INTERNATIONAL JOURNAL OF COMPUTERS COMMUNICATIONS & CONTROL
ISSN 1841-9836, 11(3):428-440, June 2016.

Checking Multi-domain Policies in SDN

F.A. Maldonado-Lopez, E. Calle, Y. Donoso

Ferney A. Maldonado-Lopez
Systems and Computing Engineering Department
Universidad de los Andes, Bogotá, Colombia
fa.maldonado1897@uniandes.edu.co

Eusebi Calle
BCDS, Broadband Communication and Distributed Systems
Universitat de Girona, Spain
e.calle@udg.edu

Yezid Donoso*
Systems and Computing Engineering Department
Universidad de los Andes, Bogotá, Colombia
*Corresponding author: ydonoso@uniandes.edu.co

Abstract: Programmable Network like SDN allows administrators to program network infrastructure according to service demand and custom-defined policies. Network policies are interpreted by the centralized controller to define actions and rules to process the network traffic on devices that belong to a single domain. However, actual networks are *multi-domain* where several domains are interconnected. Then, because SDN controllers in a domain cannot define nor monitor policies in other domains, network administrators cannot ensure that their own policies, origin policies are being enforced by the domains not directly managed by them (i.e. foreign domains). We present AudiT, a multi-domain SDN policy verifier that identifies whether an origin policy is enforced by foreign domains. AudiT comprises (1) model for network topology, policies, and flows, (2) an Audit protocol to gather information about the actions performed by network devices to carry the flows of interest, and (3) a validation engine that takes that information and detects security policy violations, and (4) an extension to the OpenFlow protocol to enable external auditing. This paper presents our approach and illustrates its application using an example considering multiple SDN networks.

Keywords: Network Operating Systems, Software-Defined Networking, Network management, Policy Verification

1 Introduction

In Software-Defined Networking (SDN), network administrators use software languages to define how network traffic is processed and delivered. They use these languages to implement network policies, concrete rules about how a network must deal with specific types of traffic known as *flows*. For instance, these languages can be used to specify which users or network machines can connect to specific servers; which network devices must be used to deliver specific types of traffic, or which bandwidth can be assigned to specific flows.

SDN is based on the separation of control and data planes. On one hand, the control plane remains on a centralized server called *controller* that makes decisions about how the traffic is processed. The controller is responsible for managing connections, addressing, and routing protocols. Applications at the controller use specific-SDN protocols such as OpenFlow [14] to instruct elements in the data plane how to process and deliver the network traffic. The data plane consists of network devices, or datapaths, responsible for packet forwarding and switching.

In the SDN architecture, a single controller manages policies and behavior of a network domain. Only the domain controller has access to the rules used by each network device in its own domain. Thus, neither a network device nor a controller can access information about rules from other network domains. Although this is perfect to deal with policies in a single domain, a network administrator cannot observe how a external network, out of its domain, handles the traffic.

With the current SDN architecture, the administrator is not able to enforce nor monitor multi-domain policies. That is because forwarding rules must be implemented on multiple domains. For instance, today is very common that network traffic is delivered using a internal network such as LANs, and external networks such as WANs and Internet. If the network administrator wants to enforce or monitor a network policy, she can define applications in her own network domain but cannot do it in the external domains. Unfortunately, she cannot check if external domains enforce a network policy because she cannot determine how the traffic is delivered in those external networks.

This situation can be specially critical in network security policies. For example, essential traffic that is delivered through external networks can be duplicated or redirected to other network machines using a simple application in the external domain SDN controllers. Due to network administrators cannot get access to the rules in the external networks, they are disabled to detect these situations neither validate if a policy is achieved.

We propose a mechanism to audit network policies in multiple domains called AudIt. Our approach overcomes these SDN limitations and allows to network administrators to validate if the network policies are enforced by a external domain. AudiT comprises three modules: an extension to the OpenFlow protocol to enable external auditing, the AudIt interface for network devices that gathers information about the actions performed in external domains to carry the flows of interest, and a validation engine that runs into the internal network controller and detects policy violations.

Other mechanisms were suggested to check network policies. For example, Hinrichs [4] developed a declarative language called Flow-based Management Language (FML) to describe network policies and configuration in a high-level and declarative approach. This FML is a high-level declarative language, based on flows, that checks the first packet of every flow against the policy. Lately, Monsanto et. al. [16] introduce a declarative language called NetCore. It is a high-level declarative language that describes the desired behavior of the network but does not deepen the implementation of that behavior. With NetCore is possible to express packet forwarding policies for SDN. Afterwards, Soulé et. al. present Merlin [21]. Merlin is also a declarative language based on logical predicates and regular expressions with which a network administrator can write network policies.

In contrast to previous works, we propose to express network policies as predicates but use a SAT solver and a model finder to evaluate predicates, find inconsistencies and detect policy violations. AudIt uses Alloy [6] to describe the network topology, policies and network traffic. Mirzaei *et al.* proposed used Alloy to verify network properties in [15]. In this case they model internal states of a network and OpenFlow switches.

In summary, we introduce the foreign controller verification problem, we define multi-domain policies in programmable networks, a mechanism to gather information from external SDN domains, and a validation engine that uses gathered information to check if a network policy is enforced by the external domain.

Rest of this paper is organized as follow: Section 2 explains the problem of auditing own policies at external domains; then we present a model to illustrate network topology, paths, forwarding rules and policies in Section 3. Then, Section 4 introduces AudiT protocol and functionality, and presents an example. Finally, Section 5 concludes the paper and presents future work.

2 Auditing policies in multi-domain networks

A *Network policy* is a set of conditions, constraints, and settings about how a specific type of traffic must be managed by a network. It also includes which users and hosts are authorized to create connections, and the circumstances under which they can or cannot connect. Network policies are the accurate and unambiguous way to specify the traffic behavior.

Initially, Stone *et al.* proposed a path-based policy language (PPL) that abstracts topological (physical) paths and flows to check network properties [22]. Now, with programmable networks as SDN, new network policy abstractions are under development, therefore the challenges in policy checking open a rich field of study. High-level declarative language were proposed to represent network policies with more expressiveness. Declarative languages such as FML [4, 5] express network policies in terms of flows. For general purposes, Hinrichs developed a declarative language called Flow-based Management Language (FML) to describe network policies and configuration in a high-level and declarative approach [4]. FML is based on flows, and checks the first packet of every flow against the policy. FML identifies a network flow by: *source* and *target* for users, hosts, and access points, in addition to protocols and requests

A flow is the specification of a traffic, sometimes is called a *session*, that contains common attributes such as source, destination, protocol, but also can specify more granular characteristics as duration, valid time, users, data format and so on. Then those policies are processed using DATALOG to find matching flows. Other languages were designed for SDN are Merlin and NetCore. Merlin [21] is a framework to write network policies for SDN. NetCore [16] is a language for describe forwarding rules and it is integrated with another framework called Frenetic, a project from Cornell and Princeton universities. These languages allow network administrators to define policies in a single-domain networks. They did not contemplate checking policy enforcement on a third-domain.

In contrast to previous approaches, AudIt offers not only the ability to write and check network policies, it is unified with the controller and extract forwarding data and check it. AudIt also uses the flow specification and checks if the set of flows is valid for a given topology. Moreover, AudIt reports inconsistencies in terms of flows not only as instructions at the hardware implementations.

For example, suppose that a network policy defines *only computers assigned to members of IT department can get access to database servers.* We can write this *policy* as:

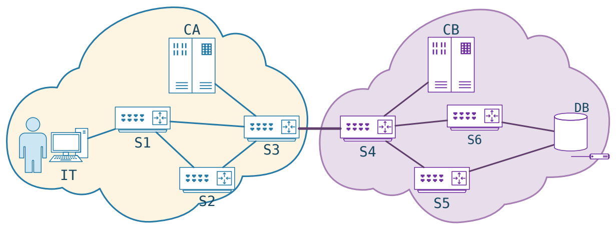
$$allow(\texttt{src,target}) \;\texttt{|}\; \texttt{src} \in IT \wedge \texttt{target} \in DataBase \tag{1}$$

$$deny(\texttt{src,target}) \;\texttt{|}\; \texttt{src} \notin IT \wedge \texttt{target} \in DataBase \tag{2}$$

Expression (1) means that the network must allow flows from IT to database servers. Due to this policy must be closed, (2) denies any flow from other machines to the same servers. In SDN networks, policies are enforced by its domain controller which rules the behavior of every forwarding device –switch– in its domain. The big question is: Can the network administrator monitor that this policy is achieved?

2.1 Policies in Multi-domain networks

In multiple-domain networks, each domain is managed by its own controller. In our example scenario depicted in Figure 1.a, the domain A is ruled by its *controller* CA, and operates the IT department and its users. The external domain B, managed by the controller CB, operates



a) Multi-domain network scenario.

S1 flowtable

Match	Action
Src = IT ∧ Dst = DB	⟨ Fwd S3 ⟩

S3 flowtable

Match	Action
Src = IT ∧ Dst = DB	⟨ Fwd S4 ⟩

b) Flowtables for devices in domain A.

Figure 1: Domain A may send a policy to be implemented in domain B, but there is not guarantee B implements the policy correctly.

the database servers. Network administrator supervises her own controller CA, and may install forwarding rules on devices S1, S2, and S3 to deal traffic generated by the IT department. Network controller CA cannot access the rules in forwarding devices in the domain B neither compel controller CB to install required forwarding actions into the devices S4, S5, and S6.

A *multi-domain policy* must be enforced by own and external domains. The controller CA may share the policy with the controller CB, and awaits that CB implements the policy in its devices. However, there is not certainty that delivered traffic in the external domain B obeys any policy defined by A. Following the example, the administrator of domain A cannot enforce policies related to deliver traffic to the database servers, because the domain B is external.

2.2 Challenges in multi-domain networks

Each network controller is in charge of configuring switching devices on its domain. Figure 1.b shows the required configuration installed on devices in the path of domain A that process the flow until it reaches the next domain. *Network configuration* are rules that implements the policy. In this case, the configuration conducts the *flow traffic* from IT department to database servers. This traffic arrives to switch S1, then is forwarded to switch S3, and finally it is forwarded to domain B interface, switch S4. Clearly, A controller unknowns and cannot handle implemented configurations in external domains. However, administrator want to know if their policies are enforced in external domains. Because of database servers are located in external network, for instance it is hosted by another company, the above policy redirects flows from the IT department to external servers but deny the flows originated from sources other than IT department. However, usually companies rely on external networks such as WANs and Internet to deliver network *flows*.

Since the configuration of network device is protected information, it is only accessed by its own domain controller, and administrator wants to check if the external controller applies a policy on its domain, we have identified a main challenge: how to detect if a policy is enforced by a external domain? and how to audit the policy enforcement without reveal risky information?

2.3 Policies in Programmable Networks

In SDN environment there are some languages to describe network policies. For example, NetCore is a high-level declarative language that describes the desired behavior of the network but does not deepen the implementation of that behavior. With NetCore is possible to express packet forwarding policies for SDN networks [16]. Another work, Merlin [21] is also a declarative language but based on logical predicates and regular expressions which can be solved using linear programming to determine forwarding paths.

Verification of SDN configurations is focused on check network properties that follow a rule. For instance VeryFlow [10] creates a network-wide invariants and checks them against rules. FatTire [19] uses regular expressions and writes policies in this way to be able to validate. Other works attend to find *conflict rules*, rules that contradict earlier ones. In such a way FortNOX [18] checks new flow-rules against a flow-constraint set, and authenticates the source of rules by means of digital signatures. Another illustration is NetPlumber [9] that searches if a candidate rule introduces network misconfigurations or policy violations . It executes a procedure called Header Space Analysis (HSA) over dependency graphs to find conflicts. These approaches examine the forwarding tables from each network device and could check if they conform with the specified policy. However, none of these approaches support the validation of policies in external domains.

From policies to flow-rule implementation Network applications – or functionality– run on a controller and define the general behavior or *policies* by installing specific configurations on each switching device. Regularly, those programs use OpenFlow (OF) [17] to communicate controllers and forwarding devices, and install, modify, or get *flow-rules* that specify how a device deals with specific traffic. A flow-rule is a pair `<match,action>` map on the device's *flowtable*. A flow-rule defines which `action` is performed once a packet header matches the `match` pattern.

OF defines a set of messages to control the internal information on each device, and rules used to process a flow. In summary, OF messages can add, modify, and query rules from device's flowtable. Actions also include: dropping a packet (`DROP`), forwarding a packet to a specific port (`FWD`), or report the set of installed rules (`STATUS`). The rule-set is *closed*, and the packet is reported to the controller if its header does not match any rule.

3 Topology and Policy Models

We describe a model that involves the physical topology and paths; and network operation definitions such as flows, policies, and conflicts. First, we use the following specification for networks. We will use the *relation of correspondence* later on when we write the model in Alloy.

Definition 1 (Network Graph)**.** A network graph $\mathcal{G}$ is a duple (N, L) such that N is the set of nodes, and links $L := (N, N, C)$ is the *correspondence* C with domain and co-domain $N \to N$.

Considering the links relation L, we write the function $\mathsf{links}(n)$ to denote $\{m|(n, m) \in L\}$, the set of all the nodes m connected to a node n. In addition, for convenience, we write $n \to m$ to express *from n to m* sometimes instead of $(n, m) \in L$. A well-formed network $\mathcal{G}$ must satisfy the following rules: 1) Network is connected, $\forall n \in N|\,\mathsf{links}(n) \neq \emptyset$, i.e. there are not isolated nodes; 2) no self-loops, $\nexists n \in N|n \to n$, i.e. there are not links from a node to itself; and, 3) for all link, there is an arrival node, $\nexists n \in N|\mathsf{links}(n) \cap n = \emptyset$.

It is important to note that nodes in our model does not represent a network device – router nor switch–, a node denotes a *device port*. Then under this abstraction, the link relation represents forwarding rules, not just physical links in the topology.

Definition 2 (Path)**.** A path p is a tuple (s, t, N_p, L_p). A source node s, a target node t, a subset of nodes $N_p = \{n_1, n_2, \ldots, n_k\}$, and a subset of links that creates the sequence $L_p = \{(s, n_1), (n_1, n_2), \ldots, (n_{k-1}, n_k), (n_k, t)\}$.

A path, can be described as a list of nodes that maintains a sequence. Path nodes $N_p \subset N$, path links $L_p \subset L$. A well-formed path satisfies: 1) all implicated nodes in the links belong to node set, $\forall (n_a, n_b) \in L_p \implies \{n_a, n_b\} \in N_p$, 2) $\{s, t\} \in N$, source and target nodes are in the network, and 3) source node *opens* and target node *finishes* a path, $\nexists (n_1, n_2) \in L_p | n_2 = s \vee n_1 = t$.

At this point is appropriated to describe the *transitive closure*. We use this concept to tackle the reachability property when describe a path. A binary relation R is *transitive* if contains tuples in the way $a \to b$ and $b \to c$, but also contains $a \to c$. This relation is noted as R^+ and contains R. Finally, a path has no loops, considering the relation L and the function $\mathsf{links}^+(n)$ is the set of all nodes that can be reached from n. Then a path has no loops if $\nexists n \in N_p | n \in \mathsf{links}^+(n)$. Also for convenience, we denote a path as a node sequence as $\langle s, n_1, n_2, \ldots, t \rangle$. We include a wildcard symbol $(*)$ to denote any unspecified node or sequence of nodes. For example, the path $p = \langle A, *, C \rangle$ is the path that starts at node A and ends at C.

3.1 Traffic flows

Flow is the fundamental abstraction for our model. For the reader it is similar to *communication session* supported by a set of paths and device configurations. Traffic flow defines the high-level network parameters needed to create a competent communication channel. A flow provide enough detail to describe a set of feasible sessions, and provides a form to group and manage these sessions.

Definition 3 (Flow)**.** A flow is a sequence of traffic constraints $f = (f_1, \ldots, f_n)$. Each term f_i is a restriction over a traffic characteristic, strongly related to filters on packet fields.

Due to we illustrate flows as traffic constraints. Reader should note we indicate packet-field match as those constraints. The used definition allows us to construct flexible and composed communication flows. A term of flow involves transport-layer protocol, source / destination at third layer, or applications. Also, we use set operators over these packet fields to define the flow. For instance, $flow_a = \{$`protocol = TCP`, `src_ip = 192.168.5.10`, `dst_ip = 192.168.7.10`$\}$ details a traffic flow between those IP addresses and TCP as transport protocol. Note that this flow only defines the traffic in one way. It means, the other direction is not under this definition.

However, flow definition is only associated to communication characteristics and packet fields, but not the set of paths that supports the flow.

3.2 Policies, conflicts and semantics

In order to define the set of paths that implements a policy and then identify policy conflicts and violation we follow the guidelines of Harel and Rumpe [8] to specify a modeling language $Ł$ describing the *syntactic domain* $\mathcal{L}_L$, the *semantic domain* $\mathcal{S}_L$ and the *semantic function* $\mathcal{M}_L : \mathcal{L}_L \to \mathcal{S}_L$, also traditionally written $[\![\cdot]\!]_L$.

The policy is a set of rules that achieves a management procedure. Forwarding rule is the action that a node executes to forward a packet into a computer network. Rules are described in terms of flows, by the previous definition.

Definition 4 (Policy)**.** A network policy is a tuple $\pi = (f, P, C, \alpha)$ s.t. f is the target flow composed of packet-field values, P is the set of paths that support the flow, C is a set of conditions, over the flow f or path P, and α is an action, regularly $\{permit, deny\}$.

Essentially, a policy resolves whether allow the flow f over the set P conclude on a specific action α. It is decisive and produces a configuration network that allows or deny the traffic flow. For example, the network administrator wants to apply the policy: *Ana is a user with profile of IT member, who is in the subnetwork 192.168.5.*/24 (S1), is allowed to access the database at subnetwork 192.168.7.*/24 (S6) and port 1521, and her traffic must go through the router S3.*

Now the manager has to detail the policy. In order to do that she solves the following steps:

1. path `S1_S6` := `<S1,*,S3,*,S6>`,

2. transport protocol: = `TCP`,
3. port number: = 1521;
4. the conditions `user` = `Ana`, and `Ana` $\in$ `IT member`;
5. finally the policy decision: *permit*

In this way, we can find ω, the set of configurations and instructions that implements the paths and the policy π. Note that IP addresses, user groups, traffic class and protocols should be modeled as *sets*. On the other hand, ordered items such as *time* are modeled as *sequences* to be able to compare them using $\leq$ and $\geq$ operators.

$$\omega \;=\; impl(p : path|(\texttt{S1\_S6}) \wedge \texttt{protocol} = \texttt{TCP} \wedge \texttt{port} = \texttt{1521} \wedge \texttt{Ana} : user \in \texttt{IT member}) \quad (3)$$

This representation of a policy, utilizing logical conjunctions, allows us to express this policy as a conjunctive normal form *predicate* (CNF), and logically solve it. Moreover, we are able to check a formal solution using a model finder as Alloy [7], compare solutions, or find inconsistencies.

Definition 5 (Policy Semantic)**.** A semantic of a policy $[\![\pi, \mathcal{G}]\!]$ is the set of paths that implements the flow over a path on $\mathcal{G}$ and achieves the policy.

$[\![\Upsilon]\!] = \Omega$ is the semantic of all network policies and produces the set of all paths implemented on the network. The complete network configuration is denoted by Ω. The semantic function $[\![\pi, \mathcal{G}]\!]$ of a policy contains the sets of paths, flow definitions, conditions and the network $\mathcal{G}$ that satisfies the policy π. Obviously, the policy π is valid in a network $\mathcal{G}$, if $[\![\pi, \mathcal{G}]\!]$ is not empty.

Definition 6 (Policy Conflict)**.** A policy conflict occurs when a set of policies are not implemented by any path or there are inconsistencies that prevent the generation of a path.

Essentially, if the semantic of a policy is empty, means that there is not set of configuration of paths that satisfies the policy. Given two valid policies π_1 and π_2, they are not conflicting in the network $\mathcal{G}$ if $[\![\pi_1 \cup \pi_2, \mathcal{G}]\!]$ is not empty. That is, two policies are not conflicting if there is a set of paths, flows, and restrictions in the network $\mathcal{G}$ that satisfies both policies. In contrast, we say that two policies π_1 and π_2 are conflicting in $\mathcal{G}$ if $|[\![\pi_1 \cup \pi_2, \mathcal{G}]\!]| = 0$.

Definition 7. (Minimal diagnosis) Given a set of policies $\pi \subseteq \Upsilon$ such that $[\![\Upsilon \setminus \pi]\!] \neq \emptyset$, the minimal set π is the minimal diagnosis.

We refer to this minimal set as the littlest configuration applicable without conflicts. Now, we need a tool, called a *verifier*, able to find (calculate) the semantic function $[\![\pi, \mathcal{G}]\!]$ and verify if that set is empty and the minimal diagnosis of that set. We use similar tools also for validating paths on network infrastructure [12], and recently we show how to use of minimal diagnosis to detect and prevent firewall-rule conflicts on software-defined networking [13].

4 Checking multi-domain policies with AudIt

AudIt is an auditing extension for OpenSwitch protocol and OpenFlow controller that allows domain controllers to validate security policies on a foreign domain. Our proposal creates a language definition and transformation to audit network policies. We use Alloy to obtain a set of tuples that satisfy the policies (exactly the semantic function). If Alloy does not find any element (the set is empty), the policy set is invalid or conflicting.

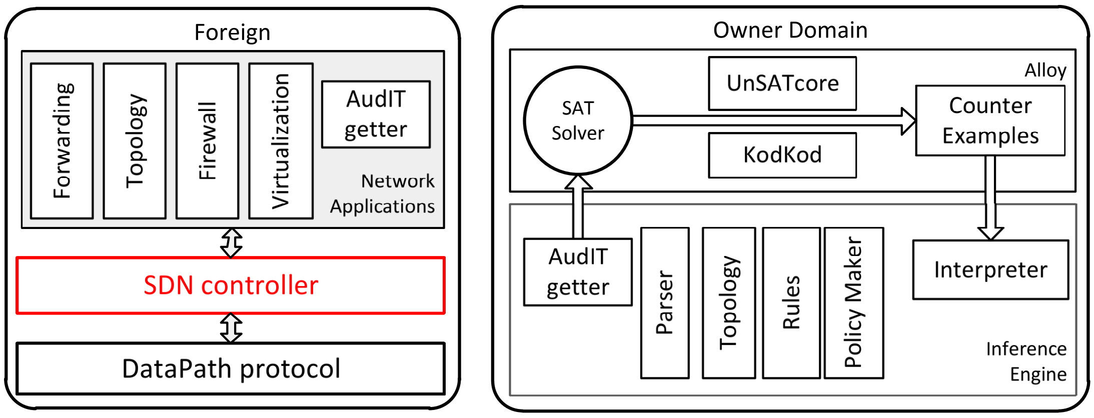


Figure 2: AudIt architecture

AudIt works as a validation protocol that allows a controller to gather auditing information from external domains and validate the origin policy. It performs two phases: gathering network information and validation process. First, the controller in the origin domain gets information we called *audit packets* that is routed through the network as regular traffic. Then, when auditor packet reaches devices in the external domain, these network devices report a subset of its own flowtable to the controller in the origin domain. Finally, the controller in the origin domain process the gathered flowtables to obtain all the processing rules related to the flows of interest, and executes the *validation engine* that checks if the external domain is accomplishing the security policy.

4.1 AudiT Extension

OpenFlow specifies a set of control messages between controller and forwarding devices. Control messages include: *modify-state* to add or delete flowtables in the device, *collect-statistics* to read counters and device statistics, *managing* groups of flowtables. Controller is also able to request device *status*, where the device reports the flowtable to the controller. AudIt uses regular controller primitives to request information from the flowtable on devices of external domain.

Information gathering Once the controller enables AudIt on each network device, and the audit packet arrives, the device invokes `OFPMP_TABLE_FEATURES` and the header match to filter a subset of the rule table that matches the header. Thereby, it extracts a set of all the *related flowtable entries* (RFE).

$$\mathtt{RFE} = \{e | e.\mathtt{src} \odot p.\mathtt{src} \cup e.\mathtt{target} \odot p.\mathtt{target}\} \tag{4}$$

For simplicity, our example only considers IP addresses and masks in the IT-database scenario. Our RFE are defined by (4). Where p is the policy and $\odot$ is the match relation.

AudIt message Figure 3 shows the structure of an AudIt message. It comprises the same flow header in order to be routed through the same path; moreover, it includes origin controller identifier, controller authentication data, other AudIt settings, and the list of fields and rules to be filtered by the device.

Flow header	Origin Controller ID	AudiT settings	Controller Signature
List of fields		List of policies	

Figure 3: Structure of an AudIt packet. The list of policies are constraints over packet fields.

4.2 AudIt protocol

Figure 4 shows the proposed protocol that allows controllers to enable AudiT protocol, gather information from foreign devices, and check policies.

1. Involved domains subscribe an *audit agreement* that specifies the permission to create, send and process audit packets. Then, all implicated domains update their module that recognizes the audit request and overwrites `AUDIT_ENABLE` variable.

2. Origin domain A shares the traffic policy over IT's traffic with B. Security Policy described in section 1 - 2: *DataBase is only accessed from IT department.* External-domain controller CB enforces the security policy in its network, translates the policy into rules applicable to its infrastructure.

3. Origin controller creates an audit packet. Audit packet contains all packet fields of the flow traffic. This procedure request information about how the traffic is delivered. Thus, foreign network devices process the audit packet as they process regular data flow, or use a interface to return the Related Flowtable Entries (RFE).

4. Foreign devices reply the audit packet with the RFE. The list of entries from its flowtable.

5. At the origin, the controller of A executes the validation engine, determines if there is a subset of rules that violates the policy, and writes a conflict report.

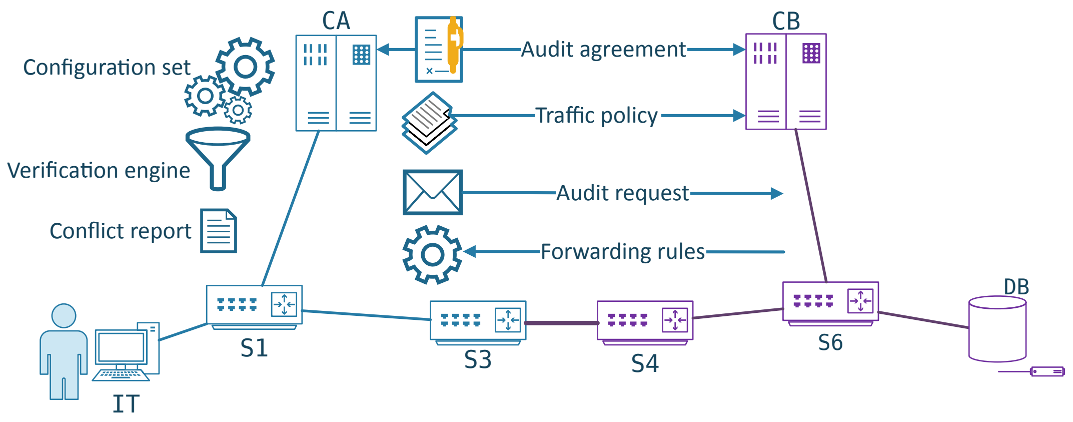


Figure 4: AudiT protocol execution. Devices from domain B report packet rules to CA, then A verifies traffic policy and generates an auditing report.

4.3 Multi-domain Policy Checking

At the end, origin controller owns all rules related with the traffic policy that comes from the external domain, and validates the set of related flowtable entries (RFE) against the policy to identify violations. Figure 5 shows the set RFE that AC gathers from domain B. It is a list of rules related with the traffic policy defined in expressions 1 and 2. Then, the validation engine determines if this subset of rules violates the policy. This policy-rule validation engine could be similar to [9].

S4 flowtable

Match	Action
Src = IT ∧ Dst = DB	⟨ Fwd S6 ⟩

S6 flowtable

Match	Action
Src = IT ∧ Dst = DB	⟨ Fwd DB ⟩

Figure 5: Related flowtables entries from devices in domain B.

4.4 Inference Engine based on SAT

We develop an *inference engine* able to check implementation procedures against network policy. Topology is defined as a set of *nodes*. *Links* is a closure relation of arity two over the node set. Specifically for this project, we model device ports as nodes, and links by a closure relation over nodes. Figure 6 shows how the topology is represented in terms of device ports. A forwarding rule, the simplest instruction that redirects a packet from one port to another is represented as part of the path. Under this perspective, the configuration is part of the topology. Forwarding rules are shown in the figure as dotted lines. These soft-links are considered as regular topology once the model is built.

An optimization opportunity arises due to forwarding rules create *soft-links* that are interpreted as part of the topology. If the traffic policy is quite specific, the resultant topology is disconnected graph, even is reduced to some paths. This abstraction of nodes as device ports, and soft-links can reduce the complexity at evaluation time.

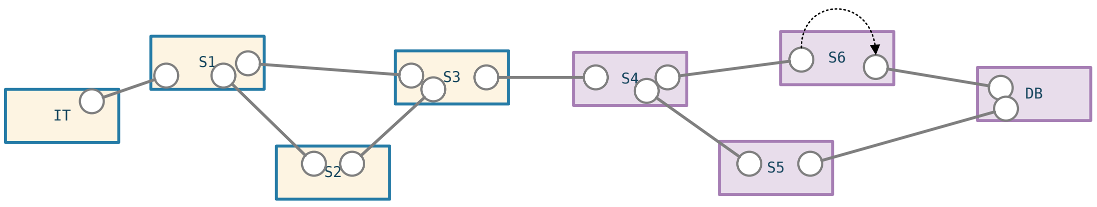


Figure 6: Representation of a network topology based on ports from the original deployment. Blocks are network devices and circles are ports. Forwarding rules are depicted as dotted lines that connect two ports.

Flow is depicted as the list of constraints over packet header, traffic movement sense, and topological considerations. For example, the flowtable described in Figure 1.b is the interpretation of constraints, source and destination addresses, over fields of a packet header. Reader should note that flowtable also denotes the *soft-link* between two ports generated by the forwarding rule. Nevertheless, this soft-links are part of the topology equally as wired links do. In other words, the model does not discerns one from another. Communication details such as protocols or port numbers are considered sets if these elements are part of the packet header. Due to our work uses set theory notion of order is not considered in this model, for that reason we cannot have policies with arithmetic conditionals. For example, the expression *if the number port is greater that 1024, then ...* is invalid in our approach.

Our inference machine is implemented on Alloy [6]. It is fed with external-domain information gathered by the audit procedure or through services that exposes forwarding information.

5 Conclusions and Future Work

We presented OpenFlow AudIt, a mechanism that checks if foreign domains are enforcing multi-domain network policies. AudIt helps to overcome policy-checking limitations of the SDN architecture. It comprises (1) an extension to the OpenFlow protocol to enable external auditing, (2) an Audit protocol to gather information about rules applied to specific network *flows*, and (3) a validation engine that uses flow information and determines if the external network is enforcing specific traffic policies. Additionally, Audit can identify policy violations. It informs the network configuration, rule of flow that infringes the policy and its identifier. In general terms, AudIt allows network administrators to gather information from external domains and determine if network policies are enforced in multiple domains.

5.1 Experiments and results

We implement and test our AudIt protocol using the Floodlight controller [1]. Test cases are divided into two groups: information gathering, and violation inferencing. We run the controller on a server and deploy a test-network using mininet [2], which operate as external domain and implement the example topology used by Sethi in [20]. From another terminal, which operates as owner domain, we run our AudIt interface and extract traffic information from the controller. AudIt implementation creates a topology representation, a policy inventory, and a configuration repository. Thereafter, the inference engine is executed. AudIt writes, policies, configurations and topology as Alloy instructions and executes the satisfiability solver.

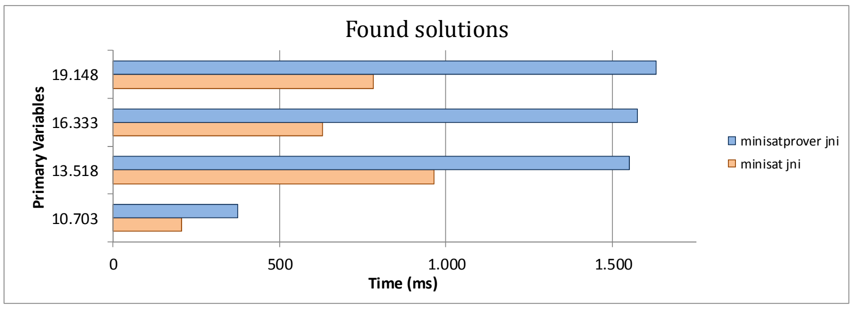


Figure 7: Solutions from Alloy implementation. The same implementation is evaluated using two solvers: minisat and minisatprover with minimal unsatisfiable core.

Figure 7 shows two evaluations over the same topology and set of policies. We implement the same FatTree topology also described in [20] to compare our approach. AudIt takes less than a second using the minisat solver, which only finds if an instance accomplishes the set of policies. On the other hand, if the network administrator wants to determine the set of policies violated by the external domain, she executes AudIt with the minisatprover option and could take up to $1.5s$. These measures are lower than the values reported on [20], for the same Fat Tree topology composed of 20 switches, 16 hosts, and 48 links. We test forwarding and reachability on a Intel i5 at 3.0 GHz, with 3.74 GB RAM. With the intension to show how state explosion and variable affect the performance we test AudIt for 930K, 1.5M, 2.2M and 2.8M of states, which are represented on primary variables shown in Figure 7.

However, AudIt does not have complete information about the network as opposed to Net-Plumber [9]. Moreover, Audit requires the deployment of our OpenFlow extensions into the

[1]`http://www.projectfloodlight.org/floodlight/`
[2]`mininet.org`

network devices in those external domains. Commercial products (i.e. switches from companies such as IBM or HP) do not support the deployment of new extensions without a firmware update. We expect that future experimental implementation shows the benefits of Audit and can be a foundation to introduce multi-domain policy validation into the standard.

Letting external domains gather information about network flow processing may represent a potential security risk for external controllers. In addition, controllers in external domains may include programs that hide information or mimic policy enforcement. Future work focuses on evaluating security risks on our experimental implementation in order to determine which additional mechanisms are required to ensure safe auditing of multi-domain policies.

Acknowledgment

Ferney A. Maldonado-Lopez is a recipient of a the fellowship N528–2011 for doctoral studies by the Colombian Department of Science, Technology and Innovation COLCIENCIAS. We would like to thank to COMIT members for their reviews and meaningful recommendations.

Bibliography

[1] Al-Shaer, E.; Marrero, W.; El-Atawy, A.; Elbadawi, K. (2009); Network configuration in a box: towards end-to-end verification of network reachability and security, *17th IEEE International Conference on Network Protocols, ICNP 2009*, 123-132.

[2] Canini, M.; Venzano, D.; Perešíni, P.; Kostić, D.; Rexford, J. (2012); A NICE way to test OpenFlow applications, *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, USENIX Association, 10-10.

[3] Gude, N.; Koponen, T.; Pettit, J.; Pfaff, B.; Casado, M.; McKeown, N.; Shenker, S. (2008) NOX: towards an operating system for networks, *SIGCOMM Comput. Commun. Rev.*, ACM, 38: 105-110.

[4] Hinrichs, T. L.; Gude, N. S.; Casado, M.; Mitchell, J. C.; Shenker, S. (2009); *Expressing and Enforcing Flow-Based Network Security Policies*, University of Chicago, Technical report, 1-20.

[5] Hinrichs, T. L.; Gude, N. S., Casado, M.; Mitchell, J. C.; Shenker, S. (2009); Practical Declarative Network Management, *1st ACM Workshop on Research on Enterprise Networking, 2009*, 1-10.

[6] Jackson, D. (2002); *Alloy: A Lightweight Object Modelling Notation*, ACM Trans. Softw. Eng. Methodol.; April 2002.

[7] Jackson, D. (2006); *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, 2006.

[8] Harel, D. and Rumpe, B. (2004); Meaningful Modeling: What's the Semantics of "Semantics"?, *Computer*, IEEE Computer Society Press, 37: 64-72.

[9] Kazemanian, P.; Chang, M.; Zheng, H.; Varghese, G.; McKeown, N. (2013); Real time Network Policy Checking Using Header Space Analysis, *Proceeding on Network System Design and Implementation (NSDI)*, USENIX, 99-112.

[10] Khurshid, A.; Zou, X.; Zhou, W.; Caesar, M.; Godfrey, P. B. (2013); VeriFlow: Verifying Network-Wide Invariants in Real Time, *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Proceeding HotSDN '12 Proceedings of the first workshop on Hot topics in software defined networks*, 49-54 .

[11] Mai, H.; Khurshid, A.; Agarwal, R.; Caesar, M.; Godfrey, P. B.; King, S. T.(2011); Debugging the data plane with Anteater, *SIGCOMM Comput. Commun. Rev.*, ACM, 41: 290-301.

[12] Maldonado-Lopez, F.; Chavarriaga, J. and Donoso,Y. (2014); Detecting Network Policy Conflicts Using Alloy, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, Springer Berlin Heidelberg, 8477: 314-317.

[13] Maldonado-Lopez, F. A.; Calle, E. and Donoso, Y.; (2015);Detection and prevention of firewall-rule conflicts on software-defined networking, *Reliable Networks Design and Modeling (RNDM), 2015 7th International Workshop on*, 259-265.

[14] McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.; Rexford, J.; Shenker, S.; Turner, J. (2008); OpenFlow: enabling innovation in campus networks, *SIGCOMM Comput. Commun. Rev.*, ACM, 38: 69-74.

[15] Mirzaei, S., Bahargam, S. and Skowyra, R. (2013); *Using Alloy to Formally Model and Reason About an OpenFlow Network Switch*, Technical Report, http://hdl.handle.net/2144/11416.

[16] Monsanto, C.; Foster, N.; Harrison, R.; Walker, D. (2012); A Compiler and Run-time System for Network Programming Languages, *SIGPLAN*, ACM, 47: 217-230

[17] Open Networking Foundation *OpenFlow Switch Specification*, v.1.3.1, ONF Open Networking Foundation, 2012

[18] Porras, P.; Shin, S.; Yegneswaran, V.; Fong, M.; Tyson, M.; Gu, G. (2012) A security enforcement kernel for OpenFlow networks *Proceedings of the first workshop on Hot topics in software defined networks*, ACM, 121-126.

[19] Reitblatt, M.; Canini, M.; Guha, A.; Foster, N.(2013); FatTire: declarative fault tolerance for software-defined networks, *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, ACM, 109-114.

[20] Sethi, D.; Narayana, S. and Malik, S. (2013); Abstractions for model checking SDN controllers, *Formal Methods in Computer-Aided Design (FMCAD), 2013*, 145-148.

[21] Soulé, R.; Basu, S.; Kleinberg, R.; Sirer, E. G.; Foster, N. (2013); Managing the Network with Merlin, *12th workshop on Hot Topics in Networks, HotNets'13, Nov. 2013*, 1-8.

[22] Stone, G.; Lundy, B. and Xie, G. (2001); Network Policy Languages: A survey and a new approach, *IEEE Network*, 15: 10-21.