Modeling and verifying the topology discovery mechanism of OpenFlow controllers in software-defined networks using process algebra

Xiang, S.Q.; Zhu, H.B.; Wu, X.; Xiao, L.L.; Bonsangue, M.M.; Xie, W.L.; Zhang, L.

Science of Computer Programming 187 (2020) 102343

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Modeling and verifying the topology discovery mechanism of OpenFlow controllers in software-defined networks using process algebra ☆

Shuangqing Xiang [a,c], Huibiao Zhu [a,*], Xi Wu [b], Lili Xiao [a], Marcello Bonsangue [c], Wanling Xie [d], Lei Zhang [e]

[a] Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
[b] School of Computer Science, The University of Sydney, Australia
[c] LIACS, Leiden University, Niels Bohrweg 1, 2333 CA, the Netherlands
[d] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China
[e] Shanghai Key Laboratory of Multidimensional Information Processing, East China Normal University, Shanghai, China

ARTICLE INFO

Article history:
Received 16 December 2018
Received in revised form 19 October 2019
Accepted 28 October 2019
Available online 4 November 2019

Keywords:
Formal verification
Modeling
SDN
Secure topology discovery
TopoGuard

ABSTRACT

Software-Defined Networking (SDN) is an emerging paradigm, providing separation of concerns between controllers that manage the network and switches that forward data flow. SDN enables network programmability and reduces the complexity of network control and management. The OpenFlow protocol is a widely accepted interface between SDN controllers and switches. OpenFlow controllers are the core of Software-Defined Networks (SDNs). They collect topology information to build a global and shared view of the network, which is used to provide services for topology-dependent core modules and applications. Therefore, the accuracy of the centralized abstract view of the network is of outermost importance for many essential SDN operations. However, the topology discovery mechanism used in almost all the mainstream OpenFlow controllers suffers from two kinds of topology poisoning attacks: Link Fabrication Attack and Host Hijacking Attack. TopoGuard is a wide-spread secure OpenFlow controller, which improves the standard topology discovery mechanism, providing automatic and real-time detection of these two attacks. However, the mechanism of TopoGuard lacks formal verification, especially in the situation where some hosts are migrating to their new locations. In this paper, we propose a general parameterized framework, including the Communicating Sequential Processes (CSP) models of the network components and the interfaces among them. Two loopholes of TopoGuard are found by implementing and verifying the proposed system model, which is an instance of the framework, in the model checker Process Analysis Toolkit (PAT). Moreover, we propose a new topology discovery mechanism based on TopoGuard, which solves the two loopholes.

☆ This paper extends the work published at TASE 2018 [1]: 12th International Symposium on Theoretical Aspects of Software Engineering.

* Corresponding author.
E-mail address: hbzhu@sei.ecnu.edu.cn (H. Zhu).

https://doi.org/10.1016/j.scico.2019.102343
0167-6423/

1. Introduction

In traditional computer networks, the control and the data planes are tightly coupled, and the whole structure is highly decentralized. Therefore, it is hard to effectively extend networks with new functionalities and reason precisely about their behaviors. Software-Defined Networking (SDN) has recently emerged as a new network paradigm, paving the way towards a next generation networking, producing an innovative research and development environment [2], and making it easier to monitor and manage network behaviors. A controller is the strategic point in a software-defined network, responsible for the management of the whole network, while the switches in the data plane are responsible for forwarding packets as well as reporting device status according to the instructions from the controller. The OpenFlow protocol [3] is the de-facto standard communication protocol between a controller and a switch, implementing the so called southbound interface between the control plane and the data plane. TopoGurad [4] is an OpenFlow controller, which provides a feasible mechanism to secure the topology discovery process. Later on, we will refer to an OpenFlow controller and an OpenFlow switch as OF controller and OF switch, or simply controller and switch.

An OF controller collects the entire network's topology information to create an abstract global view of the network, which enables a smooth and efficient network operation [5]. Many services in the application plane (e.g., routing, policy, etc.) and some core controller modules are highly dependent on the topology information maintained by the controller. Therefore, the accuracy of the topology information known by the controller is of great importance. The OpenFlow Topology Discovery Protocol (OFDP) is the topology discovery protocol used in almost all the mainstream open source OF controllers (Floodlight, OpenDayLight, Maestro, NOX, Ryu, etc.). However OFDP suffers from two information attacks: Host Hijacking Attack and Link Fabrication Attack, which take advantage of the loopholes in OFDP to poison the topology information known by the controllers. The two kinds of attacks are unique to SDN, which may lead to serious hijacking, denial of service or man-in-the-middle attacks [4,6–8]. Host Hijacking Attack concerns injecting fake host-generated packets into the network to deceive the controller into believing that the target host has migrated to another location, whereas Link Fabrication Attack involves an attacker creating a new malicious link in the network. In both cases the attacking goal is to gain control over traffic.

Some work has already been proposed to prevent the two kinds of attacks. In [8], a countermeasure-based method is proposed to overcome Link Fabrication Attack. The method adds a cryptographic Message Authentication Code (MAC) in each Link Layer Discovery Protocol (LLDP) packet to authenticate the packet's integrity. The work in [7] proposes a real-time accurate verification solution, which uses network flow graphs to detect attacks that violate those learned flow graphs/modules. In addition, Hong et al. [4] also designed a real-time low-overhead defensive solution, which adds some small changes into the topology discovery modules of OF controllers, and the secure OF controller is called TopoGuard. Although TopoGuard is in the experimental stage, the simple and effective design as well as the high performance of TopoGuard makes it as one of the most feasible methods to prevent the two kinds of attacks. However, the mechanism of TopoGuard lacks of formal verification, especially in the situation where the hosts are in the process of migration. And this is the reason why we choose it as the foundation to design a more secure topology discovery mechanism.

Communicating Sequential Processes (CSP) [9,10] has been widely applied in many fields, including distributed systems, concurrent systems, and networks [11–13]. Inspired by [11], we model the behaviors of TopoGuard, OF switches, hosts and two kinds of attackers as CSP processes. This paper is an extension of the work that is published at TASE 2018 [1].

Many other approaches have been proposed to verify SDNs. Some of them are designed to verify configurations of the data plane [14–18]. For example, FLOVER [18], a model checking system, can translate OpenFlow rules and a network security policy into an assertion set, which can then be processed and verified by an SMT solver. NetKAT [19], equipped with a sound and complete equational theory, is used for specifying and verifying packet-processing functions in SDNs.

Some other works provide tireless abstraction of the three layers of SDN and focus on the verification of the entire network [20–22]. For example, Flowlog [22] provides a Datalog-based tireless SDN programming language and enables built-in verification. Flowlog programs can be complied into Alloy, a verification tool, to verify some canonical properties [23]. [21] combines model checking and symbolic execution to mitigate the state-space explosion. CSP, like [22] and [21], enables the description of controller programs, switches and hosts through a unified abstraction. One of the most important features of CSP is that it allows the description of the communication among objects. CSP provides channels to model communications, which can help us build the model for the basic structure of SDNs through an intuitive way.

The main contributions of this paper are listed as below:

- We propose a parameterized framework that models the basic structure of SDNs, including three layers (control plane, data plane, and external environment) and the communications among them. The external environment of the networks discussed in this paper contains four kinds of hosts (compromised moving host, compromised unmoving host, normal moving host and normal unmoving host) and two kinds of attackers (link attacker and host attacker). One may extend an instance of the framework to create a specific model.
- We implement and verify the system model, which is an extended instance of the general framework. Two loopholes are found: One may make the system vulnerable to Host Hijacking Attack, and another one may hinder legal migrations.
- Through analysis, we come to a conclusion that the verification results of the two system models still hold for more complex networks that are discussed in Section 5.5.
- A new topology discovery mechanism based on TopoGuard is proposed, which is verified to cope with the two loopholes.

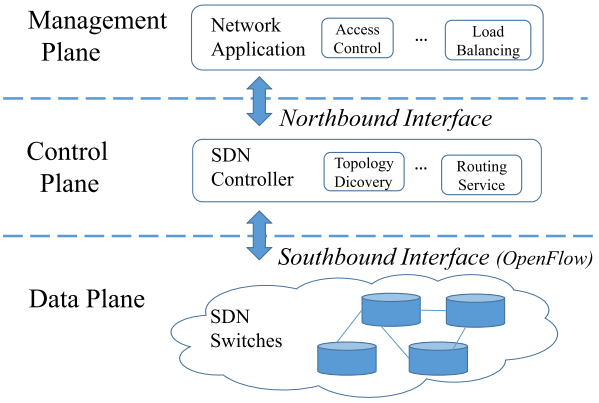


Fig. 1. SDN structure.

The remainder of this paper is organized as follows. We briefly introduce the preliminaries about the relevant concepts of the OpenFlow Protocol, CSP and PAT in the next section. Section 3 is devoted to introducing OFDP, the two kinds of attacks, and the mechanism of TopoGuard. The CSP models for TopoGuard, switches, hosts and attackers are described in Section 4. In Section 5, we implement as well as verify the proposed system model in PAT. Moreover, we discuss whether or not the verification results still hold for other SDNs. A new mechanism based on TopoGuard is proposed in Section 6, where we build the improved system model and verify it in PAT. Finally, we conclude the paper and present the future directions.

2. Preliminaries

2.1. OpenFlow protocol

The structure of SDN is shown in Fig. 1. The OpenFlow protocol is the most dominant southbound interface between the control plane and the data plane, through which an OF controller can manage and configure a remote OF switch. TopoGuard is applied to single-controller OpenFlow-based networks. One such network composes of several components: namely, one controller, a number of switches, and some hosts. We firstly introduce some glossaries according to [24].

OF Message: A controller may communicate with switches using OF messages, which are sent over connections.
Packet: A packet is treated as a unit to be forwarded among switches and hosts.
Connection: A connection carries OF messages between a switch and a controller, and it may be implemented using various network transport protocols.
Link: A link carries packets between a switch and another switch/a host. An internal link is a link between two switches.
Port: A switch has a set of OpenFlow ports, including several local reserved ports and some normal ports. *CONTROLLER* is the most common reserved port, which is used to exchange OF messages with the controller. Additionally, a switch may receive/forward packets via its normal ports.
Ingress Port, Ingress Switch: Consider a packet-in message that carries a packet. The switch that has sent the message to the controller is the ingress switch, and the port on which the packet was received by the ingress switch is the ingress port.
Flow table: A flow table in a switch contains some rules. When a packet comes into the switch via a normal port, the switch will search the flow tables to find a matching rule. Each rule is composed of some match fields, a set of instructions that will act on a specific packet, etc.
Matching: On receipt of a packet, a switch firstly extracts the header fields of the packet as well as some other information like the packet's ingress port, which are used to carry out a flow table lookup.

There are dozens of OF messages defined in the OpenFlow protocol. Here we only list three kinds of OF messages, which are used in the modeling of TopoGuard:

PktIn: We consider a switch who has received a packet with no matched rule. Then, the switch will send a packet-in message that carries the packet to the controller.
PktOut: A controller uses a packet-out message to ask a switch to forward a packet through a designated port.
PtStatus: A switch uses port-status messages to inform a controller of port changes.

2.2. CSP

Communicating Sequential Processes (CSP), first proposed by C.A.R. Hoare, is a process algebra for describing and analyzing the interactions in concurrent systems [9]. CSP processes are composed of primitive processes and actions. The processes in this paper are defined using the following syntax. a and b stand for atomic actions and c is the name of a channel.

$$P, Q ::= Skip \mid a \rightarrow P \mid c?x \rightarrow P \mid c!y \rightarrow P \mid P \| Q \mid P|||Q \mid P \Box Q \mid P; Q \mid P \triangleleft b \triangleright Q \mid (x := e\,; P)$$

- *Skip* represents a process that does nothing but terminates successfully.
- $a \rightarrow P$ denotes a process that first engages in the action a, and then behaves the same as the process P.
- $c?x \rightarrow P$ gets a message via the channel c and assigns it to variable x. Then the process behaves like P.
- $c!y \rightarrow P$ sends a message y via channel c and then behaves like P.
- $P \triangleleft b \triangleright Q$ describes that if the condition b is true, the behavior of the process is like P, otherwise, as Q.
- $P \| Q$ describes the concurrency of P and Q, and they are synchronized with their common actions or a pair of communication actions ($c!y$ and $c?x$).
- $P ||| Q$ denotes two processes which run concurrently without synchronization.
- $P \Box Q$ is a process that behaves either as P or Q, and the choice is made by the environment on the very first step.
- $P;Q$ represents a process that performs P and Q sequentially.
- *(x=e ;P)* is a process which behaves like P, except that the initial value of x is defined to be the value of the expression e. Here, a function in our model is equal to a sequence of assignments in CSP.

2.3. PAT

As a self-contained framework for composition, simulation and reasoning of concurrent systems [25] and other possible domains, Process Analysis Toolkit (PAT) [26] is designed as an extensible and modularized framework based on CSP. PAT implements various model checking techniques catering for different assertions such as deadlock freeness, reachability, and LTL with fairness assumptions in distributed systems [27]. The process notations in PAT are similar to those in CSP except some different symbols and expressions. For example, the general choice $\Box$ in CSP is expressed as [] in PAT, and $P = mod\{v = v + 1\} \rightarrow Skip$ in PAT denotes that the global variable v can be updated by an action named *mod*. The grammar of PAT is shown as below:

- *#define N x*: defines a global constant named N with initial value x.
- *var Topo[N][M]*: defines a global two-dimensional array named *Topo*.
- *channel C x*: declares a channel with identity C. x is the buffer size. Notice that a channel with buffer size 0 sends/receives messages synchronously.
- $P = mod\{v = v + 1\} \rightarrow Skip$: denotes that the global variable v can be updated by an action named *mod*.
- *#define cond* $v > 0$: defines a proposition named *cond*.
- *#assert P reaches cond*: defines an assertion that asks: whether process P can reach a state at which *cond* is satisfied. In order to determine whether the assertion is true, PAT's model checker performs a depth-first-search algorithm to repeatedly explore unvisited states until a state at which the condition is true is found or all states have been visited [26].
- *#assert* $P \models F$: asks whether every execution of P satisfies the LTL formula F.

3. Threats in topology discovery and overview of TopoGuard

3.1. Topology discovery

Topology discovery in an OpenFlow-based software-defined network concerns two services in a controller: (1) Link Manager for detecting internal links; (2) Host Tracker for locating hosts. Before the two services initiate, there is a Switch Discovery step, in which the controller discovers all the switches and gets the basic capabilities, like the ID of a switch and the ports on it, through the HandShake protocol [28].

3.1.1. Link manager

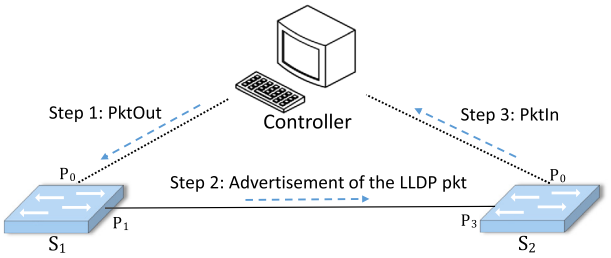
As the standard topology discovery protocol in OpenFlow-based SDNs, OFDP leverages the Link Layer Discovery Protocol (LLDP) [29]. The main parts of an OFDP/LLDP packet are shown in Table 1. The destination MAC address of an LLDP packet is set to a fixed bridge-filtered multicast address.

Then, we illustrate how OFDP works using a simple example shown in Fig. 2.

At first, the controller creates an LLDP packet for the port P_1 on the switch S_1 and sends it to S_1 though a packet-out message. Then, the switch S_1 will extract the LLDP packet from the packet-out message and forward it through P_1 according to the instruction in the packet-out message: *outport* = P_1. After receiving the LLDP packet from S_1, the switch S_2 will send a packet-in message that carries the LLDP packet, to the controller according to a pre-installed rule in its flow table. The

Table 1
The main parts of an OFDP/LLDP packet.

DstMAC	SrcMAC	EthType	Chassis ID	Port ID	Optional TLVs
01:80:C2:00:00:0E	Outgoing Port MAC	0x88CC	Switch ID	Outgoing Port ID	E.g., Signature



(a) Steps to detect the internal link $< S_1, P_1, S_2, P_3 >$.

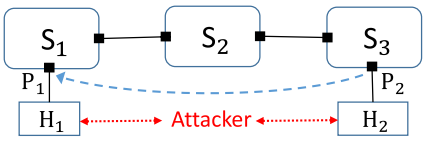
MsgType	OutPID	PktType	SrcSID	SrcPID
PktOut	P1	LLDP	S1	P1

(b) The packet-out message in step 1.

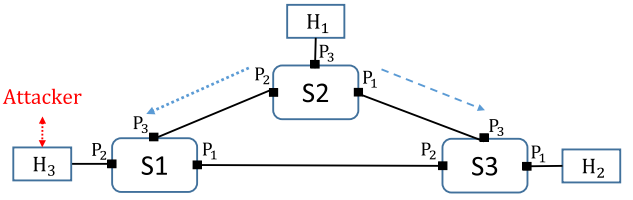
MsgType	InSID	InPID	PktType	SrcSID	SrcPID
PktIn	S2	P3	LLDP	S1	P1

(c) The packet-in message in step 3.

Fig. 2. An example of OFDP.



(a) Link Fabrication Attack.



(b) Host Hijacking Attack.

Fig. 3. Two examples of the two attacks.

packet-in message contains the identities of the ingress switch S_2 and the ingress port P_3. Besides, the LLDP packet has the information about the source switch ID and source port ID (i.e. Chassis ID and Port ID in Table 1). Therefore, the controller will deduce that there exists an internal link from (S_1, P_1) to (S_2, P_3).

3.1.2. Host Tracker

Host Tracker allows for monitoring traffic, assisting in traffic routes, and determining the source of host-generated packets [30]. After detecting a packet-in message that contains a host-generated packet (e.g. an Address Resolution Protocol (ARP) packet), Host Tracker will extract the host's ID from the packet. In addition, Host Tracker will obtain the host's location information, including the ID of the switch that connects the host (InSID) and the ID of the host's attachment port (InPID).

3.2. Threats in topology discovery

3.2.1. Link Fabrication Attack

An example of Link Fabrication Attack is shown in Fig. 3(a). We assume that the host H_1 and the host H_2 have been compromised by an attacker. At first, H_1 (the link attacker) sends a fake LLDP packet of which the Chassis ID field and the Port ID field are S_3 and P_2 to the switch S_1. A fake LLDP packet is generated by tampering a genuine LLDP packet received by H_1, or it is a copy of a normal LLDP packet received by H_2 [4]. Then, S_1 will send a packet-in message that carries the fake LLDP packet to the controller. Finally, the controller will deduce the existence of an internal link from P_2 on S_3 to P_1 on S_1, which is denoted by the dashed line.

3.2.2. Host Hijacking Attack

As shown in Fig. 3(b), before the attack, the traffic from host H_1 to host H_2 is illustrated by the dashed line. Then the host attacker injects a fake ARP packet of which the Source Host ID field is H_2 into the network through H_3. After observing the fake ARP packet, the controller will believe that H_2 has moved to P_2 on S_1. Therefore, the Routing Service in the controller will compute a new route for the traffic from H_1 to H_2. For example, a rule which says that the packet sent from H_1 to H_2 should be forwarded to port P_2 will be inserted into the flow table of S_2. As a result, the traffic from host H_1 to H_2 will be directed to H_3 and then obtained by the attacker. The traffic hijacked by the attacker is shown by the dotted line.

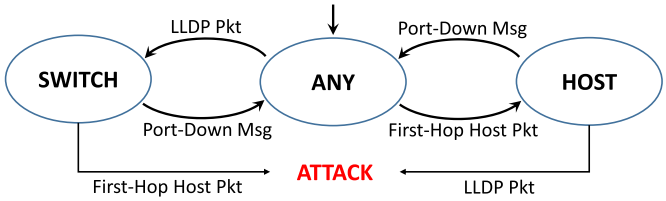


Fig. 4. Value transition of Device Type.

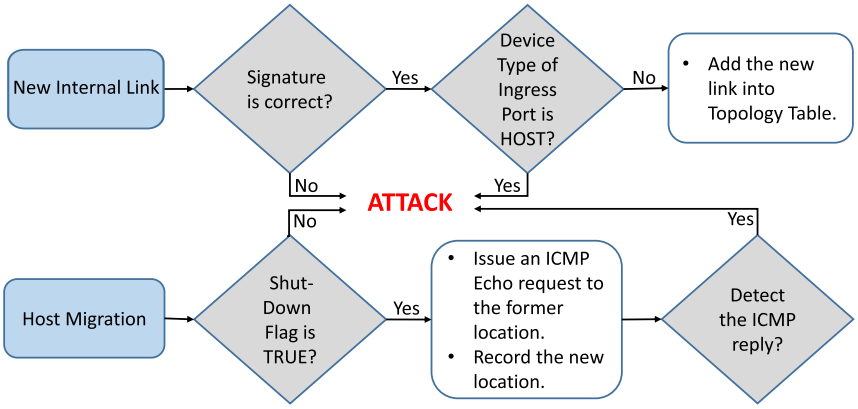


Fig. 5. The mechanism of TopoGuard.

3.3. TopoGuard

TopoGuard is a secure OF controller, which is extended from NOX. The improvement to OFDP proposed by TopoGuard could also be applied to other OF controllers which use OFDP as their topology discovery protocol. TopoGuard maintains three attributes for a switch port: Device Type, Host List and Shut-Down Flag.

Host List When TopoGuard detects a host-generated packet, if there is no record of the host in any host list, in which case the packet is a first-hop host packet, TopoGuard will add the host into the host list of the packet's ingress port.
Shut-Down Flag It will be set to TRUE once TopoGuard receives a related port-down signal, which is carried by a port-status message.
Device Type As shown in Fig. 4, the initial value of Device Type of a specific switch port is ANY. Then Device Type of the port is set to HOST once TopoGuard observes a first-hop host packet that is received by the port, or the value is set to SWITCH when an LLDP packet received via the port is monitored. The value is set back to ANY when TopoGuard detects a port-down signal from the port.

Fig. 5 shows the mechanism of TopoGuard on notification of a new internal link or a host migration. If a packet-in message that carries an LLDP packet arrives, and the internal link deduced from the message has not been observed before, a new internal link event is triggered. In this case, TopoGuard will check the controller-signed signature in the packet to ensure that the packet has not been tampered. Then, TopoGuard will detect if there is any host lies on the path of the LLDP propagation by checking Device Type. If Device Type of any of the two ports on the new link is HOST, the controller will issue an alert and deny the link update.

If TopoGuard detects a host-generated packet from a port, and the host's location deduced from the packet is different from the one recorded in Host Table, TopoGuard will check the legality of the host migration. Before the host migration finishes, the controller must have received a port-down signal of the port [4]. If Shut-Down Flag of the port is TRUE, then TopoGuard will modify the location information of the host. After that, TopoGuard will send an ICMP Echo request to the former location of the host and wait for a response within a reasonable time. If TopoGuard receives the corresponding ICMP Echo reply later, which indicates the migration is malicious, it will withdraw the previous location update and raise an alert.

4. Formalizing SDNs with TopoGuard and attackers

In order to check whether or not the mechanism in TopoGuard can successfully prevent the two kinds of attacks described in Section 3, we model the behaviors of TopoGuard using CSP in this section. Additionally, we also formalize OF switches, four kinds of hosts and two kinds of attackers. In addition, we define some tables to store the topology information collected by the controller, flow rules, etc. CSP processes can read or write those tables.

In topology discovery process, there are many kinds of attacks that aim at different objects. In this paper, we focus on the two kinds of new attacks described in Section 3, which are peculiar to SDNs, instead of the attacks against classical protocols or algorithms.

We first make some important assumptions before modeling.

4.1. Assumptions

1. As mentioned before, there are three parts in topology discovery: switch discovery, link discovery and host discovery. The switch discovery step does not require any additional protocol since when an OpenFlow switch establishes a connection to the OpenFlow controller, the switch information should be stored in the OpenFlow controller for future management [4]. Of course, there are attacks that aim at switch discovery step. For example, attacks that take advantages of the loopholes of TCP or TLS, but this paper concentrates on Host Hijacking Attack and Link Fabrication Attack, which aim at host discovery and link discovery step respectively. Therefore, we assume that the Switch Discovery step is secure.

2. If the signature algorithm for constructing LLDP packets are secure enough, it is obvious that TopoGuard will be able to detect Link Fabrication Attack with tampering style. Discussing the security of classical signature algorithms is out of the scope of this paper, and therefore, we omit the signature field from LLDP header and let an attacker launch Link Fabrication Attack in the relay manner.

3. In our model, before initiating an attack, an attacker has gained enough information about its target.

- For Link Fabrication Attack, the attacker has to get a copy of the target LLDP packet (a normal LLDP packet received by H_2 in Fig. 3(a)), which can be achieved by setting a physical link or tunnel.
- For Host Hijacking Attack, the identity of the target host (H_2 in Fig. 3(b)) is needed. The attacker may retrieve the target identifier through an ARP request.

None of the possible behaviors above will set the device type of the port that belongs to an internal link to HOST, or set the device type of the port that connects to a host to SWITCH. Therefore, these behaviors will not influence the verification results.

4. This abstraction comes from the third assumption. During the process for a host attacker to get the identifier of its target host, both of the compromised host and the target host will generate some host-generated traffics. Therefore, when the host-hijacking attack starts, the controller has already recorded the location of the corresponding hosts and changed the attributes of related ports. As a result, in our model, we will manually set the initial locations of the hosts and the attributes of related ports.

4.2. Definitions

In our model, all the communications among the components are formalized as passing messages through channels. We show some channels and the corresponding processes in Fig. 6. A directed edge in the figure denotes a channel between two processes as well as the direction of message transmission.

CS$_x$, ***S***$_x$***C***: represent the communications between the controller and the switch S_x. The controller receives OF messages via S_xC and sends OF messages through CS_x. Since every OF switch in the network is connected to the controller, there are also two channels between the switch S_y and the controller. Here we omit them from the figure for simplicity.
S$_x$***P***$_i$: represents forwarding packets to the switch S_x through its port P_i, assuming that there is an internal link between P_i ofS_x and P_j of S_y. S_x waits to receive packets from S_y through channel S_xP_i. S_x can also send packets to S_y through channel S_yP_j. Besides, assuming that the host H_u is connected to the port P_m of S_x, then the host may forward packets to S_x through channel S_xP_m.
SH$_u$: denotes forwarding packets to H_u. The link between H_u and its connected switch S_x is modeled as the channel SH_u and the channel S_xP_m, and the host is connected to port P_m on S_x. Notice that if H_u is compromised by an attacker, the attacker may inject packets into the network through the channel S_xP_m.
Monitor$_v$: is used by a link attacker to monitor the traffic that arrives at the host H_v.
PtChange: is designed to make sure that the controller will receive a port-down signal before the host H_u finishes its migration.
Host, Link: represent delivering packet-in messages to Host Tracker and Link Manager respectively.
CSend: is used by both Host Tracker and Link Manager to ask the controller to issue OF messages.

The definitions of the OF messages and the packets used in our model are illustrated in Table 2. Here @ is a separator. Notice that the format of a packet in our model is an abstraction of its practical format. That is, we omit some fields that are not necessary.

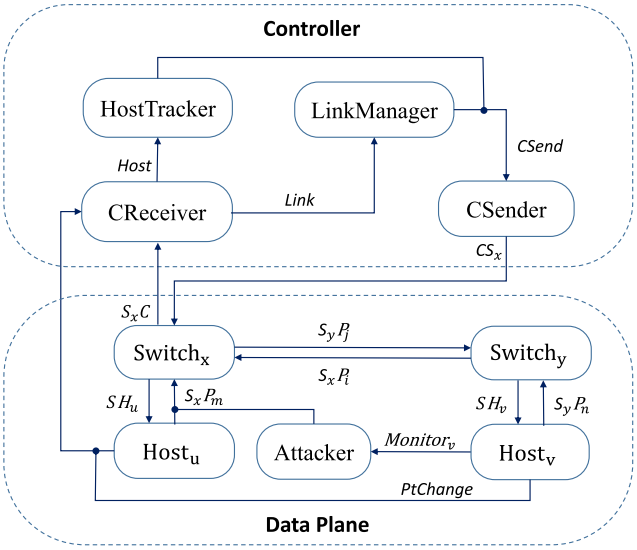


Fig. 6. The channels used in modeling.

Table 2
Definitions of OF messages and packets.

PktType	Format
ICMPEcho	PktType@Type@DstHID
LLDP	PktType@SrcSID@SrcPID
ARP	PktType@SrcHID
OFMsgType	**Format**
PktIn	MsgType@InSID@InPID@Pkt
PktOut	MsgType@OutPID@Pkt
PtStatus	MsgType@Type@SID@PID

ICMPEcho: Since we are more concerned about the interactions among the network components than how to design rules for different protocols, we just define an abstracted packet type here. DstHID is the ID of the destination host. The value of Type can be Request or Reply.
LLDP: an LLDP packet has three fields: packet type, source switch ID and source port ID.
ARP: SrcHID in an ARP packet represents the ID of the source host that has generated the packet.
PktIn: Similar to the definition of packets, an OF message has a message type field. InSID and InPID denote the ingress switch ID and ingress port ID respectively. Pkt denotes the packet that is carried in the OF message. The format of Pkt could be the format of ICMPEcho, LLDP, or ARP.
PktOut: OutPID denotes the port through which the packet will be sent out by the switch.
PtStatus: The value of Type could be PtDown or PtUp. Switch ID together with port ID denote the port that triggers the port-status message.

We list all the tables in Table 3. The definitions we give are abstracted from the real implementations or specifications. A table is indexed by the bold field/fields. A flow rule in Flow Table has four matching fields, which are the primary key of the table, and the value of a field may be a wildcard. OutPID represents an output action, denoting the port through which the switch should forward a matched packet. Notice that Host Table and Link Table store the topology information collected by the controller, while HostLoc and InterLink store the real topology of the network. HID in Host Table denotes host identity. SID and PID in Host Table represent the switch identity and the port identity respectively, showing the attachment point of a host. The information of an internal link from port P_{PID_1} of switch S_{SID_1}, represented by (SID_1, PID_1), to port P_{PID_2} of switch S_{SID_2} is stored in Link Table.

4.3. The TopoGuard controller

Then we use CSP to model the TopoGuard controller. As shown in *C1*, the process *Controller* describes the behaviors of TopoGuard. An OF controller has two modules to receive and send OF messages respectively. Therefore we define two processes: *CReceiver* and *CSender*, which may communicate with the processes *LinkManager* and *HostTracker*.

$$C1 : Controller =_{df} (CReceiver|||CSender)||(LinkManager|||HostTracker)$$

Table 3
Definitions of some tables.

Name	Owner	Definition	Functionality
Flow Table	Switch	**InPID*****PktType*****SrcHID** ***DstHID***OutPID	Storing rules which are used in the matching step.
Host Table	Controller	**HID***SID*PID	Recording host location information collected by the controller.
Link Table	Controller	**SID1*****PID1*****SID2*****PID2**	Storing the internal link information collected by the controller.
Port Table	Controller	**SID*****PID***DeviceType *ShutDownFlag*HostList	Holding the attributes of the switch ports.
HostLoc		**HID***SID*PID	Recording the real locations of the hosts in the network.
InterLink		**SID1*****PID1*****SID2*****PID2**	Storing the information of the links among switches.

Then we describe the behaviors of the receiving module. As shown in *C2*, if more than one switch tries to send an OF message to the controller at the same time, the controller will randomly choose one switch to pass a message. In addition, the controller may receive port-status messages via the channel *PtChange*.

In practical, a port-change signal is detected by a switch, and then the switch will send a port-status message to the controller. Here in order to make sure that the controller will receive a port-down signal before the corresponding host migration finishes, we omit the self-detection process of a switch and let a migrating host to inform the controller of a correlated port-status change.

As shown in $C3'$, the Shut-Down Flag of the port denoted by the port-status message will be set to FALSE, if the Type field of the message is PtDown.

Normally, an OF controller has a separate module to decide where to deliver a received packet-in message. Here we integrate this decision-making functionality into *CReceiver*. As shown in *C3*:

- If a packet-in message that carries an LLDP packet arrives, then $CReceiver_1$ will send the message to *LinkManager* to handle a possible link update.
- *HostTracker* will be invoked if the packet-in message carries a host-generated packet.

$$
\begin{aligned}
&C2 : CReceiver =_{df} \\
&\quad ((\Box_{x\in\{0\ldots SNum-1\}} S_x C?msg \rightarrow CReceiver_1(msg)) \\
&\quad \Box(PtChange?msg \rightarrow HandlePtChange(msg))); CReceiver \\
&C3 : CReceiver_1 =_{df} \\
&\quad (Link!msg \rightarrow Skip) \\
&\qquad \lhd MsgType[msg] == PktIn \wedge msg[3] == LLDP \rhd \\
&\quad (Host!msg \rightarrow Skip \\
&\qquad \lhd msg[3] == ARP \vee msg[3] == ICMPEcho \rhd Skip) \\
&C3' : HandlePtChange(msg) =_{df} \\
&\quad setSDFlag(SID[msg], PID[msg], FALSE); Skip \lhd Type[msg] == PtDown \rhd Skip
\end{aligned}
$$

Some notations used in *C2*, *C3* and $C3'$ are listed as below:

- *SNum* is a constant that stands for the number of the switches.
- *msg* is a tuple of varying length that represents an OF message.
- *msg[3]* is the value of the fourth element of the tuple *msg*, which represents the fourth field of the OF message that just received.
- *MsgType[msg]* is equal to *msg[0]*, denoting the MsgType field of an OF message.
- *Type[msg]* is equal to *msg[1]*, standing for the Type field of a port-status message.
- *PktIn, LLDP, ARP, ICMPEcho, and PtDown* are enumeration values.
- The function *setSDFlag(SID[msg],PID[msg],FALSE)* modifies the attribute Shut-Down Flag of the port. Notice that the syntax of CSP does not contain function. Here, a function in our model is equal to a sequence of assignments in CSP.

C4 describes the module that is responsible for issuing OF messages to switches. On receiving an OF message from another process, *CSender* will issue the message to the designated switch.

$$C4 : CSender =_{df} CSend?x.msg \rightarrow CS_x!msg \rightarrow CSender$$

- *x.msg* is a tuple where *msg* represents the OF message that is waiting to be issued and the first element *x* denotes the target switch to which the OF message will be sent.

Then, from *C5* to *C7*, we show how to model Link Manager, which is responsible for constructing LLDP packets as well as handling the received LLDP packets. As shown in *C5*, we build two processes to model the two functionalities mentioned above.

$$C5: LinkManager =_{df} LLDPConstructor|||LLDPHandler$$

As described in *C6*, in order to discover all the internal links in the network, Link Manager will generate an LLDP packet (represented by *LLDP.x.i*) for each port on each switch except for the reserved ports. Each switch owns one reserved port to communicate with the controller, and the ID of the reserved port is zero. Therefore, the range of x starts from 1 instead of 0. Then each LLDP packet is encapsulated into a packet-out message, and Link Manager will deliver these messages to the sending module. The packet-out message *PktOut.i.LLDP.x.i* will ask switch S_x to forward the LLDP packet *LLDP.x.i* through port P_i since the OutPID field of the message is i.

$$C6: LLDPConstructor =_{df} |||_{x\in\{0...SNum-1\},i\in\{1...PNum_x-1\}}(CSend!x.PktOut.i.LLDP.x.i \rightarrow Skip)$$

- *PNum* is a constant that stands for the number of the ports on a switch. We assume that each switch has the same number of ports.

As shown in *C7*, at first, the process *LLDPHandler* will check whether the internal link that is deduced from the received packet-in message has been detected before. The checking step is modeled as the function *searchLink(msg)*. If the return value of the function is TRUE, *LLDPHandler* will check the Device Type of the two ports on the new internal link. *getTD* gets the value of the attribute Device Type of a designated switch port by searching Port Table. If any of the two ports connects to a host, an alert will be triggered. The alert is modeled as executing the action *attack*. If the new internal link is legal, *LLDPHandler* will insert it into Link Table and update Device Type of the two ports.

$$\begin{aligned}
C7: LLDPHandler =_{df}\ & (Link?msg \rightarrow \\
& is_new_link = searchLink(msg); \\
& ((port_type_1 = getDT(SrcSID[msg], SrcPID[msg]); \\
& \ port_type_2 = getDT(InSID[msg], InPID[msg]); \\
& \ (attack \rightarrow Skip \\
& \quad \triangleleft port_type_1 == HOST \vee port_type_2 == HOST \triangleright \\
& \ insertLink(msg); setDT(msg); Skip)) \\
& \quad \triangleleft is_new_link == TRUE \triangleright \\
& \ (Skip))); LLDPHandler
\end{aligned}$$

- *is_new_link* is a global variable that is used to store the return value of the function *searchLink(msg)*.
- *searchLink(msg)*, *insertLink(msg)*, etc. are functions.

We can also describe Host Tracker as the CSP process *HostTracker*. Later in Section 6, we will give the model for the improved Host Tracker, and the main structure of the process *NewHostTracker* is the same as the structure of the process *HostTracker*. Additionally, we have described the mechanism of handling host-migration event in Section 3, and the formalization of this mechanism is straightforward. Therefore, we omit the details of the process *HostTracker*.

4.4. Switch

After describing the model of the controller, now we show how to formalize the switches. In order to reduce the state space, we pre-install all the necessary rules into flow tables. For example, we pre-install a rule for each switch to direct an ICMP Echo reply to the controller. A switch is composed of a packet handler, a message handler, a reserved port, some normal ports and a flow table. The packet handler is responsible for handling all the arrived packets, while the message handler will deal with OF messages. The behavior of a normal port with identity i on S_x is modeled as the process $PktReceiver_{xi}|||PktSender_{xi}$. The process $MsgReceiver_x|||MsgSender_x$ behaves as the reserved port with identity 0. In summary, a switch with identity x is modeled as follows:

$$\begin{aligned}
S1: Switch_x =_{df}\ & PktHandler_x||MsgHandler_x||(MsgReceiver_x|||MsgSender_x) \\
& (|||_{i\in\{1..PNum-1\}}(PktReceiver_{xi}|||PktSender_{xi}))
\end{aligned}$$

Then we explain how the sub-processes of $Switch_x$ interact with each other, and how they communicate with the environment.

As shown in Fig. 7(a), if more than one normal port has received a packet, one of these packets will be chosen to be handled, and the others will wait. The switch will search its flow table to find a matched rule for the chosen packet. The matching step, which is modeled as $PktHandler_x$, is shown in Fig. 8. The two branches of the matching step are listed as below.

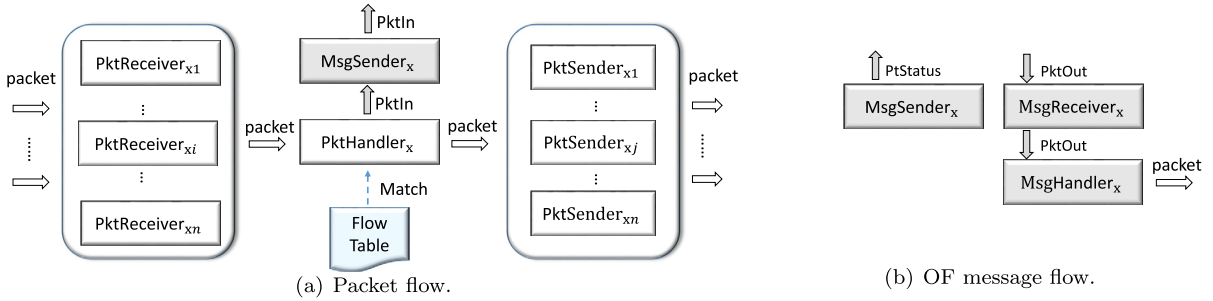


Fig. 7. Interactions of the sub-processes of the switch model.

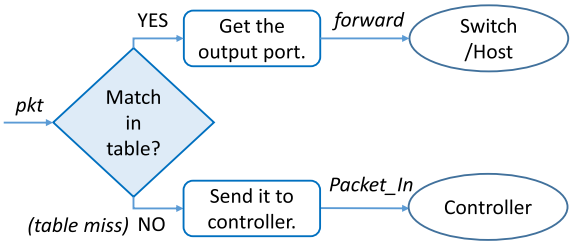


Fig. 8. The matching step.

- If there exists a matching rule for the packet in the flow table, then the packet will be forwarded to another switch or another host. If the value of the OutPID field in the matching rule is p, then $PktHandler_x$ will send the packet to $PktSender_{xp}$.
- Otherwise, a packet-in message that carries the packet will be sent out through the reserved port, and this behavior is formalized as sending the packet to the process $MsgSender_x$.

As shown in Fig. 7(b), a switch may receive packet-out messages from the controller. If the OutPID field of a packet-out message is p, then $MsgHandler_x$ will send the packet to the process $PktSender_{xp}$. The sub-processes of the process $Switch_x$ are listed from *S2* to *S7*. On receiving a packet from port P_i, the behaviors of switch S_x are formalized by the processes *S2* to *S4*.

$$S2 : PktReceiver_{xi} =_{df} S_xP_i?pkt \rightarrow HandlePkt_x!i.pkt \rightarrow PktReceiver_{xi}$$

$$\begin{aligned} S3 : PktSender_{xi} =_{df}\ & SendPkt_{xi}?pkt \rightarrow index = getSwitchPort(x,i); \\ & (sw = index[0];\ pt = index[1];\ S_{sw}P_{pt}!pkt \rightarrow PktSender_{xi}) \\ & \quad \triangleleft index! = EMPTY \triangleright \\ & (ht = getHost(x,i);\ SH_{ht}!pkt \rightarrow PktSender_{xi}) \end{aligned}$$

$$\begin{aligned} S4 : PktHandler_x =_{df}\ & HandlePkt_x?i.pkt \rightarrow out\_port = matchPkt(i,pkt); \\ & (SendMsg!PktIn.x.i.pkt \rightarrow PktHandler_x) \\ & \quad \triangleleft out\_port == EMPTY \triangleright \\ & (j = out\_port;\ SendPkt_{xj}!pkt \rightarrow PktHandler_x) \end{aligned}$$

- *getSwitchPort(x,i)* is a function that will search the table InterLink to find the switch port that connects to P_i of S_x. And the result is stored in the tuple *index* of which the first and the second elements are the switch identity and the port identity respectively.
- *getHost(x,i)* will search the table HostLoc to find the host that connects to P_i of switch S_x.
- *matchPkt(i,pkt)* denotes the matching step. The value of the OutPID field of the matched rule is stored in the global variable *out_port*.

Processes *S5* to *S7* describe that switch S_x receives an OF message from the controller and then handles the message. After receiving an OF message, $MsgReceiver_x$ will send the message to $MsgHandler_x$ through the channel $HandleMsg_x$. If the message is a packet-out message, the packet carried in the message will be extracted and sent to the process $MsgSender_x$ through the channel $SendPkt_{xp}$.

$$S5 : MsgReceiver_x =_{df} CS_x?msg \rightarrow HandleMsg_x!msg \rightarrow MsgReceiver_x$$
$$S6 : MsgSender_x =_{df} SendMsg_x?msg \rightarrow S_xC!msg \rightarrow MsgSender_x$$
$$S7 : MsgHandler_x =_{df} HandleMsg_x?msg \rightarrow$$
$$(pkt = getPkt(msg);\ p = OutPID[msg];\ SendPkt_{xp}!pkt \rightarrow MsgHandler_x)$$
$$\triangleleft MsgType[msg] == PktOut \triangleright$$
$$(MsgHandler_x)$$

- *getPkt(msg)* will extract the packet in the message and store the packet into *pkt*.
- *OutPID[msg]* is equal to *msg[1]*, denoting the OutPID field in an packet-out message.

4.5. Host

The hosts in the network can be grouped into two categories: unmoving host and moving host. Later in the next part, after introducing the attacker model, we will give another two host models that describe hosts that are monitored by attackers. We consider a host with identity u, and its location is $l=(x,i)$, denoting that host H_u connects to port P_i on switch S_x. As analyzed in Section 4.1, we will manually set the initial locations of the hosts and the attributes of related ports.

First, we describe the behaviors of a host that will not move to another location. As shown in H_2, host H_u waits to receive packets from channel SH_u. On receiving an ICMP Echo request with u as its DstHID field, the host will reply an ICMP Echo packet: *ICMPEcho.Reply.u*.

$$H_1 : UnmovingHost(u, x, i) =_{df} UnmovingHost_1(u, x, i);\ UnmovingHost(u, x, i)$$
$$H_2 : UnmovingHost_1(u, x, i) =_{df} SH_u?pkt \rightarrow$$
$$(S_xP_i!ICMPEcho.Reply.u \rightarrow Skip$$
$$\triangleleft PktType[pkt] == ICMPEcho \wedge Type[pkt] == Request \wedge DstHID[pkt] == u \triangleright$$
$$Skip)$$

- *Request* and *Reply* are two enumeration values.

Then we will expand the unmoving-host model to enable the migration of a host.

- The migration of a host contains a period of downtime, that is to say, the host will stop working for a while before starting to work in its new location.
- Besides, the switch port that connects to the host will be shut down before the host finishes its migration.

Process *H3* describes a host with location $l=(x,i)$. The host will move to a new location $l'=(y,j)$ at some time. As we mentioned before, an unmoving host may receive a packet and possibly inject an ICMP Echo reply into the network, and the migration may happen at any time. Therefore, we could combine the two actions in the unmoving-host model with a migration process in any order, which produces three combinations connected by $\Box$. If the process H_3 could perform more than one branch at the same time, it will make its choice arbitrarily.

Process *H4* describes the three steps for a host to migrate:

- (1) The controller will notice the change of the correlated port. As we analyzed before, by making a communication on channel *PtChange*, the controller will receive a port-down signal.
- (2) The host migrates from $l=(x,i)$ to $l'=(y,j)$, and we model it by modifying the location of host H_u in the table HostLoc.
- (3) The host injects an ARP packet into the network, which will help the controller to record the host's new location.

$$H3 : MovingHost(u, x, i, y, j) =_{df} ((UnmovingHost_1(u, x, i);\ Migration(u, x, i, y, j))$$
$$\Box(SH_u?pkt \rightarrow Migration(u, x, i, y, j))$$
$$\Box(Migration(u, x, i, y, j);\ UnmovingHost(u, y, j))$$
$$H4 : Migration(u, x, i, y, j) =_{df}$$
$$PtChange!PtStatus.PtDown.x.i \rightarrow (move(u, x, i, y, j);\ (S_yP_j!ARP.u \rightarrow Skip))$$

Notice that we are not concerned about the specific time period that a host is shut down, but only focus on the order in which the actions occur. Once the process *Migration* starts to execute, the process *MovingHost* cannot receive or handle any packet until *Migration* finishes its execution. Moreover, executable actions of other processes could happen between the action *PtChange!PtStatus.PtDown.x.i* and $S_yP_j!ARP.u$.

4.6. Attacker

We model two kinds of attackers: link attacker and host attacker to launch Link Fabrication Attack and Host Hijacking Attack respectively. Instead of letting an attacker gain necessary information for launching an attack in the model, we manually set the position that the attacker initiates attack and the information of the attacking target. As discussed at the beginning of this section, the attacker behaviors during the omitted steps will not bring attribute modifications that influence the verification results. By instantiating the parameterized attacker models proposed, one may create specific attacking scenarios.

4.6.1. Link attacker

As discussed in Section 4.1, we are only concerned about the relay-fashion Link Fabrication Attack. Considering two compromised hosts: H_v and H_u. H_u is attached to P_i of S_x. If an LLDP packet is captured by the link attacker via the channel $Monitor_v$, the attacker will insert the LLDP packet into the network via the channel S_xP_i.

$$
\begin{aligned}
&A2 : LinkAttacker(x,i,u,v) =_{df} Monitor_v?pkt \rightarrow \\
&\quad (S_xP_i!pkt \rightarrow LinkAttacker_1(x,i,u,v)) \\
&\qquad \triangleleft PktType(pkt) == LLDP \triangleright \\
&\quad (LinkAttacker(x,i,u,v)) \\
&A3 : LinkAttacker_1(x,i,u,v) =_{df} Monitor_v?pkt \rightarrow LinkAttacker_1(x,i,u,v)
\end{aligned}
$$

Moreover, we must allow the action of being monitored in the host models. The new processes are named *UnmovingHostM* and *MovingHostM* respectively. We take *UnmovingHostM* for example.

$$
\begin{aligned}
&H_5 : UnmovingHostM(u,x,i) =_{df} UnmovingHostM_1(u,x,i); UnmovingHostM(u,x,i) \\
&H_6 : UnmovingHostM_1(u,x,i) =_{df} \\
&\qquad SH_u?pkt \rightarrow Monitor_u!pkt \rightarrow \\
&\qquad (S_xP_i!ICMPEcho.Reply.u \rightarrow Skip \\
&\qquad\quad \triangleleft PktType[pkt] == ICMPEcho \wedge Type[pkt] == Request \wedge DstHID[pkt] == u \triangleright \\
&\qquad\quad Skip)
\end{aligned}
$$

4.6.2. Host attacker

Considering an attacker that is running on a compromised host H_u, and its target host is H_v. The compromised host is connected to P_i on S_x. As analyzed before, when the attacker is about to inject a fake ARP packet, the controller has already recorded the two hosts' locations and also changed Device Type of the related switch ports. In our model, we will manually set the information above before starting verification. As shown in *A1*, the fake ARP packet will be injected into the network through the channel from H_u to S_x, that is, the channel S_xP_i.

$$A1 : HostAttacker(x,i,u,v) =_{df} S_xP_i!ARP.v \rightarrow Skip$$

5. Verification

The parameterized CSP models proposed in Section 4 constitute a general framework, which can be instantiated as needed. In this section, we are concerned with three problems about TopoGuard:

- If TopoGuard can prevent Link Fabrication Attack?
- If TopoGuard can prevent Link Fabrication Attack?
- Will TopoGuard hinder legal host migrations?

In order to answer these questions, we first build a system model, which is an instance of the general framework, according to the network in Fig. 9. Then we implement the CSP models in Section 4 and the system model in PAT. After that, we verify the three assertions that are corresponding to the three problems for the system model. Finally, we discuss if the verification results hold for other SDNs.

5.1. The original system model

The network we used to build the system model is illustrated in Fig. 9. The host H_0 will not move to another location, while H_1 will move to the switch S_1 from S_2. Before H_1 moves to the port P_2 of S_1, H_2 will move to P_1 of S_2. A host attacker will inject a fake ARP packet into the network through H_0 to deceive the controller into believing that H_1 has moved to P_2 of S_0. Additionally, a link attacker will monitor H_1 and launch an attack from H_0 to try to create a fake link from P_2 of S_2 to P_2 of S_0.

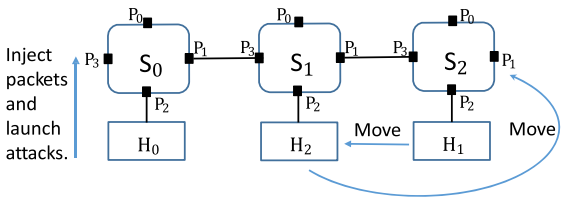


Fig. 9. The network used in verification.

Then we illustrate the CSP model as below. S_0, H_0, etc. are enumeration values. We pre-installed all the necessary flow rules into the flow tables of the switches. In addition, Device Type of P_2 on S_0 and P_2 on S_2 are set to HOST before verifying.

$$\begin{aligned} System =_{df}\ & (||_{x\in\{0..2\}} Switch_x)||Controller \\ & ||(Migration(H_2, S_1, P_2, S_2, P_1); \\ & \quad (MovingHostM(H_1, S_2, P_2, S_1, P_2)|||UnmovingHost(H_2, S_2, P_1))) \\ & ||UnmovingHost(H_0, S_0, P_2) \\ & ||(HostAttacker(S_0, P_2, H_0, H_1)|||LinkAttacker(S_0, P_2, H_0, H_1)) \end{aligned}$$

5.2. Implementing the CSP models in PAT

Once we have a PAT program and an assertion, we may ask PAT to search the state space of the program to check if the assertion is satisfied. The syntax of PAT is similar to CSP except for a few symbols and expressions. However, PAT only supports one-dimensional channel array, while CSP supports defining channels which contain multiple indexes, for example S_xP_i. The solution is to translate two-dimensional indexes into one-dimensional indexes.[1]

First of all, some of the global definitions are shown as below:

$channel\ SH[HNum]\ 0;$
$channel\ SP[SNum * (PNum - 1)]\ 0;$
$var\ LinkTable[SNum][SNum][2];$
$var\ InterLink[SNum][SNum][2];$
$var\ HostTable[HNum][2];$
$var\ HostLoc[HNum][2];$
$enum\{PtStatus, PktOut, PktIn\};$

SH[HNum] is a set of channels that denote the links from switches to hosts, corresponding to the channel SH_u in the CSP models.
SP[SNum*(PNum-1)] represents the channels used by switches to receive packets from hosts or other switches. The corresponding CSP channel is S_xP_i.
LinkTable, InterLink, HostTable, and ***HostLoc*** denote the four tables that we defined in the previous section. *LinkTable[x][y][0]/LinkTable[x][y][1]* represents the identity of the port that is part of the internal link from switch S_x to switch S_y, and the port is on switch S_x/S_y. *HostTable[u][0]* stands for the switch that is connected to H_u, while *HostTable[u][1]* represents the switch port that H_u connects.
PtStatus, PktOut, and ***PktIn*** are enumeration values, standing for OF message types.

One thing to be careful of is that, if the PAT process *PktSender(x,i)*, which represents the sending functionality of a switch port, tries to send out a packet, it has to get the channel index of the switch port or the host that connects to P_i of S_x. We define an array named *CIndex[SNum][PNum-1]* to store the channel indexes. At the beginning, we initialize this channel-index array according to two tables *InterLink* and *HostLoc*. In this step, we use two mutually exclusive sets of values to distinguish switch ports and hosts. Then by checking the value of *CIndex[x][i]*, the process *PktSender(x,i)* will know where to send the packet. For example, *CIndex[x][i]*=101 represents the channel that is used by host H_1 to receive packets, and then the process *PktSender(x,i)* will forward the packet via channel *SH[1]*.

Then, the encoding of the processes *MovingHost(u,x,i,y,j)* and *Migration(u,x,i,y,j)* is shown as follows:

$$\begin{aligned} MovingHost(u, x, i, y, j) = ((&UnmovingHost\_1(u, x); Migration(u, x, i, y, j)) \\ &[](SH[u]?pkt \rightarrow Migration(u, x, i, y, j)) \\ &[](Migration(u, x, i, y, j))); UnmovingHost(u, y, j) \end{aligned}$$

[1] The full PAT programs can be found in https://github.com/Shuang-x/PAT-TopoDiscovery.git.

Table 4
Verification results.

	Fake_Link	Fake_Host	Migration_Finish
System	Invalid	Valid	Invalid
NewSystem	Invalid	Invalid	Valid

$$Migration(u,x,i,y,i) = PtChange!PtStatus.PtDown.x.i \rightarrow$$
$$move\{HostLoc[u][0] = y; HostLoc[u][1] = j;\}$$
$$\rightarrow SP[y*(PNum-1)+j-1]!ARP.u \rightarrow Skip$$

As we can see, the PAT programs *MovingHost(u,x,i,y,j)* and *Migration(u,x,i,y,j)* are almost the same as the CSP models H_3 and H_4. In PAT, an action may be attached with a statement block of a sequential program (which may contain local/global variables, if-then-else, while, etc.) [26]. We implement a function in a CSP process as an action with a statement block in PAT. For example, the action *move* assigns the value of y to *HostLoc[u][0]* and the value of j to *HostLoc[u][1]*.

5.3. Assertions

We write three assertions in PAT to answer the three questions.

The first assertion asks whether there is any chance that Link Fabrication Attack will succeed by checking if a fake link from port P_2 on switch S_2 to port P_2 on switch S_0 will be added into the Link Table. $LinkExist[SNum][SNum]$ is a table defined in PAT to record the direction of a link between two switches.

$$\#assert\ System\ reaches\ Fake\_Link;$$
$$\#define\ Fake\_Link\ LinkTable[S_2][S_0][0] == S_2 \&\& LinkTable[S_2][S_0][1] == P_2;$$
$$\&\& LinkExist[S_0][S_2] == 0;$$

The second assertion shown as below asks if the location of the target host H_1 will be set to (S_0, P_2) in some trace, which means it is possible for the host attacker to succeed in deceiving the controller.

$$\#assert\ System\ reaches\ Fake\_Host;$$
$$\#define\ Fake\_Host\ HostTable[H_1][0] == S_0 \&\& HostTable[H_1][1] == P_2;$$

The third assertion checks if host H_1 and host H_2 will finish their migrations normally by asking if *Migration_Finish* will be eventually satisfied in every execution of the process *System*.

$$\#assert\ System\ \models <> Migration\_Finish;$$
$$\#define\ Migration\_Finish\ HostTable[H_1][0] == S_1 \&\& HostTable[H_1][1] == P_2$$
$$HostLoc[H_1][0] == S_1 \&\& HostLoc[H_1][1] == P_2$$
$$HostTable[H_2][0] == S_2 \&\& HostTable[H_2][1] == P_1$$
$$HostLoc[H_2][0] == S_2 \&\& HostLoc[H_1][1] == P_1;$$

5.4. Verification results and analyses

The verification results for the model $System$ are shown in the first row of Table 4.

- The assertion Fake_Link is invalid, which means that the link attacker is not able to succeed.
- The assertion Fake_Host is valid, showing that the host attacker may tamper the location information of host H_1. The witness trace is shown in Fig. 10.
- The assertion Migration_Finish is invalid, and one of the witness traces is shown in Fig. 11, which illustrates that TopoGuard may misjudge a legal migration.

Fig. 10 represents the witness of the assertion Fake_Host. It describes the situation that the host-hijacking attack starts during the migration of the target host H_1. When the fake ARP packet arrives at the controller, a port-down signal of P_2 on S_2 has already been received by the controller and the Shut-Down Flag of the port has been set to TRUE. As a result, the controller will believe that H_1 has left its old location and modify the location of H_1 according to the fake ARP packet. Then, because the target host cannot receive packets or reply any ICMP Echo request during its downtime, the controller will keep the update information. In other words, if the host attacker launches an attack during the downtime of the target host, it will succeed.

In addition, the success of the Host Hijacking Attack will bring some other bad influence. After the attack succeeds, the target host H_1 will inject ARP packets into the network from its new location. However, Shut-Down Flag of P_2 on S_0 is FALSE, so the controller will take the normal ARP packet from H_1 as a fake one and refuse to update the location information of H_1.

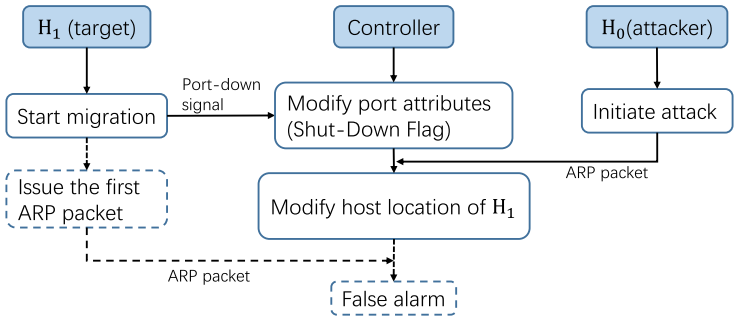


Fig. 10. Witness trace for the assertion *Fake_Host*.

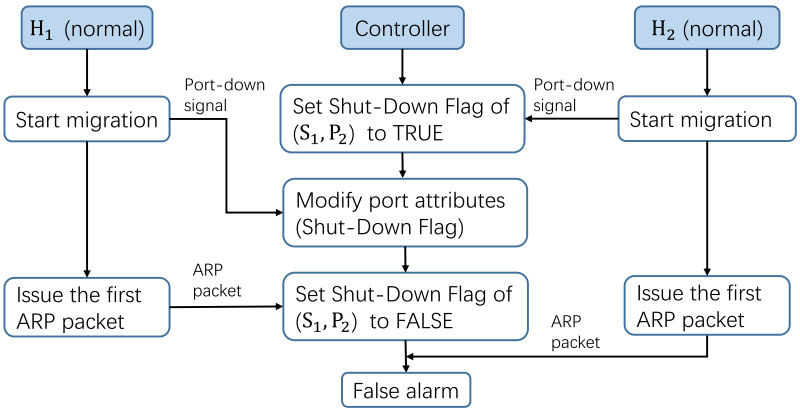


Fig. 11. Witness trace for the assertion *Migration_Finish*.

Fig. 11 is the witness trace of the assertion Migration_Finish. It shows that, before the ARP packet sent by the host H_2 from its new location (S_2,P_1) is detected by the controller, H_1 has finished its migration and the controller has observed the ARP packet generated by H_1 from its new location (S_1,P_2). Therefore, when the controller handles the first ARP packet generated by H_2 from its new location, Shut-Down Flag of P_2 on S_1 has already been set to FALSE. As a result, the controller will raise an alert and refuse to change the location of H_2.

5.5. Applying the verification results to other SDNs

In this section, we show that the verification results still hold for other networks which are extended based on the network shown in Fig. 9. It is composed of two steps. At first, we discuss the networks that have the same topology with Fig. 9 (switches) but own different environments (hosts and attackers). Then, we analyze the verification results for a much larger set of networks with more complex topologies.

5.5.1. Different environments

The intuitive idea of the environment of Fig. 9 is that we need one compromised host (H_0), one target host (H_1) and one auxiliary host (H_2). An attacker can insert fake or repeated packets into the network through the compromised host. The auxiliary host represents any host in the network except for the compromised host or the target host. The analyses for other environments are listed as follows.

- If host H_0 migrates after the attacker injects the packet, the controller will handle the fake or repeated packet before handling messages generated due to H_0's migration. Therefore, the attack will finish before the Shut-down Flag of port P_2 on S_0 is set to TRUE. That is to say, the attribute changes brought by H_0's migration will have no influence on the attack.
- If H_0 moves to another location before the attack starts, then it is equivalent to the situation where the attacker starts the attack from another location and the compromised host will not migrate.
- The two host migrations in Fig. 9 represent two categories of migrations according to the attribute-modifying mechanism of TopoGuard: (1) a host migrates to a port that has not connected to any host. (2) a host moves to a port from which another host just left.
- There is no need to let the two attackers have different locations. Because the host-hijacking attack will not set the Device Type of any port to SWITCH, and the link-fabrication attack will not set the Shut-Down Flag of any port to TRUE.

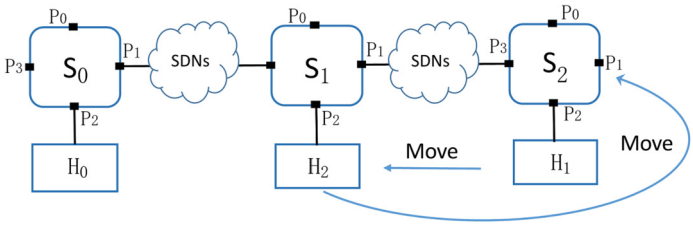


Fig. 12. Network pattern.

- There may be more than one host attacker in the network. In our model, the controller will handle messages in sequence so that the host attackers will not influence each other. As a result, one host attacker is enough, and the analysis for the number of link attackers is the same.

5.5.2. More complex topologies

As shown in Fig. 12, there could be subnetworks that lie between the attacker and the target switch/host. A subnetwork could have arbitrary topology as long as host H_0 can communicate with host H_1 to get enough information for initiating attacks.

To sum up, we come to a conclusion that the success of the two kinds of attacks does not rely on the locations of the attackers and their targets, as long as the attackers are able to gain enough information to initiate attacks. Besides, for more complex networks discussed above, the verification results still hold.

6. An improved mechanism

6.1. Mechanism

The underlying causes of the two problems described in the previous section are as below.

- One vulnerability is that the controller will believe the first packet of a host which has moved from another location.
- Another problem is that there could be a period of time before checking Shut-Down Flag of a migrating host's old location.

An intuitive solution is to strengthen the verification of the new port's attributes and eliminate the previous-port checking step. One key observation in designing TopoGuard is that the controller must receive a port-down signal before the host migration finishes [4]. According to this observation, we draw two conclusions about the networks in which TopoGuard is applied.

- The networks are not hybrid networks that contain both OpenFlow switches and traditional forwarding devices like routers. The reason is that a controller in a hybrid network does not have a global view of all the devices, and as a result, when a port of a traditional device is shut down, the controller may not receive a port-down signal.
- In the networks that have virtual machines, several hosts (physical hosts or virtual machines) may share one physical switch port. In this situation, the controller can control the virtual switches. That is to say, the global view of a controller is built on the virtual topology of the network.

To sum up, in the controller's view, a switch port may connect to at most one host. Then, when the controller notices a host migration, it should make sure that no other host is at the moving host's new location. Therefore, we add an attribute Last-Seen Host for each switch port to record the latest host that has been observed by the controller from the port. There are two situations:

- One is that the host is connecting to the port.
- Another one is that the host is the latest one that was connecting to the port.

If a host-migration event is triggered, then the controller will handle the event following the new mechanism shown in Fig. 13. Assuming that the source host of the detected host-generated packet which triggers the event is H_1. At first, the attribute Last-Seen Host of the new port is checked, and if it is empty, which means that the port has not connected to any host, the controller will record the new location of H_1. Similar as before, an ICMP Echo request will be sent to the previous location of the host. Additionally, Last-Seen Host of the new port will be set to H_1. The controller will also modify other attributes according to the mechanism of TopoGuard. If the attribute Last-Seen Host of the new port denotes a host that is not H_1, then the Shut-Down Flag of the new port should be true, otherwise the host-generated packet is a fake one.

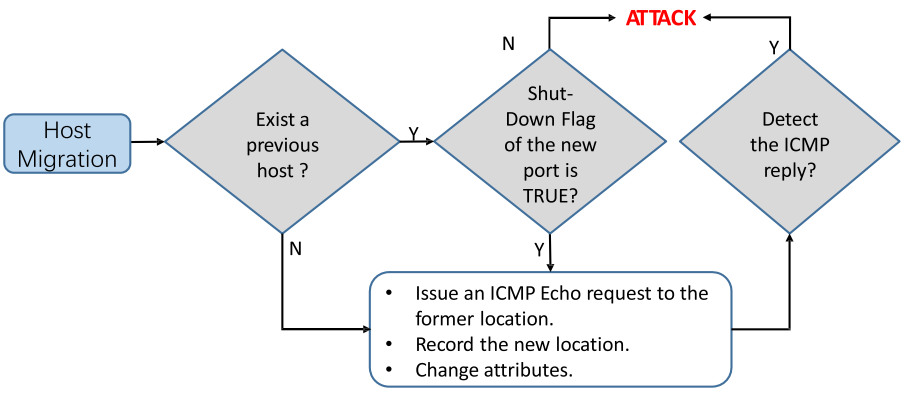


Fig. 13. The new mechanism to handle a host migration.

6.2. New Host Tracker

The main structure of the CSP model for the new Host Tracker, in which our mechanism is modeled, is shown in *C8* and *C9*. The logic inside these two processes is the same as that in the model for the original Host Tracker.

*C*8 describes that once *NewHostTracker* receives a packet-in message that carries a ARP packet, the sub-process *C*9 is invoked.

As described in *C*9, if the host denoted by the ARP packet is detected for the first time ($host\_loc == EMPTY$), and the Device Type of the ingress port is SWITCH ($port\_type == SWITCH$), then an attack alert is issued according to Fig. 4. The process *HandleNewHost* is to create a new host profile for the detected host. If the host is not new, and its location is different from the location recorded in Host Table, a host-migration event is triggered, which is modeled as *C*10.

In *C*10, *flag* stores the value of Shut-Down Flag of the new port, and *last_host* denotes the attribute Last-Seen Host of the new port. If Last-Seen Host is not empty, and the port is not shut down (Shut-Down Flag is FALSE), then an alarm is raised. Otherwise, Host Tracker will follow the steps shown in the bottom box of Fig. 13.

$$
\begin{aligned}
&C8 : NewHostTracker =_{df} Host?msg \rightarrow \\
&\quad (NewHandleARP(msg) \triangleleft PktType[msg] == ARP \triangleright HandleICMPEcho(msg)) \\
&\qquad \triangleleft MsgType[msg] == PktIn \triangleright \\
&\quad (NewHostTracker)
\end{aligned}
$$

$$
\begin{aligned}
&C9 : NewHandleARP(msg) =_{df} \\
&\quad host\_loc = searchHost(SrcHID[msg]); port\_type = getDT(InSID[msg], InPID[msg]); \\
&\quad (attack \rightarrow Skip \triangleleft port\_type == SWITCH \triangleright HandleNewHost(msg)) \\
&\qquad \triangleleft host\_loc == EMPTY \triangleright \\
&\quad (Skip \triangleleft host\_loc[0] == InSID[msg] \wedge host\_loc[1] == InPID[msg] \triangleright \\
&\quad NewHostMigration(msg, host\_loc))
\end{aligned}
$$

$$
\begin{aligned}
&C10 : NewHostMigration(msg, host\_loc) =_{df} \\
&\quad flag = getSDFlag(InSID[msg], InPID[msg]) \\
&\quad last\_host = getLastHost(InSID[msg], InPID[msg]); \\
&\quad ((setLoc(SrcHID[msg], InSID[msg], InPID[msg]); setDT(InSID[msg], InPID[msg], HOST); \\
&\quad addHostList(SrcHID[msg]); setLastHost(InSID[msg], InPID[msg], SrcHID[msg]); \\
&\quad setSDFlag(InSID[msg], InPID[msg], FALSE); \\
&\quad CSend!host\_loc[0].PktOut.host\_loc[1].ICMPEcho.Request.SrcHID[msg] \rightarrow Skip) \\
&\qquad \triangleleft last\_host == EMPTY \vee (last\_host != EMPTY \wedge flag == TRUE) \triangleright \\
&\quad (attack \rightarrow Skip));
\end{aligned}
$$

- *searchHost(SrcHID[msg])* is a function that searches Host Table to find the controller-recorded location of the host with identity *SrcHID[msg]*. The result is stored in the global variable *host_loc*, which is a tuple where the first element represents the switch identity and the second element denotes the port identity. If there is no record for the host, then *host_loc* is equal to EMPTY, which is an enumeration value.
- *getSDFlag(InSID[msg],InPID[msg])* will return the value of Shut-Down Flag of the port with identity *InPID[msg]* on the switch with identity *InSID[msg]*. If the value is TRUE, it means that this port has been shut down.

- *getLastHost(InSID[msg],InPID[msg])* will return the value of the attribute Last-Seen Host of the port designated by *InSID[msg]* together with *InPID[msg]*.
- *setLoc(SrcHID[msg],InSID[msg],InPID[msg])* modifies the controller-recorded location of the host with identity *SrcHID[msg]* to (*InSID[msg], InPID[msg]*). Besides, the functions *setDT(···)*, *addHostList(···)* and *setLastHost(···)* modify the attributes Device Type, Host List and Last-Seen Host respectively.

6.3. New system model and verification results

In the new system model, the process *NewController* is generated by replacing the sub-process *HostTracker* of *Controller* with the process *NewHostTracker*.

$$
\begin{aligned}
NewSystem =_{df}\ & (||_{x\in\{0..2\}} Switch_x)||NewController \\
& ||(Migration(H_2, S_1, P_2, S_2, P_1); \\
& \quad (MovingHostM(H_1, S_2, P_2, S_1, P_2)|||UnmovingHost(H_2, S_2, P_1))) \\
& ||UnmovingHost(H_0, S_0, P_2) \\
& ||(HostAttacker(S_0, P_2, H_0, H_1)|||LinkAttacker(S_0, P_2, H_0, H_1))
\end{aligned}
$$

The definitions of the assertions for the new system model are as same as those given in the previous section where we verify TopoGuard, and therefore we omit the repetitive parts.

$$
\begin{aligned}
& \#assert\ NewSystem\ reaches\ Fake\_Link; \\
& \cdots \\
& \#assert\ NewSystem\ reaches\ Fake\_Host; \\
& \cdots \\
& \#assert\ NewSystem\ \models <>\ Migration\_Finish; \\
& \cdots
\end{aligned}
$$

The results of the verifications are shown in the second row of Table 4, from which we can see that the new system model is able to prevent Link Fabrication Attack as well as Host Hijacking Attack, and the improvement scheme we propose will not hinder legal host migrations.

7. Conclusion and future work

We have built a parameterized framework that captures the basic structure of SDNs, including a modularized control plane, a general date plane with interfaces connected to the environment (hosts and attackers), and the communications between the control and data planes. The general framework can be extended to study controller functionalities and some data plane properties. For example, one may add a new functional module in the control plane to study some related properties by observing the states or behaviors of the network.

Link Fabrication Attack and Host Hijacking Attack are two topology poisoning attacks that are unique to SDNs. The two attacks may lead to serious hijacking, denial of service or man-in-the-middle attacks. TopoGuard, a secure extension to OF controllers, provides a topology discovery mechanism to prevent the two attacks. However, it lacks of formal verification, especially in the situation where some hosts are migrating.

In order to study the secure topology discovery mechanism that is designed to prevent the two kinds of attacks described above, we have built an instance of the framework, added two modules in the control plane, and designed the network shown in Fig. 9 to create a specific system model. Through analyses, we have shown that the specific network used in verification represents a much larger set of networks. After implementing and verifying the system model into PAT, two new loopholes of TopoGuard were found. Besides, we have shown that the success of the two kinds of attacks does not rely on the locations of the attackers and their targets, as long as the attackers are able to gain enough information to initiate attacks. In addition, the verification results still hold for more complex networks with the pattern shown in Fig. 12.

The topology discovery mechanism we proposed based on TopoGuard is a new countermeasure to prevent the two kinds of attacks. The improved system model (with the new mechanism) has been implemented and verified in PAT.

Undeniably, formal modeling and verifying is an effective method to study security problems of computer networks. In the future, we will study the computer networks consisting of traditional networks and software-defined networks. Topology discovery algorithms for these hybrid networks are notoriously difficult and very sensible to attackers, and as such formal methods can be of help.

Acknowledgement

This work was partly supported by National Key Research and Development Program of China (Grant No. 2018YFB2101300), National Natural Science Foundation of China (Grant No. 61872145), Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (Grant No. ZF1213) and the Fundamental Research Funds for the Central Universities of China.

References

[1] Shuangqing Xiang, Huibiao Zhu, Lili Xiao, Wanling Xie, Modeling and verifying topoguard in openflow-based software defined networks, in: 2018 International Symposium on Theoretical Aspects of Software Engineering, TASE 2018, Guangzhou, China, August 29-31, 2018, pp. 84–91.

[2] Diego Kreutz, Fernando M.V. Ramos, Paulo Jorge Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, Steve Uhlig, Software-defined networking: a comprehensive survey, Proc. IEEE 103 (1) (2015) 14–76.

[3] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru M. Parulkar, Larry L. Peterson, Jennifer Rexford, Scott Shenker, Jonathan S. Turner, Openflow: enabling innovation in campus networks, Comput. Commun. Rev. 38 (2) (2008) 69–74.

[4] Sungmin Hong, Lei Xu, Haopei Wang, Guofei Gu, Poisoning network visibility in software-defined networks: new attacks and countermeasures, in: 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015.

[5] Laurent Vanbever, Joshua Reich Theophilus Benson, Nate Foster, Jennifer Rexford, Hotswap: correct and efficient controller upgrades for software-defined networks, in: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013, The Chinese University of Hong Kong, Hong Kong, China, Friday, August 16, 2013, pp. 133–138.

[6] Suleman Khan, Abdullah Gani, Ainuddin Wahid Abdul Wahab, Mohsen Guizani, Muhammad Khurram Khan, Topology discovery in software defined networks: threats, taxonomy, and state-of-the-art, IEEE Commun. Surv. Tutor. 19 (1) (2017) 303–324.

[7] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, Vijay Mann, SPHINX: detecting security attacks in software-defined networks, in: 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015.

[8] Talal Alharbi, Marius Portmann, Farzaneh Pakzad, The (in)security of topology discovery in software defined networks, in: 40th IEEE Conference on Local Computer Networks, LCN 2015, Clearwater Beach, FL, USA, October 26-29, 2015, pp. 502–505.

[9] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.

[10] A.W. Roscoe, Understanding Concurrent Systems, Texts in Computer Science, Springer, 2010.

[11] Gavin Lowe, A.W. Roscoe, Using CSP to detect errors in the TMN protocol, IEEE Trans. Softw. Eng. 23 (10) (1997) 659–669.

[12] Xi Wu, Huibiao Zhu, Formalization and analysis of the REST architecture from the process algebra perspective, Future Gener. Comput. Syst. 56 (2016) 153–168.

[13] Yuan Fei, Huibiao Zhu, Xi Wu, Huixing Fang, Shengchao Qin, Comparative modelling and verification of pthreads and dthreads, J. Softw. Evol. Process 30 (3) (2018).

[14] Ramtin Aryan, Anis Yazidi, Paal Einar Engelstad, Øivind Kure, A general formalism for defining and detecting openflow rule anomalies, in: 42nd IEEE Conference on Local Computer Networks, LCN 2017, Singapore, October 9-12, 2017.

[15] Lucas Freire, Miguel C. Neves, Lucas Leal, Kirill Levchenko, Alberto E. Schaeffer Filho, Marinho P. Barcellos, Uncovering bugs in P4 programs with assertion-based verification, in: Proceedings of the Symposium on SDN Research, SOSR 2018, Los Angeles, CA, USA, March 28-29, 2018, pp. 4:1–4:7.

[16] Elvira Albert, Miguel Gómez-Zamalloa, Albert Rubio, Matteo Sammartino, Alexandra Silva, Sdn-actors: modeling and verification of SDN programs, in: Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Proceedings, Oxford, UK, July 15-17, 2018, pp. 550–567.

[17] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, Philip Brighten, Godfrey Veriflow, Verifying network-wide invariants in real time, in: Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013, pp. 15–27.

[18] Sooel Son, Seungwon Shin, Vinod Yegneswaran, Phillip Porras, Guofei Gu, Model checking invariant security properties in OpenFlow, 06 2013, pp. 1974–1979.

[19] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, David Walker, Netkat: semantic foundations for networks, in: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014, pp. 113–126.

[20] Arjun Guha, Mark Reitblatt, Nate Foster, Machine-verified network controllers, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16–19, 2013, pp. 483–494.

[21] Marco Canini, Daniele Venzano, Peter Peresíni, Dejan Kostic, Jennifer Rexford, A NICE way to test openflow applications, in: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012, pp. 127–140.

[22] Tim Nelson, Andrew D. Ferguson, Michael J.G. Scheer, Shriram Krishnamurthi, Tierless programming and reasoning for software-defined networks, in: Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014, pp. 519–531.

[23] Jackson Daniel, Software Abstractions - Logic, Language, and Analysis, MIT Press, 2006.

[24] The ONF is an operator led consortium, https://www.opennetworking.org/. (Accessed 12December2018) (Online).

[25] Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, Étienne André, Modeling and verifying hierarchical real-time systems using stateful timed CSP, ACM Trans. Softw. Eng. Methodol. 22 (1) (2013) 3.

[26] Jun Sun, Yang Liu, Jin Song Dong, Jun Pang, PAT: towards flexible verification under fairness, in: Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, pp. 709–714,.

[27] Yuanjie Si, Jun Sun, Yang Liu, Jin Song Dong, Jun Pang, Shao Jie Zhang, Xiaohu Yang, Model checking with fairness assumptions using PAT, Front. Comput. Sci. 8 (1) (2014) 1–16.

[28] Farzaneh Pakzad, Marius Portmann, Wee Lum Tan, Jadwiga Indulska, Efficient topology discovery in openflow-based software defined networks, Comput. Commun. 77 (2016) 52–61.

[29] IEEE standard for local and metropolitan area networks - station and media access control connectivity discovery, in: IEEE Std 802.1AB-2016 (revision of IEEE Std 802.1AB-2009), March 2016, pp. 1–146.

[30] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, Hrishikesh B. Acharya, Kyriakos Zarifis, Scott Shenker, Troubleshooting blackbox SDN control software with minimal causal sequences, in: ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014, pp. 395–406.