SDNShield: Reconciliating Configurable Application Permissions for SDN App Markets

Xitao Wen*, Bo Yang§, Yan Chen*, Chengchen Hu‡, Yi Wang◇, Bin Liu†, Xiaolin Chen°
*Northwestern University, §Zhejiang University, ‡Xi'an Jiaotong University,
◇Huawei Future Network Theory Lab, Hong Kong, †Tsinghua University, °Chuxiong Normal University

***Abstract*—The OpenFlow paradigm embraces third-party development efforts, and therefore suffers from potential attacks that usurp the excessive privileges of control plane applications (apps). Such privilege abuse could lead to various attacks impacting the entire administrative domain. In this paper, we present *SDNShield*, a permission control system that helps network administrators to express and enforce only the minimum required privileges to individual controller apps. *SDNShield* achieves this goal through (i) fine-grained SDN permission abstractions that allow accurate representation of app behavior boundary, (ii) automatic security policy reconciliation that incorporates security policies specified by administrators into the requested app permissions, and (iii) a lightweight thread-based controller architecture for controller/app isolation and reliable permission enforcement. Through prototype implementation, we verify its effectiveness against proof-of-concept attacks. Performance evaluation shows that *SDNShield* introduces negligible runtime overhead.**

I. Introduction

The control plane is one of the most critical yet vulnerable part of a network. Traditionally people have to put absolute trust in the reliability of the control software, which from time to time allows attackers to exploit network devices [1], [2], [3], [4]. Today, with software-defined networks (SDN), the control software becomes even more vulnerable. The mainstream SDN platforms [5], [6] foster open and prosperous markets for control-plane software [7], who provide a great range of apps for network management. However, obtained from various sources, the apps come with different quality guarantees and thus may contain flaws, vulnerabilities and even malicious logic. As a result, the security concern is ranked the top issue that prevent enterprise and data center networks from adopting SDN [8].

A natural solution to the over-privilege problem is access control. Typically, cryptographic authentication is employed by state-of-the-art SDN security solution in order to prevent unauthorized accesses. In addition to authentication, some access control enforcement mechanisms based on Android-like permissions [9], [10], [11], [12], [13] are proposed. However, although these mechanisms broadly extend the policy enforcement frontier, it remains challenging both to *specify appropriate permissions* and to *enforce permissions gracefully*. Furthermore, most current SDN controllers implement a monolithic architecture that allows app code directly executed in the controller runtime. Although efficient, such architecture provides merely no security isolation between controller and apps. Rosemary [10] explores to isolate apps with OS process containment. However, it still remains unexplored how to enable the app developers and the administrators to cooperate on composing secure permission policies.

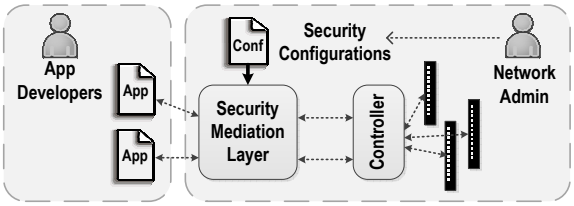

Fig. 1: Existing SDN access control systems.

Therefore, we envision the next generation SDN permission system with a fine-grained policy structure, a generic reconciliation model and a secure while efficient isolation architecture, as described in the three key features below.

1) **App developers can express fine-grained permission requests.** API-grained permissions are inadequate to specify minimum privileges for SDN apps. We need a flexible set of *permission abstractions* that allow fine-grained representation of app behavior boundary.
2) **Administrators can easily refine app permissions with higher-level security policies.** Fine-grained permissions bring the challenge in permission management. To reduce management burden, we need a reconciliation process that refines and customizes the requested app permissions with the security policies specified by the administrators.
3) **The controller needs a secure while efficient isolation architecture to enforce permissions.** Instead of a monolithic controller, we need to enforce isolation between controller and apps to ensure reliable permission enforcement and robustness. Such an isolation architecture should provide security properties while being compatible with existing apps and imposing minimal runtime overhead.

We present *SDNShield*, the first permission control system for SDN apps that achieves all the above features. An overview of the *SDNShield* architecture is shown in Figure 2. The key challenges are as follows. The behavior space of controller apps has inherent complicated permission structure and thus 1) **clean permission abstractions** are needed to capture such complexity. 2) **Policy reconciliation** needs language support, for which the permission language and security policy language need to be co-designed. 3) **Enforcement overhead** has to be minimal, since permission enforcement is on the critical path of the control plane.

We made the following contributions.

- We design fine-grained permission abstractions as a domain-specific language (§IV). The abstractions feature a two-level structure: the coarse-grained *permission token*

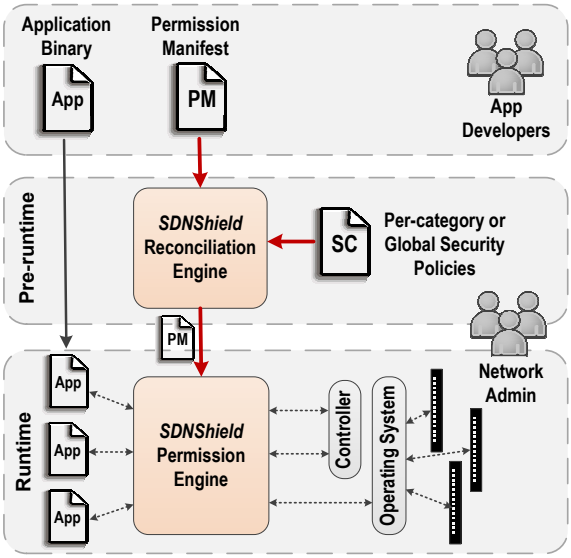

Fig. 2: *SDNShield* overview.

and the fine-grained *permission filter*. Such structure allow the flexible expression of permissions with structured parameters, such as abstract topology, while hides the complexity of the controller specifics.

- We propose the architecture to conduct reconciliation on requested permissions and security policies (§V). We define permission boundary and mutual exclusion as two basic forms of security policies, which characterize a variety of security goals.
- We design a novel controller isolation architecture that features a thread-based lightweight isolation between controller and apps (§VI). We choose to isolate app executions with Java threads by leveraging and extending Java build-in security sandbox, which provides comprehensive reference monitoring, incurs negligible overhead and requires no modification on legacy apps.

We implement a controller-independent prototype of *SDNShield* as a standalone permission engine and a Java-based controller isolation framework. With some controller specific extensions, we demonstrate its effectiveness with OpenDaylight controller platform [5], one of the most popular SDN controller platforms, as well as Floodlight OpenFlow controller [14].

We present two use case scenarios in §VII, describe our prototype in §VIII and evaluation in §IX. The results demonstrate that *SDNShield* introduces negligible latency overhead and imposes minimal impact on throughput of the controller runtime.

Although our implementation currently works with Java-based SDN controllers, the design of *SDNShield* universally applies to other SDN controller platforms. Furthermore, we also show that, with some extensions, *SDNShield* can be adapted to the emerging controllers that provide rich northbound interfaces (§VI-C).

The design of *SDNShield* is inspired by the large body of studies and practices of general access control systems (§X), especially on mobile OS and web browsers whose environment resembles that on a SDN controller. However, the unique design constraints in SDN environment, such as the resource structure, the threat models and the permission management challenges, lead us to the design choices and the technical contributions of *SDNShield*.

II. Threat Model

The novel threats on SDN apps stem from the fundamental challenge of verifying the trustworthiness of a program module [15]. Because apps directly interact with critical resources, people expect apps to have a high level of security as the controller. However, although the mainstream controller platforms are usually well verified, the quality and goodwill of all apps in the market are very challenging to guarantee with the state-of-the-art program analysis techniques, thus the existence of vulnerabilities and exploits is generally inevitable. Generally, either buggy or malicious apps can expose the controller platform to exploits. A number of traditional attack patterns, such as web-based attacks to the management interface or host-based attacks on the app host machine, can potentially give attackers the arbitrary code execution capacity on behalf of the app.

As long as an app is compromised, the attacker will have unfettered control of the controller, the OF switches in the network domain as well as the resources provided by the host operating system. Thus, the attacker can launch a full range of potential attacks. Among them, we are particularly interested in the following four classes of attacks as shown in Figure 3, for which existing SDN security approaches do not deliver good protection.

Class 1: Intrusion to data plane: With the capacity of packet-in/packet-out messages, the compromised app can sniff/inject arbitrary data-plane packets in realtime, which enables attackers to directly manipulate the traffic.

Class 2: Leakage of sensitive information: A compromised app can leak out obtained information via host network or other side channels on the host machine where the controller resides. This loophole breaches the confidentiality of the network configuration and traffic statistics, and can enable more advanced attacks by allowing attackers to conduct analysis on the flow tables, the control software and the models of OF switches.

Class 3: Manipulation of rules: A compromised app can manipulate forwarding behavior stealthily with the unfettered capacity to modify OF rules, resulting in various active network attacks, such as the man-in-the-middle attack and blackhole attack. Such capacity can be combined with network-based attacks to generate more advanced exploits.

Class 4: Attacking other apps: A compromised app can deactivate other apps, especially security apps, by overriding or bypassing their rules. Attackers can bypass the ACL rules set by a security app through *dynamic-flow tunneling* [16], which evades existing security policies by establishing a tunnel and modifying the packet header at both ends of the tunnel.

Table I compares the attack protection coverage of existing SDN security approaches and *SDNShield*. Among them, traffic isolation prevents the attacks launched from one network slice to attack another slice, but delivers no security to apps deployed on one network slice that collaboratively process the same set of traffic. Network state analysis, on the other hand, can detect global invariant violations (*e.g.,* forwarding black-hole) in forwarding rules, but is unable to detect the

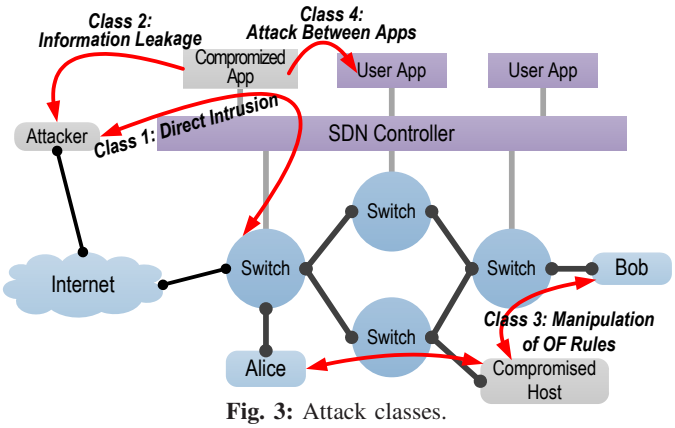

Fig. 3: Attack classes.

	Class 1	Class 2	Class 3	Class 4
Traffic isolation	CT	CT	CT	CT
NW state analysis	✗	✗	✓	✓
SDNShield	✓	✓	✓	✓

TABLE I: Comparison of protection delivered by SDN security approaches. CT: only protect against attacks across tenants.

other malicious behavior, such as traffic sniffing/injection and information leakage.

In comparison, with proper permission settings, *SDNShield* delivers good protection on all the four attack classes by preventing apps from conducting security sensitive actions it is not authorized to.

III. SDNShield Overview

SDNShield is a general SDN permission system for both app developers and the administrators that allows easy policy composition for the administrators and reliable policy enforcement on apps. We propose a configurable permission and reconciliation system that allows administrators to *configure* the app requested permissions with parameters and make it possible to write apps that gracefully react to the denied permissions. As depicted in Figure 2, the process works as follows. 1) The app developer distributes app release together with the *permission manifest* representing the potentially needed permissions. 2) Before app deployment, the administrator configures and customizes the app permissions with the local *security policies*, which contains parameters and constraints to the app permissions. 3) The *reconciliation engine* then reconciles the permission manifests with the security policies, and produces the final parameterized permissions to be enforced. 4) When the app is running, the *permission engine* mediates API calls and enforces the permissions. Utilities are provided for app developers to check for permissions and handle the permission denials before making an API call that requires permissions. [1]

SDNShield contains two major components, the permission engine and reconciliation engine.

Permission engine. *SDNShield* permission engine compiles and enforces the permission policies written in *SDNShield permission language*. When the app is loaded, the permission engine compiles the permission manifest into the runtime checking code that is associated with every API call issued from the app. During the lifetime of the app, the permission engine keeps mediating all the API calls, making sure the permission policies are consistently enforced.

In practice, the permission manifests provided by the app developers are distributed along with app release packages, since the app developers have the most complete knowledge of the apps. A permission manifest can be automatically generated from app source code with static/dynamic analysis tools employing the techniques similar to Android permission analysis [17], [18], [19]. Then, the developers can refine the permission manifest to specify optional permission switches for specific functionalities, so that no redundant permissions are requested if optional functionalities are disabled.

Reconciliation engine. *SDNShield* allows the administrators to configure app permissions via security policy. The security policy is represented with the domain-specific *SDNShield security policy language*, which provides abstractions for a variety of security constraints.

In practice, the administrators first describe their local security policies in the security policy language, and feed the policies to the reconciliation engine. The reconciliation engine then verifies the security policies by looking at the input permission manifest. Once a policy violation is detected, the reconciliation engine alerts the administrator and provides possible alternative permissions for administrator's consideration. The alternative permissions are generated by truncating the offending permissions or refining permissions with environment variables.

The composition of the local security policies is guided by potential threats. For example, to defend against *Class 1* attacks, *SDNShield* only needs to specify a security constraint to prevent an app to have *both* the packet-in/-out permission *and* the access to host network of the controller. This prevents the app from both the traffic sniffing/injection capacity and the ability to communicate with a remote attacker via host network. In real deployment, those security policies for specific threats can be distributed as templates, so as to lower the hurdle to have basic protection. For advanced protection, a GUI will be provided to facilitate the policy customization/composition process. In fact, such administration process is proven effective with the prevalence of the similar administration system for Enterprise Mobility Management (EMM), such as AirWatch [20] and Citrix XenMobile [21].

IV. Permission Abstractions

The heart of *SDNShield* is a domain specific permission language that defines the boundary of app behaviors.

Existing SDN permission systems typically model app permissions at the granularity of the APIs provided by the controller platform. For example, SE-Floodlight [11] can control whether an app can insert/modify a rule with `FLOW_MOD` permission. However, such API-level permissions are often too coarse-grained to specify a usable app permission. For example, it cannot make decisions based on the parameters of the API calls such as the switches, the subnet or the rule actions.

SDNShield enables fine-grained parameterized permission granting by introducing a two-level permission abstraction

[1] From the lesson of Android market, the app may crash or not function in the desired manner if the developer doesn't proactively handle the cases that requested permissions are denied.

Resource	Permission Token	Notes
Flow table	read_flow_table	
	insert_flow	Including insert and modify.
	delete_flow	
	flow_event	Get callback notification.
Topology	visible_topology	Specify partial or virtual topology.
	modify_topology	Change controller's view of physical topology.
	topology_event	
Statistics & Errors	read_statistics	
	error_event	
Pkt-in & Pkt-out	read_payload	Payload in pkt-in msg.
	send_pkt_out	
	pkt_in_event	
Host system	host_network	Network access outside control channel.
	file_system	
	process_runtime	Shell access, etc.

TABLE II: A subset of *SDNShield* permission tokens.

comprised of coarse-grained *permission tokens* and fine-grained *permission filters*.

A. Permission Token

Permission tokens are the coarse-grained app behavior privileges that can be either approved or denied for an app. We divide the set of SDN specific permission tokens along two dimensions, namely SDN resources and app actions. The SDN resources we consider range from network states, controller states to low-level control messages. For each resource, the app's potential actions include read, write and event notification. The permission tokens are designed orthogonal with each other, so that there is no inter-permission dependency. Other than interacting with SDN APIs, apps may also need to interact with the host machine OS via system calls. We design permission tokens to limit the system calls, which may expose apps to unnecessary attack surface. Table II shows a subset of our permission set.

B. Permission Filter

Permission tokens can only be granted in an all-or-none manner. In order to specify permissions in a finer-grained way, we introduce the *permission filter*, which is associated with a permission token to restrict its effective scope. For example, the app's `read_flow_table` permission can be restricted to a specific port number when associated with a `flow_predicate` filter as the parameter:

```
PERM read_flow_table LIMITING TCP_DST 80
```

Semantically, a permission token can be viewed as a boolean switch of a set of static APIs; while a filter is a boolean function of a runtime API call (along with all the arguments and the runtime context). We use the term *attribute* to refer to any of the runtime arguments or context of an API call. A filter splits the space of API calls into two subspaces by labeling each API call with ALLOW or DENY. In this way, a filter helps establish a middle state of a permission token by allowing a subset of API calls to pass through.

Next, we present the syntax and semantics of permission filters in detail.

a) Singleton Filter: Singleton filters are the building block of filter expressions. A singleton filter labels an API call according to a specific attribute of the API call. Different singleton filters inspect different attributes, and thus are independent of each other. Normally, an individual singleton filter is only effective to modify a subset of permissions that contain the specific attributes it inspects. *SDNShield* defines a number of singleton filters on several attributes as follows.

Flow filter. Flow filters act on flow-specific arguments of an API call. Thus, they can be associated with the permissions managing flow table resources. Based on the inspected attributes, flow filters include predicate filter, action filter, ownership filter, *etc*. As an example of flow filters, *predicate filter* compares the flow predicate in an API call with the value or the value range specified in the filter parameters, and only allows API calls with narrower predicates to pass through. A value range can be presented using a bit-wise mask:

```
PERM read_flow_table LIMITING \
IP_DST 10.13.0.0 MASK 255.255.0.0
```

this permission allows the app to see the flow entries targeting at the subnet 10.13.0.0/16. To provide more flexibility, *SDNShield* offers *wildcard filter*, which inspects the wildcard bits rather than the matched bits. When associated with `insert_flow` or `delete_flow` permission token, it forces the apps to set specific wildcard bits in their rules. For example, a load-balancing app that shuffles flows according to the lower 8 bits of IP_dst should have the permission:

```
PERM insert_flow LIMITING \
WILDCARD IP_DST 255.255.255.0
```

This permission specifies that the upper 24 bits wildcard of IP_dst of any issued rules must always be set, *i.e.,* the app can only identify flows using the lower 8 bits that are not wildcarded.

In addition, *action filter* inspects flow actions. *Ownership filter* inspects and keeps track of the issuers of all the existing flows. *Priority filter* limits the maximum/minimum priority value set in rules. *Table size filter* limits the maximum number of rules an app can put into a switch. *Packet out filter* can prevent apps from issuing packet-out with an arbitrary content. In sum, flow filters are useful to restrict apps' visibility or manipulability of flow table entries.

Topology filters. Topology filters help administrators to create abstract topology for apps. Topology filters allow two type of abstract topology. *Physical topology filter* helps to expose a subset of physical switches and links to an app. *Virtual topology filter* helps create the illusion of big switches, each of which is comprised of multiple physical switches:

```
PERM visible_topology LIMITING \
VIRTUAL SINGLE_BIG_SWITCH LINK EXTERNAL_LINKS
```

The above permission allows the app to see the topology as a single big switch by translating the topology information in the API request/response on the fly.

In addition to flow filters and topology filters, *SDNShield* also support a number of other filters. ***event callback filters*** inspect how an app interacts with an event notification. ***Statistics filter*** helps to restrict an app's visible statistics.

b) Filter Composition: While singleton filters are flexible in constraining a single attribute, the administrators may need to express complicated permissions with multiple attributes. For example, one might want to grant `read_flow_table` permission to an app, but only restricting to the flows previously issued by the app or the flows affecting the subnet $10.13.0.0/16$.

SDNShield fulfills this demand through filter composition. *SDNShield* supports the composition of singleton filters with logical operators including conjunction, disjunction and negation. Intuitively, the conjunction (disjunction) of two filters labels an API call true if and only if both operands (either operand) label(s) it true; while the negation of a filter always outputs the opposite label produced by the operand. The composition of filters enables *SDNShield* to describe the permissions with complicated logical conditions. Consider the above example, which could be expressed as

```
PERM read_flow_table LIMITING OWN_FLOWS OR \
IP_SRC 10.13.0.0 MASK 255.255.0.0 OR \
IP_DST 10.13.0.0 MASK 255.255.0.0
```

SDNShield permission language syntax is in Appendix A.

V. Security Policy Reconciliation

A. Security Policy Language

SDNShield security policy language provides abstractions for two types of constraints: *mutual exclusion* and *permission boundary*. To specify security policies, the administrator writes constraint statements that are comprised of mutual exclusion expressions and permission boundary assertions. For example, an enterprise network may wish to prevent a single app from possessing two mutually exclusive permissions: `insert_flow` and `network_access`. A second example is to bound the permissions from a tenant to a partial topology and/or a flow space.

In practice, a constraint statement, which describes a certain security property, is associated with a set of apps. The constraint enforcement engine evaluates the constraint expressions against the permissions of the associated apps, determining whether the permissions fully satisfy the constraints or not. The output includes a pass/fail label together with a message pointing out any constraint-violated permissions. After a constraint is set up on a set of apps, the constraint enforcement engine keeps track of permission updates and ensures the constraint expressions are satisfied persistently.

Mutual Exclusion*:* Mutual exclusion expressions specify a pair of permissions that should never be possessed by one single app. Mutual exclusion is often required to describe global security properties based on specific attack patterns. For example, a combination of `network_access` permission and `send_packet_out` permission could potentially enable the injection of arbitrary data-plane packets from a remote attacker. To prevent such combination, we can set them as a mutual exclusion:

```
ASSERT EITHER { PERM network_access } \
OR { PERM send_packet_out }
```

Semantically, this constraint prevents the two permissions from being possessed by a single app.

Permission Boundary*:* Permission boundary is a powerful tool to enforce static permission boundary and dynamic relations between multiple permission manifests. Permission boundary is implemented by checking logical assertions. In a logical assertion, permission expressions are computed and compared like sets using intersection, union, complement operations and inclusion relation. The semantics of the operations are naturally defined based on the allowed app behaviors. *SDNShield* supports set operations on permission expressions: intersection, union and complement. The semantics of the operations are defined similarly with the corresponding filter operations, namely conjunction and disjunction.

The basic building block of a logical assertion is the permission comparison expression. Thanks to the property of comparability of permissions, *SDNShield* offers two comparison operations between permission expressions: inclusion and equality. They can be used to express permission boundary. For example, we can specify the permission boundary for all monitoring apps to be no more than reading topology, reading port-level statistics and communicating with data collecting servers at 192.168.0.0/16:

```
LET templatePerm = {
  PERM read_topology
  PERM read_statistics LIMITING PORT_LEVEL
  PERM network_access LIMITING \
  IP_DST 192.168.0.0 MASK 255.255.0.0
}
ASSERT monitorAppPerm <= templatePerm
```

where the *less than or equal to* sign ($<=$) specifies the permission boundary, which defines the super set of the app behavior this app can possess.

Permission Customization*: SDNShield* allows two ways to customize the requested permissions. First, the app developer can leave permission filter stubs for customization purpose, *e.g.*, a stub macro named "administrative_IP_range". The administrator just need to redefine the macro to the actual value in the local environment. Second, the administrator can also restrict a specific permission by directly appending permission filters. For example, a cloud operator can restrict the operating topology of a tenant app by appending virtual topology filter to the requested permissions.

SDNShield security policy language syntax is in Appendix B.

B. Reconciliation Mechanism

To enforce the security policies, *SDNShield* needs a mechanism to evaluate the security policy expressions on concrete permission manifest, and modify the permission manifest in case of policy violations. In this subsection, we detail the reconciliation mechanism.

1) Permission Comparison: The evaluation of permission policies fundamentally depends on the comparison of the permission expressions. Abstractly, a permission expression can be viewed as a set of the allowed app behaviors. Since high-level permissions are orthogonal, the question on permission expressions can be converted to the same question on the filters associated with the same permission of the operands. For example, an insert_flow permission on a

192.168.0.0/16 IP_dst filter includes the same permission on a 192.168.1.0/24 IP_dst filter, due to the inclusion relation of the filters.

It is trivial to determine the relation of two singleton filters. In the presence of complex filters, however, things become a bit complicated.

Algorithm 1. *The inclusion relation of a pair of filters can be determined algorithmically.*

Without loss of generality, we assume we want to determine if Filter A includes Filter B. The algorithm is as follows:

Step 1: We first convert A to CNF, denoted as $(a \wedge b \wedge ...)$, and convert B to DNF, denoted as $(x \vee y \vee ...)$. In this way, the question is converted to determine whether every disjunctive clause $(a, b...)$ in A includes every conjunctive clause in B $(x, y...)$.

Step 2: We then scan over all possible pairs the disjunctive clauses in A and the conjunctive clauses in B. We denote one of the pairs as $a = a_1 \vee a_2 \vee ...$, $x = x_1 \wedge x_2 \wedge ....$ Consider the fact that filters on different dimensions are independent, hence cannot include each other. Thus, we know a singleton filter can only include another singleton filter on the same dimension of attribute. We match the singleton filters on the same dimension in a and x, and conduct singleton filter comparison. Therefore, we have a includes x if there exists $a_i \supset x_j$, otherwise a and x are independent.

2) Security Policy Reconciliation: SDNShield reconciles the permission boundary violation by conducting a conceptual intersection between the permission manifest and the permission boundary. Due to the set nature of permission expressions, we can naturally define set operations, including intersection, union and complement, on permission expressions. Permission operations are generalization of filter composition, and can be implemented using filter composition operations. Similarly, mutual exclusion violations are handled by truncating one of the exclusive permissions. And permission customization is implemented via a preprocessor that replaces stub macros with the administrator-supplied local permission settings. It is worths noting that by default *SDNShield* alerts administrators of any security policy violations, and the reconciled permissions are then offered for administrators' consideration.

VI. Permission Enforcement

A. Controller Isolation Architecture

In *SDNShield*, the existence of multiple principals with different privileges in a controller entails the demand for reference monitoring. Specifically, two types of actions, *i.e.,* the controller API access and the system calls to the host OS, should be reference monitored. Semantically, any reference monitoring paradigm that offers complete access control over both types of actions would suffice. While regarding performance, we want to minimize the latency and throughput degradation to the controller processing.

We compare the pros and cons of a number of reference monitoring paradigms in Table III. Among the candidates, isolating apps OS process container brings large context switching overhead to gain complete memory space isolation. Moreover, sandboxing OS process requires additional mechanism to mediate the interaction between a OS process and the OS kernel. Similarly, hardware virtualization (e.g., KVM) or OS-level virtualization (e.g., LXC) introduces even greater overhead to reference monitor the apps running as guest OSes or containers, since the communications between app and controller need to traverse OS kernel or even network socket. Compared with the above paradigms, programming-language-level sandbox (*e.g.*, Java VM and Python interpreter) provides a lightweight approach to offer necessary reference monitoring capacity.

	Ctrl API	System Access	Runtime Overhead	Impl'n Complexity
OS Process	✓	✗	Median	Low
HW Virt'n	✓	✓	High	Medium
PL Sandbox	✓	✓	Low	Median

TABLE III: Comparison of reference monitoring approaches. *SDNShield* opts for PL sandbox.

We design a general controller isolation architecture and embody it based on Java security model, as depicted in Figure 4. We propose to use sandboxed thread (e.g., Java thread) as the basic unit for sandboxing and privilege enforcement. In general, the app code runs in unprivileged threads and the trusted controller kernel runs in privileged threads. The thread container provides the following security properties:

- Control flow isolation. Threads isolation guarantees that a single thread isolation executes either trusted controller code or untrusted app code. The reference monitor (e.g., the customized Java SecurityManager) ensures that the app code cannot convert an unprivileged thread into a privileged one.
- Data isolation. Since the app/controller communication is now through only limited communication channels, it is easier to prevent apps from getting references to controller kernel object, therefore preventing apps' access or modification to controller states.
- System call mediation. Security sensitive system calls from apps are now mediated by the reference monitor (e.g., Java SecurityManager), thus can be restricted by security policies.

In *SDNShield*'s isolation architecture, both the controller threads and the app threads run on top of the sandbox shim layer (i.e., a Java VM). The *Kernel Service Deputy* (KSD) is located within the controller kernel acting as a choke point of all the communications between the controller kernel and the app threads. Another communication choke point is the access control point (i.e., the customized Java SecurityManager) located within the shim layer, which mediates system calls to the host OS. Note, the choke points do not mean serialized points. Actually, *SDNShield*'s isolation architecture is highly scalable regarding the number of apps by exploiting thread-level parallelization. Multiple instances of KSDs can run in parallel to offload the API requests from apps.

B. Permission Engine

Every API call issued by apps is eventually checked by the permission engine (PE). Generally, the PE reads the permission

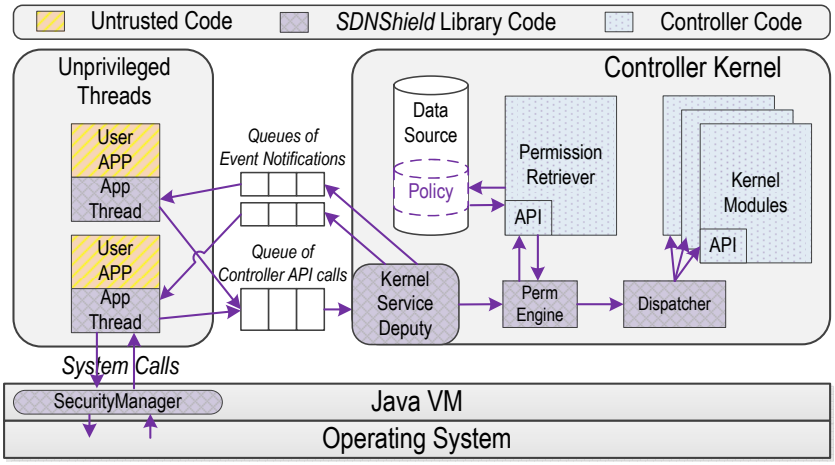

Fig. 4: *SDNShield* sandbox architecture.

manifest and compiles to the checking code to be executed at the reference monitor. We highlight two techniques of *SDNShield* in implementing the PE.

1) Evaluation of Abstract Filters: The first challenge is the semantic gap between the OF messages and the filter abstractions, *esp.*, the abstract topology filters. Controllers typically do not support abstract topology in their APIs. As a result, the reference monitor has to interpret the network view in realtime by overwriting the API calls and responses.

To fill the semantic gap, *SDNShield* actively maintains the topology mapping between the abstract topology and the physical topology, as well as the flow tables and the statistics of the virtual switches. When a flow rule is added to a virtual big switch, *SDNShield* find the corresponding physical switches and translate the flow rule to several physical rules that along the shortest path in the underlying physical topology. Statistic requests are handled similarly by querying multiple physical switches and aggregating the results.

2) Transactional API calls: Consider the scenario that multiple rules are to be installed, while one of them violates the permission. Conducting permission checking one by one may result in problematic intermediate state. *SDNShield* introduces API call transaction, which allows an app to group multiple semantically related API calls into a transaction and issue atomically. An API call transaction will only be executed if all the individual calls pass permission checking. In case of permission violation, the entire transaction will roll back and the app will be notified of the reasons for the failed API call.

C. Accommodating High-level SDN Languages

Today, the controller design increasingly moves towards rich northbound interfaces with higher-level abstractions and new programming paradigms. Hence, it is interesting to explore how the design of *SDNShield* can be adapted and applied to future SDN paradigms.

Novel Programming Paradigms: One of the major trends in SDN's northbound API is to provide app developers novel programming paradigms, such as the functional reactive programming [22], [23] and decision tree [24]. These programming paradigms typically expose SDN functionalities through explicit APIs, such as switch commands in Nettle [22] and rule insertion/invalidation in Maple [24]. *SDNShield* can leverage these explicit APIs to enforce access controls over the SDN specific resources; while system resources can still be enforced through programming-language-level sandbox.

Declarative Policy Languages: The other trend is to provide new abstractions of network resources through domain specific policy languages, such as Frenetic [25], Pyretic [26] and NetKAT [27]. Although the policy languages may contain novel abstractions, they are ultimately compiled to low-level OpenFlow instructions, where *SDNShield* access control can be enforced.

One of the challenges in enforcing *SDNShield* permission language is the difficulty to determine which app actually issues the OpenFlow instruction. In fact, the source app of an OpenFlow instruction can become ambiguous during compilation.

This challenge can be addressed by extending the compiler of the policy language and the ownership model of *SDNShield*. On the compiler side, we simply ask it to track the ownership information at a finer granularity during the policy composition process, and expose the information to *SDNShield*. Once *SDNShield* obtains the ownership information, it can split the rule and feed them to the permission engine respectively. Furthermore, we can extend *SDNShield* to allow an API access to be partially denied when some of the owner apps lack certain permissions. We leave the systematic solution as our future work.

VII. Application of SDNShield

In this section, we illustrate how *SDNShield* together with proper permissions prevents or mitigates the control-plane attacks with two hypothetical cases.

Scenario 1: Vulnerable Monitoring App: Consider a monitoring app deployed to supervise network usage of a tenant in a multi-tenant network. The app allows web connection from the administrator for management purpose. Further, we assume the app bears a vulnerability that allows arbitrary code execution.

The app release contains the following permission manifest:

```
PERM visible_topology LIMITING LocalTopo
```

```
PERM read_statistics
PERM network_access LIMITING AdminRange
PERM insert_flow
```

where two stubs, LocalTopo and AdminRange, are left for administrator to complete. The administrator supplies the security policy and local configurations as follows:

```
LET LocalTopo = {SWITCH 0,1... LINK 3,4...}
LET AdminRange = {IP_DST 10.1.0.0 \
MASK 255.255.0.0}
...
ASSERT EITHER { PERM network_access } \
OR { PERM insert_flow }
```

Next, the reconciliation engine extends the stub macros and verifies security policies. Then, it finds the mutual exclusion violation and reconciles by truncating the `insert_flow` permission. Finally, it generates the final permissions:

```
PERM visible_topology LIMITING \
SWITCH 0,1... LINK 3,4...
PERM read_statistics
PERM network_access LIMITING \
IP_DST 10.1.0.0 MASK 255.255.0.0
```

With this permission manifest, *SDNShield* can effectively mitigate the control plane attacks. The attacks through web interface will be blocked at the very first step by checking the source IP address. Even if the attacker launches the attack from an administrator's IP address, all the active attacks will still be blocked when the attacker tries to add rules or inject packets into the network. In fact, two of the four attack classes (*Class 1 and 3*) are completely prevented, since the monitoring app does not have the basic permissions to modify the network states. *Class 4* attacks are extremely difficult to launch because the app is executed in a sandbox where inter-app communication is disallowed unless it is through socket communication and the destination matches the AdminRange. The remaining attacks in *Class 2*, are also constrained to the leakage of the topology and statistics to the administrator-controlled IP addresses.

Scenario 2: Malicious Routing App*:* Consider the administrator deploys a routing app containing malicious code. The app implements shortest path routing in normal cases, but stealthily launches control-plane attacks at times. Assume the app is configured with the following permission:

```
PERM visible_topology
PERM flow_event
PERM send_pkt_out
PERM insert_flow LIMITING \
ACTION FORWARD AND OWN_FLOWS
```

SDNShield provides three levels of protection against the attacks from the malicious app. First, the routing app cannot communicate with the outside world, which implies the attacker cannot control the app over the network or obtain the sensitive information about the network. This property eliminates the possibility of *Class 2* attack. Second, the app cannot overwrite firewall rules or establish *dynamic-flow tunnel* [16] to bypass firewall rules, because it is not allowed to modify other app's rules. As a result, *Class 3 and Class 4* attacks are mostly prevented. To entirely prevent *Class 3 and Class 4* attacks, the administrator can further specify topology and flow space isolation with topology and flow filters. Third, although the routing app can still insert malicious rules and packet-out messages, the *SDNShield* can provide activity logging, which enables *forensic analysis* after the attack happens.

VIII. Implementation

A. Permission Engine and Reconciliation Engine

We implement a prototype of *SDNShield* policy engines as controller-independent Java bundles. Our *SDNShield* prototype implementation consists of three major components: i) a permission compiler that compiles permission manifests into abstract syntax trees (ASTs), ii) a reconciliation engine that enforces security policies on a permission AST, iii) a permission checking engine that check whether a specific API call is compliant with the given permission AST. The prototype consists of $\sim$23k lines of java code.

B. Isolation Architecture

a) Java-based Reference Monitor.: We implement a Java thread based isolation architecture that can be easily plugged into Java based SDN controllers with trivial changes on the controller and no changes on apps.

Inter-thread Communication and API Wrapper. Due to the thread isolation, the direct function calls between controller and apps now need to traverse through inter-thread communication channel. We implement a concurrent queue-based communication utility to facilitate the communication. In order to minimize the changes on the legacy system, we provide wrappers for service interfaces or listener interfaces so that the apps are transparent of the underlying changes in the communication channels. As a highlight, the wrappers are generated automatically according to the source code of service interface or listener interface with code transformation techniques, so that minimal human intervention is needed to support a new controller platform.

Controller and App Initiation. During the controller initiation time, we start all controller modules and a pool of kernel deputy threads with full privilege, and start an app thread with corresponding privilege of each app. Then, each app thread executes the app initiation code to configure initial states, obtain controller services and register listeners. We make necessary changes into the controller, so that the services and listener builders supplied to the apps are all wrapped. Hence the communication is enforced to traverse through inter-thread channel. Further, we ensure that all threads spawned from a unprivileged thread inherit their parents' privilege. During the lifetime of an app, all its API calls will traverse via inter-thread channel and received by a kernel deputy thread, which checks for permissions and, if permission allows, executes the API call on the behalf of the app.

We implement access control on top of Java SecurityManager, a customizable reference monitor provided by standard Java runtime. Furthermore, SecurityManager ensures that all system calls from apps are mediated by *SDNShield* permission engine. We extends Java SecurityManager to be able to check the customized permissions defined by *SDNShield*. Figure 4 depicts the architecture of the reference monitor implementation.

b) OpenDaylight Platform Support: OpenDaylight [5] is a Java-based community-led open source SDN controller platform supported by major SDN switch vendors. To enable *SDNShield*, we implement two extensions on OpenDaylight.

Architecture. OpenDaylight has a monolithic architecture, meaning the controller core and the apps runs in a single execution instance without runtime context separation. Since the boundary between the core controller functionalities and app modules is ambiguous on OpenDaylight, we draw a boundary according to our understanding to place our permission checking. We modify the app initialization process so that OpenDaylight apps runs in *SDNShield* thread container. The modification involves less than 100 lines of code on OpenDaylight source code.

We customize the API wrapper generation tool to adapt for OpenDaylight service interfaces and notification listener interfaces. We also implement a number of API converters in order to convert the OpenDaylight internal objects into the abstractions *SDNShield* permission engine can read. Specifically, a runtime API call is wrapped into a permission checking object, which contains the information including the caller app identity, the required permission and the parameters.

In addition to runtime access control, we also implement coarse-grained loading-time access control leveraging OSGi framework security. Specifically, OpenDaylight employs OSGi framework to dynamically load Java apps into the runtime. When an app is loaded, OSGi links the APIs that the app consumes with the modules that provide the APIs. We perform permission checking when OSGi loads the apps so that if no runtime permission checking is needed in case when the app does not have the required permission tokens at all.

Permission Checking. OpenDaylight employs model-driven northbound interfaces, i.e., most controller northbound interfaces are implemented as accessing the YANG data model. As a result, some *SDNShield* permission policies are enforced by mediating the read and write access to the YANG data model. Specifically, we extend the YANG data model so that sensitive nodes are associated with the necessary permissions required to read or write it. We also modify the data broker so that all data accesses are mediated by the permission engine with the associated permissions. We mediate event notifications and remote procedure calls (RPCs) at the kernel deputy thread, where necessary permission checking is performed.

c) Floodlight Controller Support: Floodlight [14] is an open-source Java-based OpenFlow controller. We also make extensions on the above two aspects to support *SDNShield*.

Architecture. Although Floodlight has a clear boundary between the controller core and the apps on code-level, it still runs monolithically. We make similar modification into Floodlight's app loading process to isolation Floodlight apps into *SDNShield* thread containers. We also customized the API wrapper generation tool for Floodlight.

Permission Checking. Different from the model-driven northbound interfaces of OpenDaylight, Floodlight opts for an API-style northbound interfaces. We also implement a number of API conversion functions to convert Floodlight API parameters into *SDNShield* abstractions.

IX. Evaluation

In this section, we evaluate the effectiveness and performance of *SDNShield* on *SDNShield*-enabled OpenDaylight SDN controller and Floodlight OpenFlow controller.

A. Methodology

We first evaluate the effectiveness of *SDNShield* permission engine and reconciliation engine with proof-of-concept malicious apps on Floodlight. Then, we focus on understanding the overhead brought by *SDNShield*. The same experiments are repeated on both Floodlight and OpenDaylight and exhibit qualitatively similar results. In the interest of space, we only show the runtime overhead results on OpenDaylight. We omit showing the reconciliation engine overhead, since it only occurs at the app installation time and the processing time never exceeds one second during our pressure tests.

The runtime overhead of *SDNShield* mainly stems from the permission checking and the asynchronous communication due to the thread isolation. The permission checking overhead mainly involves the permission evaluation and the book-keeping tasks. The asynchronism overhead is introduced by the asynchronous function calls via thread signaling, which leads to CPU context switching and potential waiting.

We use micro benchmarks to evaluate the permission checking throughput on a single CPU core. We then simulate two use scenarios to evaluate the end-to-end overhead of *SDNShield* on the controller runtime.

We deploy the controllers on an Intel Quad-core i7 3.40GHz workstation with 8GB RAM. We deploy a customized CBench [28] as our OF message generator hosted on another dedicated physical machine. We use OpenDaylight Lithium SR1 and Floodlight v0.90 as our baseline controllers. We simulate two use scenarios.

- L2 Learning Switch. The user network configures switch flow tables using the OpenDaylight *l2switch* app, which learns host position and generates switching rules by listening to OpenFlow packet-ins containing ARP packets. In this scenario, SDNShield checks permissions at the time of both the listener notification and the switching rule issuance.
- Traffic Engineering based on Application-Layer Traffic Optimization(ALTO) information. The *ALTO* app on OpenDaylight provides real-time topology and routing cost information to upper-layer apps. We write a simple traffic engineering (TE) app that listens to the ALTO app events and reacts with flow-mods that changes the routing paths. In this scenario, SDNShield checks permissions at the time of the listener notification to ALTO app, data model publication from ALTO app, data model event notification to TE app and routing rule issuance from TE app.

B. Experimental Results

1) Effectiveness: We first verify the effectiveness of the permission enforcement on four proof-of-concept malicious apps that conduct attacks as described in Section II. The first app monitors active flows by looking at packet-in messages and injects TCP RST to all active HTTP sessions. The second app collects network topology as well as switch/port configurations, and leaks out to outside attackers via HTTP

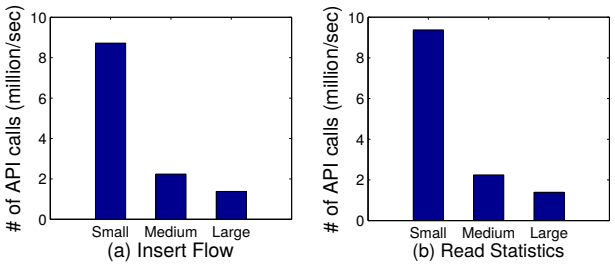

Fig. 5: Permission checking throughput on one core.

POST. The third app changes the existing routes between two hosts to traverse through a third host controlled by the attacker. The fourth app establishes a dynamic-flow tunnel through a firewall that only allows HTTP traffic at port 80. We run these apps on *SDNShield* with permissions in Scenario 1 described in Section VII and on original Floodlight. The results show original Floodlight is vulnerable to all the attacks, while *SDNShield*-enabled Floodlight is immune to all of them.

We also verify the effectiveness of the security policy reconciliation by checking over-privileged permission manifests with security policies generated based on the attack patterns. The results show that the over-privilege problem can be effectively prevented with appropriate security policies that cut off the apps' access to unnecessary resources. The only exception is that some apps (e.g., forwarding apps) essentially require access to the resources that enable certain attacks (e.g., insert forwarding rules), which is the inherent limitation of access control.

2) Permission Engine Performance: In this experiment, we evaluate the efficiency of the standalone *SDNShield* permission engine, i.e., how many the permission checks the engine can handle in a time unit on a CPU core. We measure the permission engine *throughput* with three manually generated permission manifests, which represent small, medium and large permission complexity. Three manifests respectively contain 1, 5 and 15 permission tokens, and each token is associated with 10-20 filters. The app behavior trace is a sequence of flow insertions and statistics requests that guarantees 5% of the API calls violate the permissions.

Figure 5 shows the permission checking throughput of two API calls (insert flow and read statistics) on a single core varying the complexity of the permission policies. The results show that permission checking latency is always less than one microsecond. Furthermore, since the permission checking is stateless, we can easily scale out the permission engine with parallelism.

3) End-to-end Performance: We then evaluate the overall overhead of *SDNShield* in the context of an OpenDaylight controller.We compare OpenDaylight controller with and without *SDNShield* in two scenarios. We measure the control-plane latency and throughput at the simulated switches by observing the issued packet-ins/topology changes and the received flow-mods.

End-to-end Latency*:* The overall latency introduced by *SDNShield* is dominated by the permission checking and the reference monitoring. Our evaluation shows that the total latency overhead introduced by *SDNShield* is around *tens of microseconds*, with varying app complexity and CPU core abundance. The latency overhead is two orders of magnitude smaller than the typical end-to-end latency in a data center network.

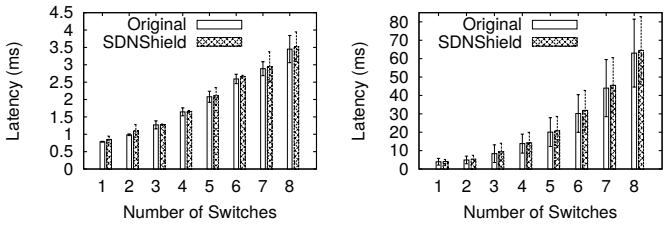

Fig. 6: Latency comparison. The bar and error bar show the median and the 10th/90th percentile of the measured end-to-end latency.

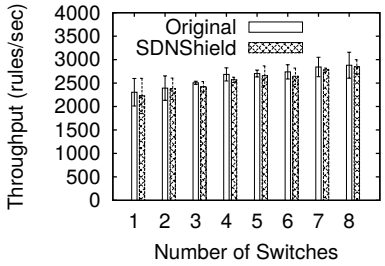

Fig. 7: Throughput comparison. The bar and error bar show the median and the 10th/90th percentile of the measured end-to-end throughput.

We conduct experiments to compare the overall control-plane latency of original OpenDaylight controller and *SDNShield*-enabled OpenDaylight controller in two use scenarios. Each experiment is repeated for 100 times. Figure 6 shows the results varying the number of switches in the network. The median result is shown as the bar and the 10/90 percentile results are shown as the error bar. We can see that the additional overhead introduced by *SDNShield* is almost unnoticeable in both experiments. Such results are confirmed by similar experiments on Floodlight controller.

End-to-end Throughput*:* We finally conduct pressure test to understand how much throughput penalty the thread isolation architecture of *SDNShield* brings, in comparison to a monolithic architecture of the original OpenDaylight. We conduct it on the L2 learning switch scenario, since the update throughput of ALTO app is limited to 2 updates per second due to unknown reasons. Figure 7 shows the results. From the figure we can see that *SDNShield* brings negligible throughput degradation compared to the original OpenDaylight controller. Such results are also confirmed by similar experiments on Floodlight controller.

Scalability*:* We evaluate the scalability of *SDNShield* to large networks and complicated apps. We collect the latency overhead varying the number of concurrent apps and the complexity of apps (measured by the API calls issued by the app). The results in Figure 8 demonstrates that the latency overhead of *SDNShield* increases linearly with the number of concurrent apps and the complexity of apps, thus *SDNShield* is highly scalable even if the number of concurrent apps and the complexity of individual apps grow in the future.

X. Related Work

a) Security on OpenFlow infrastructure: The demand for a more secure OF infrastructure has been partially addressed by prior research efforts. Network slicing [29],

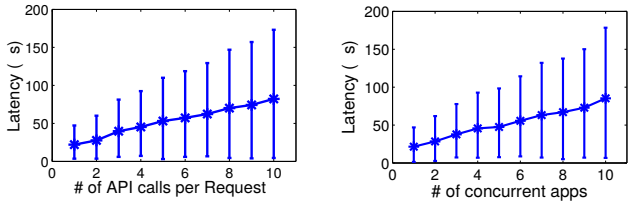

Fig. 8: Scalability of latency overhead with # of API calls per request and # of concurrent apps. The error bars show the 10th and 90th percentile. # of concurrent apps is fixed to 1 in the left subfigure; # of API calls per request is fixed to 1 in the right one.

[30], [31], [32] focuses on isolating control for disjoint traffic slices and prevents cross-slice attacks. In contrast, *SDNShield* delivers protection to not only disjoint network slices but also the apps that sequentially or collaboratively processing the same set of traffic. Network state analysis techniques [33], [16], [34] verifies network properties, such as reachability and middlebox traversal order, by analyzes network-wide flow rules. *SDNShield* could detect a broader set of control plane attacks and conduct access control with fine-grained permissions.

A few proposals also adopt the approach of permission control in SDN controllers [9], [10], [11], [12], [13]. Different from previous efforts, *SDNShield* is the first to provide the fine-grained parameterized permission abstractions and to introduce the reconciliation mechanism that customizes requested permissions with security policies.

b) Access control system design: The design of access control systems has been studied for decades. The large body of researches on design principles and practices [35], [36], [37] inspires our design of *SDNShield* mandatory access control system. Also, we learn important lessons from the recent studies on designing mobile OS permission systems. Particularly, some works design fine-grained permissions and enforcement mechanisms [38], [39], [40] for finer access control of general resource access. Other works [41], [42], on the other hand, identify that higher permission complexity may decrease users' ability and willingness to review the declared permissions. In comparison, *SDNShield* features the parameterized fine-grained permission abstractions for SDN environment. To provide better usability, *SDNShield* designs security policy reconciliation mechanism that generate fine-grained permissions by customizing declared permissions with local security policies.

c) High-level SDN languages and controllers: Today's OF controllers feature with low-level northbound interfaces that are closely coupled with the underlying switch hardware rather than the general network programming concepts. In contrast, intensive recent research efforts have been spent on identifying the "right" high-level abstraction for control plane software composition. These emerging proposals [23], [24], [25], [26] embrace a variety of programming paradigms and introduce abstractions on all levels of network programming entities, such as packet, network topology and composition operation. Specifically, Frenetic [25] designs different pieces of high-level language for state querying, packet forwarding and logic composition. Their following work, NetCore programming language [43], expands the query language and the packet-processing language, and provides formal semantics for the novel language. A parallel work Procera [23] embraces a fully reactive programming paradigm and introduces a different abstraction of the network resources. Pyretic [26] proposes dramatically different programming models for packet, network topology and policy operation.

All these SDN policy languages serve as vehicles to specify concrete network forwarding policies. In comparison, SDNShield permission language specifies the behavior privileges of SDN apps, and thus has different semantics and scope with the above SDN policy languages. Moreover, none of these works devote sufficient attention to potential control plane attacks. We believe *SDNShield* is orthogonal to the above works and we show that *SDNShield* can be adapted to provide access control to them. Although *SDNShield* does not currently have an implementation on any of the next-generation OF controllers, we think with reasonable engineering efforts, *SDNShield* can be adapted and deployed on these high-level abstractions of network programming.

XI. Conclusion

With the involvement of third-party apps, OF controllers are subject to the privilege abuse problem, which actuates a series of control-plane attacks that could compromise the entire network. To deal with the threats, we propose *SDNShield*, a privilege enforcement system comprising a fine-grained permission system and an automatic reconciliation system that merges requested permissions with local security policies. Our prototype implementation demonstrates the effectiveness and the acceptable efficiency of *SDNShield* compared with baseline.

Acknoledgement

This material is based upon work supported in part by the NSF under grant No. CNS-1219116 and by NSFC under grant No. 61472359, 61472209, 61432009, 60963021.

References

[1] Cisco ios ftp server remote exploit by andy davis. http://bit.ly/Yj21sO. Accessed: 7/30/2015.

[2] Mike lynn's 'exploit', in plain (non-technical) english. http://bit.ly/10zx5ou. Accessed: 7/30/2015.

[3] A remote cisco ios exploit. http://bit.ly/WAYG2V. Accessed: 7/30/2015.

[4] F. Lindner. Developments in cisco ios forensics. *Recurity Labs, White Paper*, 2008.

[5] OpenDaylight Platform. http://bit.ly/1HJi57D. Accessed: 7/30/2015.

[6] ONOS. http://bit.ly/1KrvSl9. Accessed: 7/30/2015.

[7] HP SDN App Store. https://hpn.hpwsportal.com/catalog.html. Accessed: 7/30/2015.

[8] SDN Security Attack Vectors and SDN Hardening. http://bit.ly/1KO0Zgc. Accessed: 7/30/2015.

[9] Xitao Wen, Yan Chen, Chengchen Hu, Chao Shi, and Yi Wang. Towards a secure controller platform for openflow applications. In *HotSDN'13*.

[10] Seungwon Shin, Yongjoo Song, Taekyung Lee, and etc. Rosemary: A robust, secure, and high-performance network operating system. In *CCS '14*.

[11] Phillip Porras, Steven Cheung, et al. Securing the Software-Defined Network Control Layer. In *NDSS*, 2015.

[12] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. Fresco: Modular composable security services for software-defined networks. In *NDSS'13*.

[13] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *USENIX NSDI'15*.

[14] FloodLight OpenFlow controller. http://bit.ly/UIL73z. Accessed: 7/30/2015.

[15] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A nice way to test openflow applications. In *USENIX NSDI'12*.

[16] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for openflow networks. In *HotSDN'12*.

[17] Ryan Johnson, Zhaohui Wang, Corey Gagnon, and Angelos Stavrou. Analysis of android applications' permissions. In *SERE-C 2012*.

[18] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *CCS 2012*. ACM.

[19] Wei Xu, Fangfang Zhang, and Sencun Zhu. Permlyzer: Analyzing permission usage in android applications. In *ISSRE 2013*. IEEE.

[20] AirWatch. http://www.air-watch.com/. Accessed: 7/30/2015.

[21] Citrix XenMobile. https://www.citrix.com/xenmobile/. Accessed: 7/30/2015.

[22] Andreas Voellmy and Paul Hudak. Nettle: Taking the sting out of programming network routers. In *PADL*, pages 235–249, 2011.

[23] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: a language for high-level reactive network control. In *Proceedings of HotSDN 2012*.

[24] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: simplifying sdn programming using algorithmic policies. In *Proceedings of the ACM SIGCOMM 2013*.

[25] Nate Foster, Arjun Guha, et al. Languages for software-defined networks. *Communications Magazine, IEEE*, 51(2):128–134, 2013.

[26] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In *USENIX NSDI'13*.

[27] Carolyn Jane Anderson, Nate Foster, et al. Netkat: Semantic foundations for networks. In *POPL '14*.

[28] Cbench controller benchmarker. http://bit.ly/YyICCs. Accessed: 7/30/2015.

[29] Rob Sherwood, Glen Gibb, et al. Can the production network be the testbed?

[30] Ali Al-Shabibi, Marc De Leenheer, et al. OpenVirteX: Make Your Virtual SDNs Programmable. HotSDN '14.

[31] Teemu Koponen, Keith Amidon, et al. Network virtualization in multi-tenant datacenters. In *NSDI*, 2014.

[32] Roberto Doriguzzi Corin, Matteo Gerola, et al. Vertigo: network virtualization and beyond. In *EWSDN*. IEEE, 2012.

[33] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Godfrey. Veriflow: verifying network-wide invariants in real time. In *USENIX NSDI'13*.

[34] Peyman Kazemian, Michael Chan, and etc. Real time network policy checking using header space analysis. In *NSDI*, pages 99–111, 2013.

[35] Peter A Loscocco, Stephen D Smalley, and etc. The inevitability of failure: The flawed assumption of security in modern computing environments. In *NISSC*, 1998.

[36] Ravi S Sandhu and Pierangela Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40–48, 1994.

[37] Pierangela Samarati and Sabrina Capitani de Vimercati. Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design*. Springer.

[38] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS*, volume 310, pages 20–38, 2013.

[39] Jinseong Jeon, Kristopher K Micinski, et al. Dr. android and mr. hide: fine-grained permissions in android applications. In *SPSM 2012*. ACM.

[40] Soteris Demetriou, Xiaoyong Zhou, et al. What's in Your Dongle and Bank Account? Mandatory and Discretionary Protection of Android External Resources. In *NDSS*, 2015.

[41] A.P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *WebApps*, 2011.

[42] A.P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *SOUPS*, 2012.

[43] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *ACM SIGPLAN Notices*, volume 47, pages 217–230. ACM, 2012.

APPENDIX A
SDNShield PERMISSION LANGUAGE SYNTAX

Permission

perm := *perm_expr* | *perm perm_expr*
perm_expr := `PERM` *perm* | `PERM` *perm* `LIMITING` *filter_expr*
filter_expr := *filter_expr* `AND`/`OR` *filter*
| `NOT` *filter_expr* | `(` *filter_expr* `)` | *filter*

Filter Categories

filter := *flow_f* | *topology_f* | *callback_f* | *statistics_f*
flow_f := *pred_f* | *action_f* | *owner_f* | *table_size_f*
| *priority_f* | *pkt_out_f*
pred_f := *field val* | *field val* `MASK` *val*
| `WILDCARD` *field val*
action_f := `DROP` | `FORWARD` | `MODIFY` *field*
owner_f := `OWN_FLOWS` | `ALL_FLOWS`
priority_f := `MAX_PRIORITY` `<INT>` | `MIN_PRIORITY` `<INT>`
table_size_f := `MAX_RULE_COUNT` `<INT>`
pkt_out_f := `FROM_PKT_IN` | `ARBITRARY`
topology_f := *phy_topo_f* | *virt_topo_f*
phy_topo_f := `SWITCH` *switch_set* `LINK` *link_set*
virt_topo_f := `VIRTUAL` *switch_map* `LINK` *link_set*
callback_f := `EVENT_INTERCEPTION` | `MODIFY_EVENT_ORDER`
statistics_f := `FLOW_LEVEL` | `PORT_LEVEL` | `SWITCH_LEVEL`

Helpers

field := `IP_SRC` | `IP_DST` | `TCP_SRC` | `TCP_DST` ...
val := `<INT>` | `<INT>.<INT>.<INT>.<INT>`
ip_fmt := `<INT>.<INT>.<INT>.<INT>`
switch_set := `{` *sw_idx* `,` ... `}`
switch_map := `{` *switch_set* `AS` *sw_idx* `,` ... `}`
link_set := `{` *link_idx* `,` ... `}`

APPENDIX B
SDNShield SECURITY POLICY LANGUAGE SYNTAX

Security Policy Language

expr := *binding* | *constraint*
constraint := `ASSERT` *exclusive* | `ASSERT` *assert_expr*
exclusive := `EITHER` *perm_expr* `OR` *perm_expr*
assert_expr := *assert_expr* `AND`/`OR` *boolean_expr*
| `NOT` *assert_expr* | `(` *assert_expr* `)`
| *boolean_expr*
boolean_expr := *perm_expr cmp_op perm_expr*
cmp_op := < | > | == | <= | >=
binding := `LET` *var_perm* = `{` *perm_expr* `}`
| `LET` *var_perm* = `APP` *app_name*
| `LET` *var_perm* = *perm_expr*
perm_expr := *perm_expr* `MEET`/`JOIN` *var_perm*
| `(` *perm_expr* `)` | *var_perm* | `{` *perm* `}`
var_perm := `<STRING>`
app_name := `<STRING>`