A Lone Wolf No More: Supporting Network Intrusion Detection with Real-Time Intelligence

Bernhard Amann[1], Robin Sommer[1,2], Aashish Sharma[2], and Seth Hall[1]

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

Abstract. For network intrusion detection systems it is becoming increasingly difficult to reliably report today's complex attacks without having external context at hand. Unfortunately, however, today's IDS cannot readily integrate *intelligence*, such as dynamic blacklists, into their operation. In this work, we introduce a fundamentally *new capability* into IDS processing that vastly broadens a system's view beyond what is visible directly on the wire. We present a novel *Input Framework* that integrates external information in real-time into the IDS decision process, independent of specific types of data, sources, and desired analyses. We implement our design on top of an open-source IDS, and we report initial experiences from real-world deployment in a large-scale network environment. To ensure that our system meets operational constraints, we further evaluate its technical characteristics in terms of the intelligence volume it can handle under realistic workloads, and the latency with which real-time updates become available to the IDS analysis engine. The implementation is freely available as open-source software.

1 Introduction

For network intrusion detection systems (IDS) it is becoming increasingly difficult to reliably report today's complex attacks purely by looking at traffic on the wire, without having any further external context at hand. For example, often the best way to detect botnet communication is to monitor for connections to known C&C servers that the security community has already identified. Likewise, external malware registries can help determine if downloaded files contain malicious code. A variety of efforts are collecting and disseminating such third-party *intelligence* systematically, including blacklists such as Google's Safebrowsing URL list [11] and VirusTotal's hash-based malware identification [29]. More sophisticated federated sharing initiatives—operated, e.g., by REN-ISAC for the education community [18] and the Department of Energy's *Joint Cybersecurity Coordination Center* (JC3) [1]—enable real-time propagation of incident information across their member institutions.

Unfortunately, however, today's IDS cannot readily integrate such external information into their processing. Their standard approach for using intelligence remains to statically convert it into their rule languages, which severely limits

the attainable benefits. If they offer direct interfaces to the external world at all, they typically restrict them to a small set of individual hard-coded applications.

In this work, we introduce a fundamentally *new capability* into IDS processing that vastly broadens a system's view beyond what is visible directly on the wire. We present a novel *Input Framework*that integrates external intelligence in real-time into the IDS decision process, independent of specific types of data, sources, and desired analyses. We design the framework so that it offers a simple interface to IDS users while providing the flexibility to interface to a range of local and remote intelligence sources. On the architectural side, we ensure that even with rich external information, the Input Framework heeds to the stringent performance requirements of high-volume, soft real-time packet processing.

We implement the Input Framework design on top of the open-source Bro IDS. By offering a Turing-complete scripting language for expressing local policies, Bro is ideally suited to exploit the full power of the new capability. Using our implementation, we demonstrate three real-world use cases: *(i)* integration with REN-ISAC and JC3 feeds; *(ii)* online virus checks of executables observed on the network; and *(iii)* real-time database queries, with results integrated back into the IDS decision process on the fly.

The goal of our work is to provide a new capability suitable for operational deployment. As such, it is crucial to ensure that interface and implementation meet operational demands, and we hence evaluate our work from that perspective. First, we report operational experiences from a large-scale network environment where operators are already deploying the implementation experimentally. Second, we instrument our implementation to understand its technical properties, such as the volume of intelligence it can handle in parallel to processing traffic, and the latency with which updates become available to the IDS analysis.

Based on encouraging feedback from operators, we anticipate that our implementation will become part of operations at further sites in the near future. We release our code as open-source under a BSD license, and most of it is already integrated into the standard Bro distribution. We emphasize, however, that conceptually our approach is not limited to the specific IDS we used, but can similarly enable others systems to leverage intelligence effectively.

We structure the remainder of this paper as follows. §2 discusses related work. §3 presents the design and architecture of the Input Framework, and §4 details our implementation. §5 discusses three concrete use cases, including initial deployment experiences with one of them at a large-scale network site. We evaluate the performance of our implementation in §6, and we finally summarize in §7.

2 Related work

The Input Framework provides a generic platform for integrating external intelligence into an IDS' live packet processing. While we are not aware of any current IDS that provides such a capability with a similar degree of flexibility, a number of existing efforts provide evidence for the utility of our approach.

We find a wide variety of online services available that provide third-party intelligence to network sites aiming to support their security efforts. Most commonly, these offer blacklists of known bad actors or content. Examples include web sites listed on Google's Safebrowsing URL list [11]; mail servers on Spamhaus's Block List (SBL, [23]); malware listed by registries like VirusTotal [29] and Team Cymru's malware hash registry [7]; and suspicious IP addresses reported to DShield [10]. Whitelists alternatively help avoiding false positives; NIST for example provides a list of hashes of known *benign* files that are part of OSs and applications [2]. More general data sources can provide further context like *whois* domain information and Team Cymru's *Bogon List* of unroutable IP space. In addition to such public services, a number of closed federations have emerged that distribute non-public incident information across member institutions. This information is much more context rich than the simple aforementioned blacklists, often containing features like the IP address, URL, downloaded malware md5 hashes, and timestamps for each incident. Examples include REN-ISAC's *Security Event System* [18], DOE's JC3 feeds [1], and Argonne Lab's *Federated Model* [6]. In §5 we show how our Input Framework integrates with the former two specifically. A particular benefit of such federations is that sites with lower technical expertise can benefit from findings and capabilities of their peers. In addition, a site may also have further local resources to support an IDS: a database of valid user accounts can help detecting brute-force SSH attacks, and a list of software running on local end hosts suggests whether a victim is vulnerable to a specific exploit. It is generally all such context information that we collectively refer to as *intelligence*.

Past studies show the benefit of integrating external information into security decisions. A recent study [19] at NCSA found that for 27% of all tracked incidents *external notifications* triggered their investigation (and not the local IDS). Verizon reports that "third parties discover data breaches much more frequently than do the victim organizations themselves" [28]; they found 92% of all breaches to fall into that category (49% when only considering larger organizations). It is such experience that motivates federations like SES to automate intelligence sharing. Another study [14] shows that attacks on different sites are often correlated and hit the separate networks within minutes. The authors recommend to rapidly share IDS state as a countermeasure. In [20], the authors analyze e-mail spam blacklists and find that local aggregation and reputation assignment can improve their accuracy. Our approach aligns with that by making complex intelligence available to the IDS and not only working on blacklists with predetermined yes/no decisions. We also find a number of specific detectors in the literature that leverage intelligence as key ingredients, such as BotHunter [12].

Current IDS do not provide flexible mechanisms to integrate external information. In our experience network operators today leverage intelligence by writing scripts that either turn it into static IDS configurations or post-process the *output* of an IDS offline. Indeed, Snort [17] distributes most of its blacklists in the form of rules [4]; and the software underlying SES provides an option to directly output Snort rules generated from received intelligence [5]. Likewise, users of Suri-

cata [24] and Bro [16] often auto-generate static configurations. Doing so however tends to incur major performance hits, and also makes updating an expensive operation that typically requires a restart. Worse, with signature-based systems this approach constraints any analysis to basic byte-level pattern matching—a model rarely constituting a good fit for higher-level intelligence that often only *augments*, and not controls, security decisions.

Newer Snort versions feature an IP reputation preprocessor [21] that directly imports IP-based black- and whitelists. It, however, requires specifically formatted input, does not operate on other information than IP addresses, and cannot leverage the lists for any analysis going beyond simple allow/drop decisions. Bro provides a generic communication interface [22] that can update state dynamically. However, while a user could use this for integrating intelligence (and some do, for a lack of alternatives), it remains low-level with no specific support for interfacing intelligence sources.

In the literature, work on adding context to IDS decisions tends to focus on correlation between IDS nodes (e.g., [8, 26, 27]), not higher-level intelligence sharing with external entities. On the commercial side, many security appliances seem to leverage forms of intelligence. For example, Symantec's firewalls support blacklists and blacklist sharing [25], and Damballa's product line includes *dynamic reputation* modules [15] based on Notos [3]. While taking a similar approach as we advocate, the Input Framework is not limited to a specific data source or analysis. Generally, we note that for commercial solutions it tends to be hard to say what they do *exactly*, as specifics of their internals are rarely public.

3 Design

We now discuss the design of the Input Framework. We begin by looking at the type of state that it targets in §3.1: external "intelligence" of low to medium volume with potentially frequent updates. We discuss our main design objectives in §3.2 and then present the high-level Input Framework architecture in §3.3.

3.1 Intelligence

Of all the run-time state that a typical IDS manages, the Input Framework targets a specific subset that today's systems support only insufficiently. Most IDS implementations focus on two groups of state (see Fig. 1): *(i) network state* derived directly from the monitored packet input, and *(ii) configuration state* describing the types of analyses to perform, such as a set of signatures or specific hosts to watch out for. The former group consists of volatile, high volume data (e.g., the current set of active connections along with TCP and application-layer information), and requires sophisticated schemes for efficient management [9]. The configuration of an IDS, on the other hand, is of low volume and static: changes tend to require an expensive reload operation that interrupts the current analysis, often in the form of a full system restart.

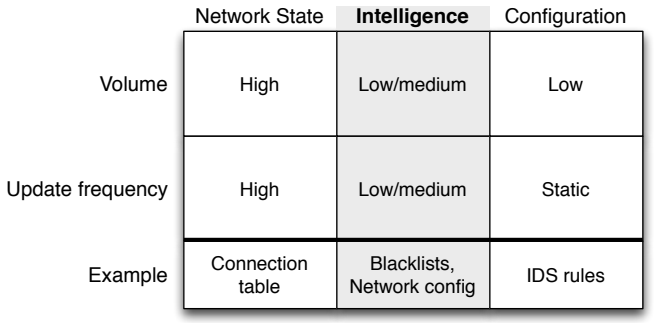

Fig. 1. IDS State

We argue, however, that there is a third group of state that we term *intelligence* state: externally provided context that, when correlated with the traffic on the wire, can significantly increase the system's detection capabilities. As discussed in §2, such state includes blacklists of known bad actors and specifics of the local environments. Conceptually, intelligence falls in between the two former groups: it is of much lower volume, and more stable, than network state. Intelligence however changes frequently, possibly multiple times per second, and thus cannot become part of the IDS's static configuration. Our Input Framework specifically focuses on integrating such state into the IDS analysis.

3.2 Objectives

The goal of the Input Framework's design is to offer a flexible mechanism to integrate intelligence from a variety of sources, without negatively impacting the IDS' main task of analyzing a high-volume packet stream under soft real-time constraints. To this end, we identify the following objectives for its design:

Adaptable to different sources. A crucial design goal is the ability to interface to a range of potential input sources and formats. We require the Input Framework to accommodate sources as different as flat files of ASCII and binary data, sockets for live feeds, databases, and web services. Along with that comes the requirement to support different modes for updates, including processing intelligence in regular batches as well as "pull" and "push" operation for real-time streams. While adding new input sources necessarily requires tailored interface code, the Input Framework should make extensions easy.

Simple, yet flexible user interface. The interface that the Input Framework exposes to the user should be easy to use and concise, with reasonable defaults where it offers options. It needs to provide a unified view of all input sources, abstracting from their individual characteristics. As intelligence will originally arrive in a variety of formats that external parties determine, we need to provide customization hooks that allow for on-the-fly preprocessing and filtering. However, all external intelligence should fully integrate with the

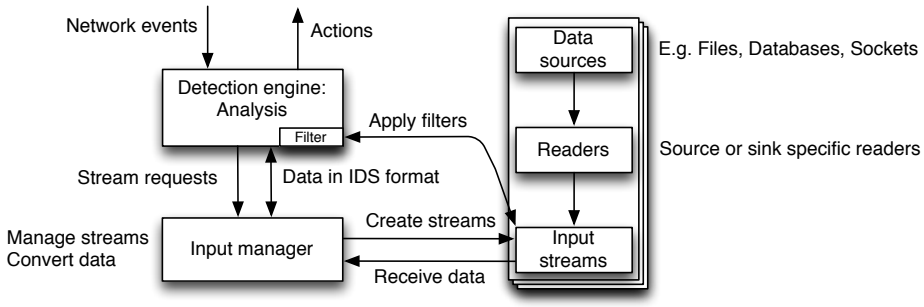

Fig. 2. Framework Architecture

IDS' standard analysis capabilities. Where possible, we want to transparently incorporate intelligence into the existing decision process.[3]

Asynchronous Operation. All I/O must execute fully asynchronously. Accessing external state can take a significant amount of time during which packet processing needs to proceed normally without blocking. This is particularly crucial for high-latency sources such as databases or web services hosted remotely. Likewise, as a stream of intelligence is coming in, processing must interleave with traffic analysis to avoid causing packet drops. Nevertheless, from the analysis' perspective, the state must be consistent at all times.

Real-time Operation. The Input Framework needs to make incoming intelligence available to the IDS analysis rapidly. While we cannot control lags introduced by an external source (like the time it takes a database to respond to a query), we strive to keep the Input Framework's internal latency low.

3.3 Architecture

Fig. 2 shows the architecture for the Input Framework that we design to address the above objectives. In the following we discuss the frameworks main components in an order roughly following the data flow. As some parts of the architecture depend on the specific IDS that one integrates with, we can only sketch them in abstract terms. However, the discussion will become more concrete in §4 where we describe our implementation inside a specific IDS.

When integrating the Input Framework, the IDS' core *analysis and detection engine* remains mostly unchanged but gains the capability to specify external intelligence sources that it wants to access. From the engine's perspective, each intelligence source corresponds to a *stream* that it creates on-demand as its

[3] The specifics here depend on the capabilities the IDS provides. For example, for a typical signature-based IDS it should be straight-forward to adapt the rule language for doing simple match/no-match decisions derived from external blacklists. However, it will be challenging to use such lists as a reputation indicator that only *contributes* to a decision if such a concept does not already exist in the system.

configuration defines. When opening a stream, it passes along the necessary control information such as type of input (e.g., file, database), location (i.e., a filename or a remote socket), as well as the expected layout of the data that the stream will later forward. For the latter, we represent all intelligence in a unified, column-based format and pass a description along with the stream request.

The Input Framework's *manager* is the central interface between analysis engine and external intelligence sources. It receives the engine's request to open a stream, spawns a new *reader* instance, and instructs it to connect to the corresponding source. We differentiate between types of readers: a *file reader*, e.g., reads from local files, and a *database* reader queries a remote database.

The readers forward all intelligence to the manager, which passes it on the analysis engine. However, rather than directly making it available, data from the readers first passes through an optional set of *filters* that may reduce and transform the input before it gets applied. As the filters run inside the analysis engine, they have access to the full IDS state.

Generally, each reader decides on the model for forwarding input to the manager. A file reader could, for example, read a file once at startup and then keep monitoring it for changes in regular intervals, passing updates on as noticed. On the other hand, a reader connecting to a real-time network feed would instead forward intelligence immediately as it arrives on its input socket.

In our architecture, manager and readers communicate via a simple API that is fully decoupled from the core of the IDS. This make it particularly easy to add new readers as they are not at all concerned with the system's potentially complex internals. In principle, one could even connect a single manager implementation to different IDS implementations just by adapting the upstream interface accordingly. On a technical level, decoupling the readers makes it straight-forward to run them in separate threads, which simplifies the implementation of asynchronous I/O. With that, the only critical point potentially impacting the IDS' packet processing remains the manager/engine interface.

4 Implementation

We implement the Input Framework architecture on top of the Bro IDS. By providing a Turing-complete scripting language for expressing custom detection policies, Bro fits well with the capabilities that the Input Framework offers: we add a new script-level API that allows users to configure external intelligence sources, which Bro then maps transparently into standard data structures. In the following, we discuss the main aspects of our implementation in terms of its internal structure (§4.1) and its user interface (§4.2).

4.1 Integration

Fig. 3 shows how our implementation integrates into Bro.When processing network input, Bro internally reduces the voluminous stream of packets to a series of

higher-level network *events* that reflect the key steps of the underlying activity. [4] A *policy interpreter* then executes scripts written in a specialized, high-level language [5] that expresses both a site's custom security policy and general forms of high-level analysis (e.g., scan detection [13]) in terms of the event stream. A crucial point is that these events are *policy neutral*: Bro itself makes no judgment as to whether the events reflect malicious or benign traffic but rather leaves that determination to a user's custom scripts.

As layed out by our general architecture (see §3.3), a central manager acts as the interface between the Bro core and intelligence sources. The manager spawns separate reader threads for each source. When reading data, these readers pass it on to the manager via thread-safe queues. The manager then feeds the information into either the event stream or directly into the policy interpreter by adding to its data structures, executing filters, and calling script layer functions. We detail all of these further below.

The interface between the manager and Bro's core is performance critical as it needs to trade-off processing incoming intelligence with that of network traffic. Generally, the latter receives priority as its volume prevents Bro from buffering packets to any significant degree. However, to satisfy our real-time constraint, incoming intelligence cannot just wait for lulls in the traffic stream (which in most environments will never occur) but instead propagates incrementally, interleaved with traffic analysis. Internally, Bro already provides an I/O loop structure that allows to balance packet processing with further asynchronous input. The input manager hooks into this loop structure as an additional data source.

4.2 User Interface

Our Input Framework implementation integrates fully into Bro's domain-specific scripting language. In the following, we walk through the main parts of the script-level interface that the Input Framework exposes to the user. As a simple running example, we consider importing a blacklist of hosts, formatted as a 3-tuple (*IP address*, *reason*, *timestamp*) where *IP address* is the host's address, *reason* a textual description of the host's offense, and *timestamp* a Unix-timestamp indicating since when the list entry exists. Stored in a tab-separated file, the list could look like this:

```
ip              reason                  timestamp
66.249.66.1     connected to honeypot   1333252748
208.67.222.222  too many DNS requests   1330235733
192.150.186.11  sent spam               1333145108
```

Reading Files. The Input Framework can directly import files such as the above into *tables*, a associative array type that Bro's scripting language provides, much like hashes in Perl. To do so, the user declares the columns to extract from

[4] Bro provides both generic transport analysis and application-specific analysis. It understands for example specifics of HTTP, DNS, SMB, and many other protocols.
[5] "A domain-specific Python."

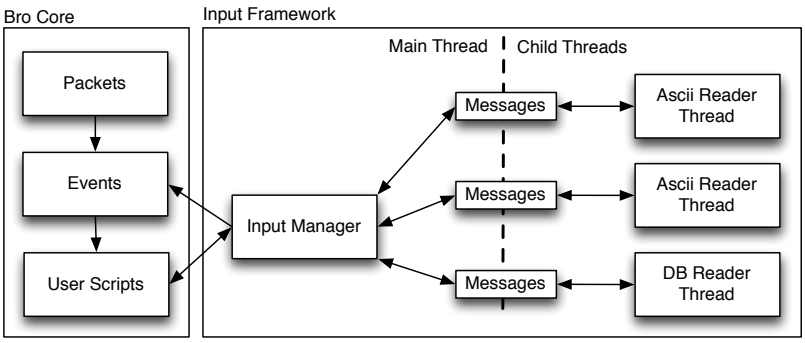

Fig. 3. Input framework implementation in Bro

the file by defining two corresponding *record* types (records are similar to *structs* in C): one for the table index and one for its values. In our example, assuming we want to use the IP address as the table index and the other two columns as its value, we can define the following types:

```
type Index: record { ip: addr; };

type Value: record { reason:    string;
                     timestamp: time; };
```

When reading the blacklist file, the Input Framework will use the records' field names (`ip` and `reason/timestamp`) to locate the corresponding columns, and it will interpret their content according the fields' types (`addr` and `time` are Bro's built-in script types for IP addresses and time values, respectively.).

Next, we define the table that will receive the content of the file:

```
global blacklist: table[addr] of Value;
```

Note that the types for table index and values correspond to the `Index` and `Value` records, respectively.[6] Now that we have defined the types and the table, we use an Input Framework API function to read the blacklist in from a file: [7]

```
Input::add_table(source="blacklist.tsv", idx=Index,
                 val=Value, destination=blacklist);
```

Once read, further script code can test whether the blacklist contains a specific address:

```
if ( 192.150.186.11 in blacklist )
    alarm(...)
```

[6] For the table index we "roll out" the fields because Bro's tables do not support record types as keys. They do however allow for index *tuples* so if our blacklist were indexed by, say, two addresses, we would write `table[addr,addr] of Value`.

[7] In this and later examples we simply Bro's syntax slightly for better readability.

When executing the `add_table` function, the Input Framework's manager internally spawns a new reader thread and then immediately returns back to the caller. While the new thread is parsing the blacklist in the background, it continuously forwards entries to the manager, which in turn adds them to the `blacklist` table incrementally. Script processing continues in parallel, and other event handlers will hence "see" each new entry immediately. In addition, the Input Framework flags completion by triggering a callback event that users can implement for logic requiring that all data has made it into the table.

Updating. Many intelligence sources will see frequent updates, either batched in regular intervals or continuously in the form of a stream. Our implementation provides three mechanisms to accommodate updates.

The most direct mechanism is to call the API's `force_update` function, which will trigger a re-read of a stream's source data (for reader types that support it). The Input Framework will then add any new values to the corresponding table, remove ones that no longer exist, and update any that have changed to their new values. Using `force_update` is a good choice if one knows when to expect a change, such as when one has run an external command. Alternatively, one can also put a reader into *automatic update* mode (via a flag passed to the `add_table` call). In this mode, the reader thread will continually check the source file for modifications and trigger the update operation automatically. Automatic updating works well with files that an external script retrieves regularly from a remote location (for instance via cron).

Finally, for readers that receive continuous streams of intelligence, they may also individually add/delete/modify table entries as they get updates. Keeping with the file example, this mode corresponds to "tailing" a file on a Unix system. Usually, however, such streaming readers will have a persistent connection to an external data source. For example, new blacklist entries could be coming in via a network connection, or a database reader may subscribe to live query updates.

Filtering. Often it is beneficial to prefilter the information coming in from an intelligence source. In our blacklist example we might, for example, only want to consider entries added within the last few days. Our implementation offers corresponding hooks that can modify or remove entries before they land in the table. Such a hook comes in the form of a callback function that the Input Framework runs for any table update under consideration. The hook receives the line to be added to the table and the information whether the entry is new, changed or updated. The hook function returns a boolean that signals if the change is to be applied or rejected. If the hook rejects a change, the element is not added for new elements, not updated for changed elements and not deleted for removed elements. In our blacklist example, we could use that to only consider blacklist entries that are not older than five days.

One assigns such a filter to a stream by passing the function as an additional argument to the `add_table` call. We emphasize that filters can access all of Bro's already accumulated state, including other script-level tables and data structures. They can even *modify* other tables, which for example allows to split intelligence from a single source across a set of related tables.

Triggering Events. Some types of intelligence do not map directly to a table structure. For example, a source may be sending information that Bro must react upon immediately, rather than storing it for later inspection. To support such applications, our Input Framework implementation offers a second, simpler reading mode in which the manager triggers an event callback for every entry it receives. The callback's arguments are similar to that of the filter function described above but do not include table-specific elements like the index value.

4.3 Reader Types

Our current Input Framework implementation supports three types of intelligence readers. As we show in our examples above, it can read ASCII files in the form of typed tab-separated columns. We kept the format's specifics compatible to Bro's structured log files, which now enables users to read *log files back in*, providing a powerful mechanism to maintain rich state persistently across restarts.[8]

The second supported format is "raw" content, in which the reader simply passes on the content of a file as a raw blob. This mode can only trigger events, not table updates, as there is no further structure associated with the data. Optionally, the raw reader can also split a file at predefined separator characters. It is, for example, possible to get one event for each line in a file. As an extension, the raw reader can take its input not only from files but also from the output of custom shell commands. This feature enables in particular to query external web services using a utility like `curl`. The following code snippet demonstrates how Bro can retrieve a JSON file on the fly:

```
# Define the type that will store the data.
type JSON: record { data: string; };

# Define the handler that will process the JSON data.
event got_data(value: JSON) {
    ... Code to process data goes here ...
}

# Trigger the request.
Input::add_event(source="curl www.host.com/list.json |",
                 fields=JSON, event=got_data, reader=Input::RAW);
```

The `add_event` call creates a new reader thread that first executes the external command and returns the output asynchronously by generating a `got_data` event. The event handler can then further parse the data.[9] Note how the `source` argument ends with a pipe symbol to indicate that the value reflects a command to execute, not a file name.

[8] Bro's logging system indeed complements our work by providing a corresponding *output framework*.

[9] Alternatively, one could parse the JSON externally as part of the executed command and use the ASCII reader to receive it in a structured form.

Finally, as our third reader type, we develop a PostgresSQL interface that executes SQL queries and forwards the result back to Bro, mapping it either transparently into a table or into events, as described above. We discuss this reader in more detail in §5.4 where we show a concrete usage scenario. We also add a PostgresSQL log *writer* to Bro. In combination, reader and writer enable users to perform arbitrary bi-directional database transactions in real-time, all in parallel to Bro's normal packet processing and with full access to its global state.

Our Input Framework implementation provides a simple self-contained API for implementing new readers that makes it straight-forward to add further types of input. In particular, we are planing to add interfaces to other databases as well as to syslog clients.

5 Deployment Scenarios

To support our claim that the Input Framework introduces a fundamentally new IDS capability, we now examine its potential from a deployment perspective. In §5.1 we first examine the need for *trusting* external intelligence as an overarching operational concern that current IDS do not sufficiently address. In §5.2 we discuss a real-world application of the Input Framework that is already in operational deployment at the Lawrence Berkeley National Laboratory. Finally, §5.3 and §5.4 discuss two further usage scenarios that we prototype as case studies and expect to similarly move into operations. We note that many specifics of our Input Framework design and implementation evolved through close interaction with network operators at a number of sites, and the discussion in this section captures much of the feedback we received.

5.1 To Trust or Not to Trust?

From an operational perspective, *trust* is a crucial concern when integrating third-party intelligence into a site's security decisions. Consider a site with a policy to automatically block connectivity for malicious external IP addresses. With external IP blacklists coming in for example from a federation like REN-ISAC, the operators need to decide which of the addresses justify a block. On a technical level, simply blocking all of them is rarely feasible due to limits on the number of rules that firewalls can handle. But more importantly, there is rarely any local control on what exactly the intelligence feeds include and hence their information requires additional vetting and a process to develop confidence in the quality of the data. In particular, operational usage needs to account for policy differences between sites—the IP address of a P2P tracker, or an *undernet* IRC server, may be criticial to block for one site, yet tolerated at another. Also, any accidental inclusions risk severely impacting legitimate traffic (finding IPs from Akamai or Amazon AWS blacklisted is not unusual). Furthermore, some feeds (like REN-ISAC's) contain additional qualifying information such as severity ratings, confidence levels assigned to an entry, or number of distinct sources reporting it. Such ratings tend to be highly subjective and are thus often insufficient to

trigger automated action on their own. They may however contribute to crossing a threshold when combined with further orthogonal evidence.

Operationally, a crucial shortcoming of many IDS implementations is their lack of support for the fine-granular decisions that such considerations require. Accordingly, we see operators falling back to externally vetting information via custom scripts before then converting them into static IDS rules. That, however, lacks any flexibility to go beyond simple black/white decisions. There is no way to convert a dynamic reputation scheme into a standard signature.

The Input Framework addresses such concerns by providing the means to incorporate intelligence into the IDS decision process itself, rather than leaving it to external pre- or postprocessing. Doing so not only fundamentally improves detection capabilities and response times, but also provides considerable workflow improvements by eliminating the external process that attempts to fit the intelligence into what the IDS configuration language supports.

5.2 Federated Blacklists

Our Input Framework implementation is in operational deployment at the Lawrence Berkeley National Laboratory (LBNL), where the cyber security team uses it to integrate both SES feeds and JC3 feeds into the Lab's Bro installation. In the following we report on their experience with the new capability after nearly 2 months of use. LBNL adopted use of the Input Framework due to its ability to continuously integrate crucial indicators into the monitoring infrastructure as quickly as they are published. Prior to the Input Framework, it was not operationally feasible to repeatedly restart their IDS potentially multiple times an hour as feeds were published in an adhoc manner. Additionally, incorporating policies by hand was an error prone process causing unintentional delays.

LBNL prefers using the SES and JC3 limited-circulation feeds over other public sources as they supply vetted data and are continuously maintained. As such, these feeds allow for tight integration with the IDS and enable to automate decisions as their semantics are well understood. The institutions behind the feeds also allow LBNL to go back upstream and inquire about potential false positives or borderline cases. The SES feed is updated automatically once per day and the JC3 feed is downloaded manually from a secure server when updates are released. The SES feed contains individual subfeeds for spam, scanners, phishing, suspicious nameservers, and suspicious networks. In general, these feeds contain different types and volumes of intelligence in the order of 300–3000 lines per subfeed. Typical entries for SES are an malicious host's IP address, event timestamp, domain, port, URL and file MD5 hash, as appropriate. Each item also comes with a separate severity rating as well as a confidence level. JC3 provides malicious domains and IP addresses, augmented with information about which sites reported the threat and also threat-level estimates.

LBNL uses the Input Framework's table interface (see §4.2) to directly import the feeds into a set of Bro tables. External scripts query the feeds from the providers and write them to disk. The Input Framework then picks up the changes transparently. LBNL also uses the filter mechanism to modify data

during imports. For example, some SES subfeeds do not contain hostnames as a separate column but only come with complete URLs to malware. However, these rarely appear as a whole in network traffic and LBNL hence uses a custom filter function to extract the hostname and turn it into a table index on the fly. Furthermore, LBNL joins several feeds into a single table by configuring multiple sources that all write to the same destination.

During the two months of deployment, this setup has proven to improve LBNL's detection capabilities in a number of ways. As an example, HTTP scanners are notoriously difficult to detect reliably because it is difficult to distinguish a malicious scanner from a search engine's web crawler. However, having intelligence feed integration, LBNL can now tie feed data to Bro's TCP-level scan detector. When the latter finds a possible scanner, the IDS checks to see if the IP is blacklisted; if so, it blocks it automatically. Combining the two detectors in this way allows to quickly block HTTP scanners without subjecting *all* blacklisted IPs to that treatment (and thus preventing many unjustified blocks). A similar approach works well for encrypted SSH and HTTPS traffic, which an IDS cannot further inspect at the content-level. The Input Framework has already triggered investigations in several such cases.

More generally, LBNL finds that steering subsets of intelligence into corresponding protocol-specific analyses leads to more reliable alarms than the standard approach of matching broadly against the bulk of the traffic. For example, LBNL uses a hook into Bro's TCP analysis to trigger intelligence lookups for the addresses of every newly established connection. Likewise, all DNS requests and replies lead to checks for the corresponding domain names. While already valuable on their own, one can also correlate matches across protocols. For instance, a blacklisted path may first appear in an HTTP request followed by a DNS lookup for a malicious domain. Indeed, the majority of intelligence-triggered alerts currently corresponds to such DNS-after-HTTP matches.

5.3 Online Virus Checks

As a second deployment scenario, we prototype online virus checks using the popular malware checking service *VirusTotal* [29]. While network-level virus checking is not a novel concept, most solutions operate as proxies that actively intercept TCP sessions. A few commercial systems seem to support passive virus checks (e.g., by FireEye and Netwitness) but we are not aware of available open-source solutions doing live packet-level scans. The Input Framework makes it straight-forward to support such functionality within an existing IDS.

In fact, with the Input Framework in place the main challenge is not interfacing to a virus checker but extracting files from network traffic. Here, we leverage the file extraction features that Bro's HTTP engine provides, including its support for content downloaded in chunks or from multiple sessions simultaneously. Hence it is, for example, possible to recognize viruses in files where a user first aborted a download and later resumed it at the aborted position.

We implemented online virus checking as a plugin to Bro's *file analysis framework* that is currently in development and scheduled to be part of an

upcoming release. The framework supports reassembly for files transferred non-linearly and provides a convenient hooking point for handling files across protocols in a standardized way. A plugin for VirusTotal provides two alternative operation modes: it can either *(i)* calculate an MD5 hash for a file on the fly and submit that to the VirusTotal API for checking against its database; or *(ii)* submit the file *in whole* to the service. In both cases, VirusTotal returns a JSON-string indicating the results that we then parse in a Bro script using the Input Framework's event interface. If there is a match, the script can take action such as notifying an operator or disconnecting the victim system. An alternative to using VirusTotal would be to instead call a local virus scan engine. Doing so can be beneficial if privacy concerns prevent online lookups.

The Bro script providing the VirusTotal functionality on top of the Input Framework is just about one hundred lines long, including empty lines and comments. To avoid an excessive number of lookups, it allows to optionally analyze only a subset of all files, such as selected file types, or content from specific IP addresses or CIDR ranges. During our testing, we indeed detected a malicious file transfer from an compromised host in a 600MB real-world trace.

5.4 Database Interface

Our initial implementation also provides a database reader that connects to PostgresSQL. The reader supports both importing data once from a DB table, and continuously as updates from live queries arrive. Compared to the text-based intelligence we have used in our examples so far, the database interface opens up further potential by providing real-time access to external intelligence that exceeds a volume that an IDS could handle itself internally.

To demonstrate this capability, we consider a setting where the IDS flags suspicious activity for its operators but also augments the alert with further context about the attack source. Specifically, we want to integrate *whois* information into the notification, such as the time when a domain was registered and its administrative contact information. The Bro-side for that comes in two parts: *(i)* we hook into Bro's processing to execute a database query via the Input Framework when an alarm triggers; and *(ii)* when the database's reply arrives, we augment the alarm accordingly and then pass it on for further processing. Somewhat simplified[10] the query looks like this:

```
add_event(source="select * from whois where domain='"+ domain +";",
   name=<uid>, event=got_reply, reader=Input::POSTGRES);
```

Here, the `uid` is an automatically generated unique identifier that later allows the `got_reply` handler to associate a reply with the corresponding query.

We test this approach using an internally maintained PostgresSQL database that contains complete whois information for several million domains, and we find

[10] We configure the database connection separately. In practice, one must ensure to sanitize the `domain` to avoid SQL injection attacks.

it to work as expected. In practice, one could extend this scenario in a number of ways. For example, rather than just augmenting alerts, the IDS can use the database information to further assess the threat, such as by elevating an alarm's priority when it involves recently registered domains.

More generally, this scenario demonstrates how the Input Framework can make intelligence available to the IDS *on demand*, without needing to move all the information into the system itself. Database connections is the most powerful of all our examples, and we expect that operators will start relying on them extensively as they become familiar with the new capability.

6 Performance Evaluation

To understand the performance of our implementation we perform a set of measurements to determine *(i)* the intelligence volume it can handle under realistic workloads; and *(ii)* the latency with which real-time input becomes available to the IDS analysis.

6.1 Benchmark Reader

We create a dedicated *benchmark reader* for our measurements. Rather than connecting to an actual intelligence source, that reader generates artificial data with characteristics that we can freely configure. The reader first examines the data types requested via the Input Framework API (see §4), and then generates corresponding table updates and events. For example, when the API requests intelligence of string type, the benchmark reader returns a random byte sequence. It recursively fills in fields of record types and can hence generate arbitrarily complex data structures on the fly. One can configure the rate with which it sends updates and also an optional increase of that rate over time.

6.2 Realistic Workloads

It is challenging to benchmark IDS systems with realistic workloads in a way that is repeatable and has reproducible results. We cannot just run the system on live traffic because continuous variations in packet volume and mix would not allow for fair comparisons of different configurations executed sequentially. We however also cannot run offline from traces as Bro would process the packets as quickly possible (i.e., at 100% CPU usage), without the normal lags seen during real-time operation that the Input Framework uses to interleave intelligence updates with the packet processing. To overcome these problems, we leverage Bro's *pseudo-realtime* mode [22] which combines the best of both worlds. In that mode, Bro reads its input from a trace, yet it mimics real-time behavior by introducing artificial delays into the packet processing, corresponding to timestamp differences of consecutive packets. Doing so results in reproducible operation that is comparable to using the Input Framework on live traffic.

For our evaluations we capture a 5-minute packet trace at the uplink of UC Berkeley. The campus' upstream connectivity consists of two 10 GE links, with daytime average rates of 3-4Gb/s total. Such a volume is much more than a single Bro instance can handle, and we thus record only a subset corresponding to a more realistic setting. Specifically, we capture the traffic that a single Bro instance analyzes in the Berkeley campus' NIDS Cluster [27], which corresponds to $\frac{1}{28}$ of all flows. The resulting trace contains 100M unique IP addresses and 330K flows. 81% of the packets are TCP, and port 80 is the most common port (31,7%). The average data rate is 222 MBit/s at about 40K packets/sec.

For our measurements we use a current development version of Bro with a recommended, complex default configuration. When running without the Input Framework, a Bro process exhibits a CPU load of 50-80% while processing the trace on our evaluation system[11], which is about the level realistic for live operation without incurring packet drops.

6.3 Sustainable Load

We measure the load of the main Bro thread when the benchmark reader generates certain fixed numbers of events per second. Besides system characteristics such as CPU and memory resources, the sustainable data rate depends on the complexity of the involved data types (i.e., the record definitions), and on the reading mode in use (table updates or events).

For simple events, consisting just of a timestamp, we measure a limit of about 42,000 events/sec. Fig. 4 compares the CPU utilization for three different rates. For each rate, the plot shows the probability density of CPU load samples measured in 1s intervals over the course of processing the 5-min input trace. For comparison, we also show the load for a baseline run that does not activate the Input Framework. We see that at 10,000 events/sec, the CPU load increases just slightly (average 51% vs. 49%). At 36,000 events/sec it increases more noticably (average 58%), and at 50,000 events/sec individual CPU samples exceed 1.0s, i.e., more than the system can support.[12]

We repeat similar measurements with more complex events as well as with simple and complex tables. For the complex case, the record type contains 14 different data fields, of which 5 are list types (which is more than operations are likely to use). The sustainable loads for complex events are about 4,000 entries per second. For simple tables, the Input Framework can handle about 20,000 entries per second and for complex tables about 2,000 entries per second. The CPU loads at those rates are similar to those in Fig. 4 and we thus skip corresponding plots. We also examine system load with very low data rates (10s of updates/sec), which is more likely what one will see in typical deployments. For these, we do not see any measurable load increase.

[11] The system has two quad-Core Xeon E4530 CPUs @2.66 GHz and 12 GB RAM.

[12] With Bro's pseudo-realtime mode, a CPU sample >1s means that the time required for processing 1s of network traffic exceeded 1s of real-time, which in live operation would have resulted in packet loss.

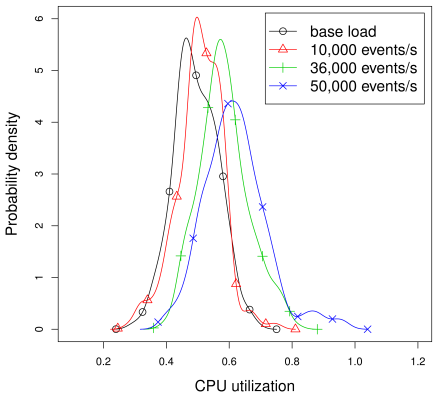

Fig. 4. Input Framework load increases

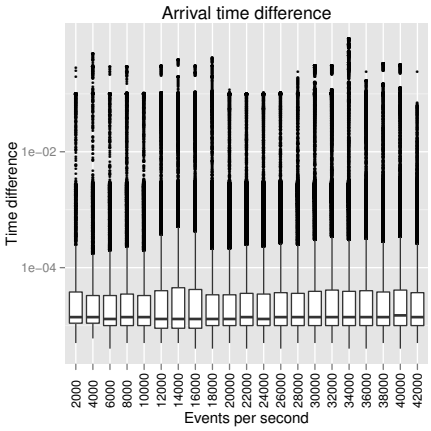

Fig. 5. Event latency evaluation

Overall, we conclude that even with complex intelligence data, our implementation can sustain more than 3,000 insertions/sec while processing a typical packet load with a complex IDS analysis configuration. With less complex input, it achieves rates matching that of the packet input(!). The observed CPU increase is hardly surprising at such high rates. Operationally, however, the most relevant result is a different one: having headroom to accommodate high update frequencies is good, yet most deployments will never see such rates. For them, the we find that the Input Framework does not increase CPU usage.

6.4 Latency

From an operations perspective, the time it takes to make intelligence updates available to the IDS analysis is another important factor for operations. Consequently, we also measure the Input Framework's latency, i.e., the difference between the time when it receives an update from a source until that becomes available at the scripting layer. We configure the benchmark reader to generate events that include the current timestamp, and a receiving Bro script then calculate the difference. As in the load evaluation, we use Bro's pseudo-realtime mode running again on the same trace file. We performe a series of measurements, each time increasing the rate at which the benchmark reader sent events. We stop the series at the maximal attainable rate of 42,000 events/sec.

Fig. 5 visualizes the measured latencies for each rate in the form of a bar plot. (Note the logarithmic scale on the y-axis.) The whiskers end at 1.5*IQR+Q3, all other points are considered outliers and plotted as a dot. We see that the latencies remain very small, averaging around `1.4ms`. There are a few infrequent cases that have latencies in excess of 100 msec (less than 0.4%)—however, even in the worst case, the latency is under 900 ms. The minimum time difference is 4 μsec and hence in the order of measurement inaccuracies. Interestingly, the latencies do not change much at all as the rate increases, indicating that as long as the Input Framework can operate at a rate, it will forward updates rapidly.

Overall, we conclude that the Input Framework does not add significant delays after receiving intelligence from a source.

7 Conclusion & Outlook

The global security community is collecting a treasure trove of third-party intelligence that can support operations staff in automating incident detection and investigation, including many forms of blacklists recording known bad actors and malicious content. Unfortunately, network intrusion detection systems still miss out on fully leveraging this potential for making more reliable decisions as they do not offer corresponding interfaces for flexibly integrating such knowledge.

In our work, we present a novel architecture that adds unconstrained intelligence access as a new capability to the IDS toolbox. We design an Input Framework that adapts to a variety of sources, provides a simple yet flexible user interface, and integrates smoothly with an IDS' main task of analyzing high-volume packet streams under soft real-time constraints. We implement an initial version of the Input Framework on top of the open-source Bro IDS. We also prototype a set of usage scenarios that exploit the power of the new capability, including integration with federated intelligence sharing initiatives, online virus checks for downloaded files, and real-time interaction with a PostgresSQL database for assessing the relevance of alarms on the fly. Furthermore, separate benchmark measurements confirm that our implementation is well-suited to handle frequent real-time intelligence updates while adding virtually no delay before making it available to the IDS analysis.

This Input Framework implementation is already in operational deployment at the Lawrence Berkeley National Laboratory where the Lab's security team finds it to significantly improve their detection capabilities. With the Input Framework's generic approach to integrating intelligence, we are looking forward to the operations community developing further powerful applications as they become familiar with the new capability.

Acknowledgments: We would like to thank the Lawrence Berkeley National Laboratory for their collaboration. This work was supported by the U.S. Army Research Laboratory and the U.S. Army Research Office under MURI grant No. W911NF-09-1-0553; a fellowship within the Postdoc-Programme of the German Academic Exchange Service (DAAD); by the Director, Office of Science, Office of Safety, Security, and Infrastructure, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231; and by the US National Science Foundation under grant OCI-1032889. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors or originators and do not necessarily reflect the views of the DAAD, the ARL/ARO, the DOE, or the NSF, respectively.

References

1. Department of Energy Cyber Joint Cybersecurity Coordination Center. *http://www.doecirc.energy.gov/*

2. National Software Reference Library. *http://www.nsrl.nist.gov/*
3. Antonakakis, M., Perdisci, R., Dagon, D., Lee, W., Feamster, N.: Building a Dynamic Reputation System for DNS. In: USENIX Security (2010)
4. Blacklist.rules, ClamAV, and Data Mining. *http://vrt-blog.snort.org/2011/02/blacklistrules-clamav-and-data-mining.html*
5. Collective Intelligence Framework. *http://code.google.com/p/collective-intelligence-framework/*
6. Cyber Fed Model – Community-Wide Cyber Security Alert Distribution. *http://web.anl.gov/it/cfm/*
7. Cymru, T.: Malware Hash Registry. *http://www.team-cymru.org/Services/MHR/*
8. Debar, H., Wespi, A.: Aggregation and Correlation of Intrusion-Detection Alerts. In: RAID (2001)
9. Dreger, H., Feldmann, A., Paxson, V., Sommer, R.: Operational Experiences with High-Volume Network Intrusion Detection. In: ACM CCS (2004)
10. DShield.org Recommended Block List. *http://feeds.dshield.org/block.txt*
11. Google Safe Browsing API. *http://code.google.com/apis/safebrowsing*
12. Gu, G., Porras, P., Yegneswaran, V., Fong, M., Lee, W.: BotHunter: Detecting Malware Infection through IDS-driven Dialog Correlation. In: USENIX Security (2007)
13. Jung, J., Paxson, V., Berger, A.W., Balakrishnan, H.: Fast Portscan Detection Using Sequential Hypothesis Testing. In: IEEE Security and Privacy (2004)
14. Katti, S., Krishnamurthy, B., Katabi, D.: Collaborating against common enemies. In: IMC (2005)
15. Ollmann, G.: Blacklists & Dynamic Reputation. White paper (2011), *http://www.damballa.com/downloads/r_pubs/WP_Blacklists_Dynamic_Reputation.pdf*
16. Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. Computer Networks 31(23–24), 2435–2463 (1999)
17. Roesch, M.: Snort: Lightweight Intrusion Detection for Networks. In: Systems Administration Conference (1999)
18. Security Event System. *http://www.ren-isac.net/ses*
19. Sharma, A., Kalbarczyk, Z., Barlow, J., Iyer, R.K.: Analysis of Security Data From a Large Computing Organization. In: IEEE DSN (2011)
20. Sinha, S., Bailey, M., Jahanian, F.: Improving SPAM Blacklisting through Dynamic Thresholding and Speculative Aggregation. In: NDSS (2010)
21. Snort 2.9.1 release announcement. *http://blog.snort.org/2011/08/snort-291-has-been-released-including.html*
22. Sommer, R., Paxson, V.: Exploiting Independent State For Network Intrusion Detection. In: ACSAC (2005)
23. The Spamhaus Block List. *http://www.spamhaus.org/sbl*
24. Open Information Security Foundation: Suricata Download. *http://www.openinfosecfoundation.org/index.php/downloads*
25. Symantec - Configuring blacklisting for base event types with IDS/IPS on Symantec Gateway Security 5400 Series 2.x. *http://www.symantec.com/business/support/index?page=content&id=TECH81936*
26. Valdes, A., Skinner, K.: Probabilistic Alert Correlations. In: Proc. RAID (2001)
27. Vallentin, M., Sommer, R., Lee, J., Leres, C., Paxson, V., Tierney, B.: The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In: RAID (2007)
28. Verizon: 2012 Data Breach Investigations Report. Tech. rep., *http://www.wired.com/images_blogs/threatlevel/2012/03/Verizon-Data-Breach-Report-2012.pdf*
29. VirusTotal Public API. *https://www.virustotal.com/documentation/public-api/*