Securing the Software-Defined Network Control Layer

Phillip Porras, Steven Cheung, Martin Fong, Keith Skinner, and Vinod Yegneswaran
Computer Science Laboratory, SRI International
333 Ravenswood Avenue, Menlo Park, CA 94025
{porras,cheung,mwfong,skinner,vinod}@csl.sri.com

***Abstract*—Software-defined networks (SDNs) pose both an opportunity and challenge to the network security community. The opportunity lies in the ability of SDN applications to express intelligent and agile threat mitigation logic against hostile flows, without the need for specialized inline hardware. However, the SDN community lacks a *secure control-layer* to manage the interactions between the application layer and the switch infrastructure (the data plane). There are no available SDN controllers that provide the key security features, trust models, and policy mediation logic, necessary to deploy multiple SDN applications into a highly sensitive computing environment. We propose the design of security extensions at the control layer to provide the security management and arbitration of conflicting flow rules that arise when multiple applications are deployed within the same network. We present a prototype of our design as a Security Enhanced version of the widely used OpenFlow Floodlight Controller, which we call *SE-Floodlight*. SE-Floodlight extends Floodlight with a security-enforcement kernel (SEK) layer, whose functions are also directly applicable to other OpenFlow controllers. The SEK adds a unique set of secure application management features, including an authentication service, role-based authorization, a permission model for mediating all configuration change requests to the data-plane, inline flow-rule conflict resolution, and a security audit service. We demonstrate the robustness and scalability of our system implementation through both a comprehensive functionality assessment and a performance evaluation that illustrates its sub-linear scaling properties.**

I. Introduction

SDN frameworks, such as OpenFlow (OF), embrace the paradigm of highly programmable switch infrastructures [1] managed by a separate centralized control layer. Within the OpenFlow network stack, the *control layer* is the key component responsible for mediating the flow of information and control functions between one or more *network applications* and the *data plane* (i.e., OpenFlow-enabled switches). The network applications are typically traffic-engineering applications, such as flow-based load or priority management services, or perhaps applications designed to mitigate hostile flows or to enable flows to bypass faulty network segments.

To date, OpenFlow controllers [2], [3], [4], [5] have largely operated as the *coordination point* through which network applications convey flow rules, submit configuration requests, and probe the data plane for state information. As a controller communicates with all switches within its network, or network slice [6], it provides the means to distribute a coordinated set of flow rules across the network to optimize flow routes and divert and balance traffic to improve the network's efficiency [7].

From a network security perspective, OpenFlow offers researchers a unique point of control over any flow (or flow participant) deemed to be hostile. An OpenFlow-based security application, or *OF security app*, can implement much more complex flow management logic than simply halting or forwarding a flow. Such apps can incorporate stateful flow-rule production logic to implement complex quarantine procedures of the flow producer, or they could migrate a malicious connection into a counter-intelligence application in a manner 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 [8]. Thus, there is a compelling motivation for sensitive computing environments to consider SDNs as a potential source of innovative threat mitigation.

However, there are also significant security challenges posed by OpenFlow, and SDN's more broadly. The question of what network security policy is embodied across a set of OF switches is entirely a function of how the current set of OF apps react to the incoming stream of flow requests. When peer OF apps submit flow rules, these rules may produce complex interdependencies, or they may produce flow handling conflicts. The need for flow-rule arbitration, as new candidate rules are created by the application layer, is an absolute prerequisite for maintaining a consistent network security policy.

Within the OpenFlow community, the need for security policy enforcement is not lost. Efforts to develop virtual network slicing, such as in FlowVisor [6] and in the Beacon OpenFlow controller [9], propose to enable secure network operations by segmenting, or slicing, network control into independent virtual machines. Each network domain is governed by a self-consistent OF app, which is architected to not interfere with those OF apps that govern other network slices. In this sense, OpenFlow security has been cast as a non-interference property. However, even within a given network slice, the problem remains that security constraints within the slice must still be enforced.

Contributions: This paper explores the challenges of defining a security mediation layer between the OpenFlow application layer (where both security and traffic-engineering application must co-exist) and the data-plane (where switches

implement the flow policies embodied in the flow rules produced by the OF Apps). To address the security mediation challenge, we propose the design of a security enforcement service embedded within the control layer, which mediates all flow rule submissions and control protocol requests produced by OF apps operating within the same network slice.

We examine the basic question of what it means to provide a complete mediation of all communications between the application layer and the data plane, and explore why the absence of such mediation makes OpenFlow unsuitable for secure network deployment. We then present a complete design description of a control-layer security mediation service, which we have implemented and deployed on one of the most widely used open-source controllers available today.

Among the critical technical challenges that our design must address is that of reconciling the dynamic production of flow-rule logic with the need to maintain consistent security policy constraints. Within OpenFlow networks, these security policy constraints are essentially flow rules, produced by an administrator or dynamically inserted by OF security apps in response to perceived threats. While we present our solution to the flow policy mediation challenge through the design description of SE-Floodlight, the described security features are fully applicable to the broader family of OpenFlow controllers.

II. Defining Security Mediation in an OpenFlow Network

This section enumerates several of the security challenges that face those seeking to deploy applications within an OpenFlow network. Here, we examine these challenges by way of motivating examples. We then follow this discussion with the presentation of a secure control-layer design that overcomes these challenges.

A. Challenge 1: Application Co-existence

Consider the basic challenge of implementing two or more OF apps within the same network, as depicted in Figure 1. Suppose A1 initiates a series of flow rule insertions designed to quarantine the flows to and from a local Internet server, which is operating in a malicious manner. A2, a load-balancing app, redirects incoming flow requests to an available host within the local Internet server pool. Suppose the Internet server quarantined by A1 subsequently becomes the preferred target for new connection flows by A2, as this quarantined server is now the least loaded server in the pool. Who should arbitrate the quarantine imposed by A1 against subsequent conflicting rules produced by A2, and which OF app (A1 or A2) should prevail?

Such conflicts in traffic engineering objectives may happen often. An excellent example of an OpenFlow Stack implementing multiple (competing) traffic engineering algorithms is Google's OpenFlow-based B4 private WAN network manager [7]. Using an OpenFlow implementation of a centralized WAN-wide traffic engineering (TE) application, Google's B4 accommodates three parallel traffic-engineering components:

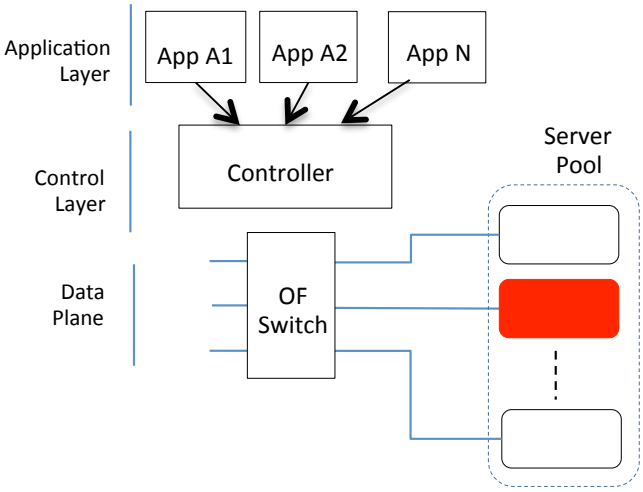

Fig. 1. Multi-app co-existence scenario

- *traffic throttling*: arbitrates competing application flow demands during periods when its networks resources are constrained.
- *flow balancing*: utilizes SDN's virtual circuit facilities to leverage available network capacity according to application priority.
- *fine-grained path manipulation*: it dynamically reallocates bandwidth in the face of link/switch failures or shifting application demands.

In combination, these traffic engineering functions enable Google to run the B4 private-WAN at near 100% utilization, with all links averaging 70% utilization over sustained periods, shattering the need for the typical 3-fold over-provisioning of capacity recommended by the standard networking model. In short, the centralized control paradigm of SDNs provides B4 the agility to dynamically navigate and throttle flows given the global operating needs of the private-WAN.

But how do these competing TE functions co-exist when they embody such diverse traffic TE objectives? How are conflicts resolved when a fair load-balancing strategy would dictate flows be directed down one network path, while a fault or flow-priority based assessment dictates otherwise.

To date, as in B4, the solution for OpenFlow developers has been to design competing TE logic into a monolithic application, where the arbitration of conflicting flow rules occurs fully within the app. Unfortunately, monolithic app designs are not extensible, reusable, secure, or reliably maintainable.

In this paper, we explore the potential for implementing conflict detection and resolution inline at the OpenFlow control layer. An intelligent control layer capable of arbitrating flow rule conflicts *could* offer an alternative for developing modular and reusable OF apps that may be co-instantiated within a single network. The trade-off of implementing multi-app arbitration logic at the control layer is that when an OpenFlow application loses arbitration, it must be informed of the failure, the basis for rejection, and must incorporate its own

recovery logic (Section IV-F). This approach to implementing exception logic to recover from loss from conflict arbitration is an essential element toward the design of robust and modular OF applications.

B. Challenge 2: Flow Constraints vs. Flow Circuits

Defining filters to constrain communications between network entities is a central element for defining network security policies. However, the OpenFlow protocol's introduction of the `Set` action empowers apps to instruct a switch to rewrite the header attributes of a matching flow. Indeed, perhaps the central benefit of SDNs is this ability to deploy software-enabled orchestration of flows to manage network resources in an agile manner. However, this inherent flexibility in the OpenFlow protocol to define complex virtual circuit paths, also introduces significant management challenges, such as the *origin binding* [10] problem.

The creation of virtual circuits offers a particular challenge to designing a control layer capable of maintaining a consistent network security policy, where the conflicts between incoming candidate flow rules and existing flow rules are detected and resolved before rules are forwarded to the switch.

For example, let us consider the submission of a flow rule by a security application A1, which seeks to prevent two hosts from communicating.

Rule 1: (*criteria*) A to B, (*action*) drop

As this flow constraint (i.e., dropping flows from A to B) is installed to protect the network, it should hold that any subsequent candidate flow rule should be rejected if it conflicts with Rule 1. Now consider the potential submission of three subsequent flow rules submitted by app A2, which may arrive in *any* order.

Rule m: * to D, set D→B, Output to table
Rule n: A to *, set A→C, Output to table
Rule o: C to B, forw

We observe that, individually, none of the three flow rules (m, n, o) conflicts with Rule 1. The action "Output to table" indicates that once the set operation is performed, the result should continue evaluation among the remaining flow rules.

In one possible rule submission ordering [m, n, then o], if the controller were to allow rule o to be forwarded to the switch, then a logic chain will arise. That is, a flow from A to D results in the following: D's address is set to B by rule m, A's address is set to C by n, and rule o cause the flow to be forward to B. In effect, any submission order of m, n, and o, leads to the circumvention of Rule 1.

The inability to reliably handle invariant property violations in recursive flow rule logic chains established by such virtual circuits, using the `Set` action, is a fundamental deficiency of existing systems such as VeriFlow [11] and commercial systems like VArmour [12]. Without a scalable inline solution, this basic challenge will hinder the deployment of multi-application deployments of OpenFlow in networks that require strong policy enforcement.

C. Challenge 3: An Application Permission Model

In addition to creating flow rules, OpenFlow provides apps with a wide range of switch commands and probes. For example, applications may reconfigure a switch in a manner that changes how the switch processes flow rules. Apps may query statistics and register for callback switch events, and they can issue vendor-specific commands to the switch.

While network operators might choose to run a third party OF app, OpenFlow offers them no ability to constrain which commands and requests the apps will issue to the switch. This notion of constraining a third party OF app's permissions may seem analogous to that of smartphone users who can choose to accept or deny a mobile app permission to access certain phone features or services. Why provide an OF App unneeded access to issue low-level vendor-specific commands, if the intent of the running app has no requirement for such calls to be issued?

D. Challenge 4: Application Accountability

The absence of design considerations for multi-app support in OpenFlow also results in a lack of ability for the control layer to verify which app has issued which flow rule. Co-existing applications could issue interleaved flow rules, all of which are then treated identically by the control layer and data plane. An arbitration system cannot assign unique precedence or priority to flow rules or switch commands produced from any application. Rather, OpenFlow's design requires that arbitration of conflicting flow rules occurs among the applications themselves. This approach limits the reusability of OF apps and results in the monolithic application designs, such as Google's B4 example.

E. Challenge 5: Privilege separation

The legacy of well-known OpenFlow controllers have essentially treated the application layer as a *library layer*, in which the traffic engineering logic is instantiated as interpreted scripts, module plugins, or loadable libraries that execute within the process space of the controller. This has largely been for the purpose of performance, but as shown in [13], the current state of established controllers are highly susceptible to network collapse from minor coding errors, vulnerabilities, or malicious logic embedded within a traffic engineering application. Even a slow memory leak within a TE application can cease the entire control layer, rendering the network unresponsive to new flow requests.

While [13] explored the robustness challenges of merging the application layer into the control layer, the implications of current controller architectures are equally problematic for implementing security mediation services. Among its implications, *privilege separation* dictates that the element responsible for security mediation should operate independently from those elements it mediates. Thus, for the OpenFlow control layer to operate as a truly independent mediator, applications should not be instantiated in the same process context as the controller.

In OpenFlow, separation between the application and control layer is achieved through a *Northbound API*, which is

essentially an API defined to transmit messages between the OpenFlow application and control layers, where each operates in a separate process context. There is substantial effort to establish a Northbound API standard [14], but in addition to providing the process separation necessary for fair mediation, the Northbound API should also provide strong authentication and identification services to link each app to all flow rules it has created.

F. Toward a Security-Enhanced Control Layer

In the remainder of this paper, we present our design of a security enhanced control layer that address all of the above outlined challenges. Based on this design, we created a security-enhanced version of an existing controller, which we call SE-Floodlight. This implementation represents the first fully functional prototype OpenFlow controller designed to provide a comprehensive security mediation of multi-application OpenFlow network stacks.

III. Designing An OpenFlow Mediation Policy

To better understand the notion of secure mediation in the context of OpenFlow networks, let us delve more precisely into what information is exchanged by the parties being mediated through the control layer.

Table I enumerates the forms of data and control function exchanges that occur between the application layer and data plane within an OpenFlow v1.0 stack.[1] Each row identifies the various data exchange operations that must be mediated by the control layer, and indicates the mediation policy that is implemented. Protocol handshakes used to manage the controller-to-switch communication channel are excluded: the focus here is the mediation of application layer to switch exchanges.

SE-Floodlight introduces a security enforcement kernel (SEK) into the Floodlight controller, whose purpose is to mediate *all* data exchange operations between the application layer and the data plane. For each operation, the SEK applies the application to data plane mediation scheme shown in in Table I. The *Minimum authorization* column in Table I identifies the minimum role that an application must be assigned to perform the operation. As discussed in Section IV-A, SE-Floodlight implements a hierarchical authorization role scheme, with three *default* authorization roles. The lowest authorization role, APP, is intended primarily for (non-security-related) traffic engineering applications, and provides sufficient permissions for most such flow-control applications. The security authorization role, SEC, is intended for applications that implement security services. The highest authorization role, ADMIN, is intended for applications such as the operator console app.

The objective of the mediation service is to provide a configurable permission model for a given SE-Floodlight deployment, in which both the set of roles may be extended and their permissions may be customized for each newly defined role.

[1]While the Open Network Foundation (ONF) continues to introduce new features and data types in the evolving OpenFlow standards, we believe the broad majority of these emerging features will translate into a variation of these data exchange operations.

Flow Direction	Data Exchange Operation	Mediation Policy	Minimum Authorization
01: A to D	Flow rule mod	RCA (Section IV-C)	APP
02: D to A	Flow removal messages	Global read	APP
03: D to A	Flow error reply	Global read	APP
04: A to D	Barrier requests	Permission	APP
05: D to A	Barrier replies	Selected read	APP
06: D to A	Packet-In return	Selected read	APP
07: A to D	Packet-Out	Permission	SEC
08: A to D	Switch port mod	Permission	ADMIN
09: D to A	Switch port status	Permission	ADMIN
10: A to D	Switch set config	Permission	ADMIN
11: A to D	Switch get config	Permission	APP
12: D to A	Switch config reply	Selected read	APP
13: A to D	Switch stats request	Permission	APP
14: D to A	Switch stats report	Selected read	APP
15: A to D	Echo requests	Permission	APP
16: D to A	Echo replies	Selected read	APP
17: D to A	Vendor features	Permission	ADMIN
18: A to D	Vendor actions	Permission	ADMIN

TABLE I
A SUMMARY OF CONTROL LAYER MEDIATION POLICIES FOR DATA FLOWS INITIATED FROM THE APPLICATION LAYER TO DATA PLANE (**A to D**) AND FROM THE DATA PLANE TO APPLICATION LAYER (**D to A**)

Column 3 of Table I presents the default *Mediation policy* that is assigned to each available interaction between the application layer and data plane. First, the ability to define or override existing flow policies within a switch (row 1) is the inherent purpose of every OpenFlow application. However, SE-Floodlight introduces *Rule-based Conflict Analysis* (RCA), described in Section IV-C, to ensure that each candidate flow rule submitted does not *conflict* with an existing flow rule whose author is associated with a higher authority than the author of the candidate rule.

In OpenFlow, a rule conflict arises when the candidate rule enables or disables a network flow that is otherwise inversely prohibited (or allowed) by existing rules. In OpenFlow, conflicts are either direct or indirect. A *direct conflict* occurs when the candidate rule contravenes an existing rule (e.g., an existing rule forwards packets between A and B while the candidate rule drops them). An *indirect conflict* occurs when the candidate rule logically couples with pre-existing rules that contravenes an existing rule. We discuss rule conflict evaluation in Section IV-C.

A second class of operations involve event notifications used to track the flow table state. These operations *do not* alter network policies, but provide traffic engineering applications with information necessary to make informed flow management decisions. The default permission model defines these operations as two forms of public read. The *Global read* represents data-plane events that are streamed to all interested applications who care to receive them (rows 2 and 3). *Selected read* operations refer to individual events for which an application can register through the controller to receive switch state-change notifications (rows 5, 6, 12, 14, and 16). Selected read notification represent replies to permission-protected operations, which are discussed next. The *Packet-In*

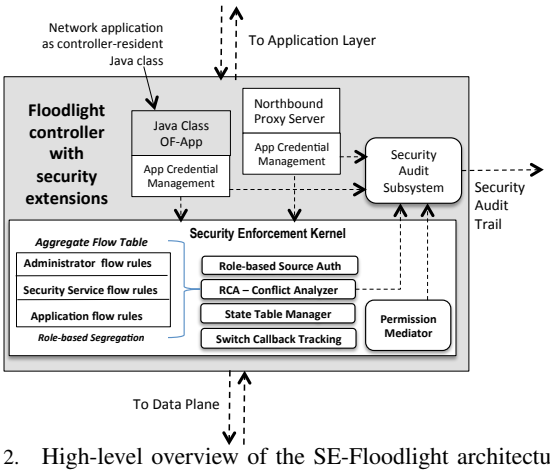

Fig. 2. High-level overview of the SE-Floodlight architecture

notification is an exception, in that it is received in response to flow rule insertions (vetted through RCA) that trigger the switch to notify that application when packets are received that match the flow rule criteria or when no matching flow rule is found.

The third class of operations involve those that require explicit permissions (Rows 4, 7-11, 13, 15, 17, and 18). These operations either perform *direct alterations* to the network flow policies implemented by the switch, or enable the operator to control switch configuration or to test switch accessibility. The intent is to grant an application permissions to the minimum set of operations necessary for it to perform the its expected functions. Custom permission sets can be granted to applications by creating additional authorization roles. Note that each permission is associated with a minimum authorization role, such that roles hierarchically above the minimum role are also granted permission to the operation. For the APP authorization role, permissions are granted for those operations that do not provide methods for altering network security policy. One exceptional operation is *Packet-Out*. SE-Floodlight restricts Packet-out to the SEC authorization role because this operation is used to formulate custom packets that are sent directly to the switch, bypassing flow table evaluation within the switch. For this operation, SE-Floodlight requires the administrator to specifically grant this permission to individual non-security-related traffic engineering applications.

IV. Design of a Secure Control Mediation Layer

Figure 2 illustrates the components integrated into SE-Floodlight, which extend baseline Floodlight to implement our security mediation service. Not shown in Figure 2 is a set of critical software patches which we introduce to alter and extend key elements of Floodlight. These alterations are necessary to ensure that our added components can perform full mediation of all data exchanges between the control layer and data plane. The extensions are also necessary to enable source-credential validation of each application message, and to enable our permission extensions to restrict certain application-layer interactions with the data plane.

From the bottom of Figure 2, the SEK empowers an OpenFlow stack to support multiple applications in parallel, introduce control-layer arbitration when apps produce conflicting flow logic, and imposes the application permission constraints described in Table I. The SEK extends Floodlight with five major components.

- A *Role-based Source Authentication* module provides digital signature validation for each flow-rule insertion request to provisionally limit a candidate flow rule's priority based upon the application's operating role (Section IV-A).

- A *Conflict Analyzer* is responsible for evaluating each candidate flow rule against the current set of flow rules within the *Aggregate Flow Table* (Section IV-C).

- The *State Table Manager* and *Switch Callback Tracking* components enable the SEK to maintain a locally accurate snapshot of the switch flow table (Section IV-E). This local representation is essential for enabling the Conflict Analyzer to determine whether a new candidate rule is in conflict with one or more flow rules already resident in the flow table.

- The *Permission mediator* mediates the non-flow-rule-based messages that are exchanged between the application layer and data plane. The Permission mediator denies access to an operation unless the application is authenticated under a role for which the administrator has granted permission to the operation.

An application may be run in two possible modes. First, in Section IV-B we present an application authentication scheme, which support author identity linkage for every SEK-mediated message. We present a *module authentication* service, which enables a Java class to be integrity-checked and assigned a cryptographically-signed operating role. Floodlight implements network applications as Java classes, as do other Java-based controllers; there is a substantial legacy base of OpenFlow applications implemented in this manner. Thus, a key pragmatic feature of SE-Floodlightis to validate the integrity of an OF app loaded as a class module at load time, and then to assign all messages it produces to the module's signed authorization role.

However, in secure deployments of SE-Floodlight, to address the challenge of *privilege separation* one should employ the digitally authenticated Northbound remote API, with per-message credential assignment. In Section IV-F, we discuss the benefits of using this API to enable the security mediation service to operate as a truly independent mediator between the application layer and the data plane. This is the method that secure deployments of SE-Floodlight would use to ensure the integrity of the security mediation service.

Finally, Section IV-G presents the first OpenFlow security audit subsystem, which tracks all security-relevant events occurring within the OpenFlow control layer. This audit subsystem satisfies an important prerequisite for environments that must address security compliance specifications pertaining

to audit of administrative functions. More broadly, the audit subsystem offers individual application accountability for all OpenFlow messages processed to and from the application layer, as well as capturing all controller-internal and switch-reported security relevant events.

A. Role-based Authorization

Today's OpenFlow controllers simply do not address contentions that may arise when co-resident OF apps produce conflicting flow rules. Before we describe our conflict detection and resolution strategy (Section IV-C), we must introduce the notion of *authorization roles*, which administrators may assign to individual OpenFlow applications.

An application's authorization role is assigned during the application authentication procedure, which is described in Section IV-B. Authorizations are group roles, whose members inherit both the rule authority used in conflict resolution discussed in Section IV-C, and the set of associated permissions presented in Section III. This authorization scheme introduces three default application authorization roles (types), which may be augmented by the operator with additional sub-roles, as desired.

Applications assigned the administrator role, ADMIN, such as an administrative console application, may produce flow rules that are assigned highest priority by the flow-rule conflict-resolution scheme. Second, network applications that are intended to dynamically change or extend the security policy should be assigned the *security application*, SEC, authorization role. Security applications will typically produce flow rules in response to perceived runtime threats, such as a malicious flow, an infected internal asset, a blacklisted external entity, or an emergent malicious aggregate traffic pattern. Flow rules produced with the SEC role are granted the second highest priority in the rule conflict resolution scheme, overriding all other messages but those from the administrator. All remaining apps are assigned the APP authorization role.

B. Module Authentication

For multi-application environments it is critical to understand which application has submitted which OpenFlow message. Module authentication provides the foundation for mechanisms such as application permission enforcement, role-based conflict resolution, and security audit for holding errant applications accountable.

SE-Floodlight enables two operating modes when mediating OpenFlow network applications. First, it can mediate an application that is implemented as an internally-loaded Java class module, which is the (legacy) method used by Floodlight. Second, SE-Floodlight introduces a client-server Northbound API, which enables operators to separate the controller process (the security mediator) from the process space of the application (the agent being mediated); see Section IV-F.

We begin by explaining the authentication and credential-assignment scheme used when the OpenFlow application is spawned as a class module. Our approach utilizes the Java *protected factory method construct*, which produces a protected subclass for each message sent from the client application Java class to the SEK. The protected factory supplies the authentication-supporting API extensions used to communicate with the SEK, and it embeds the application's credential into a protected subclass for each object passed to the SEK. Neither the factory nor its protected sub-classes are modifiable by the Java client, unless the client contains embedded JNI code or violates coding conventions, which we discuss in *Step 1* below.

Our goal is to introduce a class-based OF application messaging scheme that adds an administrator-assigned credential to each message produced by the app, while preventing the app from tampering with this credential. The following steps outlines our approach:

- *Step 1: JNI pre-inspector module:* Prior to credential establishment of the Java class module, the Java class module must be pre-parsed by an inspection utility. If the module is discovered to contain application-supplied JNI code or classes that are declared within or extend classes in "reserved" packages, the Java class will fail pre-inspection.[2]

- *Step 2: Application role assignment:* Java class integration begins with an installation phase, in which the administrator generates a runtime *credential*, which includes a signed manifest for a specified Java class module, its superclasses, and embedded classes. The credential uniquely identifies the application and incorporates the authorization group role to which the application will be assigned when instantiated. The administrator also assigns the authorization role to a *security group*, which specifies an upper limit of group priority that group members may assign to the messages sent to the controller. This upper limit enables the application to assign precedents to its own stream of flow rules within the sub-range of priorities corresponding to their role.

- *Step 3: Class validation function:* If a credential has been assigned to the class module by the administrator, the the integrity of the manifest and class files contained within the running JVM context are digitally verified. If verified, the class is provided a protected factory that will populate each created object with a non-accessible subclass that contains the role found within the application's credential (i.e., all objects produced through this factory are assigned the credential provided by the administrator in Step 2). Integrity check failures result in unloading of the application and a raised exception. If no credential is present, SE-Floodlight can be configured to either not load the module or to automatically load the module using the default (APP) credential.

- *Step 4: The application submits a message to the SEK:* When the SEK receives a message from a Java class module, it inspects the message to determine whether it contains a factory-supplied protected subclass. If the

[2]Any application requiring native code should be deployed using the Northbound API, Section IV-F, while applications containing classes that inject themselves into "reserved" packages will be summarily rejected.

message contains the protected subclass, the SEK assigns the message the group specified within the credential. If the protected subclass is not present, the SEK associates the message with a default role assigned by the administrator.[3]

C. Conflict Detection and Resolution

The SEK employs an algorithm called *Rule-chain Conflict Analysis (RCA)* [4] to detect when a candidate flow-rule conflicts with one or more rules resident in the switch flow table. By *conflict*, we mean that the candidate flow rule by itself, or when combined with other resident flow rules, enables or prevents a communication flow that is otherwise prohibited by one more existing flow rules.

1) RCA Internal Rule Representation: RCA maintains an internal representation of each OpenFlow rule, r, present in the switch. This representation is composed of the following elements:

- *r.criteria*: This corresponds to the flow-rule-match structures, as specified in the OpenFlow 1.0 specification.
- *r.action*: Corresponds to the flow-rule action field: the SET action effects are captured in the *r.SET_mods* field (below).
  - D - No output. Drop packets corresponding to this criteria
  - O_f - Forward: output to port, output to controller, or broadcast
  - O_t - Output to table: the flow may continue evaluation by other flow rules. This action enables two or more rules to be logically chained together when one rule whose action is O_t also contains logic to alter the flow's attributes to satisfy the criteria of a second rule (i.e. the two rules form a *rule chain*).
- *r.flow_altering*: Indicates whether the rule incorporates a SET action that alters the attributes involved in the match criteria
- *r.SET_mods*: Captures the alterations to the flow attributes performed by the SET action. (*r.SET_mods* equals *r.criteria* for all flow rules that exclude the SET action).
- *r.priority*: The priority of the rule as assigned by the application that authored the rule. (Each authorization role specifies a maximum priority that the applications may use.)

2) Synthetic Rules: Synthetic rules are produced by RCA and stored in the SEK's state table, but they are not forwarded to the switch. Rather, synthetic flow rules augment the state table by capturing the end points of virtual flows established through *rule chains*. From the OpenFlow controller perspective, rule chains are reducible to two deterministic end-points. A completed rule chain is one that is terminated by a flow rule whose action is either drop (D) or forward (O_f). A candidate rule can be chained to an existing rule in either of two ways. *Tail chaining* occurs with a resident rule, *r*, when *r.action* == O_t and *r.SET_mods* matches r_c*.criteria* (the criteria of our candidate rule). *Head Chaining* is the complementary case, where r_c*.action* == O_t and r_c*.SET_mods* matches *r.criteria*.

Tracking the correspondence between rules that contribute to the formation of a synthetic rule is accomplished by adding two attributes to the internal rule representation:

- *r.parents*: $\varnothing$ if this *r* is not a synthetic rule. Otherwise it specifies the two rules that were chained to produce the synthetic rule.
- *r.child* : $\varnothing$ if not a parent of a synthetic rule. Otherwise it specifies the synthetic rule created from *r*. Upon deletion of *r*, the child is garbage collected.

D. The RCA Algorithm

Set Definitions: The set of active RCA internal rules, which implicitly and explicitly correspond to flow rules resident in each switch's flow table, is denoted by the set A. Upon initialization of RCA, set A = $\varnothing$. From A, we also create a subset, H_f, which represents the set of resident rules that may become participants at the head of a rule chain, i.e., *r.flow_altering* == true and *r.action* == O_t.

Much of RCA's task is to *match* the candidate rule criteria, r_c*.criteria* against a resident rule's r_i*.criteria*. To perform rule matching, RCA employs a binary tree with source and destination n-tuples as keys. For each candidate rule, r_c, RCA performs the follow steps:

Step 1: Testing for direct conflict: For each binary tree match between candidate r_c and A, RCA determines which rule takes precedence. *Rule precedence* is assigned to the rule with the highest priority. If r_i takes precedence over r_c, then r_c is rejected with a permission (PERM) error code and RCA exits (no further evaluation is required). If r_c takes precedence over r_i, then r_i is deleted from the flow table (a delete notification is produced) and we proceed to Step 2. Otherwise, the two rule priorities are equal. In this case, if neither r_c nor r_i's action is O_t, their actions do not match, and r_c*.SET_mods* match r_i*.SET_mods*, then the *matching precedence policy* is applied. Otherwise, RCA proceeds to Step 2, and considers whether r_c will form a chain with other resident rules.

Matching Precedence Policy: When the two rule priorities are equal, then by administrative configuration, RCA will employ either a FIFO or LIFO strategy, assigning precedence to either the resident rule (FIFO) or the candidate (LIFO). FIFO is the default strategy, in that it rejects the existing candidate rule, thereby requiring the *explicit* recognition and removal of the conflicting resident rule before the candidate is added.

The next task is to identify conflicts that arise when the candidate combines with a resident rule to produce a rule chain. Here, let us introduce two synthetic rule variables: a synthetic tail-chain rule r_{tr} = $\varnothing$, and a synthetic head-chain rule r_{hr} = $\varnothing$. We also set r_{c-orig} = r_c.

Step 2: Detect a tail-chaining candidate rule: For each rule r_i, in H_f, if r_c*.criteria.src* matches r_i*.SET_mods.src* and r_c*.criteria.dst* matches r_i*.SET_mods.dst*, then we construct

[3]This may occur if the application employs Floodlight's legacy message API. No Floodlight API enables the application to bypass SEK message evaluation, and all such messages will inherit the administrator-assigned default authorization role (ideally, the lowest [APP] authorization role).

[4]The version presented here is motivated by the Alias-set Rule Reduction (ARR) algorithm presented in a previous workshop paper [15].

```
function DIRECT_FLOW_TESTING(r_c)
    for r_i ∈ A matching r_c do
        if (r_i.priority > r_c.priority) then return ;
        else if (r_i.priority < r_c.priority) then goto step2;
        else if ((r_i.action ≠ O_t) || (r_c.action ≠ O_t)) then
            goto step2;
        else if r_c.set_mods ≠ r_i.set_mods then goto step2;
        else goto match_precedence;
        end if
    end for
end function
function MATCH_PRECEDENCE(r_i, r_c)
    if (strategy == LIFO) then reject r_c and exit;
    else if (strategy == FIFO) then
        mark r_i for deletion and goto step2
    end if
end function
```

a synthetic tail chain rule that combines the flow logic of r_i followed by r_c. Synthetic rule r_{tr}.*criteria* is set to r_i.*criteria.src* and r_c.*criteria.dst*. Rule r_{tr}.*SET_mods.dst* is set to the destination address of the *last* non-synthetic rule in the chain. For synthetic rules with output O_t, r_{tr}.*SET_mods.src* is the source address of the *last* non-synthetic rule in the chain; otherwise the rule chain terminates, and r_{tr}.*SET_ mods.src* is the source address of the *first* non-synthetic rule in the chain. Finally, set r_{tr}.*priority* = lowest (r_c, r_i), and set r_{tr}.*action* to the last non-synthetic action in the rule chain. We then add r_{tr} as a child of r_c and r_i, mark r_i for deletion (its logic is now incorporated in the synthetic rule), and set r_c = r_{tr}.

```
function STEP 2: DETECT_TAIL_CHAINING(r_c)
    for r_i ∈ H_f | r_c.criteria ≡ r_i.set_mods do
        r_tr = r_i ⨁ r_c;
        r_tr.criteria = (r_c.criteria.src, r_i.criteria.dst);
        (r_f, r_l) = (first, last) non-synthetic rule in chain;
        if (r_tr.output = O_t) then r_tr.set_mods.src = r_l.src;
        else r_tr.set_mods.src = r_f.src;
        end if
        r_tr.priority = min(r_c.priority, r_i.priority);
        r_tr.action = r_l.action;
        r_c.child = r_tr;
        r_i.child = r_tr; and mark r_i for deletion
        r_c = r_tr;
    end for
end function
```

Step 3: Detect a head-chaining candidate rule: If r_c.*flow_altering* == *true* and r_c.*action* == O_t, we conduct a binary tree match of r_c.*SET_mods* to each r_i.*criteria* in A. Each match represents a head chaining link from r_c to r_i, and the synthetic rule r_{hr} is constructed to represents the rule set logic. Synthetic rule r_{hr}.*criteria* is set to r_c.*criteria.src* and r_i.*criteria.dst*. We determine r_{hr}.*SET_mods*, r_{hr}.*action*, and $r_h r$.*priority*, in the same manner as in Step 2 for r_{tr}. We then add r_{hr} as a child of r_c and r_i, mark r_i for deletion, and set r_c = r_{hr}.

Step 4: Chained rule conflict analysis: For rule chain r_c, we conduct a binary tree match of r_c.*SET_mods* against all

```
function STEP 3: DETECT_HEAD_CHAINING(r_c)
    if r_c.flow_altering and r_c.action = O_t then
        for r_i ∈ A | r_c.set_mods ≡ r_i.criteria do
            r_hr = r_c ⨁ r_i;
            r_hr.criteria = (r_c.criteria.src, r_i.criteria.dst);
            set r_hr.set_mods, action, and priority as in Step 2;
            r_c.child = r_hr;
            r_i.child = r_hr; and mark r_i for deletion
            r_c = r_hr;
        end for
    end if
end function
```

r_i.*criteria* in A. If a match is found such that r_c.*action* ≠ r_i.*action*, then a policy conflict exists and RCA determines which rule takes precedence in the same manner as in Step 1. If r_c.*priority* is lower than r_i.*priority* then r_c is rejected with a permission (PERM) error code and RCA exits. If r_c.*priority* is higher than r_i.*priority*, then r_i is deleted from the flow table (a delete notification is produced). If r_c.*priority* == r_i.*priority* and RCA enforces LIFO, then r_i is marked for deletion; otherwise r_c.*action* = r_i.*action*.

```
function STEP 4: CHAINED_CONFLICT_ANALYSIS(r_c)
    for r_i ∈ A | r_c.set_mods ≡ r_i.criteria do
        if (r_c.action == r_i.action) then continue;
        end if
        // policy conflict exists
        if (r_c.priority < r_i.priority) then
            reject r_c with PERM error and exit;
        else if (r_c.priority > r_i.priority)||(strategy == LIFO)
then
            delete r_i and notify
        else r_c.action = r_i.action
        end if
    end for
end function
```

Step 5: Integrate rule changes to H_f and A: Provisionally, if the $r \neq \varnothing$, add r_{c-orig}, r_{tr}, and r_{hr} to H_f and A. [5] Delete all r_i marked for deletion.

1) Multiple Switches: For simplicity, the algorithm described in Section IV-D is for a single switch. However, SE-Floodlight can also handle flow-rule mediation among multiple switches. To accomplish this, we provide an OpenFlow switch port connectivity table, where each row has two *switch identifier:physical port* pairs, S:P and S':P', that represent a bi-directional physical connection between two OpenFlow switches S and S'.

At a high level, the multi-switch algorithm uses RCA for individual switches. When a flow-rule transitions from one switch to another, SE-Floodlight changes the evaluation context to that of the receiving switch. Specifically, when a flow rule is evaluated for a given switch S with an *Output-to-Port* action to port P, SE-Floodlight checks for S:P in the table.

[5]The respective conditions are $r_{c-orig} = \varnothing$; $r_{tr} \neq \varnothing \wedge r_{tr}$.*child* $\neq \varnothing$; $r_{hr} \neq \varnothing \wedge r_{hr}$.*child* $\neq \varnothing$.

If S:P is found, SE-Floodlight replaces the action with a *Goto-Connected-Switch* pseudo-instruction referencing S'. When the instruction is evaluated, it then changes the table context to S' before continuing to the RCA evaluation.

2) Complexity: In Step 1, the binary tree search of r_c through A requires *O(log N)* comparisons, where N is the number of rules that are present in A. This step represents the bulk of switch forwarding and filtering rules that are produced for direct flow handling in a switch. However, traffic engineering applications will also produce rules to conduct flow rerouting logic, and rule candidates involved in altering flow-rule criteria attributes must be further processed by Steps 2-5. Tail chain analysis requires *O(log M)* comparisons, where M is the number of n-tuples that are present in H_f, $M \leq N$. Steps 3 only occurs when r_c is a chaining candidate, and requires *O(log N)* comparisons, Step 4 requires *O(log N)* comparisons when r_c produces a synthetic rule, and Step 5 requires *O(log N + log M)* comparisons. Thus, in the degenerate case, where M == N, RCA conflict resolution remains *O(log N)*.

E. Flow Policy Synchronization

A key facility of flow-rule conflict resolution is the ability for the control layer to maintain an accurate picture of the flow table state of the managed switch. SE-Floodlight maintains this state synchronization using two modules: the state table manager, and the switch callback tracer. Combined, these two modules maintain an accurate state of all active flow rules, their disposition within the switch's flow table, and the authorization role of the rule producer.

When a flow rule is inserted into the switch, the rule and the associated role of the flow-rule author, are recorded in the aggregate state table in its internal rule form. Before inserting a flow rule in the switch, the SEK enables the flow removal notification flag (OFPFF_SEND_FLOW_REM) on the rule, informing the switch to send a flow removal notification message when a deletion occurs. A flow rule may be deleted by the switch through rule-specific idle and hard timers. The SEK may also initiate the expulsion of a resident rule from the switch flow table.

When the SEK forwards a flow rule to the switch, it also records the rule or its derived rule chain in its local rule table. When a local rule is purged from the local rule table, its corresponding flow rule in the switch flow table is also purged and the SEK performs a barrier request to ensure the rule is removed prior to processing the next application layer message. The barrier notification is tracked by the switch callback tracer.

The memory complexity of the state table manager is linearly bounded by the capacity of the switch flow table. State table manager memory usage is not a function of the packet stream processed by the switch.

F. Process Separation

The Northbound API enables an application to be instantiated as a separate process and ideally operated from a separate non-privileged account. The main elements of the Northbound API are the protocol specification, the controller-side server module, and the language-specific access libraries. Connections to the server can be secured using standard SSL communication with either server or mutual authentication. In addition, the initial dialog can be configured to include an additional password-based authentication handshake.

The most unique and important aspect of this Northbound API specification is its ability to assign an authenticated application credential to every OpenFlow message that passes through the API. The initial connection establishment between an application and the Northbound API employs both a credential-based mutual authentication and password exchange over OpenSSL. Either or both authentication schemes may be used for author and role validation. Upon successful authentication, the validated role is checked against a pre-loaded configuration file, that is configured and managed by the network administrator, which contains the authorization role and privileges that this application will inherit. The role inheritance occurs by the instantiation of an individual server-side proxy object, which is assigned to marshal all messages to the SEK from this client. This proxy is instantiated using the same protected factory method described previously for client Java classes (Section IV-B), wherein this case the factory (and the application's credential) is assigned to the proxy. Upon successful authentication, all client API messages are received by its server-side proxy, and marshaled to the SEK as message objects.

Northbound API messages are divided into four types: Southbound-OpenFlow, controller-service, internal, and extensions. The Southbound-OpenFlow messages contain the OpenFlow protocol elements plus additional required context elements, such as the data-path identifier for a specific switch. While the structure of OpenFlow protocol elements embedded in the Northbound API protocol remains logically the same as in the OpenFlow specification, the elements are encoded using the GPB framework to provide a consistent and compact encoding scheme for all Northbound API protocol messages.

The controller-service messages provide remote clients with access to essential controller services, such as registering and deregistering to receive Southbound OpenFlow messages (e.g., Packet-In messages), query for permission to use an OpenFlow protocol type (e.g., send a Packet-Out message), and properly handling a switch-synchronizing barrier request-reply pair. *Extension messages* provide access to auxiliary controller functions, such as a device query for switch-port location or known IP address information. *Internal messages* include the initial handshake, any configured non-SSL authentication, setting or changing queue policy, error events related to server connections, and testing connection stability or latency with a ping request.

G. Security Audit Service

OpenFlow-based network event auditing and packet-level logging facilities have garnered increased attention in response to the dire need for advance solutions in understanding sta-

bility issues in complex traffic engineering applications, and for diagnosing ever increasingly complex network topologies. For example, systems like NetSight [16] and `ndb` [17] offer refined *packet-level tracing* facilities to understand how traffic traverses the network infrastructure. In the case of `ndb`, a postcard-based strategy is used to recover the route that packets take to complete the connection. NetSight builds upon this concept but introduces a system for tracking packet traversal history through the network, and for driving interactive debugging and monitoring systems. NetSight also records the data plane's flow table state and logs packet traces. OFRewind is an example *network-event-based* audit system for recording and playing back SDN control-plane traffic [18]. OFRewind is able to replay these network events through topologies for troubleshooting or debugging network device and control plane anomalies.

SE-Floodlight introduces yet another form of auditing in OpenFlow. Here our intention is focused on holding individual applications accountable for the OpenFlow messages they produce, and for tracking all all security-relevant events which are visible to the control plane. This OpenFlow audit subsystem satisfies an important functional prerequisite for environments requiring conformance with most security compliance specifications pertaining to audit of administrative functions. More broadly, it offers *substantial* help in tracking OpenFlow application-layer behavioral issues, such as for troubleshooting application correctness or comparing flow record production performance among similar applications that are exposed to the same network traffic.

Here, we introduce a translation of the common security audit requirements defined in criteria such as in the DoD TCSEC standard [19] into the context of the OpenFlow control layer. In doing so, we enumerate the following security-relevant auditable events, which are both necessary and present in this audit subsystem:

- use of identification and authentication events
- flow-rule modification events
- flow-rule conflict resolutions
- flow-rule delete and error notifications
- alterations to the configuration of the switch or controller, including application REST APIs
- packet in and out transactions
- statistics requests and replies
- use of vendor specific switch functions
- shutdown, startup, and connection to the audit subsystem
- SEK-specific application permission grant and denial events
- audit API used to add auditable event

The audit records are variant-prefixed binary Google Protocol Buffer records, compatible with the protobuf-2.5.0 specification [20]. Floodlight modules may obtain access to the audit control service (e.g., for administrative application developers) using an audit logger factory from Floodlight's service registry. Use of the audit API is restricted through the SEK permissions service and, by default, audit messages require a minimum SEC authorization role. Through the audit factory, Floodlight modules can create and write audit events and configure the audit service.

For each audit record, SEK reports the event time, message type, full message content, the application credential, the disposition (outcome) of the message, and optional message-specific field attributes. The audit service leverages the application authentication service to provide the digitally validated identity of the application that produced the event, or a switch identifier to uniquely identify switch-initiated audit events.

Finally, the audit subsystem provides facilities for audit event selection, event output management, and audit queue exhaustion recovery strategy. The recovery strategy offers either a configurable event *drop* policy or a *system halt* policy, in which all control layer processing is halted until the audit queue is consumed to a low-water mark. System halting when audit exhaustion occurs has been the required mode in military standards for operating system audit services, such as defined in the DoD TCSEC standard [19]. For completeness, the drop policy enables LIFO or FIFO discard, discard preferential selection based on event type, and the queue high and low watermark definitions for discard activation.

V. Functionality Assessment

In this section we consider the functional aspects of SE-Floodlight's features, both through a real use-case scenario and through a brief summary of security feature validation testing.

A. A Multi-application Use Case

Designing an OpenFlow control layer capable of securely managing multi-app network configurations allows operators to fields OF security apps in parallel with other network applications. While the security app may be deployed to detect and mitigate threats to network operations, there is still the basic need for an independent app to handle all of the foreground non-malicious flow request. One real-world live deployment scenario of SE-Floodlight has been deployed within our laboratory as a publicly available wireless network access point manager. We refer to this deployment instance as an example self-defending wireless network. The network topology is shown in Figure 3.

The data plane of the OpenFlow-based wireless network is implemented using the software switch OpenVswitch, which connects a set of 802.11b access points to the Internet. SE-Floodlight implements a control layer, which runs two OpenFlow applications. The first app is a variant of the Floodlight learning switch, operating with the APP security role, and is used to forward network flows from wireless clients on to the laboratory gateway. The second app is actually composed of three independent applications. BotHunter monitors a logical switch span port, analyzing all traffic from the wireless clients. BotHunter applies its dialog correlation algorithm to identify local wireless clients operating in a manner that is consistent with coordination-centric malware [21]. When

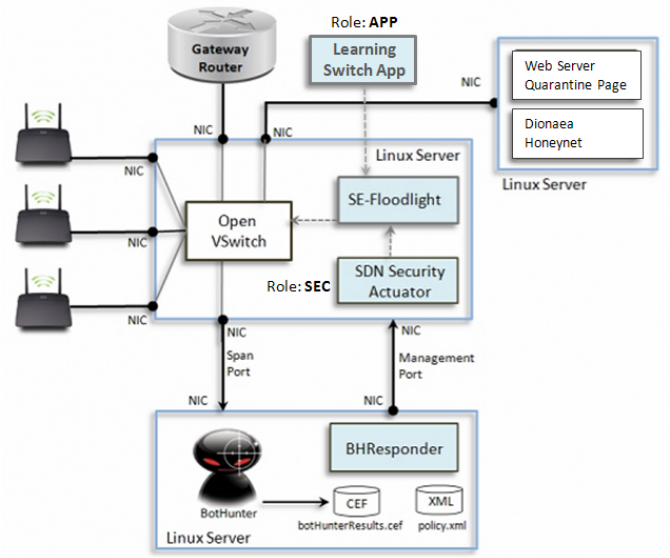

Fig. 3. The topology of a self-defending wireless network that uses SE-Floodlight to manage an anti-malware security service, called BotHunter, in parallel with a learning switch application. Combined, the applications manage wireless network traffic within our laboratory and automatically quarantine all flows to and from a local wireless client when BotHunter detects that the wireless client is infected with malware.

BotHunter produces an infection profile, the profile is forwarded to BHResponder, which implements a simple policy-matching algorithm to decide whether the local asset should be quarantined from the network. If so, BHResponder sends a quarantine security directive that identifies the IP address of the local wireless client to the SDN Security Actuator.

The SDN Security Actuator is an OpenFlow application operating with the SEC role, and uses the SE-Floodlight Northbound API to communicate with SE-Floodlight. When activated, it registers a callback to receive notification of all flow requests to and from the infected client. All connection to and from the client are then denied, except those involving DNS and Web requests. These client flows are redirected to a quarantine notification web page, indicating that the user should speak to the site administrator as soon as possible.

B. Functional Evaluation

Table II summarizes blackbox validation testing conducted on the various security mechanisms designed within SE-Floodlight.

Features	Functional Tests
Role-based Authorization	1: Rule rejection due to insufficient priority
RCA	2: Dynamic flow tunnel via tail chaining
Audit service	3: Audit log record generation
Module Authentication	4: Application credential validation 5: Remote application using NB API authentication

TABLE II
SUMMARY OF TESTS FOR EVALUATING THE SECURITY FUNCTIONALITIES OF SE-FLOODLIGHT

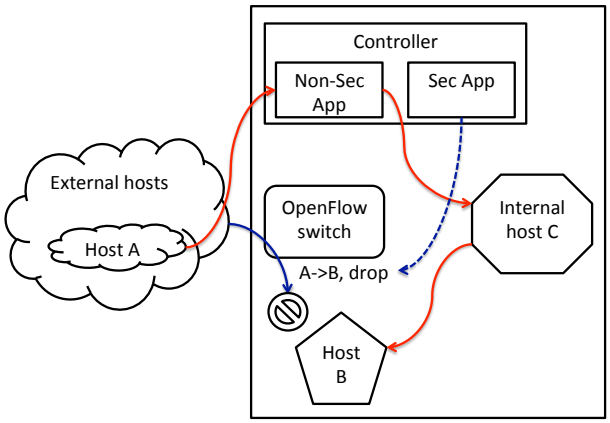

Fig. 4. Data flow for malicious traffic redirection

Test 1: Rule rejection due to insufficient priority: To test the detection of a direct rule conflict, we establish a network flow policy in which a security application installs a rule to prevent an external entity from communicating with an internal asset. We illustrate our example scenario in Figure 4.

We establish a test case in which malicious host A would like to connect to the internal host B, but is prevented from doing so via a flow rule that is inserted by an OF application running with the SEC authorization role. Next, an application operating at a lower authorization role (Non-Sec app in Figure 4) registers for PACKET_IN for flows originated from host A. Upon receipt of a matching flows, the application inserts a flow rule to rewrite C's IP addresses to B, thereby causing the flow to be diverted from C to B.

When running the malicious application with the (original) Floodlight controller, network traffic from the adversary-controlled external IP address succeeds in communicating with a host in the internal network. When SE-Floodlight is used, the conflict between the candidate flow rule and the SEC application's flow rule is detected, causing the SEK to determine that the candidate flow rule possesses insufficient authority to displace the drop rule.

Test 2: Dynamic flow tunnel via tail chaining: We evaluate the ability of SE-Floodlight to recognize conflicts that arise when the candidate rule combines to form a rule chain with existing resident rules. Here, the candidate rule does not independently violate an existing rule policy. However, when it combines with a set of existing resident flow rules, they form a virtual flow logic (captured by the RCA *synthetic rule*, Section IV-C2 for this chain) that is found to conflict with an existing flow rule. Note that the synthetic rule is assigned the lowest authorization role of all rules in the chain, such that if it is found to be lower than the conflicting rule, then our candidate rule is rejected.

We begin this test with three rules currently resident in the switch flow table:

ADMIN Rule #1: A to B; drop
ADMIN Rule #2: A to C; set A to A', output to table
SEC Rule #3: A' to C; set C to B, output to table

The first rule indicates that all flows from A to B should be filtered from our network. However, rules 2 and 3 contain logic to transform the source address from A to A' and from C to B. Next, an application running with the APP authorization role submits a flow that states:

APP Rule #4: A' to B; forward

This rule, does not directly violate any of the existing resident rules. However, when combined with rules 2 and 3, rule 4 produces a tail chain that creates a synthetic rule that inherits the lowest authorization role, APP, and which forms the logical flow rule:

APP Synthetic Rule: (A, A') to (C, B); forward

In the synthetic rule we show all attributes that the set transformations in rules 2-4, produce. That is, A is transformed to A' in rule 2, and C is transformed to B in rule 3. When the synthetic rule is compared to ADMIN rule 1, the candidate rule is found to produce an logical conflict with rule 1, and the candidate rule is therefore rejected.

With Floodlight, all rules in this scenario are successfully inserted into the flow table. When a flow is submitted to Floodlight from A to C, the flow is then faithfully tunneled from A to B, even though flow rule 1 (A to B; drop) prohibits this flow. With SE-Floodlight, rules 1-3 are successfully inserted into the flow table. However, on evaluation of rule 4, RCA computes the tail chain between the candidate and rules 2-3, determines that the result chain is in conflict with rule 1, and rejects rule 4 because the synthetic rule derives a lower authorization role than rule 1.

Test 3: Audit log record generation: The security audit service of SE-Floodlight can be configured to log controller security-relevant events of interest to the users. The audit log may be useful for network troubleshooting as well as a data source for security monitoring. We performed a test with the StaticFlowPusher Floodlight module, which provided a REST API for modifying flow tables. Also, we granted the PACKET_OUT privilege to the module. Figure 5 depicts an audit log record generated when we used the module to manually add a flow rule to a switch, with part of the Southbound OpenFlow message truncated.

```
posixTime: 2014-05-05T11:22:33.444
agentID: "n.f.security.SEFloodlight"
instanceID: "n.f.security.SEFloodlight@29422384"
status: INFO
detail {
  category: "OF"
  instanceID: "n.f.s.SEStaticFlowEntryLoader@197ebe66"
  clientID: "SEStaticFlowEntryLoader"
  group: "packetOut"
  memberID: "packetOut"
}
sbOFMessage {
  datapathID: 00:00:00:00:00:01 (1)
  ofMessage {
    ofType: FLOW_MOD
  ...
```

Fig. 5. Example audit log record

Test 4: Application credential validation: To counter the threats in which an adversary may forge an application or may compromise the integrity of an existing one, SE-Floodlight employs credential files for applications. Our test involves creating new applications and modifying existing applications.

With Floodlight, which does not support the notion of application credentials, we could create new applications and modify existing ones, and successfully ran the applications. With SE-Floodlight, the newly created applications (for which we did not create credential files) and the modified applications (for which we did not regenerate the corresponding credential files) failed to run.

Test 5: Remote application using NB API authentication: The Northbound API supports mutually authenticated connections between northbound client and SE-Floodlight, including two types of authentication: password-based and SSL-certificate-based. Also, Northbound authentication enables the use of the verified client identifier as the member ID for SE-Floodlight. We performed a test that involves running an application on a remote machine sending a password over a secure connection to perform authentication with SE-Floodlight using the Northbound API.

While Floodlight supports a REST API for remote message exchange, this interface does not support fine-grained application authentication. SE-Floodlight's module authentication service is designed to support individual authentication of applications. With SE-Floodlight, we showed that a remote application that did not have the right authentication token was prevented from performing any actions against OpenFlow switches using SE-Floodlight.

VI. Performance Evaluation

We conducted experiments to measure the latency overhead for our prototype implementation of SE-Floodlight. We define *latency* as the time that SEK takes to perform flow table maintenance and RCA resolution for a new flow rule insertion.

In our experiments, we ran an OpenFlow controller on a Linux machine, and ran Open vSwitch (a software-based OpenFlow switch; http://openvswitch.org) on another Linux machine. By using separate machines for the controller and the switch, we can more accurately measure the performance of the controller without being affected by the switch activities that do not involve the controller. To measure the latency, we implemented a timing module for SE-Floodlight, which provides a REST API for obtaining the statistics of the latency measurements.

We used nmap (http://www.nmap.org) on a host S to send UDP packets to random unique ports of another host D. On host D, we ran the discard service listening to a specified port,

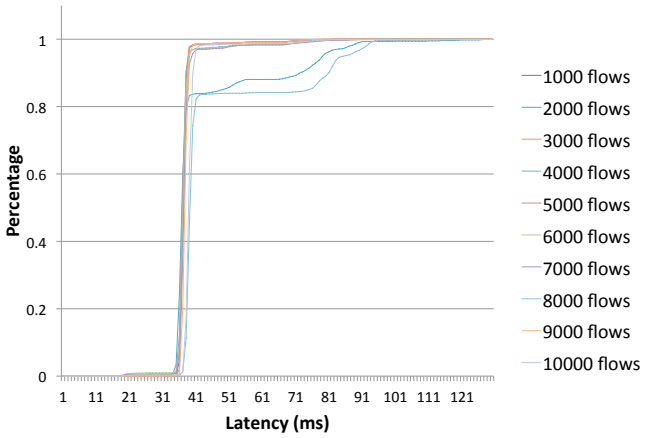

Fig. 6. Cumulative distribution function for the latency of adding new flows

and configured iptables to redirect UDP traffic for a range of ports to the discard service port. Also, we employed a purpose-built IP-address-based learning switch module for Floodlight that generated flowmods with source and destination IP address fields, source and destination ports, and the protocol. Recall that the overhead of RCA is a function of the number of active flow rules. Thus, we constructed these flowmods not to timeout to create a worst case scenario for SE-Floodlight.

To measure the latency overhead of SE-Floodlight as a function of the number of new flow rules, we conducted a series of tests in which we created different number of flow rules starting with an empty flow table. Figure 6 depicts the latency CDF for the tests, with the X-axis corresponding to latency (milliseconds) for adding new flow rules and the Y-axis corresponding to the cumulative percentage.

The results (as shown in Figure 6) indicate that the maximum latency pertaining to the majority of the flows remained relatively flat as we increased the number of new flow rules in the experiment. This suggests that our implementation has good scaling properties. The figure also illustrates that the addition of most new flows incurs an additional overhead of approximately 35 ms. Despite being an unoptimized prototype implementation, our flow management overheads seem to be in line with measured median inter-arrival times, for new flows in a datacenter, of around 30 ms [22].

VII. Limitations and Discussion

We have scoped the central security objective of this paper to that of reconciling the need for a reliably enforced security policy against the ability for SDN applications to deliver highly dynamic traffic flow management. Our security-enhanced controller, SE-Floodlight, does not enforce non-interference among co-resident applications; interference is explicitly allowed, as applications may override each other, depending on their respective authorizations. Neither does SE-Floodlight prevent disclosure of the operations performed by peer applications. For example, applications may observe flow rule delete notifications produced from peer applications. SE-Floodlight does not safeguard the SDN-stack from actions that would cause it to halt operations. SDN *resilience* at the data plane is explored in [23], which is complementary to our *security-focused* design solution.

This design presentation also does not discuss the administrative responsibilities in configuring and operating a secure control layer in an operational deployment scenario. For example, SE-Floodlight *can* be deployed in a manner that would enable a malicious application to bypass or corrupt the SEK (e.g., employing an application with embedded JNI designed to alter the control layer state). However, SE-Floodlight's Northbound API enables applications to be run in a separate process context, and ideally within a separate user account. Thus, SE-Floodlight is configurable to provide strong privilege separation between the controller and the applications that it mediates.

VIII. Related Work

SE-Floodlight is built over the foundations laid by previous studies such as Ethane [24], SANE [25] and 4D [26] that first made the case for control-flow separation and clean-slate design of the Internet and catalyzed the development of OpenFlow. We build our system on Floodlight, which is an open-source OF controller [4]; however, our methodology could be extended to other architectures like Beacon [5], Maestro [27], and DevoFlow [28]. A more basic version of our system implementation on the NOX controller was described in a workshop version of our paper [15]. FlowVisor [6] is a platform-independent OF controller that seeks to ensure non-interference *across* different logical planes (slices) but does not instantiate network security constraints *within* a slice. It is possible that an OF application could use packet modification functions to result in flow rules that are applied across multiple network switches within the same slice. In such cases, we need a security enforcement kernel such as SE-Floodlight to resolve conflicts.

The problem of routing misconfigurations has been well studied in the context of traditional enterprise monitoring and inter-domain routing protocols For example, researchers have investigated the problem of modeling network devices to conduct reachability analysis [29], [30], firewall configuration using decision diagrams [31] and test case generators [32], [33]. The router configuration checker (rcc) uses constraint solving and static analysis to find faults in BGP configurations [34].

The need for better policy validation and enforcement mechanisms in SDN networks has been touched on by prior and concurrent research efforts on dynamic access control [35], automated application testing [36] and language abstractions to guarantee consistency of flow updates [37]. In addition, there are multiple competing efforts that build a control plane for middleboxes and argue for decoupling middleboxes from controllers [38], [39], [40]. In our workshop paper [15], we presented a conflict analysis algorithm and the need for a security enforcement kernel. This paper presents an alternate conflict analysis algorithm with improved computational performance (logarithmic vs. linear) and support for multi-switch

deployments.

The FlowChecker system encodes OpenFlow flow tables into Binary Decision Diagrams (BDD) and uses model checking [41] to verify security properties. Veriflow is a real-time system that slices flow rules into equivalence classes to efficiently check for invariant property violations [11]. However, the evaluation of FlowChecker and Veriflow do not consider handling of *set* action commands, which we consider to be a significant distinguisher for OpenFlow networks, i.e., these systems do not handle recursive rewrite rules that result in dynamic flow tunnels, as described in Section II. *Header space analysis* is an offline static analysis technique for detecting network misconfigurations [42]. A recent paper, extends HSA for inline rule-conflict detection and resolution in the context of firewalls. Their system faces similar challenges to ours in terms of detecting rule conflict violations. However, their goal is to build more robust firewalls for SDN environments and does not deal with the problem of conflict resolution among competing OpenFlow applications. Further, they operate on single-switch environments and their algorithm does not handle inter-table dependencies [43].

IX. Conclusion

While OpenFlow has gained much attention in the network research community, and arguably a fair amount of momentum among modern network designers, the OpenFlow protocol and its current control-layer implementations are significantly lacking in an ability to support multiple applications within a single network. We surveyed the security challenges involved in not just managing competing OF apps within a single network, but design an extensive set of features that would enable a multi-apps OpenFlow network to run in a sensitive network computing environment that must meet stringent security requirements.

The central contribution of this work is the presentation of a reference implementation of a security-enhanced controller, called SE-Floodlight. We introduce the notion of OF-app security roles and an OpenFlow-specific permission model to cover all OpenFlow message exchanges between apps and the data plane. We also introduce the *Rule-chain Conflict Analysis* algorithm, which is an inline flow-rule conflict detection algorithm capable of identify invariant property violations in recursive flow-rule logic chains that arise from OpenFlow's virtual circuit facilities. We propose a role-based hierarchical resolution strategy that resolves conflicts, and a Northbound API that provides authenticated per-message credentials. Finally, we introduce an application-layer audit subsystem into the OpenFlow control layer, and define the mapping of events and audit attributes necessary to satisfy common security-audit requirements.

In August 2013, we released an early prototype of SE-Floodlight on the Internet, which has received over 300 downloads. The prototype is intended to stand as a reference implementations of our ongoing research toward future *secure* SDN infrastructures. We intend this work to help accelerate the eventual development of a robust OpenFlow control layer that can make compelling SDN-enabled security applications available in the networks that need them most.

X. Acknowledgments

The authors would like to gratefully acknowledge fruitful discussions with Guofei Gu, Andrew Moore, Peter Neumann, and Seungwon Shin during various stages of this project. The work described in this paper was sponsored by the Defense Advanced Research Projects Agency (DARPA) MRC2 Program and the Air Force Research Laboratory (AFRL), under contract FA8750-11-C-0249. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

References

[1] 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 Computer Communications Review*, April 2008.

[2] POX, "Python network controller," http://www.noxrepo.org/pox/about-pox/.

[3] 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 Computer Communications Review*, July 2008.

[4] FloodLight, "Open SDN controller," http://floodlight.openflowhub.org/.

[5] D. Erickson, "The beacon openflow controller," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013.

[6] 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 the Usenix Symposium on Operating System Design and Implementation (OSDI)*, 2010.

[7] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-deployed Software Defined WAN," in *Proceedings of ACM SIGCOMM*, 2013.

[8] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson, "Fresco: Modular composable security services for software-defined networks," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*, February 2013.

[9] OpenFlowHub, "BEACON," http://www.openflowhub.org/display/Beacon.

[10] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul, "FlowTags: Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013.

[11] 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.

[12] VArmour, http://www.varmour.com/.

[13] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, "Rosemary: A robust, secure, and high-performance network operating system," in *Proceedings of the ACM SIGSAC Conference on Computer Communications Security*, ser. CCS, 2014.

[14] Open Networking Foundation Northbound Interfaces Working Group, https://www.opennetworking.org/images/stories/downloads/working-groups/charter-nbi.pdf/.

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

[16] N. Handigol, B. Heller, V. Jeyakumar, D. Mazieres, and N. McKeown, "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks ," in *Proceedings of the Usenix Symposium on Operating System Design and Implementation*, 2014.

[17] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, "Where is the debugger for my software-defined network?" in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. ACM, 2012.

[18] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, "Ofrewind: Enabling record and replay troubleshooting for networks," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'11. USENIX Association, 2011.

[19] Department of Defense, *Trusted Computer System Evaluation Criteria*, Dec. 1985.

[20] K. Varda, "Protocol buffers: Google's data interchange format," Google, Tech. Rep., 2008. [Online]. Available: http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html

[21] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee, "BotHunter: Detecting Malware Infection Through IDS-driven Dialog Correlation," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007.

[22] S. Kandula, S. Sengupta, A. Greenberg, and P. Patel, "The nature of datacenter traffic: Measurements and analysis," in *In Proceedings of Usenix/ACM IMC*, 2009.

[23] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Avant-guard: Scalable and vigilant switch flow management in software-defined networks," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security*, ser. CCS, 2013.

[24] 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.

[25] M. Casado, T. Garfinkel, M. Freedman, A. Akella, D. Boneh, N. McKeowon, and S. Shenker, "SANE: A Protection Architecture for Enterprise Networks," in *Proceedings of the Usenix Security Symposium*, 2006.

[26] 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.

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

[28] 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 10th ACM Workshop on Hot Topics in Networks (HotNets)*, 2010.

[29] 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 *Proceedings of the IEEE International Conference on Network Protocols*, 2009.

[30] G. Xie, J. Zhan, D. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford, "On Static Reachability Analysis of IP Networks," in *Proceeding of IEEE INFOCOM*, 2005.

[31] A. Liu, "Formal Verification of Firewall Policies," in *Proceedings of the International Conference on Communications (ICC)*, 2008.

[32] D. Senn, D. Basin, and G. Caronni, "Firewall Conformance Testing," in *Proceedings of the IFIP TestCom*, 2005.

[33] 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," Tech. Rep., 2007.

[34] N. Feamster and H. Balakrishnan, "Detecting BGP Configuration Faults with Static Analysis," in *Proceedings of the Usenix Symposium on Network Systems Design and Implementation*, 2005.

[35] A. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: Dynamic Access Control for Enterprise Networks," in *Proceedings of the 1st ACM SIGCOMM WREN Workshop*, 2009.

[36] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, "A NICE Way to Test OpenFlow Applications," in *Proceedings of the Symposium on Network Systems Design and Implementation*, 2012.

[37] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent Update for Software-Defined Networks: Change You Can Believe In!" in *Proceedings of the ACM Workshop on Hot Topics in Networks*, 2011.

[38] B. Anwer, T. Benson, N. Feamster, D. Levin, and J. Rexford, "A slick control plane for network middleboxes," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013.

[39] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simplefying middlebox policy enforcement using sdn," in *Proceedings of the ACM SIGCOMM Conference*, 2013.

[40] A. Gember, A. Krishnamurthy, S. S. John, and A. Akella, "Virtual middelboxes as first-class entities in the cloud," in *Proceedings of the Open Network Summit*, 2012.

[41] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration Analysis and Verification of Federated Openflow Infrastructures," in *Proceedings of the 3rd ACM SafeConfig Workshop*, 2010.

[42] P. Kazemian, G. Varghese, and N. McKeown, "Header Space Analysis: Static Checking for Networks," in *Proceedings of the Symposium on Network Systems Design and Implementation*, 2012.

[43] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "Flowguard: Building robust firewalls for software-defined networks," in *Proceedings of the Third ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2014.