# OpenSec: Policy-Based Security Using Software-Defined Networking

Adrian Lara and Byrav Ramamurthy

***Abstract*—As the popularity of software-defined networks (SDN) and OpenFlow increases, policy-driven network management has received more attention. Manual configuration of multiple devices is being replaced by an automated approach where a software-based, network-aware controller handles the configuration of all network devices. Software applications running on top of the network controller provide an abstraction of the topology and facilitate the task of operating the network. We propose OpenSec, an OpenFlow-based security framework that allows a network security operator to create and implement security policies written in human-readable language. Using OpenSec, the user can describe a flow in terms of OpenFlow matching fields, define which security services must be applied to that flow (deep packet inspection, intrusion detection, spam detection, etc.) and specify security levels that define how OpenSec reacts if malicious traffic is detected. In this paper, we first provide a more detailed explanation of how OpenSec converts security policies into a series of OpenFlow messages needed to implement such a policy. Second, we describe how the framework automatically reacts to security alerts as specified by the policies. Third, we perform additional experiments on the GENI testbed to evaluate the scalability of the proposed framework using existing datasets of campus networks. Our results show that up to 95% of attacks in an existing data set can be detected and 99% of malicious source nodes can be blocked automatically. Furthermore, we show that our policy specification language is simpler while offering fast translation times compared to existing solutions.**

***Index Terms*—Software-defined networking, OpenFlow, network security, policy-based network management, policy specification.**

## I. Introduction

WITH the advent of software-defined networks (SDN) [2], efforts to automate and simplify network operation have become popular [3]–[5]. In SDN, the complexity of the network shifts towards the controller and brings simplicity and abstraction to the network operator. As we move away from manual configuration at each device, we get closer to automated implementation of network policies and rules. SDN decouples the control plane from the data plane and migrates the former to a logically centralized software-based network controller. More complex network-control applications can thus be implemented at the controller and exploit the fact that they are network-aware due to the centralized nature of the control plane.

OpenFlow [6] is a protocol that standardizes how an SDN controller communicates with the network devices. An OpenFlow-compliant switch exposes to the controller an abstraction of its flow table and allows the controller to manipulate it by inserting, modifying or deleting rules in the table. Using OpenFlow, an application running on the network controller can thus control how one or more layer 2 switches forward incoming packets.

We propose OpenSec, an OpenFlow-based network security framework that allows campus operators to implement security policies across the network. To motivate this work, suppose a campus operator needs to mirror incoming web traffic to an intrusion detection system (IDS) and e-mail traffic to a spyware detection device. Our goal is to leverage SDN to allow the operator to write a high-level policy to achieve this, instead of having to manually configure each device. Furthermore, suppose the IDS detects malicious traffic and the sender needs to be blocked from accessing the network. Instead of having the operator configure the edge router to manually disable access to the source, we are interested in blocking the sender automatically.

Because OpenSec provides an abstraction of the network, the operators can focus on specifying simple and human-readable security policies, instead of on configuring all the devices to achieve the desired security. OpenSec consists of a software layer running on top of the network controller, as well as multiple external devices that perform security services (such as firewall, intrusion detection system (IDS), encryption, spam detection, deep packet inspection (DPI) and others) and report the results to the controller. The main goal of OpenSec is to allow network operators to describe security policies for specific flows. The policies include a description of the flow, a list of security services that apply to the flow and how to react in case malicious content is found. The reaction can be to alert only, or to quarantine traffic or even block all packets from a specific source.

We have built OpenSec taking three design requirements into consideration. First, policies should be human-readable. Simplicity is one of the main goals of our framework and although current work has focused on creating human-readable policies [7], [5], [8], we argue that there is still room for improvement to make the policy languages human readable. Second, data plane traffic should be processed by the processing units (network devices, middleboxes or any other hardware that provides security services to the network). When the controller is responsible for all the tasks it becomes a bottleneck and the solution does not scale well. In OpenSec, the controller is

Manuscript received November 17, 2014; revised April 25, 2015, September 2, 2015, and December 16, 2015; accepted December 21, 2015. Date of publication January 12, 2016; date of current version March 9, 2016. This work was supported by the National Science Foundation under Grant CNS-1040765, Grant CNS-1345277, and Grant ACI-1541442. An earlier version of this paper appears in the Proceedings of IEEE GLOBECOM 2014 [1]. This manuscript was prepared while A. Lara was a Ph.D. candidate at the University of Nebraska-Lincoln. The associate editor coordinating the review of this paper and approving it for publication was F. De Turck.

The authors are with the Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE 68588-0115 USA (e-mail: adrian.lara@ecci.ucr.ac.cr; byrav@cse.unl.edu).

Digital Object Identifier 10.1109/TNSM.2016.2517407

subject to a low workload and is responsible for implementing policies and modifying forwarding rules based on the security alerts received from the processing units. Third, the framework should react to security alerts automatically to reduce human intervention when suspicious traffic is detected.

The goal of OpenSec is to facilitate a simple, human-readable language to automatically implement network security policies. Further, we undertake a first step towards automated, policy-based reaction to security alerts using OpenFlow. In this paper, additionally, we evaluate the scalability of OpenSec using a real dataset with more than 14 million flows and we add more evaluation scenarios.

To experiment with the proposed framework, we consider two network scenarios. First, to evaluate OpenSec at a realistic scale, we use an existing dataset available at the University of Twente that comprises of traffic directed to a honeypot [9]. Second, we use OpenSec to deploy a Science demilitarized zone (DMZ) to enable higher throughput for scientific data transfers. Next, we focused our evaluation on four metrics. First, we measured the time needed by OpenSec to implement security policies based on the number of switches, processing units and existing policies in the system. Second, we measured the delay needed by the framework to react to alerts raised by processing units. Third, we show the benefits of automated blocking. Fourth, we discuss the trade-off of moving middleboxes away from the datapath and mirroring traffic as opposed to doing in-line processing.

Our results shows that OpenSec scales well because the delay in reacting to alerts remains constant when the traffic rate increases. We also show the benefits of moving the middleboxes away from the data path in terms of achieved throughput. Finally, we compare OpenSec with existing solutions such as Procera [7], CloudWatcher [8] and Fresco [10] and show that the performance of the proposed framework is equal or better than similar works.

The rest of this paper is organized as follows. In Section II we survey related work and in Section III we motivate our work. Next, in Section IV we describe the OpenSec framework and its main components. After that, we describe the operation of OpenSec in Sections V and VI. In Section VII we describe the experimental setup. In Section VIII we evaluate the framework in terms of scalability and performance and we compare OpenSec against similar work in Section IX. Finally, we conclude in Section X.

## II. Related Work

### A. Policy-Based Management Without SDN

A significant amount of work has focused on policy specification [11], policy refinement [12]–[15], conflict detection [16], [17] and policy analysis [18] in networks. Policy-based management (PBM) has also been applied to network management [19] and security [20].

Agrawal et al. [18] provide an overview of how policy-based management can be applied to networked systems. In particular, they explain how Policy Management for Autonomic Computing (PMAC) can be applied to network management. In a nutshell, PMAC is a generic policy middleware that supports extensive and flexible policy languages. Also, a Policy Definition Tool (PDT) should be provided to allow users to create and modify policies. Finally, an automated manager is responsible for collecting policies and implementing them.

Rubio-Loyola et al. [15] propose a a method to refine policies in policy-based management systems. Policy refinement allows to derive low-level enforceable policies from high-level guidelines. The authors provide a list of steps needed to convert high-level goals into low-level policies and describe a framework that supports all the required steps.

Charalambides et al. [16] address conflict resolution in PBM, a crucial aspect when managing a system using policies. Indeed, as the authors point out, when several policies coexist it is likely to encounter that two or more policies give a different output for the same input. This study addresses the problem of conflict resolution when using policies to provide Quality of Service (QoS).

OpenSec is similar to these methods in that it proposes a centralized system capable of receiving policies as input and analyzing them, checking for conflicts and implementing them. In this work, however, we provide a detailed explanation of how policies can be converted into OpenFlow messages to update the forwarding rules dynamically. Also, the policies used in OpenSec are low-level specifications because they already include a list of OpenFlow matching fields that should be used. Thus, the main contribution of OpenSec is the automated administration of processing units and dynamic reaction to security alerts using SDN, as opposed to deriving low-level policies from high-level goals.

### B. Policy-Based Network Management Using SDN

With the advent of SDN, the field of network management has evolved to become more dynamic [21]–[23]. Casado et al. proposed Ethane [4]. In Ethane, an operator creates a policy using the Flow-based Security Language (FSL) to create a high-level access control list. Ethane allows an operator to write an access control policy with good granularity while still using high-level language (for example, using "testing nodes" instead of a subnet mask). Although OpenSec does not focus on referring to network objects by name, we do provide a broader set of security services besides access control. Also, OpenSec includes a reactive component to security alerts. When anomalous traffic is detected, OpenSec can modify traffic rules as specified by the policy, which adds a reactive component missing in Ethane.

Foster et al. propose Frenetic [24], a programming language to program OpenFlow-based networks. Frenetic provides an interface to query traffic information. Frenetic can also be used to create a policy to react to network events. In our opinion, Frenetic focuses on simplifying how to program network events and how to retrieve traffic information. OpenSec focuses on hiding such complexity and allowing a security operator to work at a higher level, since it was designed to implement and enforce security policies, rather than to provide another mechanism to handle events sent by the network switches.

Finally, Bari et al. [25] proposed PolicyCop, an autonomic QoS policy enforcement framework for SDN. This framework allows to specify service level agreements (SLAs) to implement and enforce QoS in an OpenFlow-based network. The step-by-step method used by PolicyCop to convert policies into flow rules is similar to that of OpenSec. However, our focus is on reacting to network security alerts instead of QoS violations.

### C. Candidate Frameworks for Comparison Against OpenSec

Among the proposed frameworks that convert policies to OpenFlow messages, three are candidates for comparison against OpenSec: CloudWatcher [8], Fresco [10] and Procera [7]. We describe these frameworks next and we compare them against OpenSec in Section IX.

Shin et al. proposed CloudWatcher [8], a security monitoring framework for the cloud that has several similarities with our work. Using CloudWatcher, a network operator can use a policy to describe a flow and describe which security services must be applied to it. For example, if traffic within a subnet must be subject to denial of service attack detection and intrusion detection, then a policy can be used to describe this. The authors focus on how the controller can find the optimal route to send the traffic to those processing units and the policy language is not described in detail. OpenSec goes one step further by allowing the operator to describe how to react in case malicious traffic is detected. Our focus is more on how to implement the policies instead of optimal routing decisions to find the processing units.

Voelli et al. proposed Procera [7], a "functional reactive programming" framework where a user can write a high-level policy to define how to handle network events. Just like Frenetic, Procera also aims to simplifying how to deal with network events. They have in common that they both seek a simpler interface to program the network and to react to network events. Also, Procera addresses an important topic: enforcing network policies. Our approach to enforce the implementation of security policies is based on the technique proposed in Procera.

Fresco [10] is another OpenFlow-based security framework that exposes security modules to external users, who can in turn define security policies using such modules. To use Fresco, an operator must define the type, input, the output, the parameter, the action and the event. Fresco is also similar to Procera and Frenetic in the sense that it allows manipulating network events and handling them through pre-defined modules.

One thing that differentiates OpenSec from Procera and Frenetic is how it reacts to security alerts raised by external units. Although this feature is supported by the latter two, it is not considered the main way to operate them. In OpenSec, it is. In Frenetic and Procera, all reaction is defined by the code written by the operator. In a nutshell, Frenetic and Procera allow for a more granular control of the flow setup whereas OpenSec hides such events from the operator and reacts automatically.

Although other studies have addressed policy-based network administration using OpenFlow, as well as providing security through SDN, OpenSec's innovative approach allows operators to customize the security of the network using human-readable policies and to customize how the controller reacts automatically when malicious traffic is detected.

## III. Motivation

The main goal of OpenSec is to be a human-friendly, dynamic and automated security framework. Next, we describe three design requirements of our framework: moving middleboxes away from the main datapath, reacting automatically to security events and creating a simple policy specification language.

### A. Moving Middleboxes Away From the Main Datapath

Sekar et al. show that, in a given enterprise, there are almost as many network appliances as there are routers [26]. Moreover, they point out how middleboxes do not favour network innovation, as they are closed system with no room for experimentation. Indeed, middleboxes are harder to update, upgrade or replace when compared to standard Ethernet switches.

The first goal of OpenSec is to move the middleboxes away from the choke points of the topology traversed by all traffic. Instead, these devices should be located outside of the main path between the LAN and the Internet and should act as security processing units that are visited only by the traffic that needs to be processed. Using a smarter OpenFlow-based control plane, the OpenSec should dynamically create rules to re-route traffic. This is important in terms of performance and reliability, since a L2 switch is easier to maintain, upgrade or replace in comparison to specialized hardware. When a specific flow is subject to deep packet inspection (DPI), for example, then the controller adds a rule that forces such traffic to visit the DPI processing unit.

To allow OpenSec to scale better, the processing units are responsible for analyzing the traffic and detecting malicious flows. Sampling traffic at the controller increases the chances of introducing a bottleneck and increases the complexity. Also, the connectivity between switches and controller is usually of low bandwidth. In contrast, the data plane allows a faster bit rate and the processing units are optimized to handle big flows. In OpenSec, the controller remains listening to alerts and reacts to those alerts by deciding how to modify the traffic rules.

### B. Reacting Automatically to Security Events

The second goal of OpenSec is to enable automated reaction to security events. When a middlebox detects suspicious traffic, it issues a security alert. Traditionally, these alerts are received by a network operator who then decides how to react to them. With OpenSec we aim at automating this reaction so that the framework either blocks the traffic, or simply alerts the operator of the detected malicious traffic. The reason why this is feasible is because, in general, an operator can plan ahead of time how critical a flow is based on the service provided through that flow. In a production network, for example, an operator would want a denial of service attack to be stopped as soon as possible. In contrast, a testing environment could be less critical. Thus, we designed OpenSec to allow the operator to specify ahead of time what the automated reaction should be, so that in case of malicious traffic, the human participation is minimized.

TABLE I
SYNTAX TO CREATE POLICIES USING OPENSEC

| | Value | Description |
|---|---|---|
| Flow | inPort, VLAN, etherSrc, etherDst, ipSrc, ipDst, TCP-SrcPrt, TCP-DstPrt | Uses OpenFlow match fields to describe a flow |
| Service | Encrypt, IDS, DPI, spam, DDoS or any other service registered | Identifies a security service that should be applied to the flow |
| React | alert, quarantine, block | Determines how to react if the service reports malicious content |

### C. *Creating a Simple Policy Specification Language*

The third goal of OpenSec is to provide a simple policy specification language to allow operators to redirect traffic to the middleboxes and to enable automated reaction. Among all related work, Procera (see Section II-C) is probably the one that has focused most on designing human-readable policy definition. However, we argue that understanding a definition written in Procera is not straightforward. For example, the following instructions define a rule that allows all traffic:

**Procera: proc** *world* → **do**; **returnA:** λ *req* → *allow*

This statement still contains symbols that make it complicated to read. Instead, we aim at statements such as:

**OpenSec: Flow:** VLAN=192; **Service:** DPI; **React:** alert.

This OpenSec sample policy is described in Section IV-A. However, note that three components can easily be identified: the matching pattern, the security units that must be visited by this flow and the type of automated reaction.

Procera relies on reactive programming and its goal is different from OpenSec. In Procera, the goal is to program the network using policies and this includes handling events generated by the switches. In contrast, OpenSec does not communicate the network events to the end-user. Instead, they are automatically processed. This allows us to use a much simpler syntax to describe the flow, identify one or more services and specify how to react when malicious traffic is detected. Policies can be defined using the keywords shown in Table I.

## IV. OPENSEC COMPONENTS

OpenSec is an SDN framework capable of forwarding flows to security processing units based on policies and to automatically react to events raised by these middleboxes. Using this framework, security devices such as intrusion detection middleboxes, firewalls or encryption units can be removed from the main data path between the LAN and the Internet. OpenSec leverages a smart control plane to allow end-users to direct only part of the traffic to these security units.

In this section we describe the main components of the framework: a policy specification language, a northbound interface, a policy manager, a set of processing units, a security event processor, an OpenFlow controller and a data repository. Then,

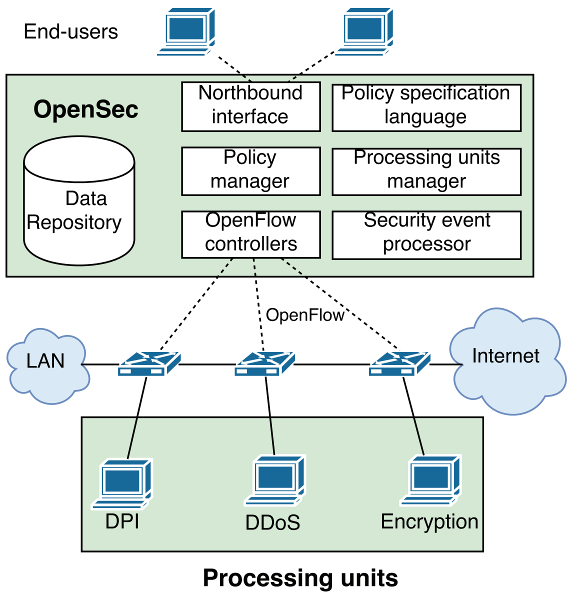


Fig. 1. The OpenSec framework: Security functions are provided by the processing units; traffic is routed to each processing unit based on requirements given through security policies; the reaction to security alerts is automated.

in sections V and VI we explain how these components interact with each other to implement the policies and to react to security alerts issued by the processing units.

### A. *Policy Specification Language*

OpenSec's policy specification language allows us to specify a matching pattern, a list of security units that should be traversed by such traffic and an automated reaction in case of receiving a notification from a unit (see Table I). The matching fields correspond to those available in OpenFlow 1.0. The service corresponds to any service ID registered by the processing unit manager. Finally, the reaction can be to alert only (via email), to block or to re-route traffic to a quarantine device.

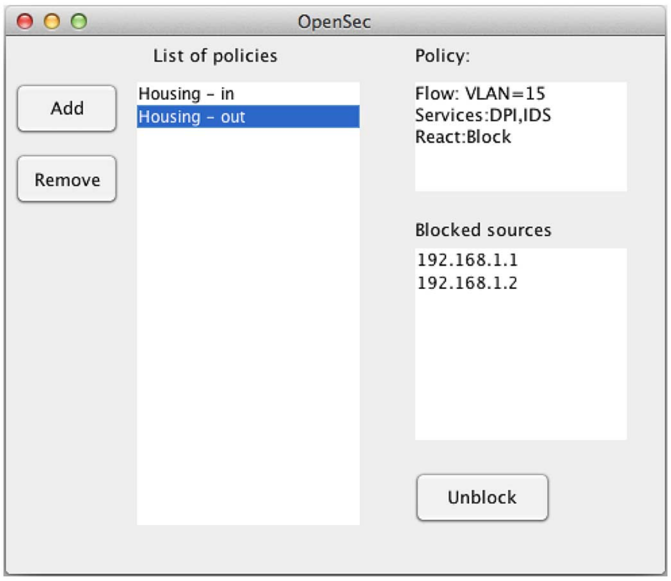
To illustrate how this language is used, we explain the example given in Section III-C:

**Flow:** VLAN=192; **Service:** DPI; **React:** alert.

The policy above specifies that all traffic tagged with VLAN 192 should be re-routed to the DPI unit. Also, if the DPI middlebox informs of a suspicious sender, OpenSec must only alert the operator via e-mail. Several match fields and several units can be listed when specifying the policy.

### B. *Northbound Interface*

The current prototype of OpenSec includes a graphical user interface shown in Fig. 2. The list on the left shows all policies currently implemented and the buttons on the left allow for adding or removing a policy. On the right side, the detailed policy is shown on top and the sources that have been blocked automatically using that policy are shown below. Finally, the operator can unblock a source. A similar GUI is provided to the user to show the information of each registered processing unit, similar to the data shown on Table II.



Housing – in
Housing – out

Flow: VLAN=15
Services:DPI,IDS
React:Block

192.168.1.1
192.168.1.2

Fig. 2. OpenSec's graphic user interface. This interface allows the network operator to add, remove and view policies. It can also be used to re-authorize blocked sources.

TABLE II
REGISTERED SECURITY PROCESSING UNITS FOR FIG. 1

| Service type | Switch ID | In-interface | Out-interface |
|---|---|---|---|
| DPI | 1 | 25 | 26 |
| DDoS | 2 | 48 | 49 |
| Encrypt | 3 | 25 | 26 |

OpenSec should allow external applications to implement network security policies automatically. As a consequence, there is a need for a northbound Application Programming Interface (API) between OpenSec and the applications. The development of such an API is part of our future work.

## C. Policy Manager

The policy manager is a core component of OpenSec. It is responsible for parsing new policies sent by the GUI and converting them to OpenSec objects. Next, it must implement the policy using the southbound interface component (controller). Finally it must also check periodically that the policy is implemented appropriately. The operation of the policy manager is explained in detail in Section V.

## D. Processing Units

OpenSec relies on external processing units (or middleboxes) to analyze traffic. The units are customized to perform the required security scan, such as a firewall, an IPS or DPI. When suspicious traffic is detected, the processing unit issues an alert to the OpenSec controller so that actions can be undertaken based on existing policies. For this to work, all units must be known to the OpenSec controller.

OpenSec implements a processing unit manager that collects all the registrations and creates a list of units and the locations in the network where they can be found. In our current implementation, each unit is mapped to a service id (DPI, IPS), a switchID, an input port and an output port (see Table II). This is all the data needed by OpenSec to manipulate the flow table of the devices in order to re-route traffic to the processing units.

Note that, if a processing unit is vendor-specific and this automatic registration cannot be implemented, a network operator can easily complete the information in the controller manually. In our current implementation, both automatic and manual registrations are supported.

## E. Security Event Processor

One of the most important features of OpenSec is the automatic reaction to security alerts. Usually, a network operator will react to an alert by either ignoring it or blocking the source of the suspicious traffic. In OpenSec, the network operator can define such a reaction in advance using three possible choices: *alert, quarantine* or *block*. The security event processor is responsible for collecting the notifications issued by the processing units and modifying forwarding rules according to the policies involved. This component is explained in detail in Section VI.

## F. OpenFlow Controller

OpenSec uses OpenFlow to interface with the switches. To do so, a module running in the Floodlight controller [27] implements the required interfaces to listen to network events and communicate with switches. When a request is received from the policy implementer to push a new rule, this module is responsible for sending the message to the right switches. A more detailed explanation of how an OpenSec policy is converted to a list of OpenFlow messages is provided in Section V.

Although the controller of OpenSec is centralized, multiple controllers can work together to increase the availability of the control plane. In the current implementation we rely on a single software to do this. However, it has been proposed to have multiple controllers working in a synchronized way [28]–[31]. We do not address the implementation of a distributed control plane in this paper.

## G. Data Repository

OpenSec uses a data repository to store several pieces of information. First, all implemented policies are stored to check for conflicts when new policies are received, and also to know how to react to security events raised by the processing units. Similarly, all the information needed to route traffic to the middleboxes (device id, switch id, input port and output port) is also stored. Finally, OpenSec also records when hosts are blocked from accessing the network.

Next, we describe how these components interact with each other. First, we explain the operation of OpenSec to implement security policies. Then, we describe how the framework reacts to alerts issued by processing units.

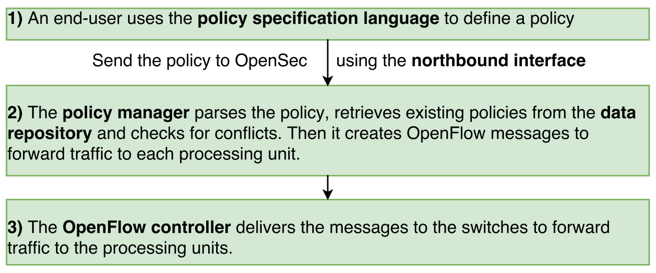

**1)** An end-user uses the **policy specification language** to define a policy
Send the policy to OpenSec
using the **northbound interface**
**2)** The **policy manager** parses the policy, retrieves existing policies from the **data repository** and checks for conflicts. Then it creates OpenFlow messages to forward traffic to each processing unit.
**3)** The **OpenFlow controller** delivers the messages to the switches to forward traffic to the processing units.

Fig. 3. Steps needed to implement a policy.

## V. Operation of OpenSec: Policy Implementation

In this section we describe how the policy manager of OpenSec converts a new policy created through the GUI into forwarding rules in the switches. Figure 3 provides an overview of the entire process.

First, the policy manager receives the policy definition file from the northbound interface (GUI) component. Next, it converts it into a Policy object that can be processed by OpenSec to update flow rules in the switches. First, the parser builds a Flow object and sets the attributes based on the values given by the policies. A Flow object is a simple structure where we store the VLAN, MAC source and destination, IP source and destination and TCP port. Next, the parser queries the processing units manager (described below) to retrieve the switch id associated to the processing units specified in the policy, as well as the port needed to send data to those units. This is added as a collection to the Policy object. Finally, a flag is used to indicate whether the reaction in case of a security event is to alert, quarantine or block. Algorithm 1 shows how policy definitions are parsed.

Note that we intentionally keep the policy parsing function OpenFlow-independent. The fields used to describe a flow are standard and the Policy object can be used by any other SDN protocol to update the forwarding tables of switches.

Once the policy has been parsed, the policy manager verifies that it does not conflict with existing ones. Checking for conflicting rules is particularly challenging in deployments with multiple controllers sharing network control through a slicing technique such as FlowVisor [32]. In such scenario, the checker should verify that controllers do not use unauthorized resources and do not override rules pushed by other controllers. In the context of OpenSec, all rules are pushed by the controller and this simplifies the task. Our policy checker verifies that the Flow attributes of each Policy object do not conflict with each other. For example, two identical Flow objects cannot be part of different policies, since this will become ambiguous when the rules are pushed to the switches. Policy checking becomes increasingly important as the north API of OpenSec becomes more sophisticated. We plan to allow for multiple security applications to automatically add new policies and the policy checker needs to verify that those applications do not interfere with each other. However, we do not address this issue in the current work.

Once a policy has been parsed and checked for conflicts, the policy manager can convert all objects into OpenFlow-compliant rules. The policy implementer is the only component in OpenSec that is currently tied to the OpenFlow protocol.

**Algorithm 1.** Parsing a file into a policy object

```
Data: new file path $path$
//Parse new policy from file Policy policy = new Policy( )
lines = readFile($path$)
for each line f in lines do
    if line starts with 'Flow' then
        //Create a match based on fields match =
        createMatch(line);
    end
    if line starts with 'Service' then
        //save codes of units (DPI, DoS,...) servicesCollection
        = getServices(line);
    end
    if line starts with 'React' then
        //Remember expected reaction
        reaction = getReaction(line);
    end
    policy.setMatch(match);
    policy.setUnits(units);
    policy.setReaction(reaction);
    policy.setVlan(getNextVLAN( ));
end
```

It converts the Flow objects created by the policy parser into the specific instance needed by the OpenFlow controllers (described below) to push new flow rules into the switches. Note that several rules can be created simultaneously if the policy specifies multiple security units. Rules will be pushed into the switches attached to those devices.

When a new rule is pushed to a switch using OpenFlow, the message issued by the controller must include an input port, a match and a set of actions. Note that at least one rule will be inserted for each processing unit listed in the policy. Therefore, for each unit, OpenSec first finds the input port where traffic is expected. This is available in the data repository where all existing forwarding rules are inserted (see Table II). We assume that a rule already existed to carry the traffic either in our out of the local network and, as a consequence, we expect that OpenSec will find a rule that matches the pattern given in the policy. Once the input port has been found, the match is created based on the policy matching information. Finally, the action is computed as follows. First, an output port must be added so that traffic reaches the middlebox. Second, a VLAN tag must be added to uniquely map this traffic to a policy. Indeed, when a processing unit informs the controller that malicious traffic has been detected, the VLAN id and the source IP address are provided in the notification. These two fields uniquely map a source to a policy, allowing OpenSec to react to traffic coming from the identified source as specified by the policy. This is explained in more detail in Section VI. All the steps described are shown in Algorithm 2. As a result, $FlowMod$ OpenFlow messages are issued for each middlebox to match on the input port and the policy matching fields and to mirror traffic to the units through the port retrieved from the database.

Finally, one important step of policy-based management is policy enforcement. This step consists of periodically checking

**Algorithm 2.** Implementing a policy

```
Data: Policy policy
//Implement policy in network
servicesCollection = policy.getServicesCollection();
for each service u in unitsCollection do
    //For each service code, find the unit
    Unit unit = unitManager.getUnit(service);
    //Get DPID, inPort and match to create a flow rule
    dpid = unit.getDPID( );
    inPort = unit.getInPort( );
    match = policy.getMatch( );
    //Get input port from existing rule
    inputPort = database.findInputPort(match);
    //Get next available VLAN tag
    vlanTag = database.getNextTagAvailable();
    //Update existing rule
    writeFlowMod(match on: inputPort and match, actions: add
    vlan tag, output to port inPort);
end
```

that policies are effectively implemented. To do so, the controller issues *packet_in* messages that match the fields of each implemented policy. Next, the controller verifies that the issued packets were routed appropriately. One possible way to check this is to craft packets that will actually trigger alerts. Another one is to have the units notify the controller that a test message has been received. We are currently working on this component and therefore do not provide evaluation results yet.

### A. Step-by-Step Example

To summarize this section, we provide a step-by-step example of how an operator can implement a policy using OpenSec. Suppose that we implement the following policy using the sample network in Fig. 1:

**Flow:** VLAN=192; **Service:** DPI; **React:** block.

This policy ensures that traffic tagged with VLAN 192 is mirrored to the DPI unit. Also, any source sending malicious traffic should be blocked.

To implement the policy:

1) A network admin should write the policy using the GUI shown in Fig. 2.
2) OpenSec parses the policy and locates the switch where the DPI unit is connected, as well as the interface (switch 1, port 25 as shown in Table II).
3) OpenSec assumes that one or more rules already exist so that traffic from VLAN 192 can go through the network.
4) OpenSec finds the rule in switch 1 that matches packets tagged with VLAN 192 to get the appropriate input port.
5) OpenSec also finds the next VLAN tag available to identify traffic from this policy, assume it's VLAN 20.
6) OpenSec modifies the rule so that traffic is forwarded as specified by the original rule, but also forwarded to port 25. Additionally, the VLAN tag 20 is also added.

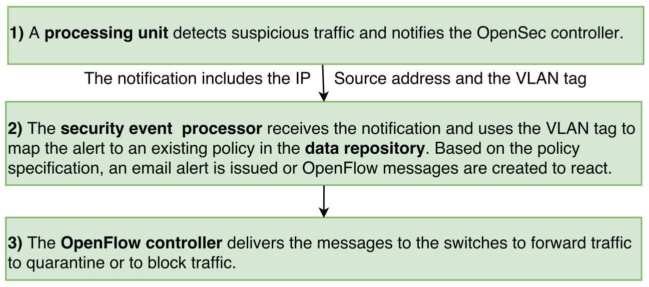

**1)** A **processing unit** detects suspicious traffic and notifies the OpenSec controller.
The notification includes the IP Source address and the VLAN tag
**2)** The **security event processor** receives the notification and uses the VLAN tag to map the alert to an existing policy in the **data repository**. Based on the policy specification, an email alert is issued or OpenFlow messages are created to react.
**3)** The **OpenFlow controller** delivers the messages to the switches to forward traffic to quarantine or to block traffic.

Fig. 4. Steps needed to react to a security event.

Once a policy has been implemented, traffic is routed to the processing units and security alerts might be issued by the middleboxes. Next we describe how OpenSec reacts to these alerts.

## VI. Operation of OpenSec: Reaction to Security Events

One key feature of OpenSec is the ability to automatically react to security alerts without involving the network administrator. In this section we describe in more detail how this process is achieved. The overall process is shown in Fig. 4.

OpenSec relies on the processing units to perform security scans on incoming traffic. In the current implementation, each middlebox must be configured by the network operator to do a specific task. However, enabling automated configuration by OpenSec is a possible future work direction. To be compatible with OpenSec, a processing unit must be capable of sending a message to the controller indicating that a tuplet {VLAN, IP source} is behaving suspiciously. As we described earlier, OpenSec tags traffic with a VLAN to uniquely map it to a given policy. Therefore, by sending the VLAN tag and the IP source, the processing unit identifies the sender but also indicates to OpenSec which policy caused this flow to be routed to the unit.

While the processing units scan traffic, the security event processor listens to notifications from processing units using application layer sockets. When a new message arrives, a new threat handles it. First, the process reads the VLAN tag and the IP source of the suspicious node. Next, the process queries the data repository to get the policy mapped to the received VLAN tag. Finally, the process retrieves the type of reaction specified in the policy. As a result, OpenSec now knows if the source must be blocked or sent to the quarantine unit. If the specified reaction is 'alert,' forwarding rules are not modified and the network administrator is notified by e-mail. Otherwise, we describe next how to modify the forwarding rules.

The creation of new rules depends on the reaction type. To quarantine traffic, a processing unit logging all traffic is attached to one of the switches on a given port. By default, a rule exists on all switches to forward to the quarantine unit all traffic tagged with a specific VLAN tag. Therefore, to react to such an attack, OpenSec must simply insert a rule at the edge switch that will tag all incoming traffic from the suspicious source with the VLAN tag associated to the quarantine unit. Similarly, if the policy indicates that traffic should be dropped, then OpenSec inserts a rule at the edge switch to do so.

### A. *Step-by-Step Example*

Consider once again the example described in Section V-A that forwards traffic to a DPI unit and blocks suspicious sources. Now we describe the steps followed when an alert is sent by the DPI unit to OpenSec.

1) The DPI processing unit is configured by the administrator to perform some security scan.
2) The processing unit detects malicious traffic coming from source 174.145.23.3 and sends a notification to the controller: {20, 174.145.23.3} (20 is the VLAN used in the example started in Section V-A, the IP source is only an example).
3) OpenSec retrieves from the data repository the policy mapped to VLAN tag 20 and the reaction specified by the policy (block).
4) OpenSec issues a flowmod message to the edge switch asking to block all traffic coming from source 174.145.23.3.

Next we describe the experimental setup used to evaluate OpenSec.

## VII. Experimental Setup

To evaluate the performance of OpenSec, we first describe three experimental scenarios deployed on the GENI testbed. For all deployments, layer 2 connectivity is provided between all nodes with 100 Gbps capacity. All nodes run the Ubuntu 12.04 operating system and OpenFlow-compliant switches are implemented using Open vSwitch. Finally, the average latency between a switch and a controller is 1 ms. Likewise, the average latency between processing units and the OpenSec controller is 5 ms.

### A. *Scenario 1: Multi-Switch and Multi-Unit Network*

For scenario 1, we deployed in GENI a linear topology with seven switches (1-7) and two processing units connected to each switch (1.1, 1.2, 2.1, 2.2, etc.). The goal of this network is to evaluate the time needed by OpenSec to implement rules across multiple devices. Therefore, the processing units do not implement any security function, as we only use them to force OpenSec to send OpenFlow messages to different switches.

Using this scenario, we implement the following three policies.

**Policy 1.1: Flow:** VLAN=1; **Service:** 1.1, 2.1, 3.1; **React:** alert.

**Policy 1.2: Flow:** VLAN=2; **Service:** 1.1, 1.2, 2.1, 2.2, 3.1, 3.2; **React:** alert.

**Policy 1.3: Flow:** VLAN=3; **Service:** 1.1, 1.2, 2.1, 2.2, 3.1, 3.2, 4.1, 4.2, 5.1, 5.2, 6.1, 6.2, 7.1, 7.2; **React:** alert.

### B. *Scenario 2: Incoming Traffic in a Campus Network*

The goal of scenario 2 is to demonstrate how OpenSec can block malicious traffic from entering a campus network. To do so, we used a dataset made available by the University of Twente that comprised of traffic directed to a honeypot [9]. The honeypot was connected to the Internet and ran network services such as SSH, FTP, HTTP and so on. By design, only suspicious traffic was forwarded to that node. The traffic trace was also labeled and organized in a database of flows, alerts and alert types. As a consequence, it is very convenient for experimentation because each flow has been labeled with one attack type, such as SSH scan, SSH connection, FTP scan, FTP connection or HTTP connection. Table III shows a detailed classification of the dataset.

TABLE III
Type of Traffic in the Dataset

| Traffic type | Number of IP sources | Number of flows |
|---|---|---|
| SSH_conn | 103,104 | 13,939,813 |
| FTP_conn | 5 | 12 |
| Entire dataset | 107,988 | 14,170,132 |

We also developed two security units: one for intrusion detection and one for deep packet inspection. The IDS unit runs Bro [33], an open source network analysis framework capable of intrusion detection. The DPI unit is built on top of nDPI [34], an open source DPI tool. nDPI supports all major networking protocols at any layer, such as IPv4, IPv6, UDP, TCP, HTTP, DNS, SSH, SMTP, Flash and many others. The IDS unit is configured to detect ping flooding and SYN flooding attacks by sending an alert when a given source sends more than 25 ICMP echo requests per second. Similarly, if a node sends multiple TCP handshake requests and then drops the connection, an alert is also raised.

The experimental setup using the GENI testbed for this scenario is as follows. We replayed the honeypot traffic dataset from a node in the Internet to replicate all the messages. To detect suspicious SSH connections, we configured Bro to alert when a single source attempts more than six connections every five seconds or less. This decision is based on the traffic received by the honeypot where a majority of ssh connections were attempted every five seconds in average. Moreover, we created a policy that mirrors to an IDS all incoming traffic destined to port 22.

For this experiment, we implemented the following policy:

**Policy 2.1: Flow:** VLAN=15; **Service:** IDS, DPI; **React:** block.

### C. *Scenario 3: Deploying a Science DMZ*

Science networks carry high-speed data transfer flows that need high bandwidth and are very susceptible to packet loss. Therefore, the goal of a Science DMZ is to route traffic through a path with customized controls that ensure an acceptable level of security while guaranteeing a high-speed loss-free channel. In Scenario 3, we implement a Science DMZ using OpenSec.

We created the network shown in Fig. 5b in GENI. The testing devices (email senders, data transfer nodes and web-server users) are deployed in the GENI aggregate located at the University of Illinois. The campus network is deployed in the

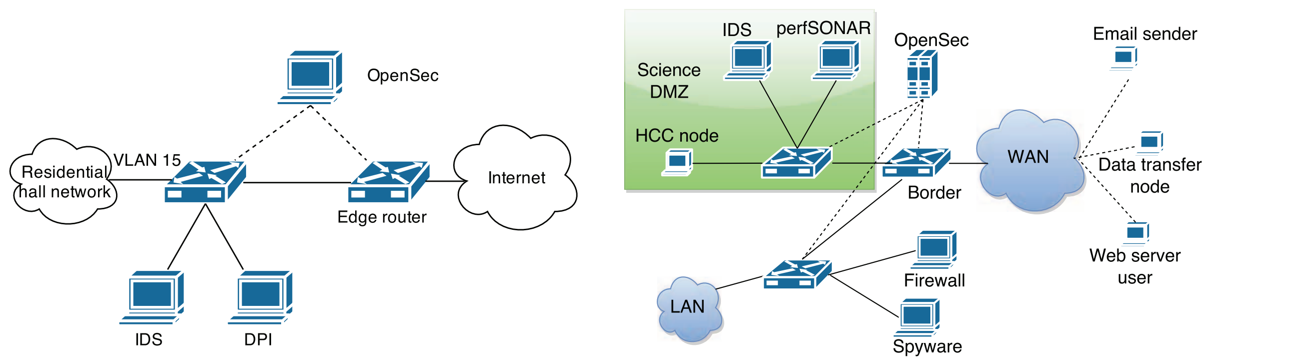


Fig. 5. (a) Shows the experimental topology deployed in GENI for scenario 2. (b) Shows a Science DMZ for a campus network deployed in GENI for scenario 3.

University of Utah InstaGENI aggregate. In this experiment, the aggregates are "stitched" together through a layer-two tunnel. This connectivity is provided by the GENI testbed. We also scale up the experiment using a scaling technique provided by GENI so that there are 50 nodes deployed in the Illinois aggregate. Out of 50 nodes, 10 send scientific data at a high rate, 25 send requests to the web server located in the LAN and 15 send email traffic.

For this experiment, we implemented the following three policies. The first policy is responsible for Science DMZ security controls. The second one forwards all LAN traffic to the firewall and the third one sends LAN traffic destined to TCP port 25 to the spyware detection unit.

**Policy 3.1: Flow:** EtherPort =1; **Service:** perfSONAR, IDS; **React:** alert.

**Policy 3.2: Flow:** EtherPort =1; **Service:** Firewall; **React:** alert.

**Policy 3.3: Flow:** TCP-dp=25; **Service:** Spyware; **React:** block.

The first row of Table VI shows the policies used to realize the required security behavior. On the Science DMZ side, all traffic is mirrored to the IDS and the perfSONAR units. On the LAN side, all traffic is sent to the firewall and then forwarded to the LAN once it has been inspected by the firewall. Also, incoming mail is forwarded to the spyware detection unit. The second row of Table VI shows how equivalent policies would be implemented using Procera, a similar framework. A comparison between these frameworks is provided later on.

### D. *Large Number of Policies*

Finally, we also experimented with implementing a large number of policies to evaluate the performance of the conflict check mechanism. To do so, we generated up to 1000 random policies and we measured the time needed to add a new policy afterwards. In the next section we show the result of our experiments.

## VIII. Evaluation

Next we evaluate the performance of OpenSec in implementing policies and reacting to security alerts. We also demonstrate the benefits of automated blocking, as well as the advantages of moving middleboxes away from the main datapath and intelligently mirroring traffic using OpenSec.

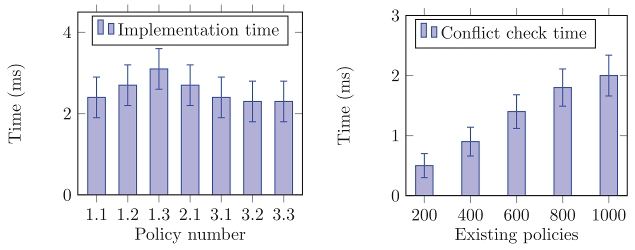


Fig. 6. (a) Shows the time needed to implement all policies. (b) Shows the time needed to check for conflicts when a new policy is added.

### A. *Performance of Policy Implementation*

We first measure the time needed to implement network policies. To do so, we computed the time needed to implement all the policies described in the experimental setup. The results are shown in Fig 6a. Policies 2.1, 3.1, 3.2 and 3.3 correspond to realistic scenarios and the implementation time is in the order of a few milliseconds. We used policy 1.3 to evaluate if the number of switches and processing units affects the implementation time. Indeed, this policy shows the highest time, but it is still in the order of milliseconds. In fact, given that the average latency between the controller and the switches is one millisecond, our results show that a majority of the implementation time corresponds to this communication.

Next, we also measured the time needed to check for conflicts when a new policy is implemented. As described in the experimental setup, we implemented up to 1,000 policies to do this and the results are shown in Fig. 6b. Although the check time increases linearly, it remains in the order of a few milliseconds. Furthermore, it is unlikely that such a large number of policies will be implemented on a campus network. However, this shows that OpenSec scales well.

### B. *Performance of Reaction to Security Alerts*

Next we measure how long it takes for OpenSec to react after a security alert has been issued. Suppose that, in scenario 3 (Science DMZ), a malicious sender is detected by the DPI unit.

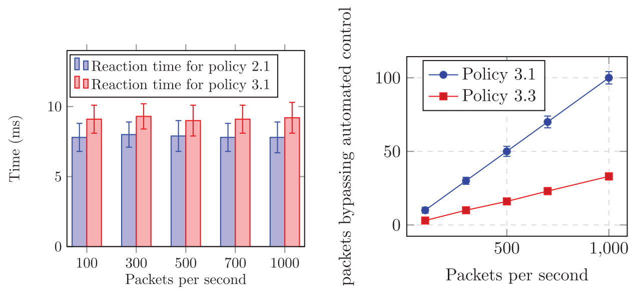


Fig. 7. (a) Shows time elapsed between the detection of malicious traffic and the blocking of the source. Independently of the traffic rate, the time needed by OpenSec to detect malicious traffic and block the sender remains constant. (b) Shows the number of packets that go through after detection of malicious traffic. As the traffic rate increases, the number of packets that go through while the blocking is being implemented grows linearly.

As shown in Fig. 7a, the delay needed to block malicious traffic remains constant independently of the traffic rate. This delay remains constant because it does not depend on the number of attacks detected or the packet arrival rate. For every alert, the controller simply finds the matching policy and modifies the forwarding rules as required. Fig. 7a also shows the reaction time for scenario 2.

Note that a certain delay exists between the detection time and the moment when traffic is actually blocked by the OpenSec controller. Due to the fact that the delay remains constant, the number of packets that bypass the quarantine unit grows linearly, as shown in Fig. 7b. In this figure, the number of packets for policy 3.1 grows faster because more packets are being received compared to policy 3.3. Indeed, for policy 3.1, all traffic directed to the Science DMZ is mirrored. In contrast, policy 3.3 only mirrors mail traffic, so only part of the experimental data is received by the spyware unit.

We conclude that the policies deployed for this evaluation work well to detect attacks that are carried over multiple packets, such as a denial of service attacks. In such scenarios, reacting to the attack after a small number of packets have reached the server is acceptable because the number of packets that go through is too small to effectively launch an attack. However, if the requirement is to detect smaller attacks that are carried over a small number of packets (SQL injection, for example), then in-line processing is needed.

## C. Benefits of Automated Blocking

Next we evaluate the benefits of automatically blocking a sender node. In scenario 2, policy 2.1 mirrors all traffic to an IDS and a DPI units. In our implementation, a source is blocked when it attempts connections every five seconds for at least six times. A close observation of the start and end times of each connection yields that, out of 1000 connections, an average of 20 will go unnoticed by OpenSec because there is a pause of more than five seconds between two or more alerts.

Without blocking, 13,939,813 flows would reach the local network. However, when the IDS detects forged SSH handshakes, the sender of the packets is automatically blocked. Table IV shows the number of flows that reach the destination instead of been blocked, as well as the number of sources blocked. Note that 95.5% of flows are stopped before reaching the destination and 99% of attacking nodes are blocked from the network.

TABLE IV
RESULTS OF IMPLEMENTING THE BLOCKING POLICY

| Total flows | Flows reaching destination | Total IP sources | Sources blocked |
|---|---|---|---|
| 13,939,813 | 628,624 | 103,104 | 102,085 |

## D. Trade-Offs of Mirroring Traffic

Next we discuss the benefits and limitations of using OpenSec to remove the middleboxes from the data path and intelligently mirror part of the traffic. First, in scenario 3 we send high-rate flows to a host in the Science DMZ and we do the same with a host in the LAN. Figure 9a shows the result of comparing these two transfers. In both cases we use the secure copy (scp) tool to transfer a large file and we measure the number of bytes per second received at each host. We observe that the end-host in the Science DMZ receives a constant number of bytes per second, whereas the rate of traffic sent to the host in the LAN decreases. The reason for this decrease is that the path through the campus network is much slower than the one traversing the Science DMZ and packet loss occurs at the firewall and the spyware detection units. Thus, the TCP implementation of the sender assumes that this is due to congestion and lowers the transmission rate. For this reason, after the initial decrease the throughput becomes stable.

Second, we measure the amount of traffic that traverses the firewall and we compare it with the amount of data routed to the spyware detection unit. Table VI shows the OpenSec policies in this experiment. Note that the third policy ensures that only TCP-25 traffic goes to the spyware unit. As a result, Fig. 9b shows the difference between the traffic going to the security units. By filtering mail traffic, we significantly reduce the load of the spyware unit. The second row of Table VI also shows how the equivalent functionality would be achieved in Procera. Notice how the syntax is closer to a programming language rather than a simple list of match fields and security services. While Procera provides other functionalities, OpenSec's syntax is sufficient to provide adequate routing of traffic to the appropriate middleboxes.

Next we show the advantage of removing middleboxes from the main data path in terms of throughput. There are two factors that impact throughput based on our experiments: increased latency and packet loss. To show the impact of latency on the throughput, we first measured how the round-trip delay increases as the traffic traverses more units placed on the datapath (see Fig. 8a). The impact of this increase on the throughput can be observed in Fig. 8b. In both figures, we show how the latency and the throughput remain constant when OpenSec is used to mirror traffic to the units instead of traversing all the middleboxes for in-line processing. Although this impact is considerable, the packet loss caused by in-line processing is even more significant. To illustrate this, suppose that a supercomputing facility wants the IDS to analyze all incoming traffic.

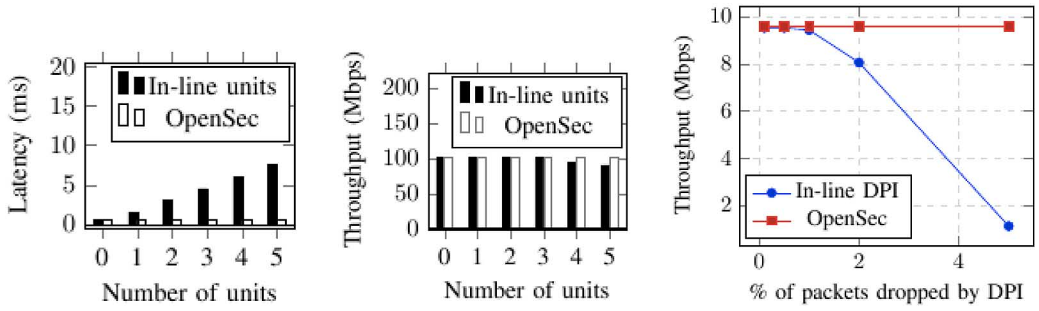


Fig. 8. (a) Shows the increase in round-trip latency as more middleboxes are traversed by the traffic, (b) Shows the decrease in throughput as more middleboxes are traversed by the traffic due to the increased latency (5% loss). (c) Shows TCP throughput achieved using OpenSec and in-line DPI using a 10Mbps link. The packet loss caused by in-line DPI reduces the throughput significantly, whereas it remains constant when using OpenSec because the traffic is only mirrored to the DPI and the packet loss is smaller.

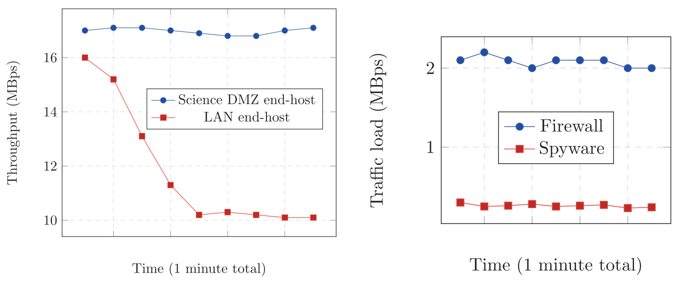


Fig. 9. (a) Shows the number of bytes received by the firewall and the spyware detection units. The amount of traffic that visits the spyware detection unit is lower because only traffic with destination port TCP 25 is routed through this unit. (b) Shows the number of bytes received by end-hosts in the Science DMZ and the LAN. The host in the science DMZ receives more traffic because the path between end-points is faster. For the host in the LAN, security devices such as the firewall decrease the performance and the traffic rate is lower.

However, security middleboxes such as firewalls and IDS units are not yet capable of dealing with big data flows without dropping packets. Dart et al. [35] show the TCP throughput can be reduced by a factor of 9 if a router is losing a very small percentage of traffic. To verify this, we simulated a DPI unit that is traversed by all traffic and then sends all data back to the main data path. To evaluate the benefit of using OpenSec, we compared the throughput achieved using an in-line IDS with the throughput achieved using OpenSec. We experimented with different percentages of packet drops and Fig. 8c shows how this solution heavily impacts the TCP throughput between the server and the client. By duplicating the traffic, OpenSec increases the amount of traffic, but also allows the data to traverse the network at a faster rate.

However, there are also limitations when mirroring traffic. We showed earlier that when mirroring is used, some malicious packets can still reach the server after the alert was issued. To counter this, OpenSec can also achieve real-time blocking using a a different policy that sends all traffic to the unit and then receives all traffic back and forwards it. This allows the processing unit to drop packets. However, we showed how having an in-line processing unit can significantly reduce the throughput of a flow. Therefore, a trade-off exists between achieving higher performance and having a faster response time to block suspicious packets.

In the next section, we compare OpenSec to similar solutions described in Section II.

## IX. Comparison With Existing Solutions

In this section we compare OpenSec against CloudWatcher [8], Fresco [10] and Procera [7], the three similar solutions described in Section II-C. A peer-to-peer comparison is provided and Table V summarizes the main differences.

### A. Procera

The main advantage of OpenSec with respect to Procera is simplicity. We showed in Table VI how OpenSec's syntax is simpler than Procera's to deploy a Science DMZ. Also, the fact that OpenSec does not expose switch events to the end-user simplifies the network administration. We do not provide a quantitative comparison because no comparable numerical results are provided in Voelli et al. [7].

### B. CloudWatcher

Next, we compared the time needed by OpenSec to translate policies into OpenFlow messages with the results achieved by CloudWatcher. The results, shown in Table VII, do not include the time needed to send the message from the controller to the switch, but only the time needed to translate a policy into a set of OpenFlow messages. Table VII shows the results when there are one, two or three processing units involved in the policy. The experiment consists of translating 50 different random policies. Because CloudWatcher evaluates multiple algorithms, a range of time is given. In all cases, OpenSec achieves a faster time because we do not consider routing in our proposed solution. However, we do note that times for both solutions are similar.

### C. Fresco

Table VIII compares the time needed to implement the network rules using only one controller, Fresco or OpenSec. The results show that OpenSec needs less time to parse the policy and push rules into the switches. The experiment consists of implementing 50 different policies with random matching fields and automated blocking.

TABLE V
SYNTAX TO CREATE POLICIES USING OPENSEC AND PROCERA

| Solution | Improvement in OpenSec | Not addressed by OpenSec |
|---|---|---|
| Procera | Human readability: OpenSec's policy specification language is simpler to understand. See Table VI to compare how policies 3.1, 3.2 and 3.3 would be written in both frameworks. Procera's syntax is closer to a programming language. | Procera enables dynamic policies, such as reacting one a user has exceeded a data quota. |
| CloudWatcher | Reaction to security alerts: OpenSec considers automated reaction while CloudWatcher does not. See Table VII for a numerical comparison. | CloudWatcher considers routing decisions to find the best processing unit. OpenSec assumes that processing units are one hop away. |
| Fresco | Human readability: Fresco allows operators to specify security modules that also look like code snippets, as opposed to security policies as in OpenSec. See Table VIII for a numerical comparison. | Fresco allows for security modules to inter-operate, whereas OpenSec's policies are independent and do not exchange any type of information. |

TABLE VI
SYNTAX TO CREATE POLICIES USING OPENSEC AND PROCERA

| | | | |
|---|---|---|---|
| OpenSec | Flow: EtherPort = 1<br>Service: perfSONAR, IDS<br>React: alert<br>*/* Security controls for the Science DMZ */* | Flow: EtherPort = 1<br>Service: Firewall<br>React: alert<br>*/* Firewall for all traffic going to the LAN */* | Flow: TCP-dp = 25<br>Service: Spyware<br>React: block<br>*/* Spyware detection of incoming mail */* |
| Procera | **proc** *world* $\rightarrow$ **do**<br>**returnA:**<br>$\lambda$ *req* $\rightarrow$ **if** EtherPort=1 **and** security event already exists<br>*alert*<br>**else**<br>*allow* $\odot$ *redirect 10.10.1.1* $\odot$ *redirect 10.10.1.2* | **proc** *world* $\rightarrow$ **do**<br>**returnA:**<br>$\lambda$ *req* $\rightarrow$ **if** EtherPort=1 **and** security event already exists<br>*alert*<br>**else**<br>*allow* $\odot$ *redirect 10.10.1.3* | **proc** *world* $\rightarrow$ **do**<br>**returnA:**<br>$\lambda$ *req* $\rightarrow$ **if** TCP-dp=25 **and** security event already exists<br>*deny*<br>**else**<br>*allow* $\odot$ *redirect 10.10.1.4* |

TABLE VII
TIME NEEDED TO CREATE OPENFLOW RULES IN OPENSEC AND CLOUDWATCHER FOR A SINGLE POLICY

| | One unit | Two units | Three units |
|---|---|---|---|
| OpenSec | 0.07 ms (s.d 0.002) | 0.07 ms (s.d 0.002) | 0.07 ms (s.d 0.002) |
| CloudWatcher | 0.1-1.1 ms | 0.1-1.15 ms | 0.1-1.2 ms |

TABLE VIII
TIME NEEDED TO CREATE AND PUSH OPENFLOW RULES IN OPENSEC AND FRESCO FOR A SINGLE POLICY

| | NOX/Floodlight | Scan detection |
|---|---|---|
| Fresco | 0.823 ms (using NOX) | 2.461 ms |
| OpenSec | 0.45 ms, s.d 0.01 (using Floodlight) | 0.46 ms, s.d 0.02 |

## X. CONCLUSION

In this paper we presented OpenSec, an OpenFlow-based framework that allows network operators to describe security policies using human-readable language and to implement them across the network. OpenSec acts as a virtual layer between the user and the complexity of the OpenFlow controller and automatically converts security policies into a set of rules that are pushed into network devices. OpenSec also allows network operators to specify how to automatically react when malicious traffic is detected. OpenSec allows for automated reaction to security alerts based on pre-defined network policies. By doing so, it contributes to hiding the complexity of the network to security operators, who only need to focus on defining the policies.

Our evaluation shows several advantages of OpenSec. First, moving the analysis of traffic away from the controller and into the processing units makes our framework more scalable. Even when the load is high, the controller is not a bottleneck. Second, OpenSec is a first step towards moving the security controls away from the core of the network. This is a key requirement in a network that leverages Cloud security, for example. Instead of controlling every device, the local network just sends data to the cloud and reacts based on the alerts received by the cloud service provider. Third, OpenSec fits well in scenarios that require mirroring of traffic to monitoring devices. This is particularly true for the Science DMZ use case that we demonstrated, because of the significant throughput gain achieved. This use case also showed how multiple policies can be combined together to achieve a campus-wide deployment. Likewise, the use case of residential networks showed how a simple policy can be used to control network access from housing networks. Moreover, our scalability results show that OpenSec achieves a constant reaction time for different traffic rates.

Future work includes securing the OpenFlow framework and enforcing policy implementation. In order to deploy OpenSec at any production level, several security measures must be taken into consideration, such as switch authentication, physically distributed control plane and also authenticating the users that are allowed to add policies to OpenSec. We plan to deploy OpenSec on the Holland Computing Center (HCC) network at the University of Nebraska-Lincoln campus. The HCC handles flows with high data rate for scientists working with the Compact Muon Solenoid (CMS) particle detector. This will provide a unique opportunity to deploy OpenSec at production scale. Finally, we plan to investigate how to speed up in-line processing using programmable boards such as NetFPGA cards to achieve a tighter integration between the OpenSec controller and the processing units.

## REFERENCES

[1] A. Lara and B. Ramamurthy, "OpenSec: A framework for implementing security policies using OpenFlow," in *Proc. IEEE Globecom Conf.*, Austin, TX, USA, Dec. 2014, pp. 781–786.

[2] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using OpenFlow: A survey," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 1, pp. 493–512, Feb. 2014.

[3] A. Lara, A. Kolasani, and B. Ramamurthy, "Simplifying network management using software defined networking and OpenFlow," in *Proc. IEEE Int. Conf. Adv. Netw. Telecommun. Syst. (ANTS)*, Bangalore, India, Dec. 2012, pp. 24–29.

[4] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 1–12, Oct. 2007.

[5] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Commun. Mag.*, vol. 51, no. 2, pp. 114–119, Feb. 2013.

[6] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[7] A. Voellmy, H. Kim, and N. Feamster, "Procera: A language for high-level reactive network control," in *Proc. Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, Helsinki, Finland, Aug. 2012, pp. 43–48.

[8] S. Shin and G. Gu, "CloudWatcher: Network security monitoring using OpenFlow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?)," in *Proc. 20th IEEE Int. Conf. Netw. Protocols (ICNP)*, Austin, TX, USA, Oct. 2012, pp. 1–6.

[9] R. Hofstede, L. Hendriks, A. Sperotto, and A. Pras, "SSH compromise detection using NetFlow/IPFIX," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 5, pp. 20–26, 2014.

[10] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson, "FRESCO: Modular composable security services for software defined networks," in *Proc. Netw. Distrib. Syst. Sec. Symp. (NDSS)*, San Diego, CA, USA, Feb. 2013, pp. 1–16.

[11] A. K. Bandara, E. C. Lupu, and A. Russo, "Using event calculus to formalise policy specification and analysis," in *Proc. IEEE Workshop Policies Distrib. Syst. Netw.*, Lake Como, Italy, Jun. 2003, pp. 26–39.

[12] A. K. Bandara, E. C. Lupu, J. Moffett, and A. Russo, "A goal-based approach to policy refinement," in *Proc. IEEE Workshop Policies Distrib. Syst. Netw.*, Yorktown Heights, NY, USA, Jun. 2004, pp. 229–239.

[13] A. K. Bandara, A. Kakas, E. C. Lupu, and A. Russo, "Using argumentation logic for firewall policy specification and analysis," in *Large Scale Management of Distributed Systems*. New York, NY, USA: Springer, 2006, pp. 185–196.

[14] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, G. Pavlou, and A. L. Lafuente, "Using linear temporal model checking for goal-oriented policy refinement frameworks," in *Proc. IEEE Workshop Policies Distrib. Syst. Netw.*, Stockholm, Sweden, Jun. 2005, pp. 181–190.

[15] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, and G. Pavlou, "A methodological approach toward the refinement problem in policy-based management systems," *IEEE Commun. Mag.*, vol. 44, no. 10, pp. 60–68, Oct. 2006.

[16] M. Charalambides *et al.*, "Policy conflict analysis for quality of service management," in *Proc. IEEE Int. Workshop Policies Distrib. Syst. Netw.*, Stockholm, Sweden, Jun. 2005, pp. 99–108.

[17] M. Charalambides *et al.*, "Dynamic policy analysis and conflict resolution for DiffServ quality of service management," in *Proc. IEEE/IFIP Netw. Oper. Manage. Symp. (NOMS)*, Vancouver, BC, Canada, Apr. 2006, pp. 294–304.

[18] D. Agrawal, K.-W. Lee, and J. Lobo, "Policy-based management of networked computing systems," *IEEE Commun. Mag.*, vol. 43, no. 10, pp. 69–75, Oct. 2005.

[19] R. Bhatia, J. Lobo, and M. Kohli, "Policy evaluation for network management," in *Proc. IEEE INFOCOM*, Mar. 2000, vol. 3, pp. 1107–1116.

[20] E. Bertino *et al.*, "Analysis of privacy and security policies," *IBM J. Res. Develop.*, vol. 53, no. 2, pp. 3:1–3:18, Mar. 2009.

[21] S. Lange *et al.*, "Heuristic approaches to the controller placement problem in large scale SDN networks," *IEEE Trans. Netw. Serv. Manage.*, vol. 12, no. 1, pp. 4–17, Mar. 2015.

[22] D. Tuncer, M. Charalambides, S. Clayman, and G. Pavlou, "Adaptive resource management and control in software defined networks," *IEEE Trans. Netw. Serv. Manage.*, vol. 12, no. 1, pp. 18–33, Mar. 2015.

[23] M. Wichtlhuber, R. Reinecke, and D. Hausheer, "An SDN-based CDN/ISP collaboration architecture for managing high-volume flows," *IEEE Trans. Netw. Serv. Manage.*, vol. 12, no. 1, pp. 48–60, Mar. 2015.

[24] N. Foster *et al.*, "Frenetic: A network programming language," in *Proc. ACM SIGPLAN Int. Conf. Funct. Program.*, Tokyo, Japan, Sep. 2011, pp. 279–291.

[25] M. Bari, S. Chowdhury, R. Ahmed, and R. Boutaba, "PolicyCop: An autonomic QoS policy enforcement framework for software defined networks," in *Proc. IEEE SDN Future Netw. Serv. (SDN4FNS)*, Trento, Italy, Nov. 2013, pp. 1–7.

[26] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi, "The middlebox manifesto: Enabling innovation in middlebox deployment," in *Proc. 10th ACM Workshop Hot Topics Netw.*, 2011, pp. 21:1–21:6.

[27] Big Switch Networks. (2016). *Floodlight* [Online]. Available: www.projectfloodlight.org/floodlight/

[28] A. Dixit, F. Hao, S. Mukherjee, T. V. Lakshman, and R. Kompella, "Towards an elastic distributed SDN controller," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 7–12, Aug. 2013.

[29] A. Tootoonchian and Y. Ganjali, "HyperFlow: A distributed control plane for OpenFlow," in *Proc. Internet Netw. Manage. Conf. Res. Enterprise Netw.*, San Jose, CA, USA, Apr. 2010, p. 3.

[30] N. Farrington *et al.*, "Helios: A hybrid electrical/optical switch architecture for modular data centers," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 339–350, Aug. 2010.

[31] F. Yonghong, B. Jun, W. Jianping, C. Ze, W. Ke, and L. Min, "A dormant multi-controller model for software defined networking," *Commun. China*, vol. 11, no. 3, pp. 45–55, Mar. 2014.

[32] R. Sherwood *et al.*, "Carving research slices out of your production networks with OpenFlow," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 1, pp. 129–130, Jan. 2010.

[33] V. Paxon *et al.* (2014). *The Bro Network Security Monitor* [Online]. Available: http://www.bro.org

[34] ntop. (1998). *nDPI: Open and Extensible LGPLv3 Deep Packet Inspection Library* [Online]. Available: http://www.ntop.org/products/ndpi/

[35] E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski, "The science DMZ: A network design pattern for data-intensive science," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal. (SC)*, Denver, CO, USA, Nov. 2013, pp. 1–10.

**Adrian Lara** received the B.S. and M.Sc. degrees in computer science from the University of Costa Rica, Costa Rica, and the Ph.D. degree in computer science from the University of Nebraska-Lincoln, Lincoln, NE, USA, in 2006, 2011, and 2015, respectively. He is currently an Invited Professor with the Computer Science and Informatics Department, University of Costa Rica. His research interests include software-defined networking, network security, and future Internet architectures.

**Byrav Ramamurthy** is currently a Professor and the Graduate Chair with the Department of Computer Science and Engineering, University of Nebraska-Lincoln (UNL), Lincoln, NE, USA. He has held visiting positions at the Indian Institute of Technology Madras (IITM), Chennai, India, and the AT&T Labs-Research, NJ, USA. He has authored over 160 peer-reviewed journal and conference publications. He has graduated 11 Ph.D. and 44 M.S. students under his research supervision. He is author of the book *Design of Optical WDM Networks—LAN, MAN and WAN Architectures* and a coauthor of the book *Secure Group Communications over Data Networks* (Springer, 2000 and 2004). He serves as the Editor-in-Chief for the *Photonic Network Communications* journal (Springer). He was the Chair of the IEEE ComSoc Optical Networking Technical Committee (ONTC) from 2009 to 2011. He has served as the TPC Co-Chair for the IEEE INFOCOM 2011 conference to be held in Shanghai, China. His research has been supported by the U.S. National Science Foundation (NSF), the U.S. Department of Energy (DOE), the U.S. Department of Agriculture (USDA), National Aeronautics and Space Administration (NASA), AT&T Corp., Agilent Tech., HP, OPNET Inc., and the University of Nebraska-Lincoln (UNL). He was the recipient of the College of Engineering Faculty Research Award for 2000 and the UNL CSE Department Student Choice Outstanding Teaching Award for Graduate-level Courses from 2002 to 2003 and 2006 to 2007.