To appear in the ISOC Network and Distributed System Security Symposium, February 2013.

FRESCO: Modular Composable Security Services for Software-Defined Networks

Seugwon Shin[1], Phillip Porras[2], Vinod Yegneswaran[2], Martin Fong[2], Guofei Gu[1], Mabry Tyson[2]

(1) Texas A&M University
College Station, TX 77840
seungwon.shin@neo.tamu.edu
guofei@cse.tamu.edu

(2) SRI International
Menlo Park, CA 94025
{porras,vinod,mwfong}@csl.sri.com
mabry.tyson@sri.com

Abstract

OpenFlow is an open standard that has gained tremendous interest in the last few years within the network community. It is an embodiment of the software-defined networking paradigm, in which higher-level flow routing decisions are derived from a control layer that, unlike classic network switch implementations, is separated from the data handling layer. The central attraction to this paradigm is that by decoupling the control logic from the closed and proprietary implementations of traditional network switch infrastructure, researchers can more easily design and distribute innovative flow handling and network control algorithms. Indeed, we also believe that OpenFlow can, in time, prove to be one of the more impactful technologies to drive a variety of innovations in network security. OpenFlow could offer a dramatic simplification to the way we design and integrate complex network security applications into large networks. However, to date there remains a stark paucity of compelling OpenFlow security applications. In this paper, we introduce FRESCO*, an OpenFlow security application development framework designed to facilitate the rapid design, and modular composition of OF-enabled detection and mitigation modules.* FRESCO*, which is itself an OpenFlow application, offers a Click-inspired [19] programming framework that enables security researchers to implement, share, and compose together, many different security detection and mitigation modules. We demonstrate the utility of* FRESCO *through the implementation of several well-known security defenses as OpenFlow security services, and use them to examine various performance and efficiency aspects of our proposed framework.*

1 Introduction

OpenFlow (OF) networks distinguish themselves from legacy network infrastructures by dramatically rethinking the relationship between the data and control planes of the network device. OpenFlow embraces the paradigm of highly programmable switch infrastructures [23], enabling software to compute an optimal flow routing decision on demand. For modern networks, which must increasingly deal with host virtualization and dynamic application migration, OpenFlow may offer the agility needed to handle dynamic network orchestration beyond that which traditional networks can achieve.

For an OpenFlow switch, the data plane is made programmable, where flows are dynamically specified within a *flow table*. The flow table contains a set of *flow rules*, which specify how the data plane should process all active network flows. In short, OpenFlow's flow rules provide the basic instructions that govern how to forward, modify, or drop each packet that traverses the OF-enabled switch. The switch's control plane is simplified to support the OpenFlow protocol, which allows the switch to communicate statistics and new flow requests to an external OpenFlow network *controller*. In return, it receives flow rules that extend its flow table ruleset.

An OF controller is situated above a set of OF-enabled switches, often on lower-cost commodity hardware. It is the coordination point for the network's flow rule production logic, providing necessary flow rule updates to the switch, either in response to new flow requests or to reprogram the switch when conditions change. As a controller may communicate with multiple OF switches simultaneously, it can distribute a set of coordinated flow rules across the switches to direct routing or optimize tunneling in a way that may dramatically improve the efficiency of traffic flows. The controller also provides an API to enable one to develop

OpenFlow applications, which implement the logic needed to formulate new flow rules. It is this application layer that is our central focus.

From a network security perspective, OpenFlow offers researchers with an unprecedented singular point of control over the network flow routing decisions across the data planes of all OF-enabled network components. Using OpenFlow, an *OF security app* can implement much more complex logic than simplifying halting or forwarding a flow. Such applications can incorporate stateful flow rule production logic to implement complex quarantine procedures, or malicious connection migration functions that can redirect malicious network flows in ways not easily perceived by the flow participants. Flow-based security detection algorithms can also be redesigned as OF security apps, but implemented much more concisely and deployed more efficiently, as we illustrate in examples within this paper.

We introduce a new security application development framework called *FRESCO*. *FRESCO* is intended to address several key issues that can accelerate the composition of new OF-enabled security services. *FRESCO* exports a scripting API that enables security practitioners to code security monitoring and threat detection logic as modular libraries. These modular libraries represent the elementary processing units in *FRESCO*, and may be shared and linked together to provide complex network defense applications.

FRESCO currently includes a library of 16 commonly reusable modules, which we intend to expand over time. Ideally, more sophisticated security modules can be built by connecting basic *FRESCO modules*. Each *FRESCO* module includes five interfaces: (i) input, (ii) output, (iii) event, (iv) parameter, and (v) action. By simply assigning values to each interface and connecting necessary modules, a *FRESCO* developer can replicate a range of essential security functions, such as firewalls, scan detectors, attack deflectors, or IDS detection logic.

FRESCO modules can also produce flow rules, and thus provide an efficient means to implement security directives to counter threats that may be reported by other *FRESCO* detection modules. Our *FRESCO* modules incorporate several security functions ranging from simple address blocking to complex flow redirection procedures (dynamic quarantine, or reflecting remote scanners into a honeynet, and so on). *FRESCO* also incorporates an API that allows existing DPI-based legacy security tools (e.g., BotHunter [12]) to invoke *FRESCO's* countermeasure modules. Through this API, we can construct an efficient countermeasure application, which monitors security alerts from a range of legacy IDS and anti-malware applications and triggers the appropriate *FRESCO* response module to reprogram the data planes of all switches in the OpenFlow network.

Contributions. In summary, our primary contribution is the introduction of *FRESCO*, which simplifies the development and deployment of complex security services for OpenFlow networks. To this end, we describe

- *FRESCO*: a new application development framework to assist researchers in prototyping new composable security services in OF-enabled networks. *FRESCO* scripts can be defined in a manner agnostic to OF controller implementation or switch hardware (an important feature given the rapid evolution of the protocol standard).

- A collection of OpenFlow security mitigation directives (*FRESCO* modules) and APIs to enable legacy applications to trigger these modules. Using *FRESCO*, security projects could integrate alarms from legacy network security DPI-based applications as inputs to *FRESCO* detection scripts or as triggers that invoke *FRESCO* response scripts that generate new flow rules.

- Several exemplar security applications demonstrate both threat detection and mitigation in an OpenFlow network, including scan detectors [16, 35, 15] and BotMiner [11]. We further show that existing security applications can be easily created with the use of *FRESCO*. For example, our *FRESCO* implementations demonstrate over 90% reduction in lines of code when compared to standard implementations and recently published OpenFlow implementations [24].

- A performance evaluation of *FRESCO*, which shows promise in developing OpenFlow security services that introduce minimal overhead for use in live network environments.

2 Background and Motivation

Our intent is to design an application framework that enables the modular design of complex OF-enabled network security services, which can be built from smaller sharable libraries of security functions. Before presenting *FRESCO*'s design, we first review some of the challenges that motivate the features of our framework.

2.1 The Information Deficiency Challenge

OpenFlow controllers do not uniformly capture and store TCP session information, among other key state tracking data, which is often required to develop security functionality (e.g., TCP connection status, IP reputation). We call this an information deficiency challenge. The *FRESCO* architecture incorporates a database module (*FRESCO*-DB) that simplifies storage and management of session state shared across applications. *FRESCO* also exports a high-level API in the *FRESCO* language that abstracts away complexities relating to switch management and specific controller implementations. This abstraction is a critical feature to enable module sharing across OpenFlow network instances that may vary in controller and OpenFlow protocol version.

2.2 The Security Service Composition Challenge

The *FRESCO* framework incorporates a modular and composable design architecture, inspired by the Click router

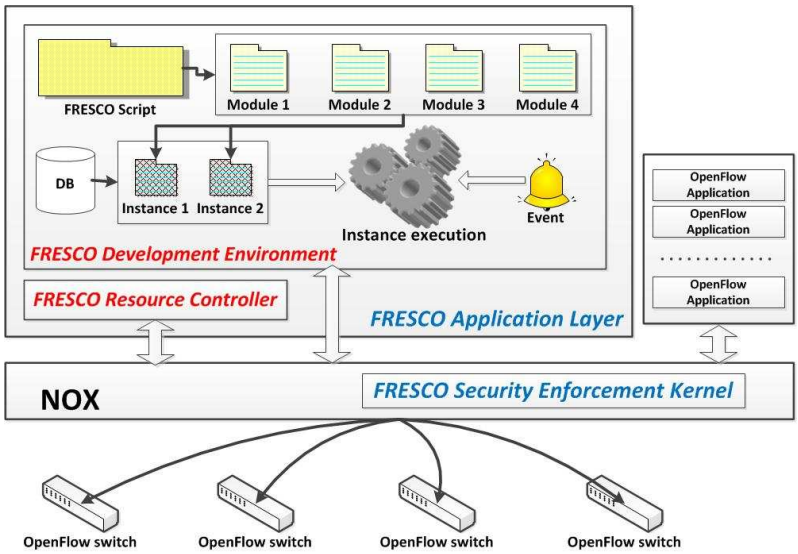

Figure 1. High-level overview of the *FRESCO* architecture.

architecture [19], which fosters rapid and collaborative development of applications through module composition. For example, a security module design to recognize certain traffic patterns that may represent a threat should be easily linkable to a variety of potential threat mitigation modules that, when triggered by the detection module, produce appropriate flow rule responses. *FRESCO* incorporates a scripting language that enables the linking of modules through data sharing and event triggering. Another important challenge is the need to provide an API that can facilitate flow rule production decisions using information produced from legacy DPI-based security applications (such as IDS or anti-malware applications).

2.3 The Threat Response Translation Challenge

The OpenFlow protocol enables the controlling software layer to communicate flow handling instructions to the data plane. However, while network security technologies do indeed produce threat alerts applicable to responses for individual flows, these technologies also have a need to express more complex (even stateful) security response directives that may span many flow rules, or even address network-wide attack scenarios. We call this the threat response translation challenge.

For example, one may wish to define a notion of host quarantine, in which all flows from an infected internal machine are blocked, with the exception that the machine's web traffic should be redirected to a web server that returns quarantine notification pages to the machine's user. One might also wish to define redirection directives that will silently redirect flows from a detected hostile external entity away from an internal production network and into a honeynet for analysis. One might even want to produce a network-wide response to shun malicious traffic, or alternatively, incorporate high-priority flow rules to ensure that emergency administrative flows succeed during a DOS attack.

Such security directives may require a complex set of flow rule production logic, which is also ideally sharable as a countermeasure library that could be coupled with many different detection algorithms.

3 *FRESCO* DESIGN

The *FRESCO* framework consists of an application layer (which provides an interpreter and APIs to support composable application development) and a security enforcement kernel (SEK, which enforces the policy actions from developed security applications), as illustrated in Figure 1. Both components are integrated into NOX, an open-source openflow controller.

FRESCO's application layer is implemented using NOX python modules, which are extended through FRESCO's APIs to provide two key developer functions: (i) a *FRESCO* Development Environment [**DE**], and (ii) a Resource Controller [**RC**], which provides FRESCO application developers with OF switch- and controller-agnostic access to network flow events and statistics.

Developers use the *FRESCO script* language to instantiate and define the interactions between the NOX python security modules (we present *FRESCO's* scripting language in Section 4.3). These scripts invoke *FRESCO*-internal modules, which are instantiated to form a security application that is driven by the input specified via the FRESCO scripts (e.g., TCP session and network state information) and accessed via *FRESCO's* DE database API. These instantiated modules are triggered (executed) by *FRESCO* DE as the triggering input events are received. FRESCO modules may also produce new flow rules, such as in response to

a perceived security threat, which are then processed by the controller's security enforcement kernel [**SEK**] (Section 5).

4 *FRESCO* Application Layer

The basic operating unit in the *FRESCO* framework is called a *module*. A module is the most important element of *FRESCO*. All security functions running on *FRESCO* are realized through an assemblage of modules. Modules are defined as Python objects that include five interface types: (i) *input*, (ii) *output*, (iii) *parameter*, (iv) *action*, and (v) *event*. As their names imply, *input* and *output* represent the interfaces that receive and transmit values for the module. A *parameter* is used to define the module's configuration or initialization values. A module can also define an *action* to implement a specific operation on network packets or flows. An *event* is used to notify a module when it is time to perform an action.

A module is implemented as an event-driven processing function. A security function can be realized by a single module or may be composed into a directed graph of processing to implement more complex security services. For example, if a user desires to build a naive *port comparator* application whose function is to drop all HTTP packets, this function can be realized by combining two modules. The first module has *input*, *output*, *parameter*, and *event*. The *input* of the first module is the destination port value of a packet, its *parameter* is the integer value 80, an *event* is triggered whenever a new flow arrives, and *output* is the result of comparing the *input* destination port value and parameter value 80. We pass the *output* results of the first module as *input* of the second module and we assign drop and forward *actions* to the second module. In addition, the second module performs its function whenever it is pushed as an *input*. Hence, the *event* of this module is set to be *push*. A module diagram and modules representing this example scenario are shown in Figure 2.

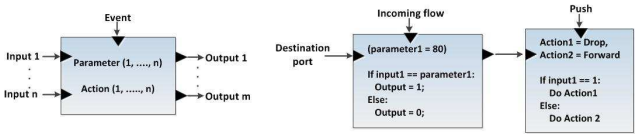

Figure 2. Illustration of *FRESCO* module design (left: model diagram; right: naive port comparator application)

An *action* is an operation to handle network packets (or flows). The actions provided by *FRESCO* derive from the actions supported by the NOX OpenFlow controller. The OpenFlow standard specifies three required actions, which should be supported by all OpenFlow network switches, and four optional actions, which might be supported by OpenFlow network switches [29]. OpenFlow requires support for three basic actions: (i) *drop*, which drops a packet, (ii) *output*, which forwards a packet to a defined port (in this paper, we sometimes use the term *forward* to denote the output action), and (iii) *group*, which processes a packet through the specified group. As these actions must be supported by all OpenFlow network switches, *FRESCO* also exports them to higher-level applications.

One optional action of interest is the *set action*, which enables the switch to rewrite a matching packet's header fields (e.g., the source IP, destination port) to enable such features as flow path redirection. Because one of the primary goals of *FRESCO* is to simplify development of security functions, *FRESCO* handles possible issues related to the *set action* by breaking the *set action* into three more specific actions: *redirect*, *mirror*, and *quarantine*. Through the *redirect action*, an application can redirect network packets to a host without explicitly maintaining state and dealing with address translation. *FRESCO* offloads session management tasks from applications and automatically changes the source and destination IP address to handle redirects. The *mirror action* copies an incoming packet and forwards it to a mirror port for further analysis. The functionality may be used to send a packet to a feature or other packet analysis systems. The *quarantine action* isolates a flow from the network. Quarantine does not mean dropping a particular flow, but rather, *FRESCO* attaches a tag to each packet to denote a suspicious (or malicious) packet. If a packet has the tag, then this packet can traverse only to allowed hosts (viz., a *FRESCO* script can fishbowl an infected host into an isolated network using packet tags).

4.1 *FRESCO* Development Environment

The *FRESCO development environment (DE)* provides security researchers with useful information and tools to synthesize security controls. To realize this goal, we design the *FRESCO* DE with two considerations. First, this environment must export an API that allows the developer to detect threats and assert flow constraints while abstracting the NOX implementation and OF protocol complexities. Second, the component must relieve applications from the need to perform redundant data collection and management tasks that are common across network security applications. The *FRESCO* development environment provides four main functions: (i) script-to-module translation, (ii) database management, (iii) event management, and (iv) instance execution.

Script-to-module translation: This function automatically translates *FRESCO* scripts to modules, and creates instances from modules, thus abstracting the implementation complexities of producing OF controller extensions. It is also responsible for validating the registration of modules. Registration is performed via a registration API, which enables an authorized administrator to generate a *FRESCO* application ID and an encryption key pair. The developer embeds the registered application ID into the *FRESCO* script, and then encrypts the script with the supplied private key. The naming convention of *FRESCO* applications incorporates the application ID, which is then used by *FRESCO* to

associate the appropriate public key with the application. In addition to registering modules, the module manager coordinates how modules are connected to each other and delivers input and event values to each module.

Database management: The DB manager collects various kinds of network and switch state information, and provides an interface for an instance to use the information. It provides its own storage mechanism that we call the *FRESCO-DataBase (F-DB)*, which enables one to share state information across modules. For example, if an instance wants to monitor the number of transferred packets by an OpenFlow enabled switch, it can simply request the F-DB for this information. In addition, this database can be used to temporarily store an instance.

Event management: The event manager notifies an instance about the occurrence of predefined events. It checks whether the registered events are triggered, and if so delivers these events to an instance. *FRESCO* supports many different kinds of events, including flow arrivals, denied connections, and session resets. In addition, the event manager exposes an API that enables event reporting from legacy DPI-based security applications, such as Snort [38] or BotHunter [12]. The security community has developed a rich set of network-based threat monitoring services, and the event manager's API enables one to tigger instances that incorporate flow rule response logic. [1]

Instance execution: This function loads the created instances into memory to be run over the *FRESCO* framework. During load time, *FRESCO* decrypts the application using the associated public key, and confirms that the ID embedded in the script corresponds to the appropriate public key. The application then operates with the authority granted to this application ID at registration time.

4.2 *FRESCO* Resource Controller

The *FRESCO* resource controller monitors OpenFlow network switches and keeps track of their status. A flow rule that is distributed from a *FRESCO* application is inserted into a flow table in an OpenFlow switch. Because the flow table has a limit on the number of entries it can hold, it is possible that a flow rule from a *FRESCO* application cannot be inserted into the flow table. However, because flow rules from a *FRESCO* application deal with security policy enforcement, such flow rules require immediate installation into the flow table of an OpenFlow network switch. Thus, *FRESCO* may forcibly evict some old or stale flow rules, both *FRESCO* and non-*FRESCO*, from the switch flow table to make space for new flow rules. This operation is done by the resource controller. Garbage collecting inactive *FRESCO* rules does not compromise the network security policy: if a prohibited flow is re-attempted later, the *FRESCO* SEK will prevent other OF applications from performing the flow setup.

The resource controller performs two main functions. The first function, which we call the *switch monitor*, periodically collects switch status information, such as the number of empty flow entries, and stores the collected information in the switch status table. The second component, i.e., the *garbage collection*, checks the switch status table to monitor whether the flow table in an OpenFlow switch is nearing capacity. If the availability of a flow table becomes lower than a threshold value (θ), the garbage collector identifies and evicts the least active flow, using least frequently used (LFU) as *FRESCO*'s default policy.

4.3 *FRESCO* Script Language

To simplify development of security applications, *FRESCO* provides its own script language to assist developers in composing security functions from elementary modules. The textual language, modeled after the Click language [19], requires the definition of six different variables per instance of modular element: (i) `type`, (ii) `input`, (iii) `output`, (iv) `parameter`, (v) `action`, and (vi) `event`.

To configure modules through a *FRESCO* script, developers must first create an instance of a module, and this instance information is defined in *type* variable. For example, to use a function that performs a specific action, a developer can create an instance of the ActionHandler module (denoted as `type:ActionHandler` within a *FRESCO* script).

Developers can specify a script's input and output, and register events for it to process by defining the script's *input*, *output*, *parameter*, and *event* variables. Multiple value sets for these variables (e.g., specifying two data inputs to `input`) may be defined by using a comma as the field separator.

Defining an instance is very similar to defining a function in C or C++. A module starts with the module name, two variables for representing the number of inputs and outputs, and left braces (i.e., {). The numbers of inputs and outputs are used to sanity check the script during module translation. Like C or C++ functions, a module definition ends with a right brace (i.e., }).

The `action` variable represents actions that a module will perform based on some conditions, where the conditions are determined by one of the *input* items. There may be multiple conditions in the `action`, which are separated by semicolons. We summarize these variables in Table 1, and Figure 3 shows example scripts of the port comparator application shown in Figure 2 (right) with two connecting modules.

***FRESCO* Script Execution:** We use a simple running example, shown in Figures 3 and 4, to illustrate the execution of a FRESCO script. First, an administrator composes a FRESCO script (shown in Figure 3) (1), and loads it into *FRESCO* (2). Next, when Host A sends a packet to port 80 of Host B through an OpenFlow switch, as illustrated in Figure 4 (3), this packet delivery event is reported to the FRESCO DE (4). The *FRESCO* DE creates instances from modules defined in the *FRESCO* script (i.e., port_comparator instance from comparator module and

[1] The example case for this scenario is shown in section 6.

Variable	*Explanation*	*Possible Values*
instance name (#input)(#output)	denotes an instance name (should be unique)	(#input) and (#output) denote the number of inputs and outputs
type: [module]	denotes a module for this instance	[module] names an existing module
input: a_1,a_2,...	denotes input items for a module	a_n may be set of flows, packets or integer values
output: b_1, b_2,...	denotes output items for a module	b_n may be set of flows, packets or integer values
parameter: c_1,c_2,...	denotes configuration values of a module	c_n may be real numbers or strings
event: d_1,d_2,...	denotes events delivered to a module	d_n may be any predefined string
action : condition ? action,...	denotes set of conditions and actions performed in the module	condition follows the same syntax of *if condition* of python language; action may be one of the following strings (DROP, FORWARD, REDIRECT, MIRROR, QUARANTINE)
{ }	denotes the module start ({) and end (})	-

Table 1. Key variables in the *FRESCO* scripting language

```
port_comparator (1)(1) {
   type:Comparator
   event:INCOMING_FLOW
   input:destination_port
   output:comparison_result
   parameter:80
/* no actions are defined */
   action: -
}
```

```
do_action (1)(0) {
   type:ActionHandler
   event:PUSH
   input:comparison_result
   output: - /* no outputs are defined */
   parameter: - /* no parameters are defined */
/* if input equals to 1, drop, otherwise, forward */
   action:comparator_result == 1 ? DROP : FORWARD
}
```

Figure 3. *FRESCO* script with two connecting modules used to build the naive port comparator

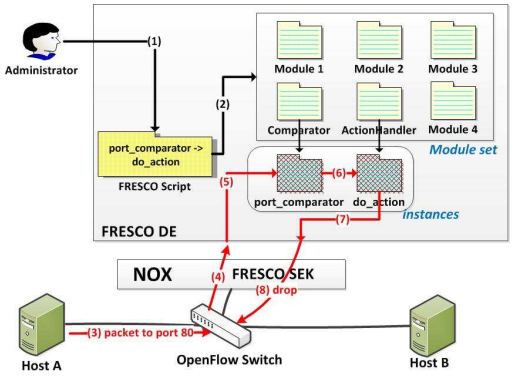

Figure 4. Operational illustration of running *FRESCO* script (case of the *FRESCO* script shown in Figure 3)

do_action instance from ActionHandler module) and dynamically loads them. The *FRESCO* DE runs each instance (5, 6), and when it receives an action from the do_action module (i.e., drop) (7), it translates this action into flow rules, which can be understood by an OpenFlow switch. Finally, these flow rules are installed into the switch through the *FRESCO* SEK (8).

5 *FRESCO* Security Enforcement Kernel

Security applications developed in *FRESCO* scripts can enforce diverse security policies, such as DROP, REDIRECT, QUARANTINE, to react to network threats by simply setting an *action* variable, as listed in Table 1. These high-level security policies can help developers focus on implementing security applications, and these policies will be automatically translated into flow rules for OpenFlow enabled switches by *FRESCO* DE (e.g., the REDIRECT action will be translated into three flow rules). Thus, developers do not need to care about network-level flow rules.

However, when *FRESCO* DE enforces translated flow rules to switches, it will face a new challenge, which stems from the fact that OpenFlow provides no inherent mechanisms to reconcile rule conflicts as competing OpenFlow applications assert new rules into a switch. For example, a set of rules designed to quarantine an internal computing server (i.e., the QUARANTINE action in a *FRESCO* script) might subsequently be overridden by a load-balancing application that may determine that the quarantined host is now the least-loaded server. One needs a mechanism to ensure that flow rules produced by a security application will take precedence over those produced from non-security-aware applications. OpenFlow also incorporates a packet alteration functions (i.e., the `set` action), specifiable within its flow rule format. This feature enables virtual tunnels between hosts, such that a virtual tunnel can be used to circumvent a flow rule that was inserted to prevent two hosts from establishing a connection.

To address this issue, *FRESCO* incorporates a security enforcement kernel (SEK), which is integrated directly into the OpenFlow controller upon which FRESCO operates. A more complete discussion of *FRESCO* SEK is provided in a published workshop paper [31]. *FRESCO* SEK offers several important features upon which FRESCO relies to ensure that flow rules derived from security services are prioritized and enforced over competing flow rules produced by non-security-critical applications:

- Rule source identification: The SEK introduces a trust model that allows FRESCO applications to digitally

sign each candidate flow rule, thus enabling the SEK to determine if a candidate flow rule was produced by a FRESCO security module, by an OpenFlow application, or by a network administrator.

- Rule conflict detection: To detect conflicts between a candidate rule set and the set of rules currently active in the switch, the SEK incorporates an inline rule conflict analysis algorithm called *alias set rule reduction*, which detects flow rule conflicts, including those that arise through set actions that are used to produce virtual tunnels. Since this is not the main focus of this paper, we include a relatively more detailed description of our rule conflict detection algorithm in Appendix A.1 for interested readers.

- Conflict resolution: When a conflict arises, the SEK applies a hierarchical authority model that enables a candidate rule to override (replace) an existing flow rule when the digital signature of the rule source is deemed to possess more authority than the source whose rule is in conflict.

6 Working Examples

We show two case studies by creating real working security applications written in *FRESCO* script.

6.1 Implementing Reflector Net

FRESCO's power stems from its ability to use OpenFlow to effectively reprogram the underlying network infrastructure to defend the network against an emerging threat. To illustrate this notion, consider a *FRESCO* application that allows OF network operators to redirect malicious scanners to a third-party remote honeypot. Using *FRESCO* script, we compose two modules that first detect an active malicious scanner, and then reprogram the switch data plane to redirect all the scanner's flow into a remote honeynet. We refer to our composed security service as a threat *reflector net*, which effectively turns the address space of any OpenFlow network that deploys this service into a contact surface for a remote high-interaction honeypot. The incentive for an operator to use such a service is that the forensic evidence collected by the honeypot can be shared back for the purpose of refining the operator's local *FRESCO*-based firewall.

First, we create and configure a simple threshold-based scan detector instance. Since *FRESCO* already provides a "ScanDetector" module, we can instantiate an instance from this module for selecting malicious external targets. For this example, let us assume that our scan analysis is triggered by an external entity producing large numbers of failed TCP connections. Thus, we establish TCP_CONNECTION_FAIL, which is captured in FRESCO's native DB service, as an input trigger event for our scan detection, which outputs a scan detection event when a threshold number of failed connections is observed.

Our FRESCO script instantiates the scan detection module using four key script variables: (i) input, (ii) output, (iii) parameter, and (iv) action. The input for this instance is a source IP address for a flow that causes TCP_CONNECTION_FAIL event. The parameter will determine a threshold value for a scan detection algorithm, and here, we set this value as 5 (i.e., if a source IP generates five failed TCP connections, we regard it as a scan attacker). The output is a source IP address and a scan detection result (noted as *scan_result*), which are delivered to the second instance as input variables. The action variable is not defined here, as the logic required to formulate and insert flow rules to incorporate duplex redirection is modularized into a second flow redirection instance. The *FRESCO* script for our flow redirection instance is shown in Figure 5 (left).

We configure a redirector instance to redirect flows from the malicious scanner to a honeynet (or forward benign flows). This function is an instance of *FRESCO's* "ActionHandler" module. This instance uses a PUSH event, which triggers the instance each time "find_scan" is outputted from the scan detection instance Finally, we need to define an action to redirect flows produced by scan attackers. Thus, we set the action variable of this instance as "scan_result == 1 ? REDIRECT : FORWARD", which indicates that if the input variable of scan_result equals 1 (denoting the scanner) this instance redirects all flows related to the source IP address. The FRESCO script for this instance is shown in Figure 5 (right).

We test this script in an OpenFlow simulation environment with Mininet [25], which is commonly used to emulate OpenFlow networks, to show its real operation. In this test, we created three hosts (scanner, target host, and honeynet) and an OpenFlow enabled switch. All three hosts are connected to the switch and able to initiate flows to each another.

As illustrated in Figure 6, the malicious scanner (10.0.0.2) tries to scan the host (10.0.0.4) using Nmap tool [28]. The scan packets are delivered through an OpenFlow switch **(1)**, where the switch then forwards the flow statistics to a *FRESCO* application (i.e., find_scan instance) through a controller. The find_scan instance determines that these packets are scan-related, and it sends the detection result to the do_redirect instance to instantiate flow rules to redirect these packets to our honeynet (10.0.0.3) **(2)**. At this time, the network configuration of the honeypot is different from the original scanned machine (10.0.0.4), which opens network port 445 while the honeypot opens network port 444. Then, the honeypot returns packets to the scanner as if it is the original target **(3)**. Finally, the scanner receives packet responses from the honeypot **(4)**, unaware that all of its flows are now redirected to and from the honeynet.

6.2 Cooperating with a Legacy Security Application

FRESCO provides an interface, which receives messages from legacy security applications, such as Snort [38] and BotHunter [12]. Usually, we use these network security ap-

```
find_scan (1)(2){
   type:ScanDetector
   event:TCP_CONNECTION_FAIL
   input:source_IP
   output:source_IP, scan_result
   parameter:5
/* no actions are defined */
   action: -
}
```

```
do_redirect (2)(0){
   type:ActionHandler
   event:PUSH
   input:source_IP, scan_result
   output: -
   parameter: -
/* if scan_result equals 1, redirect,
   otherwise, forward */
   action: scan_result == 1 ?
                  REDIRECT : FORWARD
}
```

Figure 5. *FRESCO* script with two connecting modules used to build a reflector net

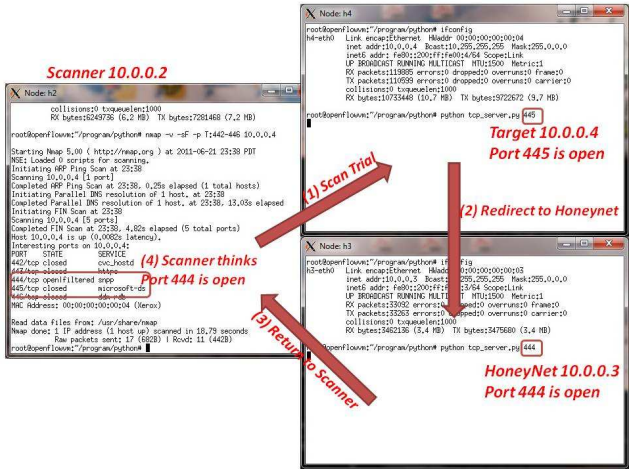

Figure 6. Operational illustration of a *FRESCO* reflector net application

plications to monitor our networks, often using DPI-based techniques to identify malicious traffic content or by simply monitoring flow patterns. Using *FRESCO*, alerts produced from such network security monitors can be integrated into the flow rule production logic of OF-enabled networks. To do this, we employ *FRESCO* actions (e.g., drop and redirect) as responses against network attacks.

One might consider reimplementing classic network security applications directly as OpenFlow applications. However, such efforts are both costly in engineering time and subject to limitations in the availability of data provided by the OF controller. Also, OpenFlow does not export full packet content over its APIs, so DPI-based security services must be implemented as external applications. To reduce the integration burden, *FRESCO* provides a function of receiving messages from third-party security applications, and we can simply design response strategies based on the messages through *FRESCO* script.

A message from a third-party security application will be delivered to a module as a type of event - MESSAGE_LEGACY, and the format of a message is of two kinds: (i) *FRESCO* type and (ii) other standardized formats such as the intrusion detection message exchange format (IDMEF) [33]. If we use FRESCO type, it is notified in the event as a keyword of FRESCO, and it can be represented as MESSAGE_LEGACY:FRESCO. If we use IDMEF, it can be shown as MESSAGE_LEGACY:IDMEF.

In the scenario, shown in Figure 7, an attacker sends a bot binary (1) to the host C, and BotHunter responds by producing an infection profile (2). Then, BotHunter reports this information (i.e., the Victim IP and forensic confidence score for the infection) to a security application written in *FRESCO* script (3). If the profile's forensic score achieves a threshold value, the application imposes a quarantine action on the victim IP. The quarantine module uses the *FRESCO* SEK to enforce a series of flow rules that implement the quarantine action SEK (4, 5). Finally, if an infected host (the host C) sends another malicious data to other hosts, such as host A or host B (6), it is automatically blocked by the switch.

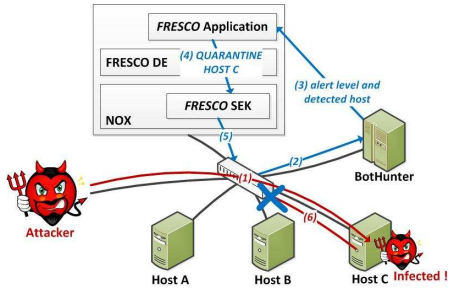

Figure 7. Operational illustration of a *FRESCO* actuator cooperating with BotHunter

To implement this function, we simply reconfigure the "do_quarantine" instance (or create another instance) used in the above example scenario for this case. This time, we instantiate the instance with four alternate variables: (i) event, (ii) input, (iii) parameter, and (iv) condition part of action. When BotHunter forwards its infection alarm using the *FRESCO* API, we set the event variable as MESSAGE_LEGACY:FRESCO. The input variables passed to this module include the victim_ip (reported as infected by BotHunter), and the infection confidence_score, which represents the degree of forensic evi-

dence recorded within the infection profile. We also specify a parameter for the confidence_threshold, and trigger the QUARANTINE action when the confidence_score exceeds the confidence_threshold. The *FRESCO* script for this instance is shown in Figure 8.

```
do_quarantine (2)(0){
   type:ActionHandler
   event:MESSAGE_LEGACY:FRESCO
   input:victim_ip,confidence_score
   output: -
   parameter:confidence_score
/* redirect all flows from source IP */
   action:confidence_score > confidence_threshold
          ? QUARANTINE(victim_ip)
}
```

Figure 8. *FRESCO* script for invoking host quarantine for BotHunter

```
1 def module_start(input_dic, param_list):
2    # initialize FRESCO native inputs
3    FR_FDB = input_dic['FR_FDB']
4    FR_event = in_dic['FR_event']
5    FR_input = input_dic['FR_input']
6
7    # initialize FRESCO variables
8    FR_ret_dic = {}
9    FR_ret_dic['output'] = []
10   FR_ret_dic['action'] = None
11
12   # start - user defined logic
13   if param_list[0] == FR_input[0]:
14      output = 1
15   else:
16      output = 0
17
18   FR_ret_dic['output'].append(output)
19   # end - user defined logic
20
21   return FR_ret_dic
```

Figure 9. *FRESCO* Comparator module

7 Implementation

We have developed a prototype implementation of the *FRESCO* architecture. The *FRESCO* Application Layer prototype is implemented in Python and runs as an OpenFlow application on NOX. The prototype operates on NOX version 0.5.0 using the OpenFlow 1.1.0 protocol, and is implemented in approximately 3,000 lines of Python. *FRESCO* modules are implemented as independent Python objects, and inputs and parameters of a module are input variables to the Python object. The return values of a Python object are considered as output values of a module.

A sample implementation of the *FRESCO* Comparator module, used in Figure 2 (right), is presented in Figure 9. All modules in *FRESCO* start with the function of module_start, and this function has two arguments: (i) `input_dic`, which is a dictionary data structure containing F-DB, event, and input values, and (ii) `param_list`, which is a list structure storing user-defined parameter values. All variables starting with "FR_" are *FRESCO* native variables. The developer fills in additional specialized logic at the bottom of the module (lines 13-18).

The *FRESCO* SEK is implemented as a native C++ extension of the NOX source code in approximately 1160 lines of C++ code. We modified the `send_openflow_command` function, whose main operation is to send OpenFlow commands to network switches, to capture flow rules from all OpenFlow applications. *FRESCO* SEK intercepts flow rules in the function and stores them into the security constraints table if the rules are from *FRESCO* applications (i.e., flow rules produced through the *FRESCO* path are considered trusted flow rules and are preserved as active network security constraints). If a flow rule is from a non-*FRESCO* application, *FRESCO* SEK evaluates the rule to determine if a conflict exists within its security constraints table. The match algorithm is specifically optimized to perform the least-expensive comparisons first. If there are conflicts, an error message is returned to the OF application. Otherwise, the rule is forwarded to the network switches. We implement and evaluate the security constraint table using the in-memory database opportunistic best-fit comparison algorithm, which reports an ability to execute queries in near-constant lookup time.

8 System Evaluation

We now evaluate the *FRESCO* framework with respect to its ease of use, flexibility, and security constraints preservation. To evaluate components in *FRESCO*, we employ *mininet* [25], which provides a rapid prototyping environment for the emulation of OpenFlow network switches. Using mininet, we have emulated one OpenFlow network switch, three hosts connected to the switch, and one host to operate our NOX controller. We perform flow generation by selecting one or two hosts to initiate TCP or UDP connections. The remaining host is employed as a medium interaction server, which responds to client application setup requests. We hosted our evaluation environment on an Intel i3 CPU with 4 GB of memory. In addition, we conduct live performance evaluations of the *FRESCO* SEK using an HP ProCurve 6600 OF-enabled switch in a test network laboratory.

8.1 Evaluating Modularity and Composability

For the evaluation, we begin with the basic problem of identifying entities performing flow patterns indicative of malicious network scanning, and compare schemes of implementing network scanning attacks with and without the use of *FRESCO*.

While network scanning is a well-studied problem in the network security realm, it offers an opportunity to examine the efficiency of entity tracking using *FRESCO*. Many well-

established algorithms for scan detection exist [16, 15, 35]. However, under OpenFlow, the potential for *FRESCO* to dynamically manipulate the switch's data path in reaction to malicious scans is a natural objective. This scenario also lets us examine how simple modules can be *composed* to perform data collection, evaluation, and response.

1. *FRESCO* Scan Deflector Service. Figure 10 illustrates how *FRESCO* modules and their connections can be linked together to implement a *malicious scan deflector* for OpenFlow environments. This scan detection function consists of the three modules described above. First, we have a module for looking up a blacklist. This module checks a blacklist table to learn whether or not an input source IP is listed. If the table contains the source IP, the module notifies its presence to the second module. Based on the input value, the second module performs threshold-based scan detection or it drops a packet. If it does not drop the packet, it notifies the detection result to the third module. In addition, this second module receives a parameter value that will be used to determine the threshold. Finally, the third module performs two actions based on input. If the input is 1, the module redirects a packet. If the input is 0, it forwards a packet. Implementing the three modules required 205 lines of Python code and 24 lines of *FRESCO* script (this script is shown in Figure 11).

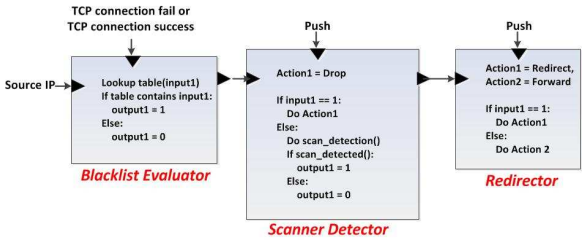

Figure 10. *FRESCO* **composition of a Scan Deflector**

2. *FRESCO* BotMiner Service. To illustrate a more complex flow analysis module using *FRESCO*, we have implemented a *FRESCO* version of the *BotMiner* [11] application. Note that our goal here is not faithful, "bug-compatible" adherence to the full BotMiner protocol described in [11], but rather to demonstrate feasibility and to capture the essence of its implementation through *FRESCO*, in a manner that is slightly simplified for readability.

BotMiner detects bots through network-level flow analysis. We have implemented the essentials of its detection functionality using five modules as shown in Figure 12. BotMiner assumes that hosts infected with the same botnet exhibit similar patterns at the network level, and these patterns are different from benign hosts. To find similar patterns between bots, BotMiner clusters botnet activity in two dimensions (C-plane and A-plane). The C-plane clustering

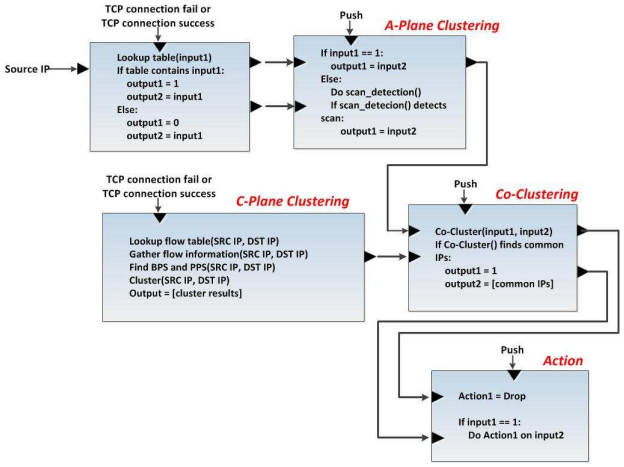

Figure 12. *FRESCO* **composition of the BotMiner service**

approach is used to detect hosts that resemble each other in terms of (packets per second) and bps (bytes per second). The A-plane clustering identifies hosts that produce similar network anomalies. In this implementation, we use the scan detector module to find network anomalies. Finally, if we find two clusters, we perform co-clustering to find common hosts that exist in both dimensions and label them as bots. BotMiner was implemented in 312 lines of python code and 40 lines of *FRESCO* script (the script for BotMiner is presented in Figure 13).

3. *FRESCO* P2P Plotter Service. We have implemented a *FRESCO*-based P2P malware detection service, similarly implemented to capture the concept of the algorithm, but simplified for the purpose of readability. Motivated by Yen's work [43], we have implemented the P2P malware detection algorithm, referred to as *P2P Plotter*, using *FRESCO*. The P2P Plotter asserts that P2P malware has two interesting characteristics, which are quite different from normal P2P client programs. First, P2P malware usually operates at lower volumes of network flow interactions than what is typically observed in benign P2P protocols. Second, P2P malware typically interacts with a peer population that has a lower churn rate (i.e., the connection duration time of P2P plotters is longer than that of normal P2P clients). The algorithm operates by performing co-clustering, to find common hosts that exhibit both characteristics (i.e., low volume and low churn rate).

We have implemented this essential functionality of the P2P Plotter algorithm as a 4-module *FRESCO* script, which is shown in Figure 14. This involved 227 lines of Python code and 32 lines of *FRESCO* script. The script for the P2P Plotter is illustrated in Figure 15. The reuse of modules (i.e., `CrossCluster` and `ActionHandler`, from the BotMiner service implementation is noteworthy, high-

```
1 blacklist_check (1)(1){
2   type:TableLookup
3   event:TCP_CONNECTION_FAIL,
          TCP_CONNECTION_SUCCESS
4   input:source_IP
5   output:blacklist_out
6   parameter:-
7   action:-
8 }
```

```
1 find_scan (1)(1){
2   type:ScanDetector
3   event:PUSH
4   input:blacklist_out
5   output:scan_out
6   parameter:-
7   action:blacklist_out == 1
          ?DROP
8 }
```

```
1 do_action (1)(0){
2   type:ActionHandler
3   event:PUSH
4   input:scan_out
5   output:-
6   parameter:-
7   action:scan_out == 1
          ?REDIRECT:FORWARD
8 }
```

Figure 11. *FRESCO* **script for Scan Detector**

```
1 table_check (1)(2){
2   type:TableLookup
3   event:TCP_CONNECTION_FAIL,
          TCP_CONNECTION_SUCCESS
4   input:source_IP
5   output:table_out,source_IP
6   parameter:-
7   action:-
8 }
```

```
1 a_cluster (2)(1){
2   type:A-PlaneCluster
3   event:PUSH
4   input:table_out,source_IP
5   output:a_cls_out
6   parameter:-
7   action:-
8 }
-
```

```
1 c_cluster (0)(1){
2   type:C-PlaneCluster
3   event:TCP_CONNECTION_FAIL,
          TCP_CONNECTION_SUCCESS
4   input:-
5   output:c_cls_out
6   parameter:-
7   action:-
8 }
```

```
1 cr_cluster (2)(2){
2   type:CrossCluster
3   event:PUSH
4   input:a_cls_out,c_cls_out
5   output:cross_out,ip_list
6   parameter:-
7   action:-
8 }
-
```

```
1 do_action (2)(0){
2   type:ActionHandler
3   event:PUSH
4   input:cross_out,ip_list
5   output:-
6   parameter:-
7   action:cross_out == 1
          ?DROP(ip_list):FORWARD
8 }
```

Figure 13. *FRESCO* **scripts illustrating composition of the BotMiner service**

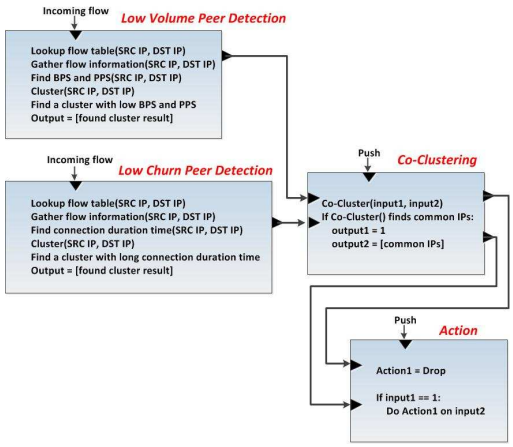

Figure 14. *FRESCO* **composition of the P2P Plotter**

lighting the reuse potential of *FRESCO* modules.

8.2 Comparing *FRESCO* Applications with Non-*FRESCO* Detectors

Network anomaly detection approaches, e.g., TRW [16], have been well-studied and are commonly used as a complement to signature-based detection systems in traditional networks. While these approaches may be instantiated as software programs or in hardware devices, the common practice is to implement them as stand-alone software programs. (We envision that the *FRESCO* development environment may be similarly used for rapid prototyping and evaluation of certain anomaly detection algorithms in OpenFlow networks.)

To highlight the advantages of *FRESCO*, we first choose an open-source network anomaly detection system and then replicate identical functionality using *FRESCO*. Specifically, we compare *FRESCO* with a recently published work [24], where the authors implemented popular network anomaly detection algorithms such as TRW-CB [34] and Rate Limit [39] as applications running on an OpenFlow network controller. We re-implement the same algorithms (i.e., TRW-CB and Rate Limit) using existing *FRESCO* modules and the *FRESCO* scripting language. We provide a comparison in Table 2, in terms of the number of lines of source code, to demonstrate the utility of the *FRESCO* development environment.

As summarized in Table 2, prior work [24] makes the case that its OpenFlow application implementation is slightly simpler than the standard implementation (i.e., the source code for the OpenFlow implementation is roughly 70% to 80% the length of the standard implementation). Using *FRESCO*, we are able to realize similar functionality with an order of magnitude fewer lines of code. That is, we have implemented the identical TRW-CB function with

```
1 low_volume_peer (0)(1){
2   type:VolumeDetector
3   event:INCOMING_FLOW
4   input:-
5   output:volume_out
6   parameter:-
7   action:-
8 }
```

```
1 low_churn_peer (0)(1){
2   type:ChurnDetector
3   event:INCOMING_FLOW
4   input:-
5   output:churn_out
6   parameter:-
7   action:-
8 }
```

```
1 cr_cluster (2)(2){
2   type:CrossCluster
3   event:PUSH
4   input:volume_out,churn_out
5   output:cross_out,ip_list
6   parameter:-
7   action:-
8 }
```

```
1 do_action (2)(0){
2   type:ActionHandler
3   event:PUSH
4   input:cross_out,ip_list
5   output:-
6   parameter:-
7   action:cross_out == 1 ? DROP(ip_list):FORWARD
8 }
```

Figure 15. *FRESCO* scripts illustrating composition of the P2P Plotter

66 lines of code (58 lines of Python and 8 lines of *FRESCO* script) and the rate limiting function with 69 lines of code (61 lines of Python and 8 lines of *FRESCO* script). These two examples represent 6% to 7% of the length of their standard implementations, and less than 9% of the recently published OpenFlow implementation.

Algorithms	Implementation		
	Standard	OpenFlow application	*FRESCO*
TRW-CB	1,060	741	66 (58 + 8)
Rate Limit	991	814	69 (61 + 8)

Table 2. Source code length for standard, OpenFlow and *FRESCO* implementations of the TRW-CB and Rate Limit anomaly detection algorithms

8.3 Measuring and Evaluating *FRESCO* Overhead

***FRESCO* Application Layer Overhead.** We compare the flow setup time of NOX flow generation with five other *FRESCO* applications and summarize the results in Table 3. To measure this, we capture packets between NOX and the OpenFlow switch, and measure the round trip required to submit the flow and receive a corresponding flow constraint. We observe that *FRESCO* applications require additional setup time in the range of 0.5 milliseconds to 10.9 milliseconds.[2]

	NOX	Simple Flow Tracker	Simple Scan Detector	Threshold Scan Detector	BotMiner Detector	P2P Plotter
Time (ms)	0.823	1.374	2.461	7.196	15.421	11.775

Table 3. Flow setup time comparison of NOX with five *FRESCO* applications

Resource Controller Overhead. The resource controller component monitors switch status frequently and removes old flow rules to reclaim space for new flow rules, which will be enforced by *FRESCO* applications. This job is performed by *FRESCO*'s garbage collector, a subcomponent of the resource controller, which we test under the following scenario. First, we let non-*FRESCO* applications enforce 4,000 flow rules to an OpenFlow network switch. In this case, we assume that the maximum size of the flow table in the switch is 4,000, and we set the threshold value(θ) for garbage collection as 0.75 (i.e., if the capacity of a flow table in a switch is $\leq$ 75%, we run the garbage collector). Our test results, shown in Figure 16, demonstrate that the garbage collector correctly implements its flow eviction policy.

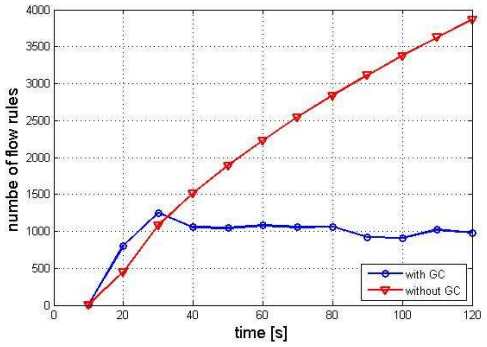

Figure 16. Operation of *FRESCO* garbage collector

9 Related Work

The OpenFlow standard has as its roots on a rich body of work on control-flow separation and clean-slate design of the Internet (e.g., [6], [10]). SANE [7] and Ethane [6] propose new architectures for securing enterprise networks. The SANE [7] protection layer proposes a fork-lift (clean-slate) approach for upgrading enterprise network security that introduces a centralized server, i.e., domain controller, to authenticate all elements in the network and grant access to services in the form of capabilities that are enforced at each switch. Ethane [6] is a more practical and backwards-

[2]These setup times were measured on mininet, which is an emulated environment running on a virtual machine. If we use a more powerful host for the controller, which is the common case in an OpenFlow network, this setup time will be reduced significantly.

compatible instantiation of SANE that requires no modification to end hosts. Both studies may be considered as catalysts for the emergence of OpenFlow and software-defined networking.

FRESCO is built over the foundations laid by these studies and shares a common objective with these systems in that it seeks to improve enterprise security using programmable network elements. However, *FRESCO* emphasizes composable security, and applications that it enables are much more sophisticated than simple access control policies. In addition, the *FRESCO* SEK focuses on providing continued enforcement of potentially conflicting flow constraints imposed by FRESCO and other OF applications. Thus, we consider our work as greatly complementary to existing studies such as SANE and Ethane.

FRESCO's focus is on the development of a holistic platform for specifying and developing OF security applications and enforcement of security constraints generated by these applications. Prior work has addressed a part of this problem, i.e., development of new languages for specifying security policies. Nettle [40] is a new language for managing OF switches that is based on functional reactive programming. Frenetic [9] and Procera [41] provide declarative query language frameworks for managing distributed OF switches, describing high-level packet-forwarding and specifying network policies. The OpenSAFE system provides a language framework for enabling redirection of traffic to network monitoring devices [3]. In contrast to these languages, the *FRESCO* development environment is specialized to serve the needs of security applications. Specifically, *FRESCO* applications issue high-level security directives (e.g., REDIRECT, QUARANTINE, MIRROR), which are then translated into OF-specific commands by the script-to-module translator. In addition, *FRESCO* applications require aggregate session and flow state information as well as directives for asynchronous delivery of switch state information that is unavailable in standard OF environments. Applications such as Random Host Mutation [14] are additional motivating examples of candidate OF security applications whose development may be accelerated using *FRESCO*.

The *FRESCO* security enforcement kernel is informed by prior research focused on testing or verifying firewall and network device configuration [36, 8, 21, 22, 2, 42, 1], e.g., using Firewall Decision Diagrams (FDDs) [21] or test case generators [36, 8]. These studies do not deal with dynamic networks. More recently, *header space analysis* was proposed, which is a generic framework to express various network misconfigurations and policy violations [17]. While HSA can in theory deal with dynamic networks, the *FRESCO* SEK differs in that it is specialized to deal with specific policy violations by OF applications, rule conflict detection, and dynamic flow tunneling. Veriflow proposes to slice the OF network into equivalence classes to efficiently check for invariant property violations [18]. The alias set rule reduction algorithm used by *FRESCO* SEK is complementary to this approach.

We build our system on NOX, which is an open-source OF controller [13]. However, our methodology could be extended to other architectures like Beacon [30], Maestro [4], and DevoFlow [26]. FlowVisor is a platform-independent OF controller that uses network slicing to separate logical network planes, allowing multiple researchers to run experiments safely and independently in the same production OpenFlow network [37]. Our work differs from FlowVisor in several ways. First, FlowVisor cares primarily about non-interference *across* different logical planes (slices) but does not instantiate network security constraints *within* a slice. It is possible that an OF application uses packet modification functions resulting in flow rules that are applied across multiple network switches within the same slice. In such cases, we need a security enforcement kernel to resolve conflicts as described in Section 5. Second, although FlowVisor improves security by separating the OF network into logical planes, it does not provide analogous capabilities to *FRESCO* for building additional security applications.

The need for better policy validation and enforcement mechanisms has been touched on by prior and concurrent research efforts. NICE provides a model-checking framework that uses symbolic execution for automating the testing of OpenFlow applications [5]. The Resonance architecture enables dynamic access control and monitoring in SDN environments [27]. The FlowChecker system encodes OpenFlow flow tables into Binary Decision Diagrams (BDD) and uses model checking [1] to verify security properties. However, the evaluation of FlowChecker does not consider handling of *set* action commands, which we consider to be a significant distinguisher for OpenFlow networks. More recently, researchers have proposed developing language abstractions to guarantee consistency of flow updates in software-defined networks [32]. In contrast, our complementary work on the *FRESCO* security enforcement kernel is focused on detection of rule update conflicts and security policy violations. The Onix platform [20] provides a generalized API for managing a distributed control plane in Software Defined Networks. The techniques and the strategies developed in Onix for managing a distributed network information base are complementary and can be integrated into *FRESCO*.

10 Conclusion

Despite the success of OpenFlow, developing and deploying complex OF security services remains a significant challenge. We present *FRESCO*, a new application development framework specifically designed to address this problem. We introduce the *FRESCO* architecture and its integration with the NOX OpenFlow controller, and present several illustrative security applications written in the *FRESCO* scripting language. To empower *FRESCO* applications with the ability to produce enforceable flow constraints that can defend the network as threats are detected, we present

the *FRESCO* security enforcement kernel. Our evaluations demonstrate that *FRESCO* introduces minimal overhead and that it enables rapid creation of popular security functions with significantly (over 90%) fewer lines of code. We believe that *FRESCO* offers a powerful new framework for prototyping and delivering innovative security applications into the rapidly evolving world of software-defined networks. We plan to release all developed code as open source software to the SDN community.

11 Acknowledgements

This material is based in part upon work supported by the Defense Advanced Research Projects Agency (DARPA) and the United States Air Force under Contract No. FA8750-11-C-0249, the Army Research Office under Cyber-TA Grant no. W911NF-06-1-0316, and the National Science Foundation under Grant no. CNS-0954096. All opinions, findings and conclusions or recommendations expressed herein are those of the author(s) and do not necessarily reflect the views of the U.S. Air Force, DARPA, U.S. Army Research Office, or the National Science Foundation. It is approved for Public Release, Distribution Unlimited.

References

[1] E. Al-Shaer and S. Al-Haj. FlowChecker: Configuration Analysis and Verification of Federated Openflow Infrastructures. In *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration*, SafeConfig, pages 37–44, New York, NY, USA, 2010. ACM.

[2] E. Al-shaer, W. Marrero, A. El-atawy, and K. Elbadawi. Network Configuration in A Box: Towards End-to-End Verification of Network Reachability and Security. In *The 17th IEEE International Conference on Network Protocols (ICNP)*, 2009.

[3] J. R. Ballard, I. Rae, and A. Akella. Extensible and Scalable Network Monitoring Using OpenSAFE. In *INM/WREN*, 2010.

[4] Z. Cai, A. L. Cox, and T. E. Ng. Maestro: A System for Scalable OpenFlow Control. In *Rice University Technical Report*, 2010.

[5] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *Proceedings of NSDI*, 2012.

[6] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *Proceedings of ACM SIGCOMM*, 2007.

[7] M. Casado, T. Garfinkel, M. Freedman, A. Akella, D. Boneh, N. McKeowon, and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. In *Proceedings Usenix Security Symposium*, August 2006.

[8] A. El-atawy, T. Samak, Z. Wali, E. Al-shaer, F. Lin, C. Pham, and S. Li. An Automated Framework for Validating Firewall Policy Enforcement. Technical report, De-Paul University, 2007.

[9] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ACM SIGPLAN Inernational Conference on Functional Programming*, 2011.

[10] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management. In *Proceedings of ACM Computer Communications Review*, 2005.

[11] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In *Proceedings of USENIX Security Symposium (Security'08)*, 2008.

[12] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting malware infection through ids-driven dialog correlation. In *Proceedings of the 16th USENIX Security Symposium (Security'07)*, August 2007.

[13] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. In *Proceedings of ACM SIGCOMM Computer Communication Review*, July 2008.

[14] J. H. Jafarian, E. Al-Shaer, and Q. Duan. OpenFlow Random Host Mutation: Transparent Moving Target Defense using Software-Defined Networking. In *Proceedings of ACM Sigcomm HotSDN Workshop*, 2012.

[15] J. Jung, R. Milito, and V. Paxson. On the Adaptive Real-time Detection of Fast Propagating Network Worms. In *Proceedings of Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2007.

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

[17] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *Proceedings of NSDI*, 2012.

[18] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Proceedings of ACM Sigcomm HotSDN Workshop*, 2012.

[19] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, August 2000.

[20] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *The Symposium on Operating Systems Design and Implementation (NSDI)*, 2010.

[21] A. Liu. Formal Verification of Firewall Policies. In *Proceedings of the 2008 IEEE International Conference on Communications (ICC)*, Beijing,China, May 2008.

[22] A. Liu and M. Gouda. Diverse Firewall Design. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 19(8), 2008.

[23] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. In *Proceedings of ACM SIGCOMM Computer Communication Review*, April 2008.

[24] S. A. Mehdi, J. Khalid, and S. A. Khayam. Revisiting Traffic Anomaly Detection Using Software Defined Networking. In *Proceedings of Recent Advances in Intrusion Detection*, 2011.

[25] Mininet. Rapid Prototyping for Software Defined Networks. `http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet/`.

[26] J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. R.

Curtis, and S. Banerjee. DevoFlow: Cost-effective Flow Management for High Performance Enterprise Networks. In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010.

[27] A. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: Dynamic Access Control for Enterprise Networks. In *Proceedings of WREN*, 2009.

[28] NMAP.ORG. Nmap: Open Source Network Discovery and Auditing Tool. `http://nmap.org`.

[29] OpenFlow. OpenFlow 1.1.0 Specification. `http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf`.

[30] OpenFlowHub. BEACON. `http://www.openflowhub.org/display/Beacon`.

[31] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A Security Enforcement Kernel for OpenFlow Networks. In *Proceedings ACM SIGCOMM Workshops on Hot Topics in Software Defined Networking (HotSDN)*, August 2012.

[32] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent Update for Software-Defined Networks: Change You Can Believe In! In *Under Submission*, 2011.

[33] RFC4765. The intrusion detection message exchange format (idmef). `http://www.ietf.org/rfc/rfc4765.txt`.

[34] S. Schechter, J. Jung, and A. Berger. Accuracy Improving Guidelines for Network Anomaly Detection Systems. In *Proceedings of International Symposium on Recent Advances Intrusion Detection*.

[35] V. Sekar, Y. Xie, M. Reiter, and H. Zhang. A Multi-Resolution Approach for Worm Detection and Containment. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, June 2006.

[36] D. Senn, D. Basin, and G. Caronni. Firewall Conformance Testing. In *The 17th IFIP International Conference on Testing of Communicating Systems (TestCom)*, pages 226–241, 2005.

[37] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the Production Network Be the Testbed. In *Proceedings of Operating System Design and Implementation*, 2010.

[38] Snort. `http://snort.org`.

[39] J. Twycross and M. M. Williamson. Implementing and testing a virus throttle. In *Proceedings of the USENIX Security Symposium*, 2003.

[40] A. Voellmy and P. Hudak. Nettle: Functional Reactive Programming of OpenFlow Networks. In *Yale University Technical Report*, 2010.

[41] A. Voellmy, J. Kim, and N. Feamster. Procera: A Language for High-Level Reactive Network Control. In *Proceedings of ACM Sigcomm HotSDN Workshop*, 2012.

[42] G. Xie, J. Zhan, D. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On Static Reachability Analysis of IP Networks. In *The 24th IEEE International Conference on Computer Communications (INFOCOM)*, pages 2170–2183, 2005.

[43] T.-F. Yen and M. K. Reiter. Are Your Hosts Trading or Plotting? Telling P2P File-sharing and Bots Apart. In *Proceedings of IEEE ICDCS*, 2010.

A Appendix

A.1 FRESCO Security Enforcement Kernel

It is possible that the flow rules created by non-security-related OF applications conflict with the flow constraints distributed by *FRESCO* applications. A conflict arises when one or more flow rules would allow a flow from one end point to another that is specifically prohibited by a flow constraint rule produced by a *FRESCO* application. To manage *FRESCO* flow constraints and perform conflict evaluation, we introduce the *FRESCO* SEK as an embedded NOX extension.

Two main components of OpenFlow rules are match conditions and actions. The former specifies packet header fields that must match for the rule's associated actions to trigger. The *FRESCO* SEK maintains the set of active constraint rules produced by registered *FRESCO* modules. Constraint rules inserted into the *FRESCO* SEK *security constraints table* are considered active, and must be explicitly deactivated by a registered *FRESCO* module. Because non-*FRESCO* applications can publish flow rules that potentially violate *FRESCO*'s network security policy, The *FRESCO* SEK employs two protection mechanisms to prevent such violations. The first mechanism is *rule prioritization*, in which flow rules produced by *FRESCO* applications are published to the switch using the highest rule priority. This immediately overrides any active flow rules in the switch's flow table that may contradict *FRESCO*'s security policy. Second, the *FRESCO* SEK applies a conflict detection algorithm between each new flow rule and the security constraints table, rejecting the new flow rule if a conflict is detected. Conflict detection is performed in two passes: alias set rule reduction, and then rule set conflict evaluation.

A conflict can also happen between security constraints enforced by different *FRESCO* applications. In this case, the *FRESCO* SEK can still detect conflicts but it needs to determine which constraint should be enforced. By default, *FRESCO* SEK keeps the first enforced constraint (i.e., ignore following conflicted constraint), but it is easy to be configured by the administrators to apply other approaches (e.g., keep the last enforced constraint, or based on some priority settings).

A.1.1 Alias Set Rule Reduction

To detect conflicts between a candidate rule set and *FRESCO*'s constraint rule sets, the source and destination IP addresses, their ports, and wild card members [3] for each rule in a rule set are used to derive rules with alias sets representing IP addresses and ports. The initial alias sets contain the first rule's IP addresses, network masks, and ports (where 0 [zero] represents any port). If the rule's action causes a field substitution via a *set action*, the resultant value is added to the appropriate alias set. These sets are

[3] For OpenFlow 1.1, the examined members include the source and destination network mask fields (for OpenFlow 1.0 these are implicitly defined by the wildcard field).

then compared to the next rule's alias sets. If there is an intersection between both the source and address sets, the union of the respective sets is used as the subsequent rule's alias sets. For example, given the *FRESCO* rule,

$$a \rightarrow \text{b drop packet} \tag{1}$$

its source alias set is (a), while its destination alias set is (b). The derived rule is

$$(a) \rightarrow \text{(b) drop packet} \tag{2}$$

For the candidate (evasion) rule set,

$$\begin{array}{l} 1\ \text{a} \rightarrow \text{c set (a} \Rightarrow \text{a')} \\ 2\ \text{a'} \rightarrow \text{c set (c} \Rightarrow \text{b)} \\ 3\ \text{a'} \rightarrow \text{b forward packet} \end{array} \tag{3}$$

the intermediate alias sets are

$$\begin{array}{l} 1\ \text{a} \rightarrow \text{c set (a} \Rightarrow \text{a')} \quad \text{(a, a') (c)} \\ 2\ \text{a'} \rightarrow \text{c set (c} \Rightarrow \text{b)} \quad \text{(a, a') (c, b)} \\ 3\ \text{a'} \rightarrow \text{b forward packet (a, a') (c, b) forward packet} \end{array} \tag{4}$$

and the derived rule is

$$\text{(a, a')} \Rightarrow \text{(c, b) forward packet} \tag{5}$$

A.1.2 Rule Set Conflict Evaluation

The *FRESCO* SEK first performs alias set rule reduction on the candidate rule set. These validity checks are then performed between each derived *FRESCO* constraint rule *cRule* and each derived flow rule *fRule*, as follows:

1. Skip any cRule/fRule pair with mismatched prototypes.
2. Skip any cRule/fRule pair whose actions are both either forward or drop packet.
3. If cRule's alias sets intersect those of fRule's, declare a conflict.

Thus, given the example security constraint table in Equation 2 and the candidate rule set in Equation 5, assuming that both rules are TCP protocol, the first candidate rule passes the first two checks. However, for the third check, because the intersection of the source and destination alias sets results in (a) and (b), respectively, the candidate rule is declared to be in conflict.

As a practical consideration, because OpenFlow rules permit both wildcard field matches and IP address network masks, determining alias set intersection involves more than simple membership equality checks. To accommodate this, we define comparison operators that determine if a field specification is (i) more encompassing ("wider"), (ii) more specific ("narrower"), (iii) equal, or (iv) unequal. Thus, an intersection occurs when the pairwise comparisons between all fields of a candidate rule are wider than, equal to, or narrower than that of the corresponding fields of the constraint table rule.

For a formalization of the above, we first define some terms: (i) S_i is the i_{th} entry of security constraints, (ii) F_i is the i_{th} entry of flow rules, (iii) $SC_{i,j}$ is the j_{th} item of the i_{th} entry of the condition part of the security constraint, (iv) SA_i is the i_{th} entry of the action part of the security constraint, (v) $FC_{i,j}$ is the j_{th} item of the i_{th} condition part of a flow rule from non-*FRESCO* applications, and (vi) FA_i is the i_{th} action part of the flow rule. At this time, both $SC_{i,j}$ and $FC_{i,j}$ are sets whose elements are one of the specific value or some ranges and $j \in \{1, 2, ..., 14\}$. Rule contradiction is then formalized using the following notation:

$$\begin{array}{l} \text{if there is any } S_i, \text{ satisfying } SC_{i,j} \cap FC_{i,j} \neq \oslash \text{ and} \\ SA_i \neq FA_i, \text{ for all } j, \text{then} F_i \text{ is conflicted with } S_i \end{array} \tag{6}$$

Finally, upon an update to the security constraints table, rule set conflict resolution is performed against all flow rules currently active within the switch. If a conflict is detected in which the switch rule is found to be wider than the *FRESCO* rule, SEK initiates a request to the switch to flush the resident rule.