Providing Dynamic Control to Passive Network Security Monitoring

Johanna Amann[1(✉)] and Robin Sommer[1,2]

[1] International Computer Science Institute, Berkeley, USA
[2] Lawrence Berkeley National Laboratory, Berkeley, USA
{johanna,robin}@icir.org

Abstract. Passive network intrusion detection systems detect a wide range of attacks, yet by themselves lack the capability to actively respond to what they find. Some sites thus provide their IDS with a separate control channel back to the network, typically by enabling it to dynamically insert ACLs into a gateway router for blocking IP addresses. Such setups, however, tend to remain narrowly tailored to the site's specifics, with little opportunity for reuse elsewhere, as different networks deploy a wide array of hard- and software and differ in their network topologies. To overcome the shortcomings of such ad-hoc approaches, we present a novel *network control framework* that provides passive network monitoring systems with a flexible, unified interface for active response, hiding the complexity of heterogeneous network equipment behind a simple task-oriented API. Targeting operational deployment in large-scale network environments, we implement the design of our framework on top of an existing open-source IDS. We provide exemplary backends, including an interface to OpenFlow hardware, and evaluate our approach in terms of functionality and performance.

1 Introduction

Network intrusion detection and prevention systems (IDS and IPS, respectively) detect a wide range of attacks, including port- and address scans for reconnaissance, SSH brute-forcing, attempts to exploit specific vulnerabilities (e.g., Heartbleed), and also complex multi-step APT-style attacks. An *IPS* operates *inline* within the network's forwarding path, enabling the system to actively react to an intrusion by, e.g., blocking the specific connection or more generally any traffic originating from the same IP address. Operationally, however, inline operation often remains impractical, as it adds a complex device into the forwarding path that increases latencies and jitter, and risks causing disruption if malfunctioning. Furthermore, for the largest of environments—such as the quickly growing set of 100 G *Science DMZs* [20]—arguable no IPS (or firewall) can today operate at their line rates at all [9]. More commonly, network environments thus deploy a passive *IDS* instead, operating out-of-band on an independent copy of the network traffic coming from a tap or SPAN port. To still support active response in that setting, some sites then provide their IDS with a separate control channel

H. Bos et al. (Eds.): RAID 2015, LNCS 9404, pp. 133–152, 2015.
DOI: 10.1007/978-3-319-26362-5_7

back to the network, most typically by having it dynamically insert ACLs or null routes into a gateway router for blocking IP addresses. Such setups, however, tend to remain narrowly tailored to the site's network topology and the specific equipment it deploys, offering little opportunity for reuse elsewhere and posing challenges for testing, maintenance, and extension.

To overcome the shortcomings of such ad-hoc approaches, we present a novel *network control framework* that provides passive network monitoring systems with a flexible, unified interface for active response, hiding the complexity of heterogeneous network equipment behind a simple task-oriented API. We structure our network control framework around four low-level traffic control primitives: dropping, whitelisting, redirection, and modification. From these, we then compose a set of higher-level tasks that an IDS can chose to deploy, such as blocking IP addresses, shunting traffic for load shedding, quarantining infected hosts, and enforcing quality-of-service guarantees. Internally, we map the primitives to rules that the network control framework forwards to a set of preconfigured backends representing the network's devices able to carry out the corresponding actions (e.g., routers, switches, firewalls). When multiple components can execute a rule, the network control framework automatically selects the most appropriate. It also transparently unifies inconsistencies between device semantics.

Our framework targets operational deployment in large-scale network environments with link capacities of 10 G and beyond. We implement the design of our framework on top of an existing open-source IDS that such environments commonly deploy. We provide exemplary backends for OpenFlow and *acld* [1], as well as a generic backend driving command-line tools (which we demonstrate with Linux iptables). Using the OpenFlow backend, we evaluate our approach through case studies involving real-world tasks and traffic. We release our implementation as open-source software under BSD license [14].

Overall, our work offers a new capability combining the advantages of unobtrusive passive monitoring with the ability to actively react swiftly and comprehensively through a unified architecture that can replace today's ad-hoc setups. While our discussion focuses primarily on the security domain, the network control framework's potential extends more broadly to traffic engineering applications well, as our quality-of-service use case demonstrates.

We structure the remainder of this paper as follows: Sect. 2 discusses use cases for our system. Section 3 presents the design of the network control framework, and Sect. 4 describes our implementation. Section 5 introduces the backends that the network control framework currently supports, and Sect. 6 evaluates the framework. Section 7 discusses the related work before our paper concludes in Sect. 8.

2 Use Cases

We begin by discussing four high-level IDS use-cases that our network control framework facilitates. Traditionally, a site would implement each of these

separately, typically with homegrown scripts that cater to their network environment. The network control framework instead offers a high-level API that supports these use cases directly, internally breaking them down into lower-level rules that it then carries out through an appropriate backend.

Dynamic Firewall. The network control framework enables an IDS to dynamically block traffic that it deems hostile. Typical examples include stopping a connection exhibiting illegitimate activity, and dropping connectivity for hosts probing the network. In contrast to a traditional firewall, an IDS can derive such decisions dynamically by analyzing session content and tracking over time the state of any entities it observes. For example, the Lawrence Berkeley National Laboratory (LBNL), a research lab with a staff size of about 4,000 and 100 G connectivity, blocks an average of about 6,000 to 7,000 IPs each day using a custom setup that interfaces the Bro IDS [17] with their border router through a separate daemon process, *acld* [1]. Indiana University, which has more than 100,000 students and multiple 10GE uplinks, blocks an average of 500 to 600 IPs per day, also using a custom setup processing data from Bro and Snort.

Shunting. Flow shunting [7,11] reduces the load on an IDS by asking the network to no longer send it further traffic for high-volume connections that it has identified as benign. In scientific environments, shunting typically targets large file transfers: once identified as such, there remains little value in inspecting their content in depth. Shedding the corresponding load leaves more resources for inspecting the remaining traffic, which in turn then allows a site to provision less IDS capacity than the full volume would require. Two sites using this approach effectively are LBNL and UIUC's *National Center for Supercomputing Applications* (NCSA). Both places currently implement shunting for GridFTP traffic with custom scripts that exploit the specifics of their network environments. On a typical day in these environments, shunting reduces the total traffic volume by about 37 % and 32 %, respectively.

Quarantine. When an IDS identifies a local system as compromised, it can—as a protective measure—redirect any new connections from that host to an internal web server that informs the user of the problem. Traditionally, implementing such a quarantine mechanism constitutes a complex task operationally, as it needs to interact closely with the local network infrastructure. For example, the Munich Scientific Network (MSN) deploys a custom NAT system [21] for quarantining that implements the corresponding logic for local end-user systems by combining a number of existing software components.

Quality-of-Service. Going beyond the security domain, the network control framework also facilitates more general traffic engineering applications. By steering traffic to different switch ports or VLANs, one can route entities over paths with different properties. For example, a Science DMZ might want to send a high-volume data transfer onto a different *virtual circuit* that provides bandwidth guarantees [15]. Another use case is bandwidth throttling. For DDOS mitigation, one can move a local target server to a different ingress path enforcing a rate-limit, thereby relieving pressure for the remaining traffic. Likewise,

a network monitor might decide to throttle individual P2P clients that it finds exceeding their bandwidth quota.

3 Design

In this work, we introduce a network control framework that enables passive monitoring applications to transparently exercise control over heterogeneous network components like switches, routers, and firewalls. In this section, we discuss the design of the network control framework, starting with its overarching objectives in Sect. 3.1.

3.1 Objectives

Our design of the network control framework aims for the following objectives:

Simple, Yet Flexible API. The network control framework's API needs to provide sufficient abstraction to make it straight-forward to use, yet remain flexible to support a variety of use cases. The API should support common high-level tasks directly (like blocking and shunting), while leaving lower-level functionality accessible that enables users to compose their own.

Unification of Heterogeneous Network Components. Sites deploy a variety of network equipment with different capabilities and semantics. The network control framework needs to unify their differences through an API that abstracts from device specifics.

Support for Complex Topologies. As networks can have complex structures, the network control framework needs to support instantiating multiple backends simultaneously, to then chose the most appropriate for each rule. For example, actions that block traffic might need to address a different device than reducing the load on the IDS through shunting. Likewise, one device may support a specific operation better, or more efficiently, than another (e.g., MAC address filtering vs. IPv6 filtering; or when dropping traffic, being closer to the source).

Unification of Forwarding and Monitoring Path. The network control framework provides control over both traffic that the network forwards and traffic that the IDS receives for processing from its tap or SPAN port. Even though the effect of manipulating them is quite different—rules on the forwarding path affect end-users, while the monitoring path only changes what the IDS' analyzes—the corresponding operations remain conceptually similar. The network control framework should thus unify the two behind a single interface.

Low Latency. The network control framework has to apply new rules rapidly. The main difference between a passive IDS and an inline IPS is the latency with which active response takes place. While network control framework can fundamentally not match an IPS' instantaneous action, the network control framework must add as little delay as possible to any latency that the devices impose that it controls.

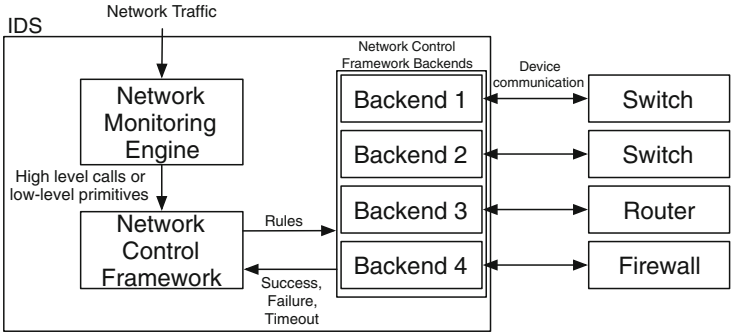

Fig. 1. Basic architecture.

3.2 Architecture

Figure 1 shows the overall architecture of the network control framework, located inside an IDS as a subcomponent. The IDS deploys its standard detection mechanisms (signatures, analysis scripts, etc.) to asses the traffic it sees. Once it decides to take action (e.g., block a scanner), it directs the network control framework to carry that out, using either its *high-level API* if that supports the use case directly through one of its operations, or *lower-level primitives* to compose non-standard functionality. In the former case, the network control framework internally maps the high-level API call to a sequence of corresponding low-level primitives. In either case, it then forwards the primitives to a set of *backends* in the form of *rules* for them to install. Each backend corresponds to a device on the network able to carry out actions. When operators register their devices with the network control framework, they provide the corresponding backend instance with the information how to communicate with the physical device (e.g., the IP address of an OpenFlow switch), as well as potentially further context information about the device's capabilities and location (e.g., the IP space behind a switch). They also assign each backend a priority. For each new rule, the network control framework then iterates through all available backends, from highest priority to lowest priority, until one confirms that it successfully executed the desired action. If no backend accepts the rule, the operation fails. In the following subsections, we elaborate further on the main parts of this overall scheme.

3.3 High-Level Operations

The network control framework supports eight predefined high-level operations, which provide for the most common IDS use cases:

drop_connection(connection, timeout) facilitates dynamic firewalling by terminating a connection through blocking its packets on the forwarding path. It receives the connection's 5-tuple as a parameter, as well as a timeout indicating the duration for the block to stay in place.

drop_address(host, timeout) operates similar to *drop_connection()*, yet blocks all traffic to or from a given IP address.

drop_address_catch_release(host, timeout) is a variant of the *drop_address()* operation that employs *catch-and-release* (C&R) [8] to reduce the number of blocks that the network needs to enforce simultaneously. C&R drops an address initially for only a short period of time (usually much less than the specified timeout). However, when that block expires, C&R keeps tracking the offending IP, and any new connection attempt will trigger an immediate reblock, now for a longer period of time. The process repeats until it reaches the maximum timeout duration. In practice, C&R significantly reduces the number of simultaneous blocks that the network has to support, enabling medium- to long-term blocking while remaining parsimonious with switch &router memory resources.

shunt_flow(flow, timeout) drops a unidirectional flow (specified in the form of its 5-tuple) from the *monitoring* path for a specified duration. This allows, e.g., to remove a large file transfer from the IDS' input stream. As on high-volume connections the bulk of the data tends to flow in one direction only, this operation leaves the other side of the session unaffected.

quarantine(Infected, DNS server, Quarantine server, timeout) isolates an internal host (infected) by blocking all of its traffic except for DNS, which it reroutes to a Quarantine server running an instrumented DNS server that always returns the IP address of itself for any hostname lookup. The quarantine server also runs a web server that can then serve a web page to the host informing end users of the reason for quarantining them.

redirect(flow, destination port, timeout) redirects a unidirectional flow to a different output port on a switch. This allows, e.g., to control quality-of-service properties by steering the traffic to a dedicated port/link.

whitelist(prefix, timeout) whitelists a network prefix so that no other network control framework operation will affect it. This serves as a safety measure, e.g., to avoid blocking critical servers or, more generally, IP space from upstream service providers like Amazon or Google.

All of these operations returns an opaque handle associated with the changes they put in place. An additional *remove(handle)* function uninstalls the operation even before it timeout expires.

3.4 Low-Level Primitives

After introducing the high-level operations of the network control framework in Sect. 3.3, this section shows how users can manually create lower-level rules through a more verbose, but powerful API that directly exposes the primitives underlying the network control framework.

Generally, a *rule* describes an action to perform on a traffic subset through a set of attributes; see Table 1. Each rule consists of three primary components: *(i)* the *type* of action to perform (e.g., *drop*); *(ii)* the *entity* to apply the action to (e.g., a specific IP address); and *(iii)* the *target* network path to operate on (forwarding or monitoring). The rule type specifies the action to perform on all of the entity's traffic, with four choices currently supported: *drop* (drop all traffic matching entity), *whitelist* (leave entity unaffected of any other

Table 1. Summary of network control framework rules.

Rule Specification:	
Type	Type of Rule (*Drop*, *Modify*, *Redirect*, or *Whitelist*).
Target	Rule targets either the *Forward* or *Monitor* path.
Entity	*Entity* (IP Address, Mac, Flow or Connection) to match.
Timeout	Interval after which the rule is expired (default: No timeout).
Priority	Rule priority; higher priority rules take precedence (default: 0).
Mod	Modification Specification (mandatory for *Modify* rules).
RedirectTo	Port Specification (mandatory for *Redirect* rules).
Location	String description of Rule. (optional)
Entities:	
Address Entity: *IP Address*	Specifies an IP address; traffic from and to address is matched.
MAC Entity: *Mac Address*	Specifies a MAC address; traffic from and to address is matched.
Connection Entity: *Source IP, Source Port, Destination IP, Destination Port*	Specifies a bi-directional connection by its 5-tuple.
Flow Entity: *Src. & Dest. Network, Src. & Dest. Port, Src. & Dest. MAC*	Specifies an uni-directional flow; wildcards allowed for all fields.
Modification Specification:	
4-tuple	Modify any or all of source, destination IP and port. (optional)
Source & Dest. MAC	Modify source and/or destination MAC. (optional)
Output port	Specify output port (optional).

rule), *redirect* (steer entity to a different output port), and *modify* (change content of entity's packets). The network control framework supports five types of entities: unidirectional *flows*, bi-directional *connections*, *IP addresses*, *network prefixes*, and layer-2 *MAC addresses*. Rules specify flows/connections through their 5-tuples, with support for wildcards as tuple elements. For IPs, prefixes, and MACs, a rule always matches any traffic involving that entity, regardless of direction. In addition to the three mandatory *type*, *entity*, and *target* attributes, Table 1 shows further options that rules support, including: *Priority* resolves conflicts if multiple rules match the same traffic; *Modify* augments modify actions by defining the change to perform; and *RedirectTo* specifies the target port. Internally, the network control framework converts all the high-level operations from Sect. 3.3 into such rules. For example, consider *shunt_flow*. In this case, the network control framework creates a rule as follows, dropping all traffic for the specified 5-tuple on the monitoring path:

```
Rule(Type=Drop, Entity=Flow([5-tuple]), Target=Monitor)
```

Implementing *quarantine* is more complex, with four separate rules:

```
Rule(Type=Drop, Entity=Flow(SrcIp=[Infected]), Target=Forward)

Rule(Type=Redirect, Priority=1,
    Entity=Flow(SrcIP=[Infected], DstIp=[DNS server], DstPort=53/udp),
    Modify(DestIp=[Quarantine Srv]), Target=Forward)

Rule(Type=Redirect, Priority=1,
    Entity=Flow(SrcIp=[Quarantine Srv], SrcPort=53/udp, DstIp=[Infected]),
    Modify(SrcIp=[DNS Server]), Target=Forward)
```

```
Rule(Type=Whitelist, Priority=1,
    Entity=Flow(SrcIp=[Host], DstIp=[QuarantineHost], DstPort=80/tcp),
    Target=Forward)
```

The first rule blocks all traffic from the infected host using the default priority of 0. The second and third higher priority rules modify *(i)* DNS requests from the quarantined host to go to the dedicated quarantine server, and *(ii)* DNS server responses to appear as coming from the original server. The fourth rule permits port 80 requests from the infected host to the quarantine server.

3.5 Adaptability to Networks

The network control framework's support for multiple backends enables pushing out a rule to the device most appropriate for putting it into effect. Consider, for example, an environment with several switches, each responsible for a specific IP subnet. One can add each of them to the network control framework by instantiating a corresponding backend, configuring each to only accept rules for the switch's IP range. When installing a rule, the network control framework will iterate through the backends until reaching the appropriate switch, which will signal that it can handle it. As another example, an environment could block traffic by deploying a combination of a firewall and a router. The router could drop individual IP addresses efficiently using its hardware lookup tables, but might not be able to match on other fields like TCP/UDP ports and would hence reject corresponding requests. The less efficient firewall would then provide a fall-back accepting all other rules.

Backend priorities allow to fine-tune the selection of backends further. When instantiating multiple backends with the same priority, the network control framework will install a rule through *all* of them. This supports, e.g., the case of multiple border routers connecting to different upstream providers: blocking IPs should take effect on all of their links. Shunting on the monitoring path provides an example for using different priorities. Generally, shunting should happen as close to the tap as possible. As a fallback, however, in case the closest switch does not support the necessary drop rule, one can always have the local IDS host itself filter out the traffic at the NIC level; that way the packets at least do not reach the IDS' processing. To support this scenario, one would instantiate a high priority backend for the switch, and a low priority backend for kernel-level filtering on the IDS system.

3.6 Unifying State Management

The network control framework installs control rules dynamically as the IDS identifies corresponding patterns. Typically, such rules remain valid only for moderate periods of time, from a few minutes to hours. Afterwards, they need to expire so that network behavior reverts back to normal and resources on the devices free up. The network control framework thus supports timeouts as an intrinsic part of its architecture; all operations and rules include them. Internally,

however, handling timeouts requires managing the corresponding state, which poses a challenge. While some backends can support rule expiration directly through device mechanisms (e.g., OpenFlow can time out rules), not all have that capability (e.g., acld). For backends without corresponding support, the network control framework includes a software implementation as an optional service that a backend can leverage. If activated, the network control framework tracks the backend's active rules with their expiration times, sending it explicit removal requests at the appropriate time. Even if a device supports rule expiration in hardware, a backend might still chose to rely on the network control framework's implementation instead if the device's expiration semantics do not align with the network control framework's API requirements. In either case, from the user's perspective the network control framework reports a notification when a rule expires, including—if the backend supports it—more detailed information about traffic it has matched during its lifetime. Generally, hardware switches track such metrics and the network control framework passes it along.

State management introduces a challenge when either the IDS or a device restarts, as generally that means the system will loose any rules it has installed. On the IDS side, one can conceptually solve that rather easily by having the system retain state persistently across restarts, either through serialization at termination time or by directly maintaining the information in a on-disk database. Once the system is back up again, it can then timeout any rules that have expired during the downtime, sending removal commands to their backends. On the device side, handling restarts proves more challenging. One approach would be replaying all the rules from IDS memory. That however could impose significant load on the device (e.g., imagine reinstalling thousands of IP address blocks). It would also require actually *recognizing* that a device has restarted, a task that turns out difficult to perform for some backends (e.g., OpenFlow switches do generally not signal restarts explicitly). Therefore, the network control framework accepts that rebooting a switch means that it will loose all its rules; the framework will continue operation as if nothing had happened. In practice the impact of this approach remains low, as due to the dynamic nature of rules in our use cases, their lifetime tends to remain short anyways. For rules that target individual flows, chances are the session will have terminated already when the device is back up. Even for long-lived flows, reverting back to normal operation occasionally usually proves fine (e.g., when shunting, load will increase back to the full level briefly). For more general rules, the higher-level analysis can often compensate for the rare case of a device restart by retriggering the original action. For example, when dropping with catch-and-release (see Sect. 2), the IDS will immediately reblock the offender on its next connection attempt. Internally, if the network control framework manages rule expiration it will eventually still send removals for rules that no longer exist after a device restart, which however the backend can ignore.

4 Implementation

We implement the design of the network control framework on top of the open-source Bro Network Security Monitor [5,17]. Bro provides an apt platform for

active response as its event-based, Turing-complete scripting language facilitates complex custom policies taking decisions. Furthermore, Bro allows us to implement the network control framework fully inside this language as well, whereas other IDS would typically require integration at a lower level.

4.1 User Interface

The network control framework's user interface consists of a new script-level Bro framework that provides script writers with an interface closely following the design we present in Sect. 3, exposing both the high-level operations as well as the low-level primitives to their custom logic. In the following, we examine two real-world examples of how a Bro user can leverage the network control framework to react to network activity the system observes.

First, consider the case of a high-volume supercomputing environment aiming to shunt all GridFTP [2] data flows, thereby lessening the load on their Bro setup. In this case, as Bro already includes the capability to identify GridFTP transfers, one can hook the network control framework's high-level *shunt_flow* function, contained in the *NetControl* namespace to Bro's corresponding event by writing a handler like this:

```
event GridFTP::data_channel_detected(c: connection) {
    NetControl::shunt_flow([$src_h=c$id$orig_h, $src_p=c$id$orig_p,
                        $dst_h=c$id$resp_h, $resp_p=c$id$resp_p], 1hr);
}
```

Second, assume we want to block the IP addresses of hosts performing a port or address scan. For that, we hook into Bro's alarm reporting ("notices"):

```
event log_notice(n: Notice::Info) {
  if ( n$note == Address_Scan || n$note == Port_Scan )
    NetControl::drop_address(n$src, 10min);
}
```

Inserting low-level rules likewise closely follows the design from Sect. 3, mapping the rule attributes from Table 1 to corresponding Bro data types. For example, the following shows the actual implementation of the `shunt_flow` operation in the Bro scripting language. For the most part, the function just converts Bro's data structures into the format that the network control framework expects:

```
function shunt_flow(f: flow_id, t: interval) : string {
  local flow = Flow(
      $src_h=addr_to_subnet(f$src_h), $src_p=f$src_p,
      $dst_h=addr_to_subnet(f$dst_h), $dst_p=f$dst_p
      );
  local e: Entity = [$ty=FLOW, $flow=flow];
  local r: Rule = [$ty=DROP, $target=MONITOR, $entity=e, $expire=t];
  return add_rule(r);
 }
```

Since the actual rule operations will execute asynchronously, the network control framework uses Bro events to signal success or failure, as well for reporting a rule's removal along with the corresponding statistics (see Sect. 3.6).

4.2 Adding Backends

As discussed in Sects. 3.2 and 3.5, the network control framework supports multiple backends simultaneously with different priorities. In our Bro implementation, one adds backends at initialization time through corresponding script code:

```
local backend = NetControl::create_backend_Foo([...]);
NetControl::activate(backend, 10);
```

The `create_plugin_Foo` function is part of the backend's implementation and receives any arguments that it requires, for example the IP address and port of a switch to connect to. `activate` then adds the newly minted instance to the network control framework, specifying its priority as well (10 in this example).

The network control framework deploys a plugin model for implementing new backends, making it easy to augment it with support for further devices. Each backend plugin has to implement three functions for *(i)* instantiating a backend of that type, *(ii)* adding rules, and *(iii)* removing rules. Instantation returns an instance of a Bro data type describing the backend with its functions and features (e.g., if the plugin can handle rule expiration itself). Both the add and removal functions receive the backend instance along with the rule as their parameters. The add function returns a boolean indicating if the backend could execute the rule.

5 Backends

In this section we present the different types of backends that our implementation of the network control framework currently supports through plugins that we have implemented: OpenFlow in Sect. 5.1, *acld* in Sect. 5.2, Bro's built-in packet filter in Sect. 5.3, and finally a generic command-line interface in Sect. 5.4.

5.1 OpenFlow

OpenFlow [13] is an SDN protocol that allows applications to control the forwarding plane of a switch (or router) by inserting or removing rules. As switches with OpenFlow support have become both common and affordable, the protocol provides an ideal target for the network control framework to support a range of devices across vendor boundaries. We added OpenFlow support to our implementation in two steps: *(i)* we created a separate abstraction inside Bro, an *OpenFlow module*, that exposes OpenFlow's primitives to Bro scripts through a corresponding API; and *(ii)* we wrote an OpenFlow backend plugin for the network control framework that uses the OpenFlow module for interfacing to

OpenFlow devices. We chose to separate the two, as OpenFlow support may prove useful for applications beyond the network control framework as well.

In a OpenFlow deployment, applications typically do not talk to devices directly, but instead interface to an *OpenFlow controller* that serves as the middle-man. The controller is the component that speaks the actual OpenFlow protocol with the switch ("southbound"), while exposing an external API (e.g., a REST interface) to clients ("northbound"). Unfortunately, there is no standardized northbound interface; depending on the choice of a controller, the mechanisms differ. For our case study, we leveraged the Ryu SDN Framework [19].

Ryu enables creating custom controllers in Python, fully supporting versions OpenFlow 1.0 to 1.3. We leveraged the Ryu API to write an OpenFlow controller interfacing Ryu to Bro's communication protocol, using the *Broker* messaging library [6]. On the Bro side, the OpenFlow module maps OpenFlow messages into corresponding Broker messages, essentially creating our own communication mechanism between the two systems.[1] Figure 2 summarizes the full architecture when using the network control framework with OpenFlow: Messages pass from the network control framework's OpenFlow backend into the OpenFlow module, which in turn sends them over to the controller via Broker. Results travel the same way in reverse.

As an additional feature, the OpenFlow backend supports callback functions that can inspect and modify any OpenFlow messages it generates before passing them on. This allows to, e.g., use fields that OpenFlow supports yet have no equivalent inside the framework (e.g., input ports, or VLAN priorities).

OpenFlow's lack of success messages posed a particular implementation challenge for the backend. Generally, OpenFlow does not acknowledge rules that were successfully installed; it only reports error cases. With the network control framework that proves problematic, as its approach to iterate through all backends make it important to confirm an action's execution. One solution would be to just assume that a rule was successfully applied after a certain amount of

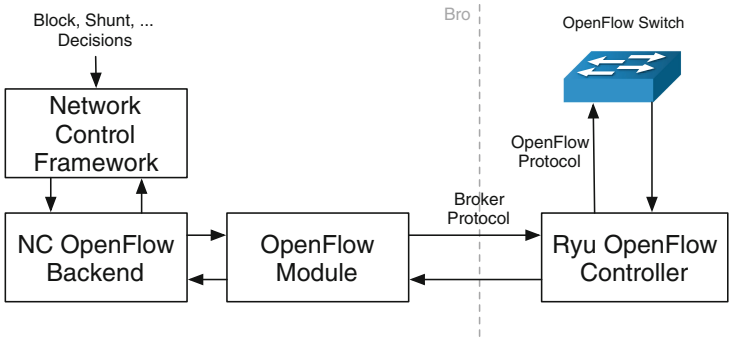

Fig. 2. OpenFlow architecture.

[1] Indeed, our Bro OpenFlow module remains independent of Ryu and could support other controllers as well if one extended them with a similar receiver component.

time has passed with no error message. However, this would require choosing a conservative timeout and hence significantly delay the success signal back to the network control framework, contrary to our objective of keeping latencies low. We instead solved this challenge by using OpenFlow's *barrier messages*. When a switch receives a *barrier request*, it will answer with a *barrier reply*, yet only after it has fully processed all preceeding OpenFlow messages. Hence, once the plugin receives a barrier reply, it knows that any operations that have not prompted an explicit error message so far, must have succeeded.

The OpenFlow backend assumes that it can insert rules without conflicting with other applications talking to the same controller and/or switch. In practice, one can typically resolve potential conflicts between applications by associating priorities with the OpenFlow rules, which the backend supports. More generally, Porras et al. [18] present an enhanced controller that mediates conflicts.

5.2 Acld

Acld [1] is a Unix daemon that acts as a middle-man for blocking IP addresses, address pairs, ports, and more; interfacing to a range of hard- and software switches and firewalls, including from Cisco, Juniper, and Force10, as well as BSD ipfw. Acld is, e.g., used by LBNL.

We created an Acld backend for the network control framework that compiles rules into acld's command syntax and then sends them over to the daemon for execution, using again Bro's communication library Broker to implement that communication.[2] Since the actions that acld supports are more limited than the rules that the network control framework can express, the backend checks first if a rule has an acld equivalent, declining it otherwise. Acld does not support rule expiry itself. Instead, the network control framework keeps track of all its rules and removes them automatically after their timeout period.

5.3 IDS Packet Filter

We also provide a backend that targets Bro itself. Bro provides an internal packet filter that allows excluding traffic from further processing early in its processing pipeline. Doing so removes the CPU overhead associated with that traffic, including in particular stream reassembly and protocol analysis. As Papadogiannakis et al. [16] demonstrate, such early filtering can significantly improve IDS performance. We implemented a network control framework backend plugin that emits rules for this Bro-internal packet filter, enabling the network control framework to execute rules for the monitoring path on the IDS system itself. Usually, this backend will represent a fall-back option: if another backend can filter the traffic earlier, that would be the better choice; but if that capability is not available, filtering late remains better than not all.

[2] Currently, Bro talks to an intermediary Python script, which in turn relays commands to acld through TCP. We plan to integrate Broker into acld directly in the future.

5.4 Generic Command-Line Interface

As a generic backend, we implemented a command-line interface that allows users to specify shell commands to execute for installing and removing rules, making it easy to support network components that come with command-line clients. As an example, we used this to implement network control framework support for Linux iptables. Our iptables implementation uses Broker again, similar to the Ryu OpenFlow interface. In this case, we implemented a Broker backend for the network control framework itself, which passes the low-level network control framework data structures to a Broker endpoint outside of Bro. We then implemented a Python script that receives these Broker messages and executes custom shell commands that the user specifies through a YAML configuration file. To pass parameters to these commands (e.g., IP addresses), the Python script replaces a set of predefined macros with the corresponding values from the network control framework rules. Each shell command executes inside a separate thread so that even rapid sequences of rules do not lead to delays.[3] For Linux iptables, we use the following command-line for blocking an IP address:

```
iptables -A INPUT [?address:-s . ][?proto:-p . ][?conn.orig_h:-s . ]
  [?conn.orig_p: --sport . ][?flow.src_h: -s . ][?flow.src_p: --sport .]
  [?conn.resp_h:-d . ][?conn.resp_p: --dport . ][?flow.dst_h: -d . ]
  [?flow.dst_p: --dport . ] -j DROP
```

Here, the macro syntax tells the Python script to replace each pair of brackets with either an appropriate command line option if the corresponding network control framework attribute is defined, or just an empty string if not. The entry to remove a rule works accordingly.

6 Evaluation

In this section we evaluate functionality and performance of the network control framework on the basis of the use cases we discuss in Sect. 2. We use the network control framework's OpenFlow backend for all experiments and measurements.

6.1 Functionality

We implemented all the use cases we discuss in Sect. 2—dynamic firewalling, shunting, quarantining, and QoS—in a variety of lab setups in different environments. For these experiments, we connected the network control framework to three different OpenFlow-capable hardware switches: an IBM G8052 (firmware version 7.11.2.0), an HP A5500-24 G-4SFP (Comware version 5.20.99), and an Pica8 Open vSwitch P-3930 (PicOs 2.5.2). In each case, we validated correct operation through manually generating corresponding traffic and confirming that the switches indeed had installed the anticipated OpenFlow rules. We conclude that our network control framework generally indeed operates as expected.

[3] As this could potentially reorder rules, users can optionally disable threading.

During our testing, we however noticed a number of differences between the OpenFlow implementations of the three switches. Most importantly, while all the switches offer OpenFlow 1.3, they differ in the feature set they support. For example, the HP A5500 only supports one output target per rule, making it impossible to duplicate traffic from one input port to two target ports—generally a desirable capability for network monitoring setups.[4] Both the IBM G8052 and the Pica8 P-3930 support this operation. Neither the IBM nor the HP switch can modify IP-level information (e.g., IP addresses or ports), preventing the network control framework's corresponding modifications from working with them. The Pica8 switch provides this functionality. Finally, the size of the switches' flow tables differ across the three devices—yet with all of them remaining rather small: the HP switch offers the largest table, yet still only supports two times 3,072 distinct entries.

6.2 Performance

In terms of performance, we examine two scenarios: the latency of blocking attacks and malicious content as well as the effectiveness of shunting traffic.

Filtering. As our first scenario, we examine the latency of blocking attacks and malicious content. When adding block rules, the main operational concern is the speed with which it takes effect; the delay between the decision and implementation should be as small as possible.

To test this scenario, we examined one hour of connection logs representing all external port 80 traffic on the Internet uplinks of the University of California at Berkeley (UCB). The upstream connectivity consists of two 10GE links with a daytime average rate of about 9 Gb/s total. During that hour, there were 9,392,623 established HTTP connections. To generate a test-load for automatic blocking, we pretended that every thousandth HTTP connection was carrying malicious content and thus had to be blocked, turning into an average of 2.6 network control framework operations per second. This level is quite a bit higher than what even large environments encounter in practice. Consulting with the operations team at LBNL, their system blocked, e.g., an average of 269 and 308 IPs per hour on May 28th and June 1st respectively. In their most active hour during those days, they blocked 616 IPs, i.e., 0.17 per second. At Indiana University, 23,875 blocks executed in total during May 2015, corresponding to 0.009 per second. Our testing workload exceeds these rates significantly, hence putting more strain on the setup than currently found in operational settings.

By extracting from the connection logs the timestamps and originator IP addresses of all "malicious" connections, we generated execution traces of network control framework operations matching what a live Bro would have executed during that hour of traffic. Replaying these traces through Bro triggers the actual OpenFlow messages with correct timing in a repeatable way. We

[4] While the lack of this feature does not affect the network control framework directly, it could prevent using it in combination with further static monitoring rules.

performed two measurements with this replay approach: *(i)* blocking all future traffic *from* the offending IP addresses, and *(ii)* blocking all future traffic *from or to* those addresses; the latter requires two OpenFlow rule insertions, doubling their frequency to an average of 5.2 per second.

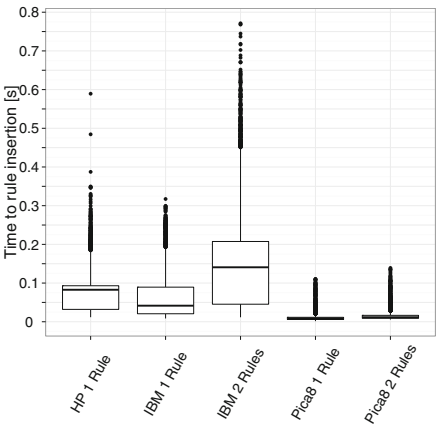

Fig. 3. Box plot of rule insertion latency with uni- and bi-directional rules for different OpenFlow hardware switches.

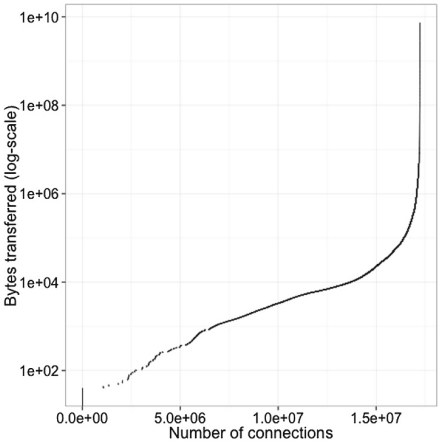

Fig. 4. TCP connection sizes at UCB (2015-05-07, 15:00-16:00) (Bytes axis log-scaled).

Figure 3 shows the delays from the moment Bro decided to insert a rule to the time when the success notification arrived back for most combinations of the three switches and the two measurements.[5] For all combinations shown, the rules took less than a second to insert, with the mean values being much lower. For example, for the IBM G8052, the median latency for bi-directional blocks was 141 ms; 42 ms for uni-directional. The Pica8 P-3039 showed the most impressive results with median times of 11 ms and 8.5 ms, respectively. For comparison, LBNL's operations team currently reports latencies around 300 ms on average when performing blocks with their home-built solution, i.e., more than an order of magnitude more than the combination of the network control framework with a (good) OpenFlow switch.

These results demonstrate that the network control framework supports high-speed rule execution even at levels substantially exceeding what operational environments currently require. Our measurements however also show the impact of hardware choices on blocking efficiency.

As a second evaluation, we determine how much data can typically still go through during the "gap time", meaning the period between when an IDS determines that a connection should be blocked, and the time when the network

[5] We excluded the HP A5500 with bi-directional rules due to problematic behavior when inserting rules that way: The average latency was >10 s, with many rules timing out and never getting an acknowledgment from the switch.

implements that decision. This is the additional latency that passive monitoring incurs in comparison to an inline IPS, with which blocks take effect instantaneously.

For this scenario, we assume a detector that inspects the header of HTTP requests to determine whether to deem them malicious, issuing blocks at the end of the header at the latest. We capture a 1 h packet trace of port 80 connections at the uplink of UCB, taken on 2015-05-01 at 16:00-17:00. Due to the fact that the traffic volume at UCB is more than a single machine can handle, we record a representative subset. Specifically, we use $\frac{1}{28}$ of all flows, corresponding to a slice of traffic that the organization's load-balancer setup sends to one of their in total 28 backend Bro machines. The resulting trace contains 159,474 HTTP connections.

We replay the trace file using a custom version of Bro, which we instrumented to output for each HTTP session *(i)* the packet timestamp of when the header was completed; *(ii)* the remaining duration of that connection from that point onwards; *(iii)* and the number of bytes transferred during intervals of I ms after completion of the header, with values for I chosen to correspond to the latencies we measured above for installing rules into the OpenFlow switches.

First, we measure how many connections terminate before they can be blocked using an uni-directional block with the different switches, assuming their block-time is either within the median or the 75 % percentile.

Table 2 shows the results of this evaluation. The table shows the median and 75 % block-speeds for the different switches. Assuming these values, we evaluate *(i)* how many connections terminate before a block can be installed, and *(ii)* what the median, mean and maximum amount of bytes are that could be transferred over the connection before the block was engaged.

These results show that, with the right hardware, the network control framework incurs latencies small enough that it would indeed have been able to stop most connections before their completion.

Shunting. As a second scenario, we examined the effectiveness of shunting traffic with the network control framework, using again network traffic from UCB's Internet uplink. This time, we examined flow logs of one hour of all TCP connections during the same peak traffic time as in Sect. 6.2. During that hour, the link saw 17,238,227 TCP connections, with a maximum volume of 7.5 GB and a total volume of 2.1 TB.

Table 2. Block times, connections that were not blocked in time, median, mean and maximum bytes transferred before block was engaged for OpenFlow switches.

Switch	Block time	Not blocked	Med. transferred	Mean transferred	Max transferred
Pica8 (Med)	8.5 ms	4,229 (2.7 %)	0	1.6 k	68 k
Pica8 (75P)	11 ms	8,273 (5.1 %)	12	2.3 k	101 k
IBM (Med)	41 ms	27,848 (17.4 %)	194	9.5 k	1.1 MB
IBM (75P)	89 ms	41,965 (26.3 %)	526	27 k	4.0 MB
HP (Med)	82 ms	38,381 (24 %)	454	23 k	4.5 MB
HP (75P)	93 ms	43,128 (27 %)	537	28 k	5.0 MB

Figure 4 plots the distribution of connection sizes, with the x-axis showing the number of connections and the y-axis their volume in log-scale.[6] We find the connection sizes highly heavy-tailed, with a small number of connections making up the bulk of the data. The mean connection size is 123 KB, the median is 2 KB.

Looking at the connections in more detail, there are 106 connections transferring more than 1 GB of data, making up 12 % of the total traffic; 1999 with more than 100 MB (36 %); and 24,106 with more than 10 MB (65 %). Assuming that we instructed the network switch to shunt each connection after reaching 1000, 100 or 10 MB respectively, we would shunt 53 %, 26 % or 6.5 % of the total TCP data transferred over the network link.

As this evaluation shows, traffic shunting can be effective even outside of scientific lab environments with their strong emphasis on bulk transfers. The university network we examine here exhibits a highly diverse traffic mix, with typical end-user traffic contributing most of the activity. Still, shunting would provide significant load reduction. Our implementation of the network control framework makes this easy to setup and control through just a few lines of Bro script code.

7 Related Work

There is a substantial body of academic work evaluating the interplay of network monitoring and software defined networking in different ways. The original OpenFlow paper [13] already suggests that applications might want to process individual packets instead of operating at the flow-level, as the OpenFlow API exposes it. Xing et al. [25] implement a prototype system using Snort to analyze packets via an OpenFlow controller. A this incurs significant computational cost, the authors use their system only up to a few thousand packets per second.

Shirali-Shareza et al. [22] examine the problem of controllers not being suitable to access packet-level information from the network. They propose an OpenFlow sampling extension, which allows the switch to only send a subset of a flow's packets to the controller. However, this approach is not suitable for use with network monitoring systems that rely on seeing the full packet stream for, e.g., TCP reassembly. Braga et al. [4] implement a lightweight DDOS flooding attack detector by regularly querying a network of OpenFlow controllers for flow information. They do not inspect raw packet contents. Van Adrichem et al. [24] present a system using OpenFlow to calculate the throughput of each data flow through the network over time by querying OpenFlow switches in variable intervals. Their results are within a few percent of direct traffic observation.

Slightly related to our work, Ballard et al. [3] present a language and system for traffic redirection for security monitoring at line rate. They implement a language to define how traffic should flow through the network as well as the system that applies the rules in an OpenFlow-capable network. Snortsam [23] is a plugin for the Snort IDS, allowing automated blocking of IP addresses on

[6] The connections reporting a size of 0 were not fully established.

a number of different hard- and software firewalls and routers. In comparison to our approach, Snortsam remains more limited, only allowing the blocking of source/destination IP addresses or single connections. SciPass [10] is an OpenFlow controller application designed to help scaling network security to 100 G networks. It supports using OpenFlow switches for load-balancing to IDS systems as well as traffic shunting. For the purpose of our paper, an application like SciPass could become another backend, just like our OpenFlow interface, and thus complement our design.

Porras et al. [18] present an enhanced OpenFlow controller mediating conflicting rules that independent applications might insert; an approach that one could use in conjunction with the network control framework's OpenFlow backend. Gonzalez et al. [11] introduce shunting as a hardware primitive in the context of an inline FPGA device with a direct interface to an IDS. Campbell et al. [7] evaluate its effectiveness inside 100 G scientific environments, using a simulation driven by Bro connection logs. The network control framework facilitates transparent operational deployment of this powerful capability. Related to shunting, Maier et al's Time Machine [12] leverages the heavy-tailed nature of traffic for optimizing bulk storage.

8 Conclusion

In this paper we present the design and implementation of a *network control framework*, a novel architecture enabling passive network monitoring systems to actively control network components, such as switches and firewalls. Our design provides a set of high-level operations for common functionality directly, while also offering access to lower-level primitives to perform custom tasks. As one of its key features, the framework supports controlling multiple network devices simultaneously, installing each rule at the component most appropriate to carry it out.

We assess the feasibility of our design by implementing the framework on top of the open-source Bro Network Security Monitor, and assess its functionality and performance through an OpenFlow backend connecting to three hardware switches in realistic settings. We find that the network control framework supports workloads beyond what even large-scale environments currently require. Going forward, we consider this framework a key abstraction for providing more dynamic security response capabilities than operators have available today. We anticipate that, in particular, the largest of today's network environments—with links of 100 G, and soon beyond—will benefit from the framework's capabilities in settings that no inline IPS can support.

Acknowledgments. We would like to thank Aashish Sharma, Keith Lehigh, and Paul Wefel for their feedback and help.

This work was supported by the National Science Foundation under grant numbers ACI-1348077 and CNS-1228792. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

References

1. ACL blocker notes. http://www-nrg.ee.lbl.gov/leres/acl2.html
2. Allcock, W., Bester, J., Bresnahan, J., Chervenak, A., Liming, L., Tuecke, S.: GridFTP: Protocol Extensions for the Grid. Grid ForumGFD-R-P.020 (2003)
3. Ballard, J.R., Rae, I., Akella, A.: Extensible and scalable network monitoring using OpenSAFE. In: INM/WREN (2010)
4. Braga, R., Mota, E., Passito, A.: Lightweight DDoS flooding attack detection using NOX/OpenFlow. In: LCN (2010)
5. Bro Network Monitoring System. https://www.bro.org
6. Broker: Bro's Messaging Library. https://github.com/bro/broker
7. Campbell, S., Lee, J.: Prototyping a 100g monitoring system. In: PDP (2012)
8. Presentation slides–Anonymized for submission (2014)
9. ESnet: Science DMZ Security - Firewalls vs. Router ACLs. https://fasterdata.es.net/science-dmz/science-dmz-security/
10. GlobalNOC: SciPass: IDS Load Balancer & Science DMZ. http://globalnoc.iu.edu/sdn/scipass.html
11. Gonzalez, J., Paxson, V., Weaver, N.: Shunting: a hardware/software architecture for flexible, high-performance network intrusion prevention. In: ACM Communications and Computer Security (CCS) Conference, Washington, D.C (2007)
12. Maier, G., Sommer, R., Dreger, H., Feldmann, A., Paxson, V., Schneider, F.: Enriching network security analysis with time travel. In: Proceedings of the ACM SIGCOMM (2008). http://www.icir.org/robin/papers/sigcomm08-tm.pdf
13. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: OpenFlow: enabling innovation in campus networks. CCR **38**(2), 69–74 (2008)
14. Network Control framework and utility code. http://icir.org/johanna/netcontrol
15. OSCARS: On-Demand Secure Circuits and Advance Reservation System. http://www.es.net/engineering-services/oscars/
16. Papadogiannakis, A., Polychronakis, M., Markatos, E.P.: Improving the accuracy of network intrusion detection systems under load using selective packet discarding. In: EUROSEC (2010)
17. Paxson, V.: Bro: a system for detecting network intruders in real-time. Comput. Netw. **31**(23–24), 2435–2463 (1999)
18. Porras, P., Cheung, S., Fong, M., Skinner, K., Yegneswaran, V.: Securing the software-defined network control layer. In: Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS), February 2015
19. Ryu SDN Framework. http://osrg.github.io/ryu/
20. Science DMZ - A Scalable Network Design Model for Optimizing Science Data Transfers. https://fasterdata.es.net/science-dmz
21. Security and NAT Gateway for the Munich Scientific Network (MWN). https://www.lrz.de/services/netzdienste/secomat_en/
22. Shirali-Shahreza, S., Ganjali, Y.: FleXam: flexible sampling extension for monitoring and security applications in openflow. In: HotSDN (2013)
23. Snortsam - A Firewall Blocking Agent for Snort. https://www.snortsam.net
24. Van Adrichem, N., Doerr, C., Kuipers, F.: OpenNetMon: network monitoring in OpenFlow software-defined networks. In: NOMS (2014)
25. Xing, T., Huang, D., Xu, L., Chung, C.J., Khatkar, P.: SnortFlow: a OpenFlow-based intrusion prevention system in cloud environment. In: GREE (2013)