A Security Policy Transition Framework for Software-Defined Networks

6

Jacob H. Cox Jr., Russell J. Clark, and Henry L. Owen III

6.1 Introduction

Software-defined networking (SDN) [1] allows for a single controller to orchestrate the actions of an entire network of switches.[1] Meanwhile, southbound interfaces, like OpenFlow [2], provide network operators with a single, vendor-agnostic interface for creating network applications, allowing for more fine-grained orchestration. In addition, these interfaces are further augmented by programming frameworks, like Pyretic [3] and Ryuretic [4] to provide greater abstraction and shield network operators from the complexities inherent in network application development. Furthermore, as organizations seek to protect their network's clients, data, and resources, greater numbers of researchers and network operators are looking toward SDN to quickly produce network security applications that address various attack vectors as they are discovered.

Unfortunately, many security solutions lack a framework for reversing or updating security measures (e.g., port blocking, traffic redirection, and other policy enforcements) once they are activated. Without a transition framework, once a client is flagged for a policy violation, revoking (or updating) the triggered security

[1] This chapter only considers a single controller, though distributed, logically centralized controllers can be used for more robust control options (e.g., fault tolerance, scalability, etc.).

J.H. Cox Jr. (✉) • H.L. Owen III
School of Electrical and Computer Engineering, Georgia Institute of Technology, North Ave NW, 30332, Atlanta, GA, USA
e-mail: jcox70@gatech.edu; henry.owen@ece.gatech.edu

R.J. Clark
College of Computing, Georgia Institute of Technology, North Ave NW, 30332, Atlanta, GA, USA
e-mail: russ.clark@gatech.edu

S.Y. Zhu et al. (eds.), *Guide to Security in SDN and NFV*, Computer Communications and Networks, DOI 10.1007/978-3-319-64653-4_6

measure is not possible without having the network operator manually update the controller with either a script or external command. In some cases, the network operator may even have to reset the controller. None of which are ideal. For in one case, network operators can become overwhelmed with a large number configuration requirements. Yet, in the other, resetting the controller can result in a loss of state for the network and deprive the network of orchestration while the controller reboots.

As Kim et al. [5,6] observe, network operators may already be responsible for as many as 18,000 network configuration changes in a given month. On traditional networks, these changes often include adding, modifying, or deleting entries in access control lists (ACLs). Similar change requirements may exist on SDNs where network operators utilize *whitelists* and *blacklists* as part of their security strategy. In both cases, each additional configuration introduces an opportunity to add an error to the network. This task is further compounded when we consider that network operators may already be maintaining ACLs that contain roughly 10,000 entries, requiring updates as much as 4,000 times per year [5]. Such requirements represent a burdensome and tedious challenge for network operators. Moreover, this burden can be even more cumbersome for many network operators who lack programming experience. Thus, forcing network operators to manually handle policy enforcements prolongs a traditional requirement that is already seen as tedious and error prone. Additionally, future and emerging networks and services are likely to present levels of complexity that are currently unforeseen and unmanageable by the traditional means. Hence, as argued by Tsagkaris et al. [7], the design and implementation of more sophisticated tools are required to simplify network management and control and also to minimize human interaction.

For the above reasons, this chapter describes a security policy transition framework [8] to automate the process of updating policy enforcements in SDNs, which can assist network operators and benefit their clients by automating security policy transitions. For instance, a transition framework can help network operators reduce their manual configuration requirements, allowing them to avoid additional network errors and to pursue more complex tasks. Second, clients receive automatic notification of their violation and instructions for regaining their network privileges. Third, it eliminates erroneous trouble tickets by informing both clients and administrators of the violation. Finally, depending on the violation and validation requirement, this framework reduces the total time required to reinstate a client's network privileges. Having triggered a policy enforcement, the client need only enter a *passkey* into a web interface (i.e., a captive portal) to regain network privileges. Additionally, this framework is easily adaptable to other protocols and cloud infrastructures.

The rest of this chapter is outlined as follows. We first discuss the motivation for a security policy transition framework in Sect. 6.2. In Sect. 6.3, we discuss related work that best correlates to security policy transitions. The components of this framework are explained in Sect. 6.4, and the Mininet-based, test environment for this framework is explained in Sect. 6.5. Use cases for this framework are then introduced in Sect. 6.6. Finally, further discussion and future opportunities for this framework are offered in Sect. 6.7 before concluding this chapter in Sect. 6.8.

6.2 Motivation for a Security Policy Transition Framework

The primary goal of the security policy transition framework [8] is to reduce the number of manual network configurations in order to reduce network errors and improve network operator efficiency. Hence, it automates system functions to alleviate human error and reduce network operator workloads. Likewise, automating the revocation (or updates) of policy enforcements, once triggered by security policies, can significantly reduce a network operator's involvement with ACLs. For example, when a flagged client is added to a *blacklist* that triggers a policy enforcement, a security policy transition framework, like the one presented in this chapter, provides the client with preconfigured options for regaining access to network services. In other cases, where automated options are not possible, a help desk – employing less-skilled attendants – can be used to provide validation services for the flagged client. When the client does meet validation requirements, either the automated system or the help desk can provide the client with a *passkey*.

For instance, patch compliance can potentially be completely automated, while infected computers that require operating system reinstalls could be handled by help desk personnel. In either case, the transition framework handles the revocation of policy enforcements once the client obtains and provides a unique *passkey*. Since this process avoids network operator involvement, operational expenses (OPEX) and customer wait times can be further improved. We now offer a more detailed discussion of the security policy transition framework as seen in Fig. 6.1.

In this framework, the network operator sets the security policies for the controller as shown in (1) of Fig. 6.1. Then, as shown in (2), when the controller detects a violation that triggers a policy enforcement, the SDN controller informs the Trusted Agent, which serves as the framework's automated system for client services and controller updates, via an in-band communication and then updates the OpenFlow switch's flow table through its southbound interface. The flow table

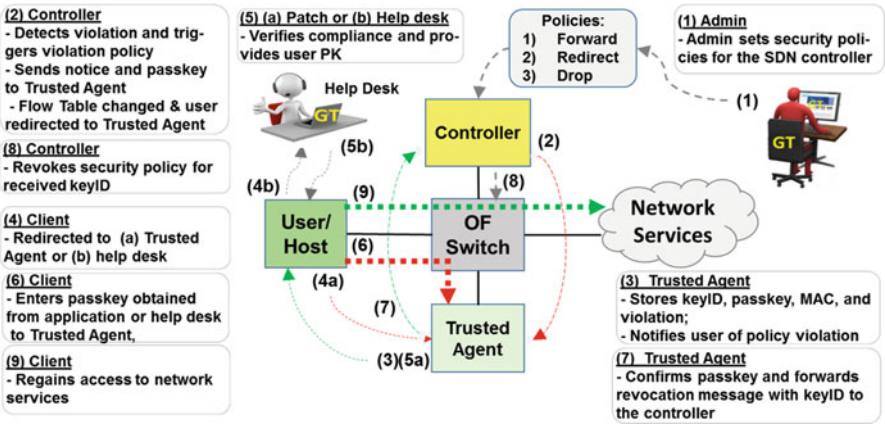


Fig. 6.1 Security policy transition framework [8]

modification results in the redirection of the flagged client's current and future traffic to a captive portal provided by the Trusted Agent. In (3), the Trusted Agent accepts and stores the client's *keyID*, *passkey*, MAC address, and violation. Thus, the Trusted Agent has knowledge of the client's violation when it presents the client with the web interface of its captive portal. This interface can then provide the client with a notice of the client's violation and instructions for regaining access to network services. For instance, if the client is flagged for patch compliance, then the Trusted Agent can make the patch available for download (4a or 5a). Once the software is installed and validated, the client can obtain a *passkey* from the validation authority (5a or 5b) and enter it into the web portal. Then, once the client enters the correct *passkey* (6), the Trusted Agent sends a revocation request to the controller (7), and the client's network privileges are reinstated as the policy enforcement is revoked (8). Hence, in this framework, the Trusted Agent services the flagged client and provides automated revocation requests to the SDN controller to remove policy enforcements.

A similar approach can be taken if the client inadvertently (or overtly) violates a network policy (e.g., packet spoofing, port scanning, rogue DHCP replies, etc.) requiring them to re-sign an acceptable use policy (AUP) after retraining. This requirement may also force the user to provide assurances that they will not repeat such actions in the future. The *passkey* can then be provided with a certificate of completion. Still, network operators may also wish to choose their level of involvement for certain policy violations. For instance, they may want to take action based on a first occurrence of a policy violation or a third, etc. In a corporation or government office, the network operator may even require the first line supervisor to login and acknowledge the incident before granting the certificate and *passkey*.

Another case occurs when clients are flagged for a computer virus that requires their system to be re-imaged. For such cases, a validation authority, such as a help desk can easily provide this service or verify that specific actions were completed. Once done, a *passkey* is provided to the client. For all of these examples, the client regains network privileges without involving the network operator. Doing so simplifies network management and control operations while also improving service efficiency for network operators and their clients. Not to mention, the incorporation of a Trusted Agent into an SDN architecture introduces additional opportunities for innovation, which we will later discuss.

6.3 Related Work

With networks growing in size, traffic volume, and requirements, the challenges of network management continue to increase. Likewise, enforcing organizational guidelines, protecting clients and data, and controlling network services all while preventing the organization's network from being intentionally or unintentionally sabotaged is an ever present network security concern. As a solution, systems often monitor client login attempts and refuse access to those who fail to authenticate or lack authorization. These solutions may even seek to monitor security policies

and automatically adjust network parameters to ensure compliance. Accordingly, various methods for controlling network access and enforcing policies exist in traditional networks and SDNs.

Traditional security management methods often rely on access control lists (ACLs), client IDs and passwords, and terminal access controller access control to enforce prearranged policies on system networks [9]. However, these policies are often reconfigured by network operators each time a security violation occurs or when a client (who triggered the security measure) regains approval to be reinstated. Additionally, protocols like 802.1X [10] can shut down ports if they detect unapproved devices connected to them, but removing these policy enforcements to reactivate these ports is often left to the network operator to resolve via a trouble ticket. Hence, these solutions place considerable configuration burden on network operators, add additional software and hardware costs, and lack an automated security policy transition framework for reinstating clients. They also add to the ambiguity of trouble tickets created by clients who do not yet know the reason for their loss of network services. Likewise, due to the ambiguity of these trouble tickets, network operators can spend unnecessary time trying to troubleshoot and determine the cause of the client's loss of services.

Consider, for example, commercial network access control (NAC) solutions, like ForeScout [11] and Cisco NAC [12]. They mostly seek to ensure that access policies for the network (and its resources) are enforced on a per person basis. These systems may even move a device to a reconfigured guest network (e.g., walled garden), so it can receive system updates (e.g., antivirus software and patches). Previously, a major restriction of NAC solutions was that they were typically limited to devices that had specific operating systems installed and/or were capable of installing a NAC agent. For instance, IEEE 802.1X requires that end devices (i.e., a client) have a supplicant installed, which is used to communicate with the central authentication server. However, modern NAC solutions have grown to offer new features. For instance, the Aruba ClearPass Policy Management Platform, the Bradford Networks' Network Sentry/NAC, the Cisco Identity Services Engine (ISE), and the Pulse Secure Policy Secure NAC solutions all offer agentless support, extended policy capabilities, onboarding support, extended guest management, extended profiles support, extended endpoint compliance, optional advanced threat protection and mitigation, expanded monitoring and reporting, and extended system integration and interoperability [13].

Still, NAC solutions can also be complicated to set up, often becoming a long-term project requiring phased deployments and more suitable to robust/mature infrastructures. Since an authentication server is required, deployment also includes more power, space, and licenses, while support for random equipment, like printers, can also be problematic. Examples also exist for defeating NAC. For instance, a security researcher demonstrated that by attaching a hub to a port, they could simply wait for the authorized client to authenticate with the NAC server and then piggy back their communications over the network by spoofing the authorized client [14]. Other researchers have implemented bridging techniques using an authorized port and an active client to achieve better results [14]. Still, when port violations are

detected, the port is generally blocked and manually cleared. So, despite a plethora of available protocols and software, network connectivity remains a manual process and a challenge to network management. Hence, network operators must still work with clients to address the flag's cause and to reinstate their privileges, which the security policy transition framework discussed in this chapter attempts to address. Additionally, NAC deployments also require proprietary switches that support NAC features, like 802.1X. However, such features are absent in SDN/OpenFlow switches, hence, SDN-based approaches are required.

SDN solutions have also developed in recent years to assist network operators with flexible network programmability for security management. For instance, PolicyCop [15] helps network operators detect policy violations. Action requests are either forwarded to an autonomic policy adaptation module or the network operator depending on the policy violation. As a result, the network operator is an essential part of this architecture having to provide manual configurations. Moreover, PolicyCop [15] does not directly consider the revocation of policy enforcements, yet it can be assumed that the network operator must provide manual interventions for those as well. Ethane [16], a precursor to SDN, provides a centralized network architecture with identity-based access control that allocates IP addresses as IP-MAC-Port associations. In this environment, clients authenticate via a webform, and their packets are then reactively evaluated by the controller for policy compliance. By doing so, Ethane allows network operators to define a single, fine-grain policy and apply it network wide. And, as a result, network clients can be held accountable for their traffic, yet Ethane also requires network operator intervention for flagged clients [16]. In contrast, FlowNAC [17] drops web-based authentication in favor of a modified 802.1X framework supporting extensible authentication protocol over LAN (EAPoL-in-EAPoL) encapsulation. In this framework, client traffic flows are associated with a target service. As a result, this system handles client access based on predefined authentication and authorization policies, but it does not consider policy violations where the client becomes flagged. Additionally, supplicant (client) software must be utilized to enable FlowNAC's features.

Kinetic, formerly known as Resonance, also represents a transition framework that offers an OpenFlow-based dynamic access control system [6]. It uses network alerts to support continuous monitoring and per interface policy control to automate dynamic security policies. Additionally, Kinetic verifies that prescribed changes align with operator requirements by employing a finite-state machine (FSM) having states that correspond to distinct forwarding behavior [6]. Transitions within the FSM are controlled by Kinetic's Event Handler, which monitors for events and triggers policy updates. However, Kinetic follows a similar vein to the solutions previously discussed in that it too requires network operators to supply the events that trigger its policy changes. In addition, since Kinetic is built atop the Pyretic [3] programming language and POX [18], which is limited to OpenFlow 1.0 [2] and only 12 packet header match fields, its packet inspection capabilities are substantially constrained. Moreover, Kinetic also lacks an automated framework for transitioning between security measures.

While all these solutions represent great strides toward better and more intuitive interfaces that simplify the application development process, they still do not provide a policy transition framework for automating the revocation or modification of a policy enforcement. Resultantly, network operators are still heavily involved in multiple, unnecessary configurations on a daily basis. Hence, we next introduce a security policy transition framework that can be implemented in SDN and NFV environments. By including this framework, network operators can improve the time associated with reinstating network clients while reducing network operator workloads and erroneous trouble tickets. Much like Kinetic [6], this framework implements an event listener (i.e., Event Handler); however, it works with a trusted entity (i.e., Trusted Agent) to determine when an activated security measure or policy enforcement should be changed. The SDN controller then assumes responsibility for implementing and enforcing security policies, while relying on the Trusted Agent to provide notification for when policy enforcements should be revoked.

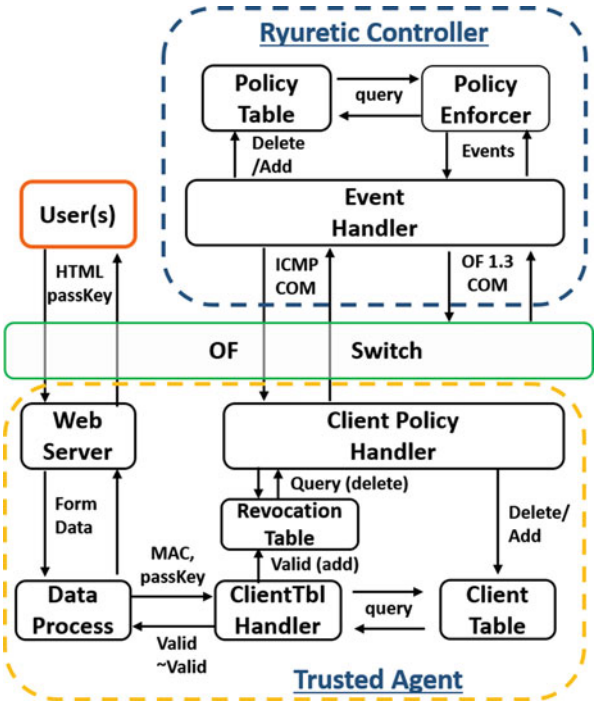
6.4 The Framework

The security policy transition framework introduced in this chapter uses Ryuretic [4] for its SDN controller applications. Ryuretic [4] is a domain-specific language offering a modular framework for application development atop the Ryu [19] controller. It also provides an intuitively simple format for network operators to select header fields within a packet (pkt[*]) and then specify what operation (*ops[*]*) occurs when a match (*fields[*]*) is found. This platform also allows programmers to craft their own packets, which is utilized to establish a communication channel between the SDN (Ryuretic) controller and its Trusted Agent using ICMP packets. This communication channel is then used to submit policy enforcement updates or revocation requests. This is discussed in greater detail in Sect. 6.4.3. Additionally, this communication allows both the Ryuretic controller and the Trusted Agent to maintain corresponding state tables as we will also discuss in Sect. 6.4.3. These and the other components comprising the controller and Trusted Agent modules (shown in Fig. 6.2) are now discussed.

6.4.1 Controller

As shown in Fig. 6.2, the Ryuretic controller for this framework is an SDN controller comprised of an Event Handler, a Policy Enforcer, and a Policy Table. These components are implemented in Ryuretic [4], which serves as an abstraction layer residing atop the Ryu [19] controller and supporting OpenFlow 1.3 [2]. With Ryuretic, network operators can choose to forward, drop, mirror, redirect, modify, or craft packets based on match parameters that they define via objects.

As shown in Fig. 6.3, when a packet-in event occurs in the Ryu [19] controller, the Ryuretic coupler generates a packet object (*pkt*) that is forwarded to the Ryuretic



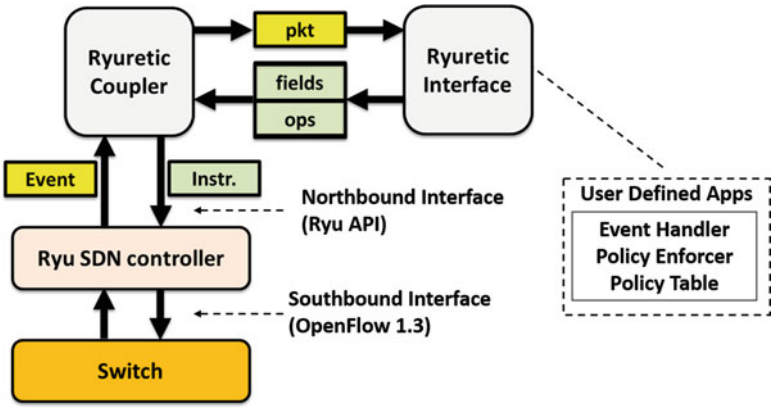


Fig. 6.2 Security policy transition framework components [8]

Fig. 6.3 Ryuretic controller

interface. This is where the network operator policies are specified. Based on these policies, the interface returns two objects (i.e., *fields* and *ops*) specifying which match-action rules to pass to the switch. These objects are then interpreted by the Ryuretic coupler and forwarded as instructions to the Ryu controller, which installs the rules to the switch. In Fig. 6.3, the Event Handler, the Policy Enforcer, and the Policy Table all exist as user-defined applications in the Ryuretic interface.

6.4.1.1 Event Handler

The Event Handler serves as the primary interface for the controller, responding to network events from the switch, security events from the Policy Enforcer, and security policy transitions from the Trusted Agent. It also handles insert and delete messages for the controller's Policy Table to maintain state for each connected client. When a packet arrives from the switch, the Event Handler passes the packet to its Policy Enforcer. If a violation is detected (e.g., a spoofed ARP packet), then the Event Handler will receive notification of the violation along with a generated *keyID* and *passkey*. It then records this information, including the client's MAC address and input port number, into the SDN controller's Policy Table, which serves as an access control list for future packet decisions. It then notifies the Trusted Agent via the in-band communication channel (shown previously in Fig. 6.2 and discussed in Sect. 6.4.3) and includes the client's table information discussed above. Finally, it provides a match-action flow rule to the OpenFlow switch to direct future packets from the flagged client to the Trusted Agent. Should the Event Handler receive a policy enforcement revocation request from the Trusted Agent, then the Event Handler reinstates the client's privileges by removing the associated client entry from the controller's Policy Table.

6.4.1.2 Policy Enforcer

The Policy Enforcer handles events passed to it from the Event Handler. It first confirms that arriving packets are not already flagged in the Policy Table. If not, the Policy Enforcer next applies selected security policies against the arrived packet. If the packet passes specified checks, then it is passed to the Event Handler for normal forwarding. Otherwise, the Policy Enforcer returns *fields* and *ops* hash tables[2] to the Event Handler – resulting in the client's traffic being redirected to the network's Trusted Agent. If the client is flagged, the Policy Enforcer also generates a randomized *passkey* and a unique *keyID*, which is passed back to the Event Handler with the client's other unique flow information (i.e., input port, MAC, and violation).

6.4.1.3 Policy Table

The Policy Table simply stores the identification and flag state information for each client. As shown in Fig. 6.4, the Policy Table stores *keyID* (primary identification key), *passkey* (for client authentication), MAC address, input port, and violation code for flagged clients (of which, all but the input port are forwarded to the Trusted Agent).

6.4.2 Trusted Agent

The Trusted Agent serves as an intermediary between the client and the network operator. For instance, the Trusted Agent is able to send revocation messages to the controller and reinstate the client's privileges in lieu of the network operator once

[2]Hash tables (Python dictionaries) are Ryuretic's method for directing network operations.

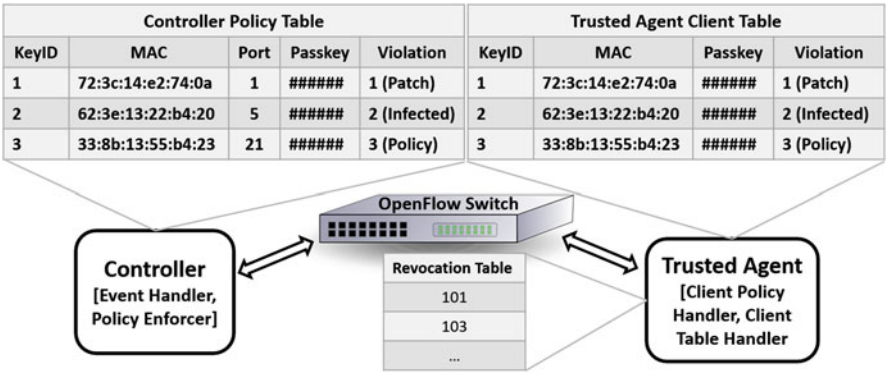


Fig. 6.4 Controller – Trusted Agent communication

the *passkey* is provided. It can also provide clients with instructions for regaining network access. Its components (See Fig. 6.2) are next discussed.

6.4.2.1 Client Policy Handler

The Client Policy Handler establishes a communication link with the controller to receive policy activation notices and submit revocation requests. When the Trusted Agent is first notified of a policy enforcement activation, it records the provided *keyID*, *passkey*, MAC, and violation associations in its Client Table as indicated in Figs. 6.2 and 6.4. The Client Policy Handler also periodically queries the Revocation Table for *keyIDs* belonging to clients who have submitted a *passkey* and are awaiting the reinitialization of client privileges. In this framework, the query is arbitrarily performed every 30 s. Ideally, this query can be performed more frequently.

6.4.2.2 Client and Revocation Tables

The Client Table allows the Trusted Agent to maintain state for flagged clients. As shown in Fig. 6.4, this table maintains the client's *keyID*, *passkey*, MAC, and violation. It is also queried by the Client Table Handler to confirm client MAC and *passkey* pairs. Furthermore, the Client Table provides violation information to the Handler, so the Trusted Agent renders appropriate instructions to the client.

The Revocation Table (also shown in Fig. 6.4) allows the Trusted Agent to queue *keyIDs* for clients awaiting privilege reinstatement. The Client Policy Handler then routinely queries the table and sends *keyIDs* in revocation messages to the controller.

6.4.2.3 Client Table Handler

The Client Table Handler queries the Client Table to verify a client's *passkey* and MAC address. If successful, the Handler loads the client's *keyID* to the Revocation Table for delivery to the controller. As a security measure, the Client Table Handler can only query the Client Table and write to the Revocation Table.

6.4.2.4 Data Processor

The Data Processor is a Common Gateway Interface (CGI) module that provides server-side scripting for the Trusted Agent's web server. It receives as input the MAC and *passkey* from form data and provides them to the Client Table Handler. In turn, the Handler returns feedback information to the client's web interface via HTML.

6.4.2.5 Web Server

While any number of web servers could be used for this component, the lighttpd [20] web server is used due to its small memory footprint and support for CGI scripts. It serves as the client's primary interface while resolving flags. It also captures the client's MAC address via a PHP script when the client enters their passkey. The passkey and MAC address are then forwarded to the Data Processor for passkey validation. Note that the web server and the client must be on the same subnet for the PHP script to capture the client's MAC address. Otherwise, it captures the MAC address of the previous hop (e.g., a router's MAC address).

6.4.3 Communication Channel

The SDN controller and Trusted Agent communicate rule insertions, updates, and revocations via crafted ICMP packets having instructions in their modified data field. Both ICMP request and reply packets (see Fig. 6.5) are used. Normally, the data field of an ICMP packet header contains information for determining round trip times (e.g., time stamps), etc. However, the transition framework's communication channel repurposes this field. Additionally, identification (ID) and sequence (SEQ) fields are set to zero. Yet, while the Trusted Agent can receive ICMP packets having complete payloads, the SDN controller only receives the ICMP packet's header information and the up to 86 bytes of the packet's data – based on observations of the Open vSwitch and Ryu controller implementation used in this work. Hence, communications from the Trusted Agent to the Ryu controller are limited to 86 bytes, as shown in Fig. 6.5b.

Field Size	Byte 0	Byte 1	Byte 2	Byte 3
IP Header (20 bytes)	Version/HL	Type	Length	
	ID		Flags/Offset	
	TTL	Protocol	Checksum	
	Source IP			
	Destination IP			
ICMP Header (8 bytes)	Type = 0, 8	Code = 0	Checksum	
	Header Data (ID, SEQ)			
ICMP Payload (optional)	Payload Data (Typically 48 bytes--can be much greater)			

a.

Field Size	Byte 0	Byte 1	Byte 2	Byte 3
IP Header (20 bytes)	Version/HL	Type	Length	
	ID		Flags/Offset	
	TTL	Protocol	Checksum	
	Source IP			
	Destination IP			
ICMP Header (8 bytes)	Type = 0, 8	Code = 0	Checksum	
	Header Data (ID, SEQ)--Set to Zero			
ICMP Payload (optional)	Payload Data TA to Controller: upto 86 bytes Controller to TA: no limit*			

b.

Fig. 6.5 ICMP Packet modification. *Ideally, within network's max transmission unit (MTU) size. (**a**) Typical contents of ICMP packet. (**b**) Modified ICMP packet

Table 6.1 Abbreviations used for controller communication [8]

Abbr.	Meaning	Summary
i	Initialize	Establish Trusted Agent parameters
a	Acknowledge	Send table entry receipt for keyID
d	Delete	Request policy deletion for specified keyID

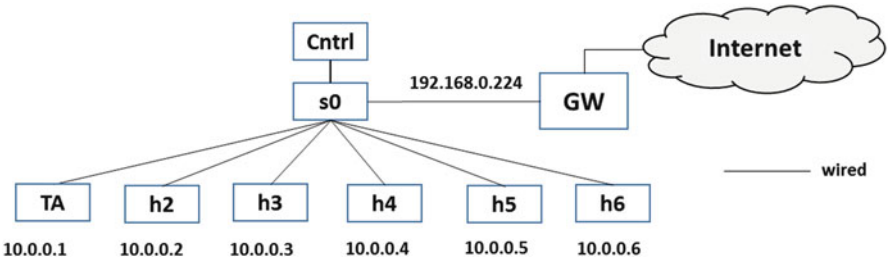


Fig. 6.6 Mininet test environment

To accommodate its data limitation and allow for future features, the Trusted Agent constrains it responses to *action*, *keyID* strings, consuming up to 8 bytes. The *action* (see Table 6.1) value is a single letter abbreviation. It identifies the message type (i.e., initialize, acknowledge, or delete). Messages not requiring a *keyID* (e.g., initialize) include a zero after the *action* value. For example, the Trusted Agent's initialization message to the Ryuretic controller appears as "i,0" in the data field of the ICMP's packet header. A revocation appears as "d,102," while an acknowledgment from the controller appears as "a,d,102."

Messages from the controller, however, have more flexibility. For instance, rule insertion methods destined for the Trusted Agent's Client Table will include MAC, *passkey*, violation, and *keyID* values in a comma-separated string. This format is recognized by the Trusted Agent and handled accordingly by its Client Policy Handler. It is through this communication channel and format that the Ryuretic controller and the Trusted Agent are able to maintain corresponding tables (the Policy Table for the Ryuretic controller and the Client Table for the Trusted Agent), which are shown in Fig. 6.4. Moreover, while limited, this solution is easily adaptable to other SDN controllers using existing protocols, making it controller neutral. In other words, any SDN controller capable of crafting ICMP packets can implement this communication channel. Still, while adequate for this implementation, the per packet data limit imposed by this communication channel serves as a challenge to encrypted and cross-domain communications, which remain open research areas in this work.

6.5 Test Environment

The test environment (shown in Fig. 6.6) is implemented in Mininet [21], a network emulator for creating virtual clients, switches, controllers, and links. All clients, including the Trusted Agent, are virtual machines with Ubuntu 14.04 operating

systems. The switch is OpenFlow 1.3 [2] capable, and Ryuretic [4] applications run atop a Ryu controller to provide network control. The testbed also provides Internet access via a virtual network address translator (NAT) or gateway router (GW). Until a client is flagged, it can ping other clients and access web services via the GW. Web services are tested using curl and *wget* commands and the Firefox Internet browser. However, the flows of a flagged client are redirected to the Trusted Agent's web server until the client provides the appropriate *passkey*. Once that is provided, the client regains network privileges within 30 s of making the entry.

6.6 Example Use Cases

In this security policy transition framework, the Ryuretic controller enforces the policies defined by the network operator. In this section, we attempt to highlight just a couple of policy violations (i.e., ARP spoofing and unauthorized NAT) that a network operator might target. We implement these attacks in the test environment discussed in Sect. 6.5. In this environment, we assume a client is behaving maliciously, so we will discuss some of the key code listings contributing to the framework's detection and notification methods. This section will also allow us to explore Ryuretic's packet crafting feature.

6.6.1 Spoofed ARP Packets

Spoofed ARP packets can poison neighboring client ARP tables and serve as a springboard for more dangerous attacks (e.g., packet dropping (black hole) and man-in-the-middle (MitM) attacks). However, this framework allows the network operator to set an ARP spoofing detection protocol in Ryuretic [4] to trigger appropriate security measures. When Ryuretic detects an arriving ARP packet, it is forwarded to the appropriate handler (see Listing 6.1). The Policy Enforcer checks the Policy Table to determine if the incoming packet should be dropped, redirected, or forwarded.

Listing 6.1 Ryuretic ARP Event Handler

```
def handle_arp(self,pkt):
  #Check Policy Table for MAC and input port
  pkt_status = self.check_net_tbl(pkt['srcmac'],
                                  pkt['inport'])
  if pkt_status is 'flagged':
    #If flagged, redirect flow
    fields,ops = self.Redirect_Flow(pkt)
  else:
    #If not flagged, test for spoof
    spoofed = self.detectSpoof(pkt)
    if spoofed != None:
      #Notify Trusted Agent of Policy Transition
      self.notify_TA(pkt)
```

```
14        fields, ops = self.drop_ARP(pkt)
15      else:
16        # Handle ARP packet
17        fields, ops = self.respond_to_arp(pkt)
18    self.install_field_ops(pkt,fields,ops)
```

If the source MAC or input port is not flagged, then the packet is evaluated by the detectSpoof() method (see Listing 6.2). If the packet is flagged as spoofed, then the Policy Enforcer notifies the Event Handler, which forwards a notification message to the Trusted Agent with the client's MAC, *passkey*, violation, and *keyID*. It then sets Ryuretic's *fields* and *ops* objects to drop future ARP replies from the client. In turn, the Trusted Agent adds the flagged client to its Client Table.

Otherwise, the packet is forwarded to the *respond_to_arp()* method for normal forwarding. As seen in Listing 6.2, the *detectSpoof()* method builds a network view to associate each client's MAC and IP address to a switch port. If a packet arrives after the network view is built with an incorrect MAC or IP address, then it is flagged as spoofed, and its future traffic is sent to the Trusted Agent, which renders a web page to explain the violation and offer instructions for regaining access to the network (e.g., submit an acceptable use policy, AUP). Currently, the security policy transition framework discussed in this chapter relies on the help desk to serve as the validating authority; however, a future implementation could make the AUP available and provide compliance validation. Once the client obtains and submits the *passkey*, their network services are reinstated – generally within 30 s of entering the *passkey*. The complete implementation is available at [22].

Listing 6.2 Ryuretic ARP poison detection method

```
1  def detectSpoof(self,pkt):
2    policyFlag = None
3    # Has port been mapped?
4    if self.netView.has_key(pkt['inport']):
5      # Does srcmac match recorded value?
6      if pkt['srcmac']!= self.netView[pkt['inport']]['srcmac']:
7        policyFlag = 'ARP'
8        # Does srcip match recorded value?
9      if pkt['srcip'] != self.netView[pkt['inport']]['srcip']:
10       policyFlag = 'ARP'
11   else:
12     # Map the port
13     self.netView[pkt['inport']] = {'srcmac': pkt['srcmac'],
14                                    'srcip': pkt['srcip']}
15     # Set policy enforcement
16   if policyFlag == 'ARP':
17     self.net_MacTbl[pkt['srcmac']] = {'stat':'flagged',
18                                       'port':pkt['inport']}
19     self.net_PortTbl[pkt['inport']] = {'stat': 'flagged'}
20   return policyFlag
```

6.6.2 Network Address Translation (NAT)

Unauthorized network address translation (NAT) devices can also compromise local networks by giving unauthorized users access to network services. One way to detect these devices is to monitor IP packet headers for a decremented time-to-live (TTL) [23]. In this example, host 2 (h2) from Fig. 6.6 attempts to run NAT services. To detect this policy violation, the Policy Enforcer utilizes a *nat_detect* module, as defined in Listing 6.3, to inspect the TTL field of each IP packet passing through the switch (s0). Consequently, inspecting every packet can impact the controller's performance. A better solution would limit packet inspections to just a few packets per flow before installing rules for future flows. Instead, this listing shows a simple example that is implemented with just few lines of code.

We first observe that most network devices have TTL values of 64 or 128. If hosts are directly connected to the switch, then it should detect one of these values. However, if these devices are connected behind a rogue NAT device with TTL decrement enabled, then the NAT will be detected, and the value returned will signal the Ryuretic controller to flag the client for a rogue "NAT" violation. The controller then updates its Policy Table and notifies the Trusted Agent before updating the switch's flow table.

Listing 6.3 Ryuretic NAT detection method creation

```
1  def TTL_Check(self, pkt):
2    policyFlag = None
3    if pkt['ttl'] not in [64, 128]:
4      policyFlag = 'NAT'
5    return policyFlag
```

6.6.3 ICMP Packet Notifications

Packet creation is a feature developed for Ryuretic allowing programmers to craft packets via its *fields* and *ops* objects. Listing 6.4 demonstrates how messages are crafted using Ryuretic within the security policy transition framework. The controller generates a packet containing the *srcmac*, *passkey*, *violation*, and *keyID* (see lines 15–16). Additionally, while not shown, the MAC and IP addresses of the controller and Trusted Agent are defined elsewhere in the code. Other ICMP message examples can be found in the Ryuretic interface file located at [22] and [24].

Listing 6.4 Ryuretic packet crafting

```
1  def update_TA(self,pkt, keyID):
2    table = self.policyTbl[keyID]
3    agent, cntrl = self.t_agent, self.cntrl
4    fields, ops = {},{}
5    fields['keys'] = ['inport', 'srcip']
6    fields.update({'dstip':agent['ip'],
7                   'srcip':cntrl['ip']})
8    fields.update({'dstmac':agent['mac'],
```

```
9                    'srcmac':cntrl['mac']})
10    fields.update({'dp':agent['dp'], 'msg':agent['msg'],
11                   'inport':agent['port'],
12                   'ofproto':agent['ofproto'],
13                   'ptype':'icmp','ethtype':0x800,
14                   'proto':1, 'id':0})
15    fields['com'] = table['srcmac']+','+str(table['passkey'])+
16                      ','+table['violation']+','+str(keyID)
17     ops = {'hard_t':None, 'idle_t':None, 'priority':0, \
18                 'op':'craft', 'newport':agent['port']}
19     self.install_field_ops(pkt, fields, ops)
```

6.6.4 Traffic Redirect

Once a client is flagged, the Ryuretic controller must next divert the client's traffic to the Trusted Agent. An example using Ryuretic [4] is provided in Listing 6.5, and additional code examples can be found at [22] and [24]. In this snippet, an IP table is tied to the Ryuretic controller's Policy Table. Notice that line 2 first sets the *fields* and *ops* objects to set match-action rules for the traffic flow. This snippet also shows how additional fields can be updated in lines 4–28. Here we see that if a client is flagged for "deny," then the traffic flow's destination information is saved to a TCP table. Its packet header data is then modified, and the packet is forwarded to the Trusted Agent. These actions occur in lines 4–17. Otherwise, if the packet originates from the Trusted Agent, the Ryuretic controller performs a reverse table lookup to associate the client with its packet, modifies the packet's source fields to reflect its original destination, and forwards the packet to the flagged client, as shown in lines 18–28 of Listing 6.5.

Listing 6.5 Ryuretic traffic redirect

```
1  def redirect_TCP(self,pkt):
2    fields,ops = self.default_Field_Ops(pkt)
3    #IP address (src & dst) maintained in IP forwarding table
4    if self.ipTbl.has_key(pkt['srcip']):
5      if self.ipTbl[pkt['srcip']] in ['deny']:
6        key = (pkt['srcip'],pkt['srcport'])
7        # Copy srcmac and dstmac to modify packet header
8        self.tcp_tbl[key] = {'dstip':pkt['dstip'],
9                             'dstmac':pkt['dstmac'],
10                            'dstport':pkt['dstport']}
11       fields.update({'srcmac':pkt['srcmac'],
12                      'srcip':pkt['srcip']})
13       fields.update({'dstmac':self.t_agent['mac'],
14                      'dstip':self.t_agent['ip']})
15       # Modify and redirect packet to TA or flagged client
16       ops = {'hard_t':None, 'idle_t':None, 'priority':100,\
17             'op':'mod', 'newport':self.t_agent['port']}
18   elif self.ipTbl.has_key(pkt['dstip']):
19     if self.ipTbl[pkt['dstip']] in ['deny']:
```

```
        key = (pkt['dstip'],pkt['dstport'])
        # Copy srcmac and dstmac to modify packet header
        fields.update({'srcmac':self.tcp_tbl[key]['dstmac'],
                       'srcip':self.tcp_tbl[key]['dstip']})
        fields.update({'dstmac':pkt['dstmac'],
                       'dstip':pkt['dstip']})
        # Modify and redirect packet to TA or flagged client
        ops = {'hard_t':None, 'idle_t':None, 'priority':100,\
               'op':'mod', 'newport':None}
    return fields, ops
```

6.7 Discussion and Future Opportunities

The security policy transition framework presented in this chapter presents many opportunities for future improvements. For instance, while this framework's limited communication channel is suitable for multiple controllers and allows for automated revocations using existing protocols, more robust communication channels are still needed. These channels could allow an east-westbound interface to better enable policy enforcement and validation across domains (which is still an open research topic) or provide for more versatile communication between the SDN controller and the Trusted Agent. As presented, the framework covered in this chapter relies on existing unmodified network protocols to implement a limited communication channel for the invocation and revocation of policy enforcements.

The transition framework discussed in this chapter is also easily adaptable to a password-based authentication framework for clients seeking to join the network. In which case, a network view can be built as clients authenticate to the network. Additionally, within the context of this framework, there is potential to provide a variety of actions for clients to take once they are redirected to the Trusted Agent. For instance, the captive portal can include patches, courses, administrative documents, initial warnings, etc. Additionally, the Trusted Agent's responsibilities could expand to include other security features. For instance, a modified Trusted Agent could provide active testing for security threats where passive monitoring is either not sufficient or too intensive for the controller to handle. With minor updates to this framework's communication channel, the SDN controller could notify the Trusted Agent of testing requirements for clients, and the Trusted Agent could instruct the SDN controller to transition the security state of a client under "test."

Using a Trusted Agent in this framework also reduces burden on network operators by reducing the manual configurations needed to remove policy enforcements, while also providing clients with immediate feedback on the status of their network privileges. With regard to network access control (NAC) systems, it is not too far a stretch to have the Trusted Agent interact with NAC authorization servers to implement comparable features as already exist today. However, this remains a focus for future work. Furthermore, since this framework's functions and components are implemented using NFV, it is also viable for cloud and virtual network environments, which serves as another future research direction.

Of course, while SDN is capable of implementing numerous security features, we still do not suggest that all security features should be handled by the SDN controller. In fact, the introduction of the Trusted Agent in this chapter further provides for the incorporation of additional security features where secondary devices serve to provide more layers to a defense-in-depth security strategy. Likewise, this framework does not replace the need for application-level monitoring. Such services are still needed to identify a client's software version, provide patch compliance, detect malware, or even apply an application-layer firewall. However, the Trusted Agent could be configured to coordinate security efforts between application-layer products and the controller.

Furthermore, as the Trusted Agent serves as a key intermediary between the client and the network operator, this framework could also benefit from the inclusion of machine learning algorithms that better cater to the client's needs while providing more automated services. Doing so could offer a more human experience along with a greater range of services for client validation. Additionally, the Trusted Agent's functions could be expanded to coordinate with existing middleboxes, manage IoT devices, and/or provide system redundancy. For example, should a primary server (e.g., DHCP, DNS, etc.) fail, the Trusted Agent could serve as a backup until the primary server is again operational. IoT device security along the network's edge is also an open research topic for which this framework might be expanded to include.

Network operators using this framework must also consider that more clients than just subscribers will operate on their networks (e.g., M2M communication or Web service interaction). If not handled appropriately, the redirection of flagged clients to a self-service interface, as proposed in this work, could cause IoT devices or user agents to assume the network has failed. Such incidents could result in the generation of erroneous trouble tickets that once again task network operators to troubleshoot connectivity issues instead of policy violations. Ideally, the network operator will *whitelist* or set aside specific ports for such devices to provide notifications if the device becomes flagged. For such cases, the Trusted Agent could also run a mail server to notify the help desk when a nonuser device is affected. Additional applications could further augment the Trusted Agent to better handle such devices as well. For instance, should IoT devices deviate from expected traffic patterns, then their subsequent flows can be forwarded to the Trusted Agent for deep packet inspection or other analysis, isolated from the network, or recorded for future analysis. SDN is uniquely situated to provide edge-based analysis of IoT device traffic, and future work will explore how a security policy transition framework as discussed in this chapter can be applied to IoT management and security.

Of course, introducing the Trusted Agent into this framework also introduces yet another attack vector. If the Trusted Agent can be compromised, then its communications to the controller for policy revocations can also be affected. However, in this system, we assume the Trusted Agent to be at least as physically secure as the SDN controller. Likewise, we utilize the controller to monitor network access to the Trusted Agent and block unauthorized traffic. As a result, only clients who have already been flagged are able to interact with the Trusted Agent via its web server, which limits packets to HTTP(S) (i.e., port 80 and 443) and DNS (i.e., port 53)

protocols. Moreover, further hardening of the transition framework should add additional network security. For instance, randomizing keyIDs, encrypting the passkey while in transit, and further securing communications between the client and Trusted Agent are all prudent measures. Another consideration is validation of this framework with standards specified by the Trusted Computing Group (TCG) [25] for Trusted Network Communications (TNC) and Security Content Automation Protocol (SCAP). However, additional security analysis, hardening, and standards compliance of this security policy transition framework, including its Trusted Agent, are left to future work.

6.8 Conclusion

With OpenFlow providing a vendor-agnostic platform for SDNs and enabling the orchestration of numerous switches, programmers are better able to implement novel network applications for security and traffic engineering. Yet, network operators still need additional measures for automating daily processes and configurations to fully utilize SDNs in physical and virtual environments. As a result, this chapter introduces the concept of a security policy transition framework, which provides automation by flagging clients, redirecting their network flows to a Trusted Agent, and revoking activated security measures once the client validates they have met specific requirements by entering a *passkey*.

In this framework, a *passkey* is obtained from a validating authority (e.g., a help desk or the Trusted Agent) and used by the client to prove that specified requirements have been met in order to rejoin the network. As a result of these features, frameworks such as the one discussed in this chapter can eliminate many daily network configuration requirements that must currently be manually performed by network operators. Other benefits include reduction of both erroneous trouble tickets and client wait times for regaining network access. However, these wait times may still vary based on the violation and system validation procedures.

Finally, this chapter introduces several potential directions that this framework may take in future iterations. For instance, machine learning could be leveraged to enhance user experience when interacting with the Trusted Agent. Additionally, the Trusted Agent may be further developed to implement active detection measures for security applications. Other future work includes security analysis and hardening of the framework itself, improving upon it communication channel with an east-westbound interface, and implementing security and management applications for IoT.

Questions

1. Overall, how will this framework or one like it aid network operators?
2. Regarding security, what additional challenges does introducing a Trusted Agent to an SDN create?

3. What features should be added to the Trusted Agent and the communication channel used in this work to support functions that go beyond policy enforcement revocation, for instance, active testing of clients?
4. Considering the ICMP-based communication channel utilized in this work, what are its primary limitations, and how might they be improved?
5. Does the communication channel used in this work represent an in-band or out-of-band form of communication? Explain your answer.
6. Is the communication channel developed in this work only applicable to the Ryuretic programming framework, or could it also be used with other controllers (e.g., POX, OpenDaylight, Floodlight, etc.)?
7. What ways might a client obtain a passkey to regain their network privileges?
8. How might network operators modify the security policy transition framework presented in this chapter to accommodate Internet of things (IoT) devices and other clients having neither access to a web browser nor an ability to respond to the Trusted Agent's web server?
9. Concerning Ryuretic, what SDN controller does it augment, and what are the objects it uses for monitoring, matching, and rule setting on packets?

References

1. McKeown N (2009) Software-defined networking. INFOCOM Keynote Talk 17(2):30–32
2. McKeown N, Anderson T, Balakrishnan H, Parulkar G, Peterson L, Rexford J, Turner J (2008) OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Comput Commun Rev 38(2):69–74
3. Reich J, Monsanto C, Foster N, Rexford J, Walker D (2013) Modular SDN programming with pyretic. Technical report of USENIX
4. Cox JH Jr, Donovan S, Clark R, Owen H (2016) Ryuretic: a modular framework for RYU. In: IEEE MILCOM2016
5. Kim H, Benson T, Akella A, Feamster N (2011) The evolution of network configuration: a tale of two campuses. In: Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference, Nov 2011. ACM, pp 499–514
6. Kim H, Reich J, Gupta A, Shahbaz M, Feamster N, Clark R (2015) Kinetic: verifiable dynamic network control. In: 12th USENIX symposium on networked systems design and implementation (NSDI 15), pp 59–72
7. Tsagkaris et al (2015) Customizable autonomic network management: integrating autonomic network management and software-defined networking. IEEE Veh Technol Mag 10(1):61–68
8. Cox JH Jr, Clark RJ, Owen HL (2016) Security transition framework for software defined networks. In: Proceedings of the 2016 IEEE the first international workshop on security in NFV-SDN (SNS2016), Nov 2016. IEEE
9. Cisco, Network management system: best practices white paper. http://www.cisco.com/c/en/us/support/docs/availability/high-availability/15114-NMS-bestpractice.html
10. Congdon P, Aboba B, Smith A, Zorn G, Roese J (2003) IEEE 802.1 X remote authentication dial in user service (RADIUS) usage guidelines (No. RFC 3580)
11. ForeScout. https://www.forescout.com/solutions/use-cases/network-access-control/
12. Cisco NAC. http://www.cisco.com/c/en/us/products/collateral/security/nac-appliance-clean-access/product_data_sheet0900aecd802da1b5.html
13. Wilkins S (2015) A guide to network access control (NAC) solutions, May 2015. http://www.tomsitpro.com/articles/network-access-control-solutions,2-916-2.html

14. Skip Al, A bridge too far: defeating wired 802.1X with a transparent bridge using Linux. https://www.defcon.org/images/defcon-19/dc-19-presentations/Duckwall/DEFCON-19-Duckwall-Bridge-Too-Far.pdf
15. Bari MF, Chowdhury SR, Ahmed R, Boutaba R (2013) PolicyCop: an autonomic QoS policy enforcement framework for software defined networks. In: 2013 IEEE SDN for future networks and services (SDN4FNS), Nov 2013. IEEE, pp 1–7
16. Casado M, Freedman MJ, Pettit J, Luo J, McKeown N, Shenker S (2007) Ethane: taking control of the enterprise. In: ACM SIGCOMM computer communication review, vol 37, no 4, Aug 2017. ACM, pp 1–12
17. Matias J, Garay J, Mendiola A, Toledo N, Jacob E (2014) FlowNAC: flow-based network access control. In: 2014 third European workshop on software defined networks, Sep 2014. IEEE, pp 79–84
18. POX. http://www.noxrepo.org/pox/about-pox/
19. Ryu. http://osrg.github.io/ryu/
20. Lighttpd. https://www.lighttpd.net/
21. Lantz B, Heller B, McKeown N (2010) A network in a laptop: rapid prototyping for software-defined networks. In: Proceedings of the 9th ACM SIGCOMM workshop on hot topics in networks, Oct 2010. ACM, p 19
22. Cox JH Jr, Ryuretic security policy transition project. https://github.com/Ryuretic/SecRev
23. Phaal P (2003) Detecting NAT devices using sFlow. http://www.sflow.org/detectNAT
24. Cox JH Jr, Ryuretic rogue access point detection. https://github.com/Ryuretic/RAP
25. Trusted Computing Group. https://trustedcomputinggroup.org/work-groups/trusted-network-communications/

Jacob H. Cox Jr. received his B.S. in EE from Clemson University, SC in 2002, and his M.S. in ECE from Duke University, NC, in 2010. He has also recently completed his Ph.D. in ECE under the supervision of Dr. Henry Owen and Dr. Russell Clark at Georgia Institute of Technology, GA. As an Army officer, Jacob served as an Army telecommunications engineer (2008–2014) with his most recent assignments being assistant professor at the United States Military Academy (2010–2013) and chief of Enterprise Operations for the South West Asia Cyber Center in Kuwait (2013–2014). His research interests include software-defined networking and network security.

Russell J. Clark received his B.S. in Mathematics and Computer Science from Vanderbilt University in 1987. He received his M.S. and Ph.D. degrees in Information and Computer Science from Georgia Institute of Technology in 1992 and 1995. For the years 1997–2000, he was a senior scientist with Empire Technologies, a network management software company. He is currently a senior research scientist at Georgia Tech's School of Computer Science where he engages hundreds of students each semester in mobile development, networking, and the Internet of things. Russell is also the founder and co-director of the Georgia Tech Research Network Operations Center (GT-RNOC) and research director for SoX/Southern Crossroads.

Henry L. Owen III received his BSEE, MSEE, and Ph.D. in Electrical Engineering from the Georgia Institute of Technology in 1980, 1983, and 1989 respectively. He joined the research faculty of the Georgia Tech Research Institute in 1980 and the Georgia Institute of Technology academic faculty in 1989. He is a member of the Computer Engineering and the Telecommunications technical interest groups at the Georgia Institute of Technology. His research interests include software-defined internetworking and security.