ParaSDN: An Access Control Model for SDN Applications based on Parameterized Roles and Permissions

Abdullah Al-Alaj
Institute for Cyber Security
C-SPECC
Department of Computer Science
UTSA, San Antonio
Texas, USA
abdullah.al-alaj@utsa.edu

Ram Krishnan
Institute for Cyber Security
C-SPECC
Department of Electrical and Computer Engineering
UTSA, San Antonio
Texas, USA
ram.krishnan@utsa.edu

Ravi Sandhu
Institute for Cyber Security
C-SPECC
Department of Computer Science
UTSA, San Antonio
Texas, USA
ravi.sandhu@utsa.edu

***Abstract*—Software Defined Networking (SDN) has become one of the most important network architectures for simplifying network management and enabling innovation through network programmability. Network applications submit network operations that directly and dynamically access critical network resources and manipulate the network behavior. Therefore, validating these operations submitted by SDN applications is critical for the security of SDNs. A feasible access control mechanism should allow system administrators to specify constraints that allow for applying minimum privileges on applications with high granularity. However, the granularity of access provided by current access control systems for SDN applications is not sufficient to satisfy such requirements. In this paper, we propose ParaSDN, an access control model to address the above problem using the concept of parameterized roles and permissions. Our model provides the benefits of enhancing access control granularity for SDN with support of role and permission parameters. We implemented a proof of concept prototype in an SDN controller to demonstrate the applicability and feasibility of our proposed model in identifying and rejecting unauthorized access requests submitted by controller applications.**

***Index Terms*—Software Defined Networking, Security and privacy, Access control, Formal models, Network security.**

I. Introduction and Motivation

Software-Defined Networking (SDN) has become one of the most important architectures for network management and, not surprisingly, helps shape how networks will be designed in the future. Decoupling the control logic of the network from the forwarding hardware has been determined as the core of SDN. This process led to the ability to control the network behavior via network apps, which often have to share the same infrastructure managed by a central controller, causing several security challenges, and access control is at the forefront of them.

Information about network resources stored in the SDN controller are highly valuable which makes it an attractive target for attackers and increases the potential to be gained via unauthorized access. Also, the potential damage that can be done increases dramatically if this unauthorized access is done by compromised, buggy, or malicious SDN apps.

In prior works on access control for SDN [1]–[3], an access control system was built based on a set of course grained permissions. For example, a permission to install a flow rule allows an app to handle any type of traffic. In other proposals that exploit the concept of roles for SDN [4], permissions are created based on object types rather than specific object instances. For example, a permission to read an object of type flow rule is generally used to read every flow rule in a switch. In another example, a permission to access a network device allows access to all network devices.

However, in real SDN deployments, a higher access control granularity is required, and there is often the need to assign permissions to a subset of object instances that share the same type. For instance, in a campus network it might be required for an app to access particular resources (e.g., switches, servers, etc.) that belong to a specific department only. Moreover, there is a need to assign permissions with higher granularity and make access decisions based on low level parameters derived from the request or contents of object itself (e.g., TCP protocol, VLAN id, IP address, Ethernet address, etc.).

This requires an access control system that restricts apps' access scope to unique object instances. An easy solution for this problem is to have an access control system in which a separate permission is created for each single object. However, adopting such approach in role-based systems has known problems and hard to manage as it requires creating and managing huge number of permissions, roles, permission-role associations, and app-role associations. A more feasible access control mechanism should allow the system administrator to flexibly specify the constraint that every app can only access and modify specific object instances commensurate to

978-1-7281-4146-6/20/$31.00 ©2020 IEEE
DOI 10.1109/CIC50333.2020.00022

its authorization requirements. Otherwise, apps may access resources not under their authority and thus conflicts may arise. For instance, apps that manage web services and require installing flow rules to handle web traffic should not be allowed to handle other traffic types.

Because of the known advantages of role-based access control especially in facilitating access control management, we propose ParaSDN, an access control model that addresses the above problems using the concept of parameterized roles and permissions.

This paper is organized as follows. In Section II, we discuss related work. Section III describes the concept of parameterized permissions and roles. Section IV describes the conceptual ParSDN model and its formal definitions. In Section V, we define app and permission assignment administrative actions. The framework architecture is described in Section VI. Section VII describes Parameter types in SDN. In VIII, we describe proof-of-concept use case and its configuration. Section IX discusses implementation and performance evaluation of the ParaSDN. Finally, Section X presents conclusion and outlines future work.

II. Literature Review

Several proposals on access control for SDN apps exist in the literature. [5]–[7] described the access control system in terms of the set of operations (APIs) as the basic unit for restricting app's activities. Although in works like [1]–[3], roles were assigned to apps, the operation (API) assigned to roles was too course grained leads to violating the least privilege principle to a large extent.

We classify SDN apps authorization into two main categories: firstly, permission-based app authorization which includes techniques wherein apps authorization is driven by direct permission-app assignment. Secondly, role-based app authorization in which app authorization is driven by permission-to-role followed by role-to-app assignment.

PermOF [5] proposed a permission system in which a permission set is directly granted to apps. The authors of [6] adopted the concept of PermOF. Inspired by Android permission system, [7] proposed a permission system based on OpenFlow messages' states that can be used as the unit to which the permission details can be applied. The authors in [8] introduced AEGIS using security access rules. Managing the aforementioned permission-based authorization systems is a widely known problem which we opt-out of. So we will only discuss role based authorization system for SDN.

FortNOX [1] implemented a role-based authorization system with three roles. FortNOX is extended and improved in SE-Floodlight [2]. In [9], a formal access control model for SDN apps based on SE-Floodlight as a reference controller is presented. SM-ONOS [10] proposed a permission system at four-level granularity. Based on API-level permissions from SM-ONOS, [11] proposed information flow control among apps for the ONOS controller. Tseng et al. [3], inspired by [2], proposed Controller-DAC with API request threshold and a priority for each app assigned either directly or via the role. These approaches have a limited granularity scope and don't provide an extensive and open fine grained capabilities. However, in our work we present a more convenient and flexible approach for creating a set of easy manageable roles built upon fine-grained and fully customized permissions that suit complex SDN use cases.

Works in [12]–[14] used the concept of parameterization with roles and privileges. However, their formalization is not well structured in a complete model, which make it hard to adopt in different contexts including SDN. In this work, we introduce a formal definition for parameterized roles and permissions that conform to the standard RBAC model and more flexible to adopt in a variety of environments including and beyond SDN.

III. ParaSDN Components Overview

In this section, we present an overview of the ParaSDN components and give examples of the syntax and semantics in the context of SDN environment.

A. Parameters

A *parameter* is a name:value pair that, when assigned to a permission, may indicate the subset of network resources that an app can access using this permission, or add restrictions on the performed operation. A parameter value can identify network resources in several ways. For example, It can usec (1) network resource IDs, for instance, a parameter 'attachment_point' can be assigned the value {0x1:1, 0x1:2, 0x2:1} to indicate the listed switch:port combinations; (2) a label that indicates a group of network resources, for instance, the label 'CS', when assigned to the parameter 'dept', may indicate all switch IDs in the CS department; (3) a property existing in the requested resource, for instance, the parameter 'traffic' with a value of 'web' indicates the set of ports used for Web protocol; or (4) a contextual property that restricts access to the resource, for instance, the parameter 'time_active' with the value of '9-17' may indicate the time frame during which an operation can be carried out.

The range of each parameter is represented by a finite set of atomic values. For example, the range of 'dept' is a set of department names that share the network infrastructure. Each parameter can either be atomic or set-valued from its declared range. For a particular parameter p, it is range is composed only from those values defined by the system administrator. Different SDN-related parameter types with examples are discussed in section VII.

B. Parameterized Permissions

A *parameterized permission* is represented by the ordered pair:

$((\text{op}_i, \text{ot}_i), \{(\text{par}_1, \text{val}_1), (\text{par}_2, \text{val}_2), ...\})$

where $(\text{op}_i, \text{ot}_i)$ combines a network operation with an object type in the ordinary permission format, and $\{(\text{par}_1, \text{val}_1), (\text{par}_2, \text{val}_2), ...\}$ is a subset of parameter:value pairs. In the parameterized permission, the object type ot_i indicates all object instances of that type on which operation op_i can be exercised.

If used alone, it provides a very course-grained access privilege and impractical for many SDN security policies. In many situations, what is required is to provide access to subset of object instances of that type. This is achieved with the help of the parameters associated with this permission. The semantics of this parameterized permission is that an app can execute the operation op_i on only object instances of type ot_i that satisfy the restrictions imposed by the parameter values.

The values of parameters in a permission are not assigned at the time of permission creation; instead, their values remains unknown until the permission is associated with a parameterized role whose parameter values already defined, i.e., permission parameters' values are steered by the values of role parameters. So, when security architects create a parameterized permission, they initialize parameter values with a special value $\perp$, which means unknown. For example, the parameterized permission:

((addFlow, FLOW-RULE), {(dept, $\perp$), (traffic, $\perp$)})

indicates that an app can insert flow rules in switches of as-yet-unknown department(s), and these rules can handle traffic of as-yet-unknown type. If the values of parameters 'dept' and 'traffic' are 'CS' and 'web', then an app can add flow rules that handle Web traffic in switches of CS department.

C. Parameterized Roles

A *parameterized role* is represented using an ordered pair:

(r_i, {(par_1, val_1), (par_2, val_2), ...})

where r_i represents a role name, and {(par_1, val_1), (par_2, val_2), ...} is a set of parameter:value pairs. Initially, all role parameters are assigned a special value $\perp$, which means unknown. For example,

(Flow Mod, {(dept, $\perp$), (traffic, $\perp$)})

is a parameterized role that includes permissions to read, update, insert, and delete flow rules in switches of as-yet-unknown department(s), and these rules can handle traffic of as-yet-unknown type. If the values of parameters 'dept' and 'traffic' are 'CS' and 'web', then an app can exercise these operations only to flow rule instances that reside in switches of CS department and handle traffic destined to Web servers.

D. Parameter Value Assignment

At the time of role engineering, there is no need to worry about actual parameter values at the level of permissions and roles. As mentioned above, parameterized permissions and parameterized roles are instantiated with parameter values assigned a special value $\perp$, which means unknown.

A parameterized permission is assigned to a parameterized role via the administrative action *assignPPerm*(pp, pr), where *pp* is a parameterized permission and pr is *a* parameterized role. At this step, no actual parameter values are assigned. This is demonstrated in step 1 of Fig. 1 (a). Because parameter values are assigned based on the requirements for an app to access system resources, their values will remain unknown until actual app-to-role assignment is executed via *assignApp*$(a, pr, valset)$, where *a* is an app, *pr* is a parameterized permission, and *valset* is the set of values to be supplied to *pr*. The values in *valset* propagates automatically to corresponding permission parameters. This app-to-role assignment and value propagation is demonstrated in steps 2 and 3 of Fig. 1 (a). The final state of the parameterized role and parameterized permission as associated with app *a* is shown in Fig. 1 (b).

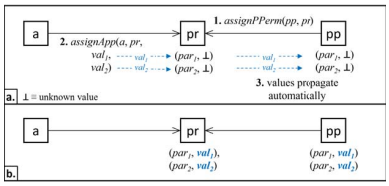

Fig. 1. Parameter values assigned via assignApp administrative action propagate automatically from role parameters to permission parameters.

E. Parameter Verification

We consider an app's access request to an object as a right of access claim by that app to that object. This claim requires verification by the access control system. We use specific functions, called Verifiers, to check the validity of this claim by comparing the parameter values in the actual access rights of the app (i.e., the available parameterized permissions of the session) with the properties of the requested object.

For example, a verifier VRuleSwitch will be called after exercising the permission ((addFlow, FLOW-RULE), (dept, CS), (traffic, web)). It is used to verify that a flow rule that is being submitted by an app for insertion is to be inserted in an authorized switch, i.e., in switches if the CS department. The verifier exploits information from the object, i.e., the flow rule, and parameter values from the parameterized permission, i.e., CS department. If the accessed switch is within the switches of CS department, a positive response is returned, otherwise the verifier returns negative response.

It worth mentioning that one verifier can serve multiple parameterized permissions. For example, the same verifier VRuleSwitch will be called with the permission ((deleteFlow, FLOW-RULE), (dept, CS), (traffic, web)). Associating one verifier with multiple parameterized permissions reduces the management effort when dealing with large number of permissions. Also, one parameterized permission might require multiple verifiers. For example, another verifier that will be invoked for any of the above parameterized permissions is VRuleTraffic which verifies that the accessed flow rule handles corrcet traffic type, i.e., web traffic. The verifiers must be called for one parameterized permission depends on the parameters associated with the permission.

IV. ParaSDN Conceptual Model and Definition

The conceptual model and the relations between the components of ParaSDN are shown in Fig. 2. ParaSDN has the following basic components: OpenFlow apps APPS, roles ROLES, operations OPS, objects OBS, object types OBTS, the parameter set PAR, and the set of parameter values VAL.

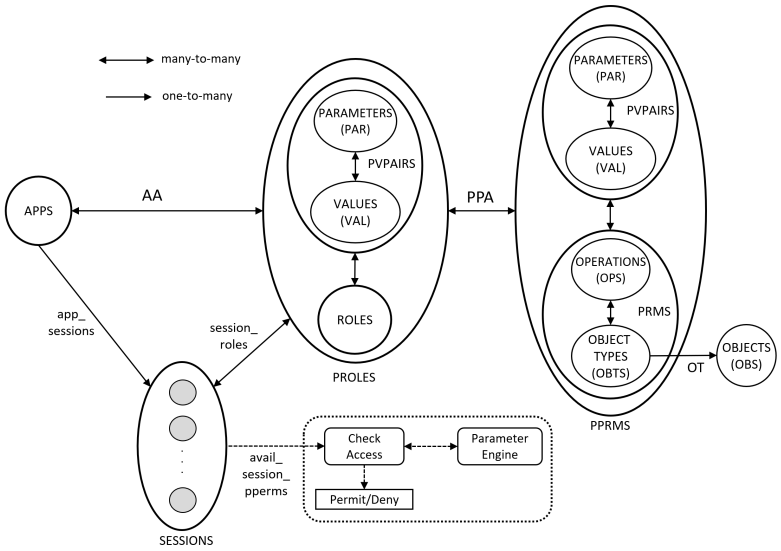

Fig. 2. ParaSDN Conceptual Model.

The basic sets and functions in ParaSDN are shown in Table I. APPS refer to the set of OpenFlow apps. ROLES is the set of role names. OPS is the set of all operations exposed to apps by the controller services and performed on objects. Inserting flow rules and reading port statistics are examples of operations. OBS is the set of object instances that are managed by the controller and should be protected from unauthorized access. They are managed by the controller to maintain a consistent state of the network infrastructure. An element in OBTS represents the type of a specific object instance. For example, FLOW-RULE, DEVICE, and LINK refer to the type of actual instances of flow rules, devices, and links respectively.

PAR represents the set of all parameters in the system. This could be atomic or set valued as determined by the type of the parameter. Type of a parameter, set or atomic, is specified by the function parType. VAL is the set of all parameter values used in the system. PRMS is the set of permissions, where a permission combines a network operation with an object type. The set SESSIONS represents a mapping between an app and an activated subset of parameterized roles. An app can have multiple sessions and a session belongs to only one app. OT is a relation for the combinations between objects and their types. PVPAIRS is a subset of parameter:value pairs. PPRMS defines the set of parameterized permissions as discussed in Section III-B. PROLES defines the set of parameterized roles as discussed in Section III-C.

The functions required for parameter verification are defined in part 4 of Table II. VERIFIERS is a set of boolean functions defined by security administrators for parameter verification. Each $V_i \in$ VERIFIERS is applied on an object and a parameter to check whether an object satisfies the requirements of the parameter. Param_verifier is a function that returns a verifier that needs to be executed at the time of access request. It maps an (object type, parameter) pairs to their related verifier.

In our model, parameter checking and verification process is an essential part of evaluating each session's access request. It requires different components to communicate as illustrated in Table II. Security administrators firstly need to define a parameter verification function V_i (or so-called a verifier) that must be executed to find whether an object fulfills the requirements of a parameter. Verifiers are defined by means of the language LVerify defined in Table III. The language LVerify allows to create conditions that involve parameter values and information about the object. In this language, ConsSet and ConsAtomic are constant sets and atomic values.

Because not all the verifiers need to be executed for a requested object, security administrators need to specify the subset of verifiers applicable to the requested object and the permission parameters. An access request might need to execute multiple verifiers depending on the parameters associated with the parameterized permission undergo the check. At the time of access request, the function CandidateVerifiers receives all parameters associated with a parameterized permission need to be checked. This function is responsible of retrieving the set of applicable verifiers and submitting this set to the function ParamCheck for evaluation. In order to do this, it passes the object type along with each parameter to param_verifier function that retrieves the applicable verifier.

It should be mentioned that the function CandidateVerifiers doesn't deal with the parameter values or the object instances themselves, however, it relies on the parameter name and the object types to fetch the relevant verifiers. On the other hand, the verifiers returned by CandidateVerifiers use information about actual object and actual parameter values for evaluation.

The function ParamCheck receives the applicable verifiers for the object and verifies if the object can be accessed based on the provided parameter values. It achieves this by invoking

TABLE I
PARASDN FORMAL MODEL DEFINITION.

1.Basic Sets:

- APPS, ROLES, OPS, OBS, OBTS, PAR, and VAL: set of apps, roles, operations, objects, object types, parameters, and parameter values.
- For each *par* ∈ PAR, Range(*par*) represents the parameter's range, a finite set of atomic values. We assume VAL includes a special value "⊥" to indicate that the value of a parameter is unknown.
- parType: PAR → {set, atomic} specifies parameter type as set of atomic valued.
- PRMS ⊆ OPS × OBTS, set of ordinary permissions.
- SESSIONS, set of sessions.

2.Assignment Relations:

- OT ⊆ OBS × OBTS, a many-to-one relation mapping an object to its type, where $(o, ot_1) \in OT \wedge (o, ot_2) \in OT \Rightarrow ot_1 = ot_2$.
- PVPAIRS ⊆ PAR × VAL, a many-to-many mapping parameter to value assignment relation. For convenience, for every pvpair = (par_i, val_i) ∈ PVPAIRS, let pvpair.par = par_i and pvpair.val = val_i.
- PPRMS ⊆ PRMS × $2^{PVPAIRS}$, a relation mapping a permission role to subset of (parameters , value) combinations. For convenience, for every pp = ((op_i, ot_i), $PVPAIRS_i$) ∈ PPRMS, let pp.op = op_i, pp.ot = ot_i, and pp.PVPAIRS = $PVPAIRS_i$.
- PROLES ⊆ ROLES × $2^{PVPAIRS}$, a relation mapping a role to subset of combinations of parameters and their values. For convenience, for every pr = (r_i, $PVPAIRS_i$) ∈ PROLES, let pr.r = r_i and pr.PVPAIRS = $PVPAIRS_i$.
- PPA ⊆ PPRMS × PROLES , a many-to-many mapping parameterized permission to parameterized role assignment relation.
- AA ⊆ APPS × PROLES, a many-to-many mapping app to parameterized role assignment relation.

3.Derived Functions:

- assigned_pperms: PROLES → 2^{PPRMS}, the mapping of parameterized role into a set of parameterized permissions. Formally, assigned_pperms(pr) = {pp ∈ PPRMS — (pp, pr) ∈ PPA}.
- app_sessions: APPS → $2^{SESSIONS}$, the mapping of an app into a set of sessions.
- session_app : SESSIONS → 2^{APPS}, the mapping of session into the corresponding app.
- session_roles: SESSIONS → 2^{PROLES}, the mapping of session into a set of parameterized roles. Formally, session_roles(s) = {pr ∈ PROLES — (session_app(s), pr) ∈ AA}.
- type: OBS → OBTS, a function specifying the type of an object defined as type(o) = {t ∈ OBTS — (o, t) ∈ OT}.
- avail_session_pperms: SESSIONS → 2^{PPRMS}, the parameterized permissions available to an app in a session. Formally, avail_session_pperms(s) = $\bigcup_{pr \in session_roles(s)}$assigned_pperms(pr).

4.Parameter Verification Functions:

- VERIFIERS = {V_1, V_2, ..., V_n} a finite set of Boolean functions. For each V_i ∈ VERIFIERS.V_i : SESSIONS × OPS ×OBS × PVPAIRS → {True, False}.
- param_verifier: OBTS × PAR → VERIFIERS, a function that maps a combination of object type and parameter to the corresponding verification function needs to be evaluated.

TABLE II
PARAMETER CHECKING FUNCTIONS.

A. Verifiers:
Language LVerify is used to define each verifier V_i(*s: SESSIONS, op*: OPS, *ob*: OBS, pvpair : PVPAIRS) in VERIFIERS.

B. CandidateVerifiers: a function that maps each object type to its applicable set of verifiers.

```
CandidateVerifiers(ot: OBTS, pvpairs : 2^PVPAIRS){
    verifiers = {};
    For each pvpair_i ∈ pvpairs do
        V_i = param_verifier(ot, pvpair_i.par);
        verifiers := verifiers ∪ {(V_i × pvpair_i)};
    return verifiers;
}
```

C. ParamCheck: a function that checks an object against all candidate verifiers until the first failure is discovered or a true is returned as the final outcome.

```
ParamCheck(s: SESSIONS, op: OPS, ob: OBS, pvpairs: 2^PVPAIRS){
    For each (V_i × pvpair_i) ∈ CandidateVerifiers(type(ob), pvpairs) do
        if ¬V_i(s, op, ob, pvpair_i)
            return false;
    return true;
}
```

TABLE III
LANGUAGE LVERIFY TO FORM VERIFIERS.

$\varphi ::= \varphi \wedge \varphi \mid \varphi \vee \varphi \mid (\varphi) \mid \neg\varphi \mid \exists x \in \text{set}.\varphi \mid \forall x \in \text{set}.\varphi \mid \text{set setcompare set} \mid \text{atomic} \in \text{set} \mid \text{atomic atomiccompare atomic}$

setcompare ::= $\subset \mid \subseteq \mid \not\subseteq$

atomiccompare ::= $< \mid = \mid \leq$

set ::= setpar.val | ConstSet
atomic ::= atomicpar.val | ConstAtomic
setpar ∈ {pvpair | pvpair ∈ PVPAIRS ∧ parType(pvpair.par) = set}
atomicpar ∈ {pvpair | pvpair ∈ PVPAIRS ∧ parType(pvpair.par) = atomic}

TABLE IV
APP AUTHORIZATION FUNCTION.

Function	Authorization Condition
checkAccess(s: SESSIONS, op: OPS, ob: OBS)	$\exists pr \in PROLES : pr \in session\_roles(s), \exists pp \in PPRMS : (pp, pr) \in PPA \wedge (op, type(ob)) = (pp.op, pp.ot) \wedge$ **ParamCheck**(s, op, ob, pp.PVPAIRS) = True.

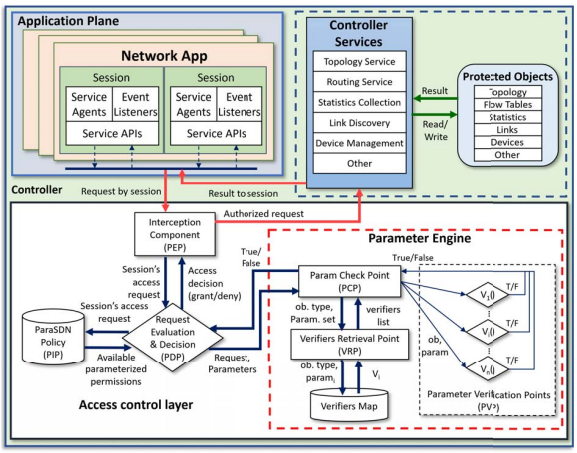

Fig. 3. General Overview of the proposed system components and Architecture

the verifiers one by one. For finding the final outcome of session's access request, the system function CheckAccess is used. This function is formally defined in Table IV. As part of the final decision, it invokes the function ParamCheck to evaluate the compliance of the object with the permission parameters. It is responsible of returning the final decision whether an app's session is or is not allowed to perform a given operation on a given object.

V. Administrative Actions for App and Permission Assignment

The specification of a complete list of administrative functions is out of the scope of this paper. We only show two administrative functions *assignApp*$(a, pr, valset)$ and *assignPPerm*(pp, pr) due to their relation to the parameter values as described in Section III-D. The formal specification of these two administrative functions is shown in Table V. The function *assignApp*$(a, pr, valset)$ assigns an app a to a parameterized role pr and assigns the values in $valset$ to parameters in pr. The values in $valset$ propagates automatically to the corresponding permission parameters in every parameterized permission pp associated with pr. A parameterized permission pp can be assigned to a role parameterized pr via the *assignPPerm*(pp, pr) function.

VI. Framework Architecture and Parameter Engine Components

In this section, we show how ParaSDN framework is designed to integrate role parameters in the decision process, and then we elaborate on how it works by presenting its operational scenario.

A. ParaSDN Framework Architecture

As shown in Fig. 3, ParaSDN consists of four main components: (1) Policy Enforcement Point (PEP), (2) Policy Decision Point (PDP), (3) Policy Information Point (PIP), and (4) Parameter Engine. The General functionality of the Parameter Engine itself is distributed among multiple components, namely, Parameter Check Point (PCP), Verifiers Retrieval Point (VRP), and multiple Parameter Verification Points (PVPs). These components function together to provide parameter evaluation essential for generating an access control decision fundamental for security policy enforcement.

When an app's session submits an access request, the authorization flow involves intercepting the session's access request by the PEP, passing the request to the PDP, querying the PIP to get the parameterized permissions available for the session, and finally calling the Parameter Engine for parameter verification. It is the PDP's decision to involve the Parameter Engine in the authorization process or not. If the ordinary permission in the parametrized permission doesn't match with the sessions request, the PDP denies access, otherwise the request is passed to the Parameter Engine for further evaluation.

B. ParaSDN Parameter Engine

The first component of the Parameter Engine is the PCP. It represents a central point in the Policy engine. It is responsible of receiving the object and the permission parameters and verifying if the object can be accessed based on the provided parameters' values. In order to do this, the PCP must check if the requested object complies with the requirements of each and every parameter associated with the permission. This is done by invoking candidate verifiers each represents a parameter verification point (PVPs).

Each PVP is a boolean expression designed by the security administrator to verify if an object satisfies the requirement of the parameter. In other words, each PVP receives an object and a parameter and evaluates the session's right to access the object based on the parameter value. If any PVP returns FALSE, which means that the requirements of that parameter is not satisfied, the PCP stops the whole parameter verification process and returns false to the PDP. On the other hand, the PCP will return true if and only if the object satisfies all the perimeter requirements, i.e., all PVPs return TRUE. The PDP logic relies on this result to allow or deny access to the requested object.

Before the PCP calls any PVP, it need to specify the subset of PVPs need to be invoked. We design the VRP as responsible

TABLE V
FORMAL SPECIFICATION OF ASSIGNAPP(A, PR, VALSET) AND ASSIGNPPERM(PP, PR) ADMINISTRATIVE FUNCTIONS.

Function	Authorization Condition	Update
assignPPerm(pp, pr)	pp ∈ PPRMS ⋀ pr ∈ PROLES ⋀ (pp, pr) ∉ PPA	PPA' = PPA ⋃ {(pp, pr)}

assignApp(a, pr, valset)	a ∈ APPS ⋀ pr ∈ PROLES ⋀ valset ∈ VAL ⋀(a, pr) ∉ AA	//Assign values to role parameters.
		For each pr_pvpair$_i$ ∈ pr.PVPAIRS, v$_i$ ∈ valset, $1 \leq i \leq$ —pr.PVPAIRS— **do**
		pr_pvpair$_i$.val = v$_i$
		//Pass parameter values from pr to its member parameterized permissions.
		For each pp ∈ PPRMS : (pp, pr) ∈ PPA **do**
		For each pr_pvpair$_i$ ∈ pr.PVPAIRS, pp_pvpair$_i$ ∈ pp.PVPAIRS, $1 \leq i \leq$ —pr.PVPAIRS— **do**
		pp_pvpair$_i$.val = pr_pvpair$_i$.val
		AA' = AA ⋃ {(a, pr)}

TABLE VI
EXAMPLES FOR FLOW-DRIVEN PARAMETERS FOR SDN.

Parameter	Description
tcp_src, tcp_dst	TCP source/distination port
udp_src, udp_dst	UDP source/distination port
vlan_id	VLAN id
ip_proto	IP protocol
ipv4_src, ipv4_dst	IPv4 source/distination address
ipv4_src_mask, ipv4_dst_mask	IPv4 source/distination subnet mask

of identifying these PVPs and submitting them to the PCP. The VRP does this by referring to the VerifiersMap which maps pairs of object type and parameter to their applicable PVP.

c

VII. PARAMETER CATEGORIES FOR SDN

We identify four categories of parameters that can be used with parameterized roles and permissions for SDN environment.

1. Topology-specific parameters: parameters to identify subsets of network switches, links, or ports. For example, the set-valued parameter switch_id with a value of 00:00:00:00:00:00:00:01 assigned to a role Topology-Visualizer restricts role holders from accessing other switches.

2. Flow-driven parameters: represent parameters to identify flow rules. They can be supplied to roles (e.g., 'Flow Mod') that authorize access to objects of type FLOW-RULE. For example, parameter tcp_dst assigned a value of 80 will identify all flow rules that manipulate traffic destined to an HTTP server. A parameter ipv4_dst_mask assigned a value of 192.168.5.0/24 identifies flow rules targeting this subnet. i.e., targeting IP addresses in the range 192.168.5.0 - 192.168.5.255 that has subnet mask of 255.255.255.0. Table VI shows examples of Flow-driven parameters.

3. Application-specific parameter: This parameter represents an app_id. It is supplied to roles to identify particular app that will operate using this role. For example, assume the parameter app_id is supplied to role 'Pool Manager' and app_id is assigned the value "Load Balancer" (assuming "Load Balancer" is an app ID for a load balancer app), this means that this role can operate only by "Load Balancer" app. Every time a request is submitted by a session using this role, a verifier function VApp_id should verify that session_app(s) = app_id('Pool Manager'), i.e., session_app(s) = "Load Balancer". Assuming this session is compromised by an app MalApp, this makes session_app(s) = MalApp. As a result, any request using this session will not be granted because the verifier VApp_id will fail since the check session_app(s) = app_id('Pool-Manager') will return false because the parameter value 'Pool Manager' is attached as the parameter value in the parameterized role. This requires sending the session id as parameter to the verifier function in order to use session_app(s) in the evaluation process which is already depicted in the formal model in Table I.

4. Organization-specific parameters: They represent parameters pertaining to internal organizational structure such as divisions and departments operating internally at some level in the organization hierarchy. For example, a parameter dept with the value of CS or CE associated with a 'Flow Mod' role identifies network resources that can be accessed by apps operating under Computer Science or Computer Engineering departments, respectively. These resources might include set of switches, ports and links under the authority of specific department. The interpretation of the organization-specific parameters and the resources associated with them is an internal organization issue. In another example, a parameter tenant with the value tenant1, authorizes an app to access tenant1 resources.

VIII. PROOF OF CONCEPT USE CASE

In this section we demonstrate and configure a use case in ParaSDN. Assume in a small campus network we have the network infrastructure as depicted in Fig. 4. The infrastructure is divided between two departments CS and CE. Assume CS dept independently manages two switches, 0x1 and 0x2 and the four hosts connected to them. Host-3 runs a web server. The CE department separately manages one switch 0x3 and two hosts host-5 and host-6. Host-5 runs a web server. Hosts 1-4 are assigned vlan_id=1, and hosts-5 and Host-6 are assigned to vlan_id=2. Switches are connected to one controller. The controller has two apps, one for each Department. 'Data

TABLE VII
CONFIGURATION OF THE PROOF OF CONCEPT USE CASE OF SECTION VIII IN PARASDN (PART1).

1. Model Basic Sets:
– APPS = {Data Usage Cap Mngr, Intrusion Prevention App}.
– ROLES = {Device Handler, Bandwidth Monitoring, Flow Mod, Packet-In Handler}.
– OPS = {queryDevice, getBandwidthConsumption, addFlow, readPacketInPayload}.
– OBS = D ∪ PS ∪ FR ∪ PIP, where D = set of all network devices, PS = set of all port statistics in all switches, FR = set of all flow rules, and PIP = set of all packet-in messages.
– OBTS = {DEVICE, PORT-STATS, FLOW-RULE, PI-PAYLOAD}.
– PAR = {vlan_id, attachment_point, dept, traffic}.
– Range(vlan_id) = {1, 2}. Range(attachment_point) = {0x1:1, 0x1:2, 0x2:1, 0x2:2, 0x3:1}. Range(dept) = {CS, CE}. Range(traffic) = {web}.
– parType(vlan_id) = atomic. parType(attachment_point) = set. parType(dept) = set. parType(traffic) = atomic.
– PRMS = {(queryDevice, DEVICE), (getBandwidthConsumption, PORT-STATS), (addFlow, FLOW-RULE), (readPacketInPayload, PI-PAYLOAD)}.
– SESSIONS = {DataUsageAnalysisSession, DataCapEnforcingSession, IntrusionPreventionSession}.
2. **Assignment Relations:**
– OT = {(d, DEVICE) : d ∈ D} ⋃ {(ps, PORT-STATS) : ps ∈ PS} ⋃ {(fr, FLOW-RULE) : fr ∈ FR} ⋃ {(pip, PI-PAYLOAD) : pip ∈ PIP}}.
– PPRMS = {((queryDevice, DEVICE), {(vlan_id, ⊥)}), ((getBandwidthConsumption, PORT-STATS), {(attachment_point, ⊥)}), ((addFlow, FLOW-RULE), {(dept, ⊥), (traffic, ⊥)}), ((readPacketInPayload, PI-PAYLOAD), {(attachment_point, ⊥)})}
– PROLES = {(Device Handler, {(vlan_id, ⊥)}), (Bandwidth Monitoring, {(attachment_point, ⊥)}), (Flow Mod, {(dept, ⊥), (traffic, ⊥)}), (Packet-In Handler, {(attachment_point, ⊥)})}
– PPA = {(((queryDevice, DEVICE), {(vlan_id, ⊥)}), (Device Handler, {(vlan_id, ⊥)})), (((getBandwidthConsumption, PORT-STATS), {(attachment_point, ⊥)}), (Bandwidth Monitoring , {(attachment_point, ⊥)})), (((addFlow, FLOW-RULE), {(dept, ⊥), (traffic, ⊥)}), (Flow Mod, {(dept, ⊥), (traffic, ⊥)})), (((readPacketInPayload, PI-PAYLOAD), {(attachment_point, ⊥)}), (Packet-In Handler, {(attachment_point, ⊥)}))}.
– AA = {(Data Usage Cap Mngr, (Device Handler, {(vlan_id, 1)})), (Data Usage Cap Mngr, (Bandwidth Monitoring, {(attachment_point, {0x1:1, 0x1:2, 0x2:1, 0x2:2})})), (Data Usage Cap Mngr, (Flow Mod, {(dept, {CS}), (traffic, web)})), (Intrusion Prevention App, (Device Handler, {(vlan_id, 2)}), (Intrusion Prevention App, (Packet-In Handler, {(attachment_point, {0x3:1})}), (Intrusion Prevention App, (Flow Mod, {(dept, {CE}), (traffic, web)}))}.
3. Derived Functions:
– assigned_pperms((Device Handler, {(vlan_id, ⊥)})) = {((queryDevice, DEVICE), {(vlan_id, ⊥)})}. assigned_pperms((Bandwidth Monitoring, {(attachment_point, ⊥)})) = {((getBandwidthConsumption, PORT-STATS), {(attachment_point, ⊥)})}. assigned_pperms((Flow Mod, {(dept, ⊥), (traffic, ⊥)})) = {((addFlow, FLOW-RULE), {(dept, ⊥), (traffic, ⊥)})}. assigned_pperms((Packet-In Handler, {(attachment_point, ⊥)})) = {((readPacketInPayload, PI-PAYLOAD), {(attachment_point, ⊥)})}.
– app_sessions(Data Usage Cap Mngr) = {DataUsageAnalysisSession, DataCapEnforcingSession}. app_sessions(Intrusion Prevention App) = {IntrusionPreventionSession}.
– session_roles(DataUsageAnalysisSession) = {(Device Handler, {(vlan_id, 1)}), (Bandwidth Monitoring, {(attachment_point, {0x1:1, 0x1:2, 0x2:1})})}. session_roles(DataCapEnforcingSession) = {(Flow Mod, {(dept, {CS}), (traffic, web)})}. session_roles(IntrusionPreventionSession) = {(Device Handler, {(vlan_id, 2)}), (Packet-In Handler, {(attachment_point, {0x3:1})}), (Flow Mod, {(dept, {CE}), (traffic, web)})}.
– avail_session_pperms(DataUsageAnalysisSession) = {((queryDevice, DEVICE), {(vlan_id, 1)}), ((getBandwidthConsumption, PORT-STATS), {(attachment_point, {0x1:1, 0x1:2, 0x2:1})})}. avail_session_pperms(DataCapEnforcingSession) = {((addFlow, FLOW-RULE), {(dept, {CS}), (traffic, web)})}. avail_session_pperms(IntrusionPreventionSession) = {((queryDevice, DEVICE), {(vlan_id, 2)}), ((readPacketInPayload, PI-PAYLOAD), {(attachment_point, {0x3:1})}). ((addFlow, FLOW-RULE), {(dept, {CE}), (traffic, web)})}.
4.Parameter Verification Functions:
– VERIFIERS = {VDeviceVlan, VStatsAttachpoint, VRuleSwitch, VRuleTraffic, VPInAttchpoint}.
– param_verifier((DEVICE, vlan_id)) = VDeviceVlan. param_verifier((PORT-STATS, attachment_point)) = VStatsAttachpoint. param_verifier((FLOW-RULE, dept)) = VRuleSwitch. param_verifier((FLOW-RULE, traffic)) = VRuleTraffic. param_verifier((PI-PAYLOAD, attachment_point)) = VPInAttchpoint.

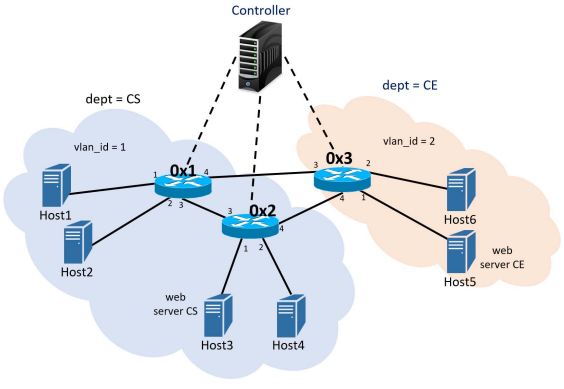

Fig. 4. Topology for proof of concept use case in section VIII.

Usage Cap Mngr' is authorized on resources of CS dept and 'Intrusion Prevention App' is authorized on resources of CE dept. The basic sets and assignment relations of the use case configuration is shown in Table VII.

The app 'Data Usage Cap Mngr' is designed to protect web server on host-3 from any denial-of-service. It needs to monitor bandwidth consumption on attachment points in the switches of CS dept. Thus, is assigned the parameterized role (Bandwidth Monitoring, (attachment_point, 0x1:1, 0x1:2, 0x2:1, 0x2:2)). When this application notices high transmission of packets destined to the web server, it inserts flow rules to block sender's traffic. This app is authorized to handle web traffic only. For that reason it is assigned to to the parameiretized role (Flow Mod, (dept, CS), (traffic, web)). 'Data Usage Cap Mngr' is allowed to read information about hosts with vlan_id = 1 only. For this reason it is assigned

TABLE VIII
CONFIGURATION OF PARAMETER ENGINE FUNCTIONS FOR PROOF OF CONCEPT USE CASE OF SECTION VIII (PART 2).

A. Verifiers:

A.1. VDeviceVlan(s: SESSIONS, op: OPS, ob: OBS, pvpair : PVPAIRS){
//assume a request from app Data Usage Cap Mngr via DataUsageAnalysisSession with the following:
//ob = host tagged with vlan_id=1
//pvpair = (vlan_id, 1)
(ob.vlan_id = pvpair.val); //will return true
}

A.2. VStatsAttachpoint(s: SESSIONS, op: OPS, ob: OBS, pvpair : PVPAIRS){
//assume a request from app Data Usage Cap Mngr via DataUsageAnalysisSession with the following:
//ob = 0x1:1
//pvpair = (attachment_point, {0x1:1, 0x1:2, 0x2:1: 0x2:2})
(ob ∈ pvpair.val); //will return true
}

A.3. VRuleSwitch(s: SESSIONS, op: OPS, ob: OBS, pvpair : PVPAIRS){
//assume a request from app Data Usage Cap Mngr via DataCapEnforcingSession with the following:
//ob = flow_rule$_{[switch_id=0x2,tcp_dst=80,...]}$
//pvpair = (dept, {CS})
//switches(CS) = {0x1, 0x2}
(∃d ∈ pvpair.val : ob.switch_id ∈ switches(d)); //will return true
}

A.4. VRuleTraffic(s: SESSIONS, op: OPS, ob: OBS, pvpair : PVPAIRS){
//assume a request from app Data Usage Cap Mngr via DataCapEnforcingSession with the following:
//ob = flow_rule$_{[switch_id=0x2,tcp_dst=80,...]}$
//pvpair = (traffic, web)
(ob.tcp_dst ∈ protocol_ports(pvpair.val)); //will return true
}

A.5. VPInAttchpoint(s: SESSIONS, op: OPS, ob: OBS, pvpair : PVPAIRS){
//assume a request from Intrusion Prevention App via IntrusionPreventionSession with the following:
//ob = packet-in message with source switch_id = 0x3
//and out_port = 1
//pvpair = (attachment_point, {0x3:1})
(attachment_point(ob.switch_id, ob.out_port) ∈ pvpair.val); //will return true
}

B. CandidateVerifiers(*ot*: OBTS, pvpairs : $2^{PVPAIRS}$){
verifiers = {};
For each p_i ∈ {dept, traffic} **do**
V_1 = param_verifier(FLOW-RULE, dept); //V_1=VRuleSwitch.
verifiers := verifiers ∪ VRuleSwitch;
V_2 = param_verifier(FLOW-RULE, traffic); //V_2=VRuleTraffic.
verifiers := verifiers ∪ VRuleTraffic;
return verifiers; //verifiers = {VRuleSwitch, VRuleTraffic}.
}

C. ParamCheck(*s: SESSIONS*, *op*: OPS, *ob*: OBS, pvpairs: $2^{PVPAIRS}$){
//Example for flow rule insertion by DataCapEnforcingSession.
verifiers = **CandidateVerifiers**(type(flow_rule$_{[switch_id=0x2,tcp_dst=80,...]}$), {(dept, {CS}), (traffic, web)}).
VRuleSwitch(DataCapEnforcingSession, addFlow, flow_rule$_{[switch_id=0x2,tcp_dst=80,...]}$, (dept, {CS}));
VRuleTraffic(DataCapEnforcingSession, addFlow, flow_rule$_{[switch_id=0x2,tcp_dst=80,...]}$, (traffic, web));
return true;
}

to the parameterized role (Device Handler, (vlan_id, 1)). The relations between theses apps and their parameterized roles are specified in AA relation in item 2 of table VII.

The function of 'Intrusion Prevention App' is to inspect packets destined to the web server in host-5. It inserts flow rules to block any malicious activity destined to this server. Because it is authorized for switch 0x3 only, the app is assigned the parameterized roles (Flow Mod, (dept, CE), (traffic, web))) and (Packet-In Handler, (attachment_point, 0x3-1).

When access requests are submitted by these apps, ParaSDN checks each access request using the CheckAccess function described in Table IV. The Parameter Engine calls the verifiers to verify if apps requests are legitimate based on parameter values. Examples of verifiers are shown in item A of Table VIII. For example, the verifier VRuleSwitch Will be called to make sure that the flow rule is inserted in a switch under the authority of the requesting app. If the 'Data Usage Cap Mngr' app tries to insert a flow rule in switch 0x2. The verifier VRuleSwitch will be elected as a candidate verifier based on the object type and the parameter. It will receive the object and the parameter (dept, CS) and verify, based on the condition, that the flow rule will be inserted in switches of CS department. Otherwise, a false is returned and access will be denied. The verifiers in item A of Table VIII gives some assumed access requests based on the use case and the corresponding verifier's decision. The two Apps achieve these tasks via sessions. These sessions and their parameterized roles are shown in item 3 of Table VII via the function session_roles.

IX. IMPLEMENTATION AND EVALUATION

In order to demonstrate our proof-of-concept prototype, we developed and ran the framework in Floodlight platform v1.2 release [15]. The Floodlight platform is deployed on a virtual machine that has 8GB of memory and runs on Ubuntu 14.04 OS installation. We created a topology with three virtual switches (Open vSwitch v2.3.90) connected to each other and each switch is connected to two hosts. Switches are connected to the controller and hosts are virtual machines that has 2GB and run Ubuntu 14.04 OS server.

We implemented our ParaSDN authorization framework in Floodlight platform and used hooking techniques without any change to the code of Floodlight modules. We implemented hooking for all operations exposed by Floodlight services to controller apps. We used AspectJ [16], which is a seamless aspect-oriented extension to Java. Our system intercepts all operations before execution. When a session issues a request the hooked API invokes the ParaSDN components for performing access verification and reply back. This system can be deployed to all other Java-based SDN controllers.

App requests are intercepted by our framework before reaching to the SDN service. Access will be provided by the service only after successful authorization check. During the lifetime of the app, our access control system keeps mediating all sessions access requests for performing security authorizations. It can identify each session, mediate each access request and send it for authorization check based on ParaSDN configuration.

To evaluate the performance of ParaSDN, we created a test app and assigned the app fifty network operations. The purpose

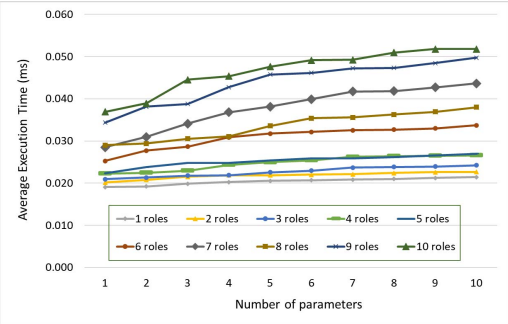

Fig. 5. Average execution time required to finish the tested operations.

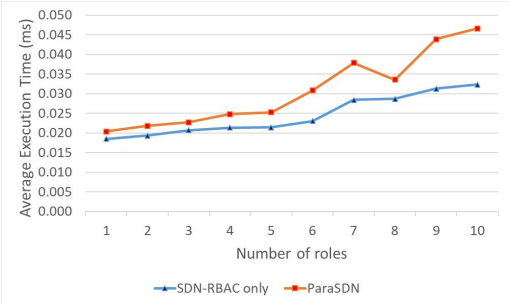

Fig. 6. Overhead imposed by parameters in ParaSDN compared to SDN-RBAC system.

is to perform a pressure test on ParaSDN by executing these operations with different security configurations. Each security configuration is characterized by the number of parameterized roles assigned to the app and number of parameters associated with each of them. We created test parameters and associated them with parameterized roles and created corresponding test verifiers. For each security configuration, the test is repeated for hundred times.

In the first configuration, the fifty operations are executed with one fixed parameterized role and varying number of test parameters. The total authorization time is reported for these fifty operations as shown in Fig. 5. Each subsequent test is performed by assigning one more parameterized role to the app and repeating the same previous test with varying number of parameters until ten roles. The execution time for all tests is reported as shown in Fig. 5.

The results in Fig. 5 demonstrates that the latency overhead of ParaSDN increases linearly with the number of parameters and the number of roles, thus ParaSDN is highly scalable even if the number of parameters and the complexity of security configuration grow in the future.

To compare the overhead imposed by parameters in ParaSDN with the one without using Parameters, i.e, the SDN-RBAC system [4], we repeated the same test on SDN-RBAC. We computed the average times required to finish all parameters with fixed number of roles and aligned the results of with that of SDN-RBAC. The results are shown in Fig. 6. The overall results show that ParaSDN adds negligible overhead to the Floodlight controller which doesn't impact the whole controller's performance.

X. Conclusion and Future Work

In this paper, we proposed ParaSDN, an access control model that provides fine grained capabilities for SDN using the concept of parameterized roles and permissions. We implemented a proof of concept prototype in an SDN controller to demonstrate the applicability and feasibility of our proposed model in identifying and rejecting unauthorized access requests submitted by controller apps. As a future work, we plan to extend our model to suit the needs for multi-controller environments in SDN-Enabled technologies like IoT and Cloud infrastructures.

Acknowledgment

This work is partially supported by NSF CREST Grant HRD-1736209 and CNS-1553696.

References

[1] P. Porras *et al.*, "A security enforcement kernel for openflow networks," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 121–126.

[2] P. A. Porras *et al.*, "Securing the software defined network control layer." in *NDSS*, 2015.

[3] Y. Tseng *et al.*, "Controller dac: Securing sdn controller with dynamic access control," in *Communications (ICC), 2017 IEEE International Conference on*. IEEE, 2017, pp. 1–6.

[4] A. Al-Alaj, R. Krishnan, and R. Sandhu, "Sdn-rbac: An access control model for sdn controller applications," in *2019 4th International Conference on Computing, Communications and Security (ICCCS)*. IEEE, 2019, pp. 1–8.

[5] X. Wen *et al.*, "Towards a secure controller platform for openflow applications," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 171–172.

[6] S. Scott-Hayward *et al.*, "Operationcheckpoint: Sdn application control," in *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*. IEEE, 2014, pp. 618–623.

[7] J. Noh *et al.*, "Vulnerabilities of network os and mitigation with state-based permission system," *Security and Communication Networks*, vol. 9, no. 13, pp. 1971–1982, 2016.

[8] H. Padekar *et al.*, "Enabling dynamic access control for controller applications in software-defined networks," in *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*. ACM, 2016, pp. 51–61.

[9] A. Al-Alaj, R. Sandhu, and R. Krishnan, "A formal access control model for se-floodlight controller," in *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*. ACM, 2019, pp. 1–6.

[10] C. Yoon *et al.*, "A security-mode for carrier-grade sdn controllers," in *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM, 2017, pp. 461–473.

[11] B. Ujcich *et al.*, "Cross-app poisoning in software-defined networking," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 648–663.

[12] L. Giuri and P. Iglio, "Role templates for content-based access control," in *Proceedings of the second ACM workshop on Role-based access control*, 1997, pp. 153–159.

[13] M. Ge and S. L. Osborn, "A design for parameterized roles," in *Research Directions in Data and Applications Security XVIII*. Springer, 2004, pp. 251–264.

[14] A. E. Abdallah and E. J. Khayat, "A formal model for parameterized role-based access control," in *IFIP World Computer Congress, TC 1*. Springer, 2004, pp. 233–246.

[15] Floodlight-Project. (2020) http://www.projectfloodlight.org/.

[16] AspectJ. (2020) Aspectj: A seamless aspect oriented extension to java. https://www.eclipse.org/aspectj/.