and Secure Communication

The design of our new inter-Bro communication system emphasizes robust and secure operation. Regarding robustness, a key point is that, from the perspective of a Bro process's main functionality, inter-Bro 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 chose to leave Bro's current single-process design intact, and to 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. In this case, the network communication does not add load to Bro's main component.

The next element of our design was to base it on semantically unidirectional communication. This means that while two peers may both send state over the same network connection, Bro's processing never expects one side to reply to something the other side sent. In particular, we do not use any form of application-layer acknowledgments. 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.

The only major drawback of this design decision is that we cannot remotely call functions that return a value (§4.1.2). There are two types of Bro functions: internal functions accessible from the script level but defined within Bro's event engine core, and script functions. We examined all internal functions as of Bro version 0.8a39 (about 100 total), and found only a few that actually have semantics that require both being called remotely and returning a value. It turns out that all of these can either be replaced by some additional script-layer logic, or they are not in fact used by any of Bro's default scripts. (The functions `active_connection`, `connection_exists`, `lookup_connection`, and `connection_record`, could be replaced by using `active.bro`; for `get_login_state` we could add a new event; `get_orig_seq` and `get_resp_seq` are only used in `terminate_connection` which (usually) has to be called locally anyway; and `get_matcher_stats` and `get_contents_type` are unused. Considering script functions, we see that instead of calling them remotely, we can as well call them locally, leveraging `&synchronize`'d state if necessary for their operation.) We conclude that the inability to remotely call functions is not a severe limitation. Finally, we note that the unidirectionality of communication only affects the core-level communication between two instances. For example, it is still quite possible to build a *script-level* handshake mechanism by passing a sequence of events between two peers. In fact, the *handover* mechanism shown in §5.1 does exactly this.

For reasons similar to those that lead us to rely on loose synchronization (see §4.2.2), we do not make any timing guarantees for the communication. For example, transfering 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 consider securing the communication, i.e., providing for confidentiality and authentication. To do so, we provide a simple script-level means to specify the use of SSL for securing the communication, which we implemented using OpenSSL [Ope].[6] Enabling encryption for confidentiality is straightforward. For authentication we make use of signed certificates. Peers are configured with the public keys of trusted certification authorities. They only approve a connection if the other endpoint presents a certificate signed by one of CAs. We note that we always have both peers authenticate themselves (in contrast to server-only SSL authentication as is often used).

A NIDS necessarily gathers a great deal of information, the access to which is often restricted and not to be shared with other parties. Therefore, trust is a vital issue when accepting connections from peers. Given the capabilities presented above, without further restrictions any peer could easily access our state as well as send us arbitrary, perhaps misleading, state. In general, we can identify four levels of increasing trust:

1. Peer may not talk to us at all.
2. Peer may get state from us, but we do not accept any state.
3. Peer may get state from us, and we accept non-control state.
4. Peer may get state from us, and we accept all state.

The levels are ordered in the way that more trust allows more activity on the other side. If already connected, just passively receiving state needs the lowest trust. Feeding us state requires more trust, and controlling our operation (e.g., by sending us a filter that turns on more analyzers) demands full trust.

The need for these sorts of different levels immediately arises operationally. For peers with which we have not developed any monitoring agreement, the first level of trust (no access) is appropriate. For sites that we wish to help but we do not know if they themselves are run competently, the second level is best. For sites with which we have a close working relationship, the third may be. Finally, when using multiple NIDS instances internally, for example for load-balancing, the fourth level likely makes sense.

Accepting state from a peer (i.e., trust level 2 and above) illustrates another implication: due to the complex semantics of the state, we cannot reasonably validate that the input is well-formed. Consider for example shallow-copied objects: when a serialization references another object, it only includes a unique ID. Upon deserialization, there is no apparent way to validate that the referenced object is *semantically* correct (we do ensure type safety though). Thus, accepting state entails trusting the peer to send valid data, and this consideration must be kept in mind when assigning trust levels.

All four levels rely on a correct identification of the remote side, which we can achieve via SSL-based authentication (and firewalling to enforce the first level). While we have not currently implemented an explicit framework to directly state "peer x is on trust level y", we have implemented hooks to enforce such trust-levels at the script-level by the user. For example, when a remote peer has successfully connected, the event engine generates an event to tell us so. Our policy script can then decide whether we want to accept state from the peer by calling a corresponding function. If we receive control-state (which currently can only be a packet filter), the decision to use it is again left to an event handler that needs to be provided by the user.

[6] We preferred SSL over other means, such as IPSec, because of its ease of use: while we have to (slightly) adapt the NIDS, the user does not need to install additional infrastructure. This is particularly important for fostering distributed, independently-administered confederations.

5 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 the 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 sight each may seem to be yet-another-extension of Bro's generally-extensible functionality, the *ease* of implementation proves the power of the approach. That the single concept of independent state enables such a diversity of new applications illustrates its *architectural* nature.

Our experiences with these applications come from monitoring the access links in several large environments: the Münchner Wissenschaftsnetz (*MWN*; research network including two universities—Technische Universität München, Ludwig-Maximilians-Universität München—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).

5.1 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. (We go in that order to avoid a monitoring outage during the changeover.) The frequency with which this is done ranges from daily (LBNL) to every few hours (MWN, UCB).

The main advantages of this sort of checkpointing are its simplicity and the robustness it provides, a sort of "memory management in depth" when coupled with Bro's fine-grained state management mechanisms. But, clearly, simply throwing away all of a NIDS's state at certain times is not the best approach. Ideally, we would like to retain a selected subset of important state, while reclaiming all of the rest.

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. The concept of persistence described in §4.2.1 now enables us to individually choose connections (by calling `make_connection_persistence`) and script-level data (via `&persistent` declarations) to be made independent, thus enabling the 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 scans, long-lived interactive connections, usernames, inferred software versions (see below), 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 current 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 it to start up, we will incur a monitoring outage.[7] We solve this problem by recognizing that instead of using temporal independent state, we can use spatial independence. 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 intentionally did 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 have decided to keep internal connection state for interactive services that tend to have long-lived connections. We do this using new default policy scripts that trigger persistence for FTP, SSH, *telnet*, and *rlogin* connections. (This list is easily customizable. In addition, one could choose to add connections for which some malicious activity was already seen.) 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 rather low in volume.[8] and suppressing unnecessary log messages is a vital NIDS capability [Axe99, Jul03]. For the second group, we differentiated 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 [Bla] by its scanning activity keeps two tables: one tracking pairs of hosts which 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.

The third group 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, for which we know they always correspond to state for persistent connections. (For example, the FTP analyzer script remembers FTP connections.) We made these kinds of tables persistent, but left all other tables un-

[7]If we start the new instance first, and have it read in the persistent state while already processing packets, we incur significant analysis inconsistencies.

[8]With the notable exception of the table `weird_ignore` recording all the "crud" [Pax99] In large networks, we see tons of crud.

modified (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 [SP03b] one of the scripts detects the software versions used by different hosts, an excellent example of information we do not want to lose. Consequently, we declared it `&persistent`.

We observed that adding persistence to a table almost always implies adding an expiration timeout, too, as it generally does not make sense to store state forever. We implemented *read* and *write* timeouts, which expire for each table element a given amount of time after the last access or modification to the element. Similar to `&create_expire` discussed above, these work for persistent tables as well as ephemeral. One exception we made to always expiring persistent state, however, was for vulnerability profiles, where we prefer to keep the information as long as possible. If required, we can always delete it manually by deleting the corresponding state file on disk.

5.2 Crash Recovery

A related application of persistent state is better recovery from crashes. Three main reasons for the crash of a NIDS are resource exhaustion, attacks, and programming errors [Pax99]. In most systems, including Bro, in each case we lose all the state so far collected by the system. But by using the `checkpoint` function (see §4.2.1) regularly, 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 —where we often experience crashes due to programming errors, and, hitherto, always lost the complete state of system as a consequence— but also in a production environment, where crashes still are 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-mortum (cf. §5.4).

In addition, we are planning to extend the handover mechanism described in §5.1 by running a "shadow" instance of Bro. It would connect to the main Bro process but otherwise stands idle, regularly checking responses to are-you-alive events. If so, it requests a copy of the main Bro's current state; if not, then it can use the last transferred state as the starting point for its own analysis.

5.3 Distributed Analysis

One 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 [SBD$^+$91] was the first to employ a sensor model, gathering low-level data remotely while performing the higher level semantic analysis centrally. On the other hand, Emerald [PN97] builds up a hierarchical structure used 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 for doing so, all of which we have been able to implement and experiment with by means of Bro's independent state. The first model supports load-balancing for monitoring high-volume links. The second supports the well-known "distributed sensor" model, and the third looks at propagating information between otherwise decoupled systems.

5.3.1 Load-balancing

On today's high-volume links,[9] it is exceedingly difficult to analyze the full packet stream with a single NIDS (unless one utilizes custom hardware [KVVK02]). 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 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 efforts so far have focused on static approaches due to their simplicity, with the two obvious ones being distribute based on: *(i)* the local IP space, or *(ii)* the application.

Dividing by IP space: To fruitfully split up the local IP space, we need knowledge about the network to find a division so that the individual NIDS systems 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).

With spatially independent state, however, we can easily solve these problems by declaring the tables `&synchronized` (per §4.2.2). Now each instance propagates its state to the peers. In fact, this is an ideal situation for loose synchronization: the propagated updates are "add this element to the table" and "increase this element's counter". Both are independent of the order in which they are applied (as long as elements are not removed, but in most cases this happens, if at all, due to automatic expiration, which each instance already does by itself). Also, for this task, a short interval of desynchronization is not of much importance.

We are currently working on exploring division by IP space operationally at MWN.

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 Bro does indeed spend quite some time processing the particular application. This is true for HTTP (due to Bro's de-

[9] E.g., the traffic level in the MWN (UCB) environment sustains more than 250 (300) Mbps averaged over an entire day.

tailed 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. It parses the negotiation of FTP data connections (`PORT`, `PASV`). A more general problem concerns analyzers that need to see traffic from all applications, such as the scan detector. For detection of vertical port scans, it counts connection attempts to different ports (applications) per host. Other examples include the ICMP analyzer, which correlates ICMP "unreachable" messages with the corresponding connections, and the analyzer that derives vulnerability profiles [SP03b].

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. Finally, we note that the two techniques of dividing by IP space and dividing by application can in principle be combined (dividing by both) in environments with a large number of analysis hosts available. Spatially independent state should be able to support this generalization, too, although we do not have concrete plans to try this out operationally.

5.3.2 Sensor model

A well-established architecture for distributed network intrusion detection is the sensor model [Amo99], 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. This becomes easy to achieve using spatially independent state.

To test this approach, we ran two instances of Bro as sensors on a dual-processor monitor machine at MWN's uplink, load-balancing the network traffic by dividing the IP space. We ran a third Bro on a second system which acted as the analyzer, receiving and processing all the events generated by the sensors' event engines. This worked without a hitch.

Then, to compare the processing loads, we set up a single sensor instance on the dual-processor monitor, and an analyzer on the second machine. The total load across both CPUs of the monitor was only slightly less than what it would have been if it was doing the full work by itself, because for the MWN setup the main processing burden is on the sensors—offloading the analysis from them is in absolute terms a relatively minor gain, and the bigger win in that environment comes from dividing up the IP space. However, due to our use of a separate process to manage inter-Bro communication (§4.3), the CPU running the main analysis benefited from considerably reduced load.

While we have not experimented with it yet, another approach made possible by spatially independent state would be 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 §5.3.1, and then we would dedicate an additional CPU to correlating their combined output at a meta-level, for example by using established correlation methods [DW01, KTK01, VS01]. (Indeed, from our experience, even just combining the log messages coherently and in a single place would be of operational benefit.)

5.3.3 Propagating information

Another potentially valuable application of spatially independent state is using it to tell other systems some facts about our analysis. Although less ambitious than fully distributed correlation, it can be quite powerful. 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.

Along these lines, a more ambitious scheme we are working on implementing is a *time machine*. Here, the second analyzer maintains a (very) large, rolling buffer consisting of all recent network traffic to the extent that available storage permits. Upon receiving an event with a suspect IP address, it can then extract not only any new traffic that a host sends, but also its *previous* traffic to the extent available in the buffer. If, for example, the buffer is a 500 GB disk partition, then this could track many minutes or hours of traffic, providing we can structure the system to stream the traffic to the disk (and delete it as it expires) sufficiently quickly.

Propagating blocked hosts: Our LBNL environment currently runs several Bro's at different entry points into the network. As discussed in [JPBB04], LBNL's security policy includes dynamically blocking scanners detected by Bro by modifying the entry 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. This way, a malicious host can be *proactively* blocked from probing the addresses behind the second entry point, rather than *reactively* blocked (which could come too late in terms of preventing damage). We plan to support this information sharing by modifying the operational Bro's to generate events when blocking new hosts and broadcasting them to their peers. We also plan to experiment with broader sharing of such information across the different institutions (MWN, LBNL, UCB) where we operate security monitoring.

5.4 Dynamic Reconfiguration, Profiling and Debugging

The final set of applications for independent state that we look at leverage that our framework not only enables data values to become independent state, but also broader notions of "values" such as the functions and event handlers in policy scripts. Here we realize the benefits made possible by converting Bro's static state into dynamic state. 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 such as 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 to their backdoor; 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 we would 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 somewhat 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 this 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 static state, i.e., the scripts. We then convert the full configuration into persistent state, stored in individual files, as described in §4.2.1. Finally, we copy the files containing state affected by our change 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

Figure 1: Port scans at two times (addresses altered).

```
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)

```
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)

```
  [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)

Figure 2: Function from `scan.bro` with timestamps.

```
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);
else
  (@01/21-12:23 #644450)
  [...]
}
```

after a considerable amount of run-time. This means that they cannot be reasonably targeted with Bro's existing tracing and interactive debugging mechanisms.

We find it is a great help if we can take a look at Bro's current state. With independent state, this is easy to achieve, since the files generated by `checkpoint` contain all the necessary information. Included in our output are timestamps when the entries were last accessed. Additionally, the ASCII output formats are also suitable for processing with Unix utilities such as *sort* and *diff*. In Figure 1, for example, we see how the table containing all currently known port scanners has changed between two points of time. Here, we can actually *see* the scan detector working. In Figure 1(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 1(b), we see the same table from about 1.5 hours later. For larger tables, the differences may be hard to see, though. However, the ASCII output formats are also suitable for processing with Unix utilities such as *sort* and *diff*, as illustrated in Figure 1(c). Now the differences become obvious; we can actually *see* the scan detector working.

Along with data values, independent state provides timestamping for script functions, too. Figure 2 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. We can again use tools like *diff* to dynamically track which portions of the code are being executed, and how frequently. The counts are particularly useful: if they differ significantly from what we expect, there is either a coding error, or a misunderstanding of the network traffic. Finally, we aim to eventually also use this information for long-term policy script maintenance, for example by locating policy script elements that have become stale and are no longer needed.

6 Summary

In this work we developed an architecture for *independent state* in network intrusion detection. While often much of a NIDS's state resides solely in volatile memory, we set out to make all of a NIDS's state exist "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, and which can later be incorporated by new processes). Our architecture 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.

Our architecture facilitates independent state for the Bro intrusion detection system [Pax99]. In the process of making all of its semantically rich, detailed state independent of any particular instance, we convert (i) *static state* (e.g., configuration and user-written scripts) to *dynamic state*, becoming changeable during run-time; and (ii) *volatile state* (state that exists during only a single quantum of time, e.g., events and operations on data) to *non-volatile* state.

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. 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 become invaluable.

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 Mark Allman and Scott Campbell for their helpful comments.

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

References

[Amo99] Edward G. Amoroso. *Intrusion Detection: An Introduction to Internet Surveillance, Correlation, Trace Back and Response*. Intrusion.Net Books, New Jersey, 1999.

[Axe99] Stefan 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.

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

[DFPS04] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. Operational experiences with high-volume network intrusion detection. In *Proc. 11th ACM Conference on Computer and Communications Security*, 2004.

[DW01] Hervé Debar and Andreas Wespi. Aggregation and Correlation of Intrusion-Detection Alerts. In *Proc. of Recent Advances in Intrusion Detection*, number 2212 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

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

[JPBB04] Jaeyeon Jung, Vern Paxson, Arthur W Berger, and Hari Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *2004 IEEE Symposium on Security and Privacy*, 2004.

[Jul03] Klaus Julisch. Clustering intrusion detection alarms to support root cause analysis. *ACM Transactions on Information and System Security*, 6(4):443–471, 2003.

[KTK01] Christopher Krügel, Thomas Toth, and Clemens Kerer. Decentralized Event Correlation for Intrusion Detection . In *Proc. of Information Security and Cryptology*, volume 2288 of *Lecture Notes in Computer Science*, 2001.

[KVVK02] Christopher Krügel, Fredrik Valeur, Giovanni Vigna, and Richard A. Kemmerer. Stateful intrusion detection for high-speed networks. In *Proc. IEEE Symposium on Security and Privacy*, 2002.

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

[Pax99] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[PN97] Phillip A. Porras and Peter G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *National Information Systems Security Conference*, Baltimore, MD, October 1997.

[Roe99] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proc. 13th Systems Administration Conference (LISA)*, pages 229–238. USENIX Association, 1999.

[SBD+91] Steven R. Snapp, James Brentano, Gihan V. Dias, Terrance L. Goan, L. Todd Heberlein, Che-Lin Ho, Karl N. Levitt, Biswanath Mukherjee, Stephen E. Smaha, Tim Grance, Daniel M. Teal, and Doug Mansur. DIDS (distributed intrusion detection system) – motivation, architecture, and an early prototype. In *Proc. 14th NIST-NCSC National Computer Security Conference*, 1991.

[SCCC+96] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS – A graph-based intrusion detection system for large networks. In *Proc. 19th NIST-NCSC*

National Information Systems Security Conference, 1996.

[Sou94] Jiri Soukup. *Taming C++ – Pattern Classes and Persistence for Large Projects*. Addison-Wesley, 1994.

[SP03a] Umesh Shankar and Vern Paxson. Active Mapping: Resisting NIDS Evasion Without Altering Traffic. In *Proc. IEEE Symposium on Security and Privacy*, 2003.

[SP03b] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proc. 10th ACM Conference on Computer and Communications Security*, 2003.

[SZ00] Eugene H. Spafford and Diego Zamboni. Intrusion detection using autonomous agents. *Computer Networks*, 34(4):547–570, 2000.

[TCP] `tcpdump`. `http://www.tcpdump.org`.

[TS02] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems – Principles and Paradigms*. Prentice Hall, 2002.

[VK99] Giovanni Vigna and Richard A. Kemmerer. Netstat: A network-based intrusion detection system. *Journal of Computer Security*, 7(1):37–71, 1999.

[VKB01] Giovanni Vigna, Richard A. Kemmerer, and Per Blix. Designing a Web of Highly-Configurable Intrusion Detection Sensors. In *Proc. of Recent Advances in Intrusion Detection*, number 2212 in Lecture Notes in Computer Science, 2001.

[VS01] Alfonso Valdes and Keith Skinner. Probilistic Alert Correlations. In *Proc. of Recent Advances in Intrusion Detection*, number 2212 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[ZP00] Yin Zhang and Vern Paxson. Detecting stepping stones. In *Proc. 9th USENIX Security Symposium*, pages 171–184. The USENIX Association, 2000.