Towards Automated Attack Discovery in SDN Controllers Through Formal Verification

Bin Yuan, *Member, IEEE,* Chi Zhang, Jiajun Ren, Qunjinming Chen, Biang Xu, Qiankun Zhang, Zhen Li, Deqing Zou, Fan Zhang, and Hai Jin, *Fellow, IEEE*

***Abstract*—Software-defined Network (SDN), presented to be a novel architecture of network because of its separation of data plane and control plane, brings centralization and extensibility to network management as well as new attacks that exploit the flexibility of SDN. OpenFlow, which is the protocol that is applied by the majority of SDN, leads to the widely used definition of the communication between the controller and the switch resulting in similar implementations regardless of different vendors. In this paper, we focus on the mechanisms of packet processing and topology discovery and their fundamental weaknesses caused by general implementations or device limitations. Despite the common vulnerabilities, the universal standard mechanisms of basic function in SDN also enlighten us to present an automated attack discovery method based on the formal verification with a generic model of SDN system. We describe the abstraction of the SDN components, their key functions, and communications along with the malicious operations that could be executed by malicious hosts and malicious switches and translate them into a formal model of the SDN system. The formal verification carried on with the assertion representing the security properties derived from the common vulnerabilities of the SDN system reports the potential attack paths each of which shows an attack process. Our evaluation shows that our method can discover feasible attack paths efficiently and effectively, with 23 attacks being identified, among which 2 are new. We further demonstrate the practicality of the 2 new attacks.**

***Index Terms*—SDN security, network security, model checking.**

I. Introduction

SOFTWARE-DEFINED Network based on the separation of control plane and data plane is a prevailing network architect of data center at present, which provide a more simple and centralized approach [1]. The centralized control plane comes up with the forwarding table based on the knowledge of the whole network which is real-time maintained during the communication with switches and the consistent global controlling strategy defined by the network administrator, and then issues the flow entries directing packet processing to switches, which simplifies the main function of switches to general packet forwarding. Therefore, SDN can provide flexible, dynamic, and programmable management and allow the network administrator to perform efficient resource strategy, traffic control, and global monitoring.

The communication between the controller and switch is carried through the SDN southbound protocol. OpenFlow [2], which is the de facto general-purpose protocol for SDN southbound interface, provides a series of standards for control plane operations that the controller adds, removes and modifies the flow entries installed in switches, while also defines the data plane behavior about packet matching and forwarding guided by flow entries [3]. However, it is important to note that OpenFlow does not explicitly define how the controller itself operates; this aspect is left to be specified by individual controller vendors. Generally, the SDN controller should include a set of common application modules to support the core functions of controller, such as topology discovery and management and flow management [4]. In terms of the topology discovery, most of the mainstream controllers adopt OpenFlow Discovery Protocol (OFDP) [5], which is tamed based on Link Layer Discovery Protocol (LLDP) to be exploited in SDN diagram, despite the vendors. All the information about the current network state is maintained through the communication between the controller and other network devices, including SDN switches and end host, and stored in controller database in order to support the application through controller's northbound API.

The benefits of SDN come with exploitable weaknesses. The flexibility of SDN, which comes from the novel structure separating the network control and packet forwarding, brings new attacks and may lead to a wider damage impact because of the centralized control plane. The controller becomes literally the single point of failure which means any sabotage and deception of controller are most likely to affect the whole network and controller can be exploited to attack other SDN devices. To make it worse, although various vendors provide numerous choices of SDN controllers, their core functions of packet process and network maintenance share the same logic pattern, which makes the SDN attack effective to different controllers regardless of their nuances. For example, the attacks aiming at the network topology view of controller

Bin Yuan, Chi Zhang, Jiajun Ren, Qunjinming Chen, Biang Xu, Zhen Li, Fan Zhang, and Deqing Zou are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China.

Bin Yuan, Zhen Li and Deqing Zou are also with Jinyinhu Laboratory, Wuhan, 430040, China.

Bin Yuan is also with Songshan Laboratory, Zhengzhou, 452470, China.

Qiankun Zhang is with the Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China. Qiankun Zhang is the corresponding author. E-mail: qiankun@hust.edu.cn.

Hai Jin is with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.

are definitely going to affect the correctness of calculation generating flow entries and further jeopardize the packet forwarding in the overall network, which makes the majority of controllers suffer. Also, most of commercial hardware SDN switches use ternary content-addressable memory (TCAM), which is known as size-limited and power-costly and leads to a limitation of flow-table space [6]. Therefore, it is easy to trigger a DoS attack against flow-table space by exploiting the packet process mechanism of the OpenFlow-based controller [7], [8].

The security issues involving SDN controllers have already drawn attention of many prior works. There are plenty of works that discovery feasible attacks focusing on certain vulnerabilities, such as topology discovery poisoning [9]–[14], resource exhaustion against SDN devices [14]–[17] and malicious application [18], [19]. Some of the attack discovery works are through exhaustive code reading and manual testing [9], while some analysis tools have been developed to analyze SDN system and effectively detect vulnerabilities based on blind fuzzing [20], model checking [21], [22] and real-time verification [14]. Nevertheless, there are several kinds of defense proposed whether by improving existing controllers [10], [11], [17], [23] or by developing new controller [18].

From massively and carefully researching the attack on SDN, we realized that there are certain exploitable points that are fatal to all kinds of SDN systems regardless of the vendors and shared by different controllers. Although the controllers may vary in specific implementations, their underlying designs are consistent, especially the fundamental functions, which increases the probability that one feasible attack against one type of controller is very likely to transfer against another.

In this paper, we propose a formal verification method to automatically discover attacks in SDN controller. Based on the awareness of the commonalities of SDN controllers, we have done a thorough investigation of the OpenFlow protocol, the implementation of mainstream controllers and SDN vulnerabilities. With the prior knowledge, we can extract an abstraction about the core components, their key functions and the critical communications in the SDN system. From the existing attack, we can also summarize the key properties that can be a disaster if violated, which are attackers' favorites. Then we present a formal description of SDN system defining the behavior of different roles in the system, such as malicious and innocent components, and their interactions as the system model in order to analyze the possible attack against SDN controller through formal verification with the model checking. Later, we evaluate the counterexamples obtained during the analysis and filter out the ones that can be carried out as practical attacks. Finally, the feasibility of these attack paths is verified by returning to the real SDN system. Our contributions are summarized as follows:

- We suggest that there are commonalities of different SDN controllers regardless of the vendors through doing comprehensive research about the attacks and defenses in SDN and examining the vulnerabilities of SDN protocol.
- We present an automated attack discovery method through formal verification based on knowledge of SDN mechanism and exploitability to instruct the practice of attacks against SDN controllers and analyze SDN security.
- We exploit our formal model with model checking and derive counterexamples which lead to exploitable ways to launch attacks on SDN controllers.
- We conduct practical experiments to validate the feasibility of new attack ways under the guidance of the analysis of counterexamples. The attacks found by our method can be verified on different controllers and different OpenFlow versions.

The structure of the following paper carries out as follows. Section II introduces SDN and summarizes the mechanisms and weaknesses of packet processing and topology discovery. In addition, we would like to discuss the problems in the existing formal validation work and the motivation for our work. Section III depict the design of our method while Section IV specifies the implementation of formal verification. Section V shows the result of our experiments and discusses the attack paths discovered by our method along with the demonstration of new attacks. We discuss the features and limitations of our method in Section VI and other related works in Section VII while concluding the paper in Section VIII.

II. Background and Motivation

In this section, we briefly introduce the overview of SDN and focus on its function in packet processing and topology discovery. We study their mechanism to conclude the weaknesses and the simple approaches to exploit the vulnerabilities. Further, we summarize the existing work on formal verification of SDN controllers to show why we model the universal mechanisms for SDN controllers.

A. SDN

The most significant difference between SDN and traditional network structure is the separation of the data plane and the control plane, where the data plane is responsible for packet forwarding as directed by flow entries and the control plane maintains the entire network view to determine and direct packet forwarding strategy.

The separation of planes leads to a highly centralized controller and simplified data plane devices. The SDN controller, as the decision maker and core of the system, has to maintain a whole view of the network, manages all the network devices and gives instruction about the packet processing according to southbound protocol, generally OpenFlow, and also provides applications with network devices and topology to facilitate more flexible and versatile management on demand for network administer through northbound API. Meanwhile, the SDN device, switches, refreshes and maintains a flow table filled with flow entries that instruct the packet processing with match field, action set deciding whether a packet should match the entry and how to deal with the packet and also handles packet forwarding. The overview of SDN structure is featured as Fig. 1.

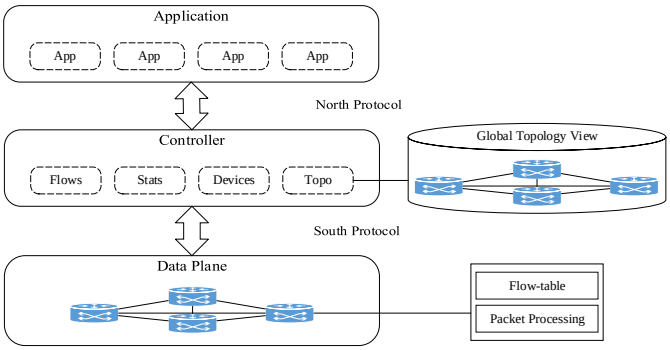


Fig. 1: SDN structure

B. Packet Processing in SDN

1) Mechanism: Basically, the controller works in a reactive way triggered by events, receives packet-in messages, and processes the packets contained in that message. To be more specific, the packets sent by an end host first arrive at the edge switch connecting to the end host. Then, in terms of OpenFlow-based SDN, the edge switch looks into its flow tables and determines how to deal with the packet. If any certain flow entry matches the packet and has the highest priority, then the switch obeys its action to process and forward the packet. Otherwise, the switch sends a packet-in message carrying the related information or even the whole packet to the controller based on the preinstalled default table miss flow entry. Then the controller makes the decision and sends a packet-out message specifying the packet processing to the switch. Sometimes the controller also installs the related flow entry using flow-mod messages to simplify the process steps of the later packet of the same flow. The processing is illustrated in Fig. 2.

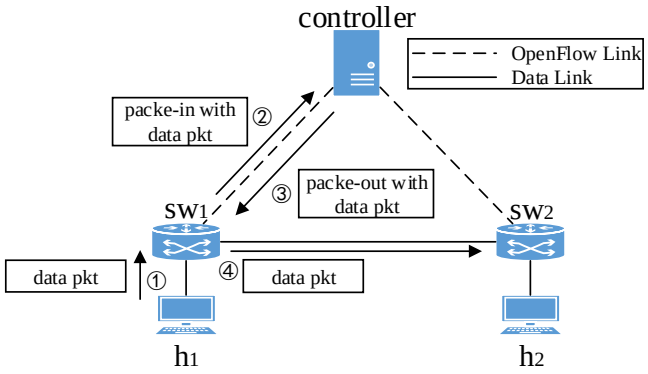


Fig. 2: The mechanism of packet processing

2) Weakness: The reactive handling approach can be abused to launch DoS attacks against the control plane [24]. Attackers can exploit the mechanism that the data plane has to request the control plane for new arrival packets to flood the control plane with requests in a short period. Eventually the number of requests exceeds the processing capacity of the controller so that the controller is unable to handle the normal flow of the normal components in the network. The requirement for attackers that adopt this way can be just an end host in the SDN network, which is readily available. Moreover, this kind of attack can lead to a further flood of responses from the control plane to the data plane. As we mentioned before, the SDN switches usually lack flow table capacity so the newly installed flow entries can cause SDN switches exhausted.

3) Example: A controller DoS can be launched by an attacker that compromised an end host in the SDN network. The attack can forge a large number of data packets with different IP and MAC addresses and send them quickly in a short period to force the edge switch to send packet-in messages to request the controller about how to forward the mismatching packets. To further make an impact on the switch's flow table, the forged packets can use random source IP and MAC addresses along with destination IP and MAC addresses that are known to exist in the network. A known destination address is chosen instead of a random destination address to induce the controller to send flow-mod messages and install new flow entries.

C. Topology Discovery in SDN

1) Mechanism: The SDN controller needs to preserve and update the network state to maintain and direct the network operations, particularly has the updated topology view about the information of links and devices to make decisions on routing.

Universally, the controllers adopt OFDP as the mechanism of link discovery in SDN, which carries out with adapted LLDP. The controller discovers switches through the TLS/TCP connection and learns the switches' configuration with the feature-request and feature-reply messages exchange. Then, the controller sends packet-out messages to all corresponding switches with LLDP packets containing unique chassis ID and port ID consistent with the switch's Datapath Identifier (DPID) and Port Number (Port No). After receiving the LLDP packets, switches forward the packet out of the certain port with a destination MAC as LLDP multicast address, which is meant to be forwarded to the nearest bridge with a single hop if the port does link to another switch. The other switch that receives the LLDP packet will fill the packet-in message. The IN_PORT field of the packet-in message is the port on which the LLDP packet was received, and the data field is the complete LLDP packet. Further, the switch forwards them to the controller according to a preinstalled flow entry that indicates all received LLDP packets to be output to the controller. Consequently, the controller processes the packet-in message and derives a unidirectional link from the port of the port ID on the chassis ID switch to the port of the IN_PORT field on the switch that generates the packet-in message. The processing is illustrated in Fig. 3.

The host tracking service is responsible for the host information, which is updated using the source host information in the data field of the latest packet-in message. Specifically, the controller records the source IP and MAC addresses and associates them with IN_PORT and DPID of the packet-in message. Commonly, the host information's index is MAC address. So that when packet-in messages arrive, the controller updates the IP address and location if the MAC address exists, otherwise creates a new host entry.

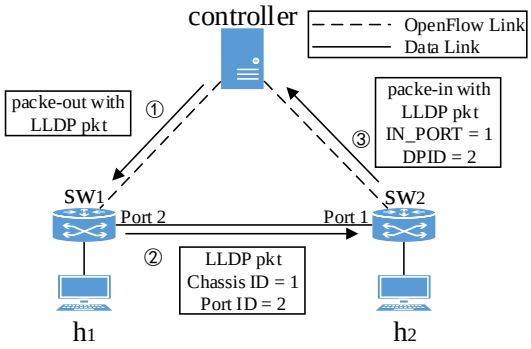


Fig. 3: The mechanism of link discovery

2) Weakness: In summary, both link discovery and host tracking are conducted by packet-in messages sent by switches from the data plane. The controller fully accepts the messages from the switches that conform to the OpenFlow protocol and uses the contents to learn the network topology, which means the mechanism is completely performed on the premise of the fidelity of data plane devices. The controller is so gullible that it can be easily deceived by a compromised switch or host with modified messages to poison the view of topology in the controller.

3) Example: The attacker with a compromised host can readily launch a topology attack with fake LLDP packets and data packets. In case of link fabrication, the attacker can tamper with the LLDP packets containing fake Chassis ID and fake Port ID and send them to its edge switch to trick the switch to send the packet-in messages to the controller. Then the reckless controller will recognize a new link between the fake port and the edge port of the compromised host. Meanwhile, it is rather simple to fake the host location. The attacker can ordinarily send a forged data packet with fake source IP and MAC address and achieve its intention. The false host information will be continuously maintained in the controller until another host starts to compete for the same address.

D. Motivation

In previous studies, formal verification has been used to validate SDN controllers. These works have modeled the components in SDN networks using different modeling languages and validated the models using different model checking tools. Which components of the SDN system are modeled by these work, and the modeling language and Verification tools used are shown in the Table I.

Based on existing formal verification work, the main problems with formal verification of SDN can be classified as follows:

Controller modeling mostly focused on applications. For SDN systems, the modeling usually includes SDN components such as controllers, switches, hosts and OpenFlow Channels. However, existing work tends to concentrate on the applications running on the controller. Most of the modeled applications, such as Pyswitch and Firewall, are deployed in the old generation of controllers (e.g, NOX [29]).

TABLE I: Formal verification related work for SDN systems

related work	Modeling Components	Modeling Languages	Verification Tools
NICE [21]	application, switches, hosts.	python	NICE
FlowLog [25]	application	FlowLog	SPIN
sdnverify [26]	application, OpenFlow Channel, switches.	Murphi	CMurphi
VeriCon [27]	application	Core SDN (CSDN)	Z3
Kuai [28]	controller events, switch events, topology.	Murphi	PReach

Moreover, some researchers propose unique modeling languages, such as FlowLog [25] and VeriCon [27]. These initiatives often encourage the SDN community to adopt their specific modeling languages (e.g., FlowLog and Core SDN) for developing SDN controller applications, with the intent that these applications could then be verified directly, without the need for additional modeling efforts. However, in practice, the SDN controller community tends to favor general-purpose programming languages like Java [30] [31] or Python [32] for application development.

Therefore, to validate an application running on an SDN controller using these formal verification tools, they must first translate the application, originally written in a general-purpose programming language (like Python), into a formal language model (like FlowLog). Moreover, this model must accurately represent the behavior of the SDN controller application to effectively leverage these tools for verifying the application's correctness.

Tools are mostly limited to a single controller platform. The existing controllers are not limited to the NOX controller platform only as described in NICE. NICE is mainly for NOX controllers using Python to write controller programs. If we want to extend NICE to ONOS or OpenDaylight, there are significant modifications that need to be made to NICE. This problem exists for other works as well. They also adapt to only a portion of the applications of a single controller.

Difficulty in defining security properties and network invariants. These tools all require the definition of properties or network invariants that need to be verified. However, the definition of these properties is not simple, and common properties are not fully adapted to specific applications. Therefore for each application, it also needs to be defined manually by the validator himself according to his understanding of the application. Improper property definitions can affect the correctness of the verification results.

Verify correctness of implementation rather than finding bugs. The current use of these tools should be for normal SDN network scenarios. In the absence of malicious SDN components, whether the controller application can work properly as expected by the developer. Lack of consideration of malicious

components in SDN networks, in these tool use cases found mostly problems in the implementation of applications running on the controller.

After nearly a decade of development, the architecture of SDN controllers has become more complex. Applications running on the controllers also have more complex functionality. However, we note that although different controllers use different languages and different architectures in their implementations, there are still some universal mechanisms that are retained, such as link discovery. As a result, security issues can sometimes be triggered in different controllers. In addition, the network components of the data plane are not always secure, and most of the existing formal verification work does not take them into account in the modeling.

Therefore, we model the universal standard mechanisms implemented for SDN controllers and consider malicious switches and malicious hosts in the network. Our method is not limited to the implementation of a specific application on the controller when modeling for the controller. Instead, we focus on the generic mechanisms of the controller, such as packet processing and topology discovery as described in Section II-B and Section II-C. Modeling universal mechanisms allows our method to work better with different controllers and versions of OpenFlow. We also no longer need to model and define security properties for each specific application. We focus more on the security issues that exist with generic mechanisms in the presence of malicious components and are less concerned with the correctness of specific mechanisms in their implementation. We describe our method in detail in Section III.

E. Spin and the Promela Language

Spin is a widely recognized model checking tool [33], predominantly utilized for validating distributed systems and protocols. Promela [34], a formal modeling language, is integral to Spin, facilitating the verification of target programs. Spin's utility in revealing security issues across various systems is well-established. Prior research has applied Spin to diverse protocols and systems, including network protocol state machines [35], IoT clouds [36], smart contracts [37], and routing protocols [38], among others.

Given that SDN networks can be viewed as distributed systems comprising various network nodes like switches, controllers, and hosts, and also as protocol systems communicating through southbound protocols (e.g., OpenFlow, OVSDB [39], NETCONF [40]), the interaction between the SDN control plane and data plane can be effectively modeled using Spin and Promela. Hence, employing Promela to construct an SDN system model and using the Spin tool for its verification is a fitting approach. Specifically, we employ the Promela language to create models of SDN systems and define the security properties these systems should adhere to. Spin processes these models and security properties to report attacks discovered.

III. System Modeling and Formal Verification

In this section, we outline the design of SDN system model in a formal approach and subsequently conduct formal verification with security properties by automated model checking.

A. Overview

To detect potential attacks and security vulnerabilities of the SDN system in the real world, we propose an approach that involves formally modeling the operations of SDN components and controllers during network operations. This modeling represents the transitions of states within the system. Subsequently, we employ model checking to verify whether the state transitions during model execution adhere to predefined security properties. The model checking tool systematically examines the transitions, reporting counterexamples when any lead the SDN model system into states that violate the specified security properties. These counterexamples serve as valuable guides to validate the existence of attacks within real-world SDN systems.

Notably, despite differences in OpenFlow versions and variations in specific implementations by different vendors, SDN controllers generally share a consistent basic functional logic, particularly in basic mechanisms such as packet processing and topology discovery. We can portray one abstract model of the actions of the SDN controller to describe the mainstream controllers.

Our method for attack discovery comprises three key components: the formal description of the SDN system model, model checking, and the subsequent analysis and validation of counterexamples, as depicted in Figure 4.

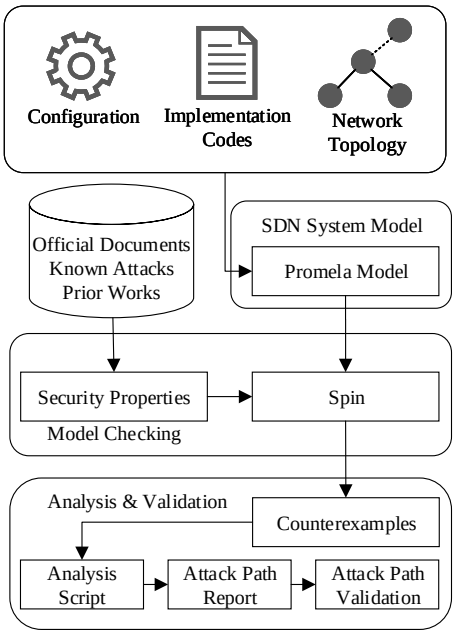


Fig. 4: The architecture of our method

We utilize the Promela language [34] to implement our model, enabling verification through the Spin off-the-shelf model checking tool [33]. This formal verification process aims to derive attack paths to assist in evaluating and increasing the security of the SDN system.

Our model's design is informed by a comprehensive exploration of the OpenFlow protocol, various controller implemen-

tations, and official documents. Simultaneously, our model incorporates a configuration file to allow the customization of the network topology and parameters. This ensures that our model can more accurately represent real SDN networks. Additional details regarding the model are displayed in Section III-D. The security properties detailed in Section III-E play a crucial role in Spin's formal verification. These security properties are derived from a comprehensive review of known attacks and prior research. Once both the model and security properties are prepared, running Spin generates counterexamples. Each counterexample represents a violation of a security property and, consequently, indicates a potential attack. Subsequently, we conduct a comprehensive analysis to comprehend and summarize the attack steps depicted in these counterexamples. These attack path reports provide a detailed illustration of the operations conducted by network components at each step, leading to a state that violates the security properties. To verify the existence of the attack in a real SDN network, we can simply follow each step of these attack path reports and confirm that the corresponding security properties are compromised in the real SDN network.

To illustrate our approach to discovering attacks in SDN, we use the example of a controller Denial-of-Service (DoS) attack. In our constructed model, which represents a universal mechanism for an SDN system, the end host, switch, and controller are depicted in the SDN architecture using the Promela language. These components are represented as sets of variables that capture their states and interactions. The model includes security properties, particularly focusing on the processing capabilities of the controller, as detailed in Section III-E. The attack path report reveals that an attacker-controlled host can initiate a controller DoS attack by flooding the system with numerous data packets, each with a distinct source address. This scenario is illustrated in Fig. 5. Since carefully crafted packets typically do not match existing flow entries, SDN switches forward them to the controller for processing. Through our analysis, we have successfully implemented this attack on SDN controllers lacking adequate protection against DoS attacks in a simulated SDN network environment using Mininet. During such attacks, the controller's processing capacity is heavily strained, leading to a significant reduction in the efficiency of processing normal network traffic. Therefore, identifying various attack paths in the system model via model checking can effectively uncover broader security issues prevalent in SDN controllers.

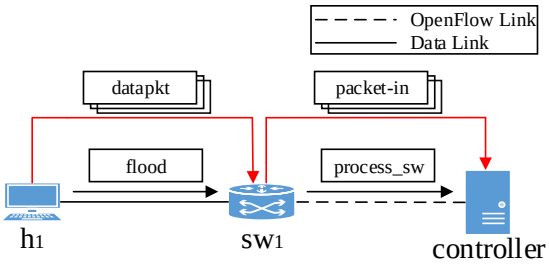


Fig. 5: Malicious host floods data packets and attacks the controller

B. Threat Model

We consider attackers to be inside the SDN since we intend to discover the specialized attacks against SDN and focus on the attacks poisoning or exploiting SDN controllers that can be launched by SDN switches and hosts.

It is rational to assume that attackers are capable of compromising some of the switches and end hosts in the network since the SDN switches are usually running in default configurations and outdated states, rendering them susceptible to compromise [41]. Besides, we trust the controller to be innocent even though it can exploit the weakness of its mechanism indirectly but cannot be taken over by the attacker directly. Meanwhile, we trust the communication between the network components, which means the man-in-the-middle attack is not taken into consideration and all the misconducts like tampering, forging and injecting are carried out by malicious participants in the data plane of the network.

Specifically, the benign components within the network are deemed capable of correctly transferring and processing data packets, aligning with the topology view in the controller that corresponds to the actual network state. The malignant ones can perform malicious operations above the normal behavior intending to affect the SDN controller and do harm to packet processing and topology discovery. We consider the actions that are feasible to execute in a real SDN network and might be able to impact on SDN controller's recognition of the network state or its physical capabilities and take advantage of the weaknesses of the controller. For a malicious host, it can send forged data packets with tampered source addresses, inject forged LLDP packets or send a load of data packets in a short term of time. For a malicious switch, it can tamper with the key fields of its received packets or drop the packets.

C. Problem Scope

In this study, our objective is to identify security vulnerabilities in SDN controllers, particularly those associated with packet processing and topology discovery, and to delineate potential attack paths. This focus is essential for comprehending and mitigating potential vulnerabilities and attack vectors, thereby enhancing SDN system security. Note that, conducting root cause analysis of these attacks, a task orthogonal to our primary focus, falls outside the scope of our current research.

D. Modeling SDN System

We present the model of the SDN system as a tuple of a state automaton $M = (S, O, V, T, s_0)$. S is the no-empty finite state set containing all states of the automaton the operations can lead to and the initial state s_0 ($s_0 \in S$) which represents that no operations have taken place in the SDN system. O is a finite set of operations in the SDN system. V is a finite set of variables that describe the situation of all components and the packets they process currently in the SDN. T is a set of transitions consisting of transitions that indicate the system transits from one state to another with an operation.

The definition of the state automaton of SDN system is a highly abstracted description based on our knowledge of

the network operation in SDN diagram. We extract the core operations during packet transmission such as routing and the topology discovery and the base operations that consist of the core operations, i.e., the sending, receiving and processing of packets by each component. In this context, we reduce the variables, for example, the packets are delineated by their key fields, which significantly influence packet processing, while non-key fields are disregarded to mitigate state complexity. A more specific introduction of the automaton is following.

1) Variables: We use a set of variables to describe the situation of components in the network, including the packets and messages that transfer between the SDN devices and controller in the network currently and need to be handled and the view of the controller. Specifically, for each switch and the controller, we define a list of pending packets *Pend* to store the receiving packets waiting to be processed. Similar to the existing work, the list is a FIFO queue and the first arriving packets will be processed earlier. For example, a data packet p is sent by host h_i and later arrives at the edge switch sw_j, which is a `send` operation from h_i with a packet p, and p will be store in $Pend_{sw_j}$ consequently. For another example, the controller *con* processes a packet-in message carried an LLDP packet and then refreshes its topology view, which is a `process_con` operation of *con* that pop out a pending packet from its $Pend_{con}$ causing the controller updates its topology view *ViewTopo* based on the link information from the LLDP packet. The host tracking process in a similar way that updates the host information in topology view *ViewTopo*. In comparison, we maintain a variable to describe the actual topology of the network as *RealTopo*. There are also variables for packet forwarding and routing, such as flow-tables of each switch $FlowTable_{sw_j}$ and mac-to-port table in controller *MacToPort*. The variables of the SDN system model are summarized as Table II.

TABLE II: Summary of variables

Variable	Description
Pend	pending packet lists in each switch and controller
FlowTable	flow-tables in each switches
MacToPort	mac-to-port table of controller
RealMacToPort	correct mac-to-port table
ViewTopo	topology view of controller about hosts and links
RealTopo	actual topology about hosts and links of the network

2) State: A state s_i ($s_i \in S$) consists of a set of variables representing the situation of the SDN system, including variables of each switch (*Pend* and *FlowTable*) and variables of the controller (*Pend*, *MacToPort* and *ViewTopo*). In real SDN systems, the variables are finite, so the states set in the model are always finite. In conclusion, a state can be expressed as Eq. (1) and the initial state s_0 as Eq. (2).

$$\begin{aligned} s_i = & Pend_{con} \cup MacToPort \cup ViewTopo \cup RealTopo \\ & \cup \bigcup_{sw_i} \{Pend_{sw_i}, FlowTabled_{sw_i}\}. \end{aligned} \tag{1}$$

$$s_0 = \emptyset \cup \emptyset \cup \emptyset \cup RealTopo \cup \bigcup_{sw_i} \{\emptyset, \emptyset\}. \tag{2}$$

3) Operation: The operations of SDN system model are relative to OpenFlow messages' and data packets' generating, processing and forwarding, which are summed up as the handling of pending packets in the controller and switches and the handling of packets sending and receiving in end hosts, all of which implies the change of components' own *Pend* and may results in the change of other ones' *Pend* as forwarding or updates of controller's views *ViewTopo* and *MacToPort* as the processing of OpenFlow messages. In addition to the operations of the benign components, we also define the operations of the malicious components in the model, such as forged packets or forged LLDP packets. Table III summarizes all nine kinds of operations in SDN system model.

TABLE III: Summary of operations

Operation	Description
`send`	host sends a normal data packet
`process_sw`	switch processes a pending packet
`process_con`	controller processes a pending message
`send_lldp`	controller sends a LLDP packet packet-out
`forge_datapkt`	host sends a data packet with forged address
`forge_lldppkt`	host sends a forged LLDP packet
`flood`	host floods data packets in a short term of time
`tamper_field`	switch tampers a key field of a pending packet
`drop`	switch drops a pending packet

`send`: A packet p is sent by host h_i and later arrives at the edge switch sw_j, which is defined as Eq. (3).

$$\begin{cases} p := newDataPacket(), \\ Pend_{sw_j} := Pend_{sw_j} \cup \{p\}. \end{cases} \tag{3}$$

`process_sw`: Switch sw_i processes a pending packet and forwards it according to the flow-table, next can be either a switch or a controller, which is defined as Eq. (4).

$$\begin{cases} p := pop(Pend_{sw_i}), \\ \{next, p'\} := proSw(FlowTable_{sw_i}, p), \\ Pend_{next} := Pend_{next} \cup \{p'\}, \\ Pend_{sw_i} := Pend_{sw_i} - \{p\}. \end{cases} \tag{4}$$

`process_con`: Controller processes a pending packet and updates its view of network or make a indication about the packet's routing strategy, which is defined as Eq. (5).

$$\begin{cases} p := pop(Pend_{con}), \\ \{next, rule, topo, m2p, p'\} := \\ \qquad proCon(ViewTopo, MacToPort, p), \\ ViewTopo := ViewTopo \cup \{topo\}, \\ MacToPort := MacToPort \cup \{m2p\}, \\ FlowTable_{next} := FlowTable_{next} \cup \{rule\}, \\ Pend_{next} := Pend_{next} \cup \{p'\}, \\ Pend_{con} := Pend_{con} - \{p\}. \end{cases} \tag{5}$$

`send_lldp`: Controller sends a packet-out message with an LLDP packet to an exact port on the switch, which is defined as Eq. (6).

$$\begin{cases} p := newLLDP(dpid, portid), \\ Pend_{dpid} := Pend_{dpid} \cup \{p\}. \end{cases} \tag{6}$$

forge_datapkt: A packet p with forged source address meant to spoof the controller is sent by host h_i and later arrives at the edge switch sw_j, which is defined as Eq. (7).

$$\begin{cases} p := newForgedDataPacket(fakeAds), \\ Pend_{sw_j} := Pend_{sw_j} \cup \{p\}. \end{cases} \tag{7}$$

forge_lldppkt: A LLDP packet p with forged $dpid$ and $portid$ meant to spoof the controller is sent by host h_i and later arrives at the edge switch sw_j, which is defined as Eq. (8).

$$\begin{cases} p := newForgedLLDP(dpid, portid), \\ Pend_{sw_j} := Pend_{sw_j} \cup \{p\}. \end{cases} \tag{8}$$

flood: A load of packets with different fake addresses is sent by host h_i in a short term of time and later arrives at the edge switch sw_j, which is defined as Eq. (9).

$$\begin{cases} Ps := newFlood(), \\ Pend_{sw_j} := Pend_{sw_j} \cup Ps. \end{cases} \tag{9}$$

tamper_field: Switch sw_i tampers a key field of a pending packet, which is defined as Eq. (10).

$$\begin{cases} p := pop(Pend_{sw_i}), \\ p' := tamper(p), \\ Pend_{sw_i} := Pend_{sw_i} - \{p\}, \\ Pend_{sw_i} := Pend_{sw_i} \cup \{p'\}. \end{cases} \tag{10}$$

drop: Switch sw_i drops a pending packet deliberately, which is defined as Eq. (11). Notably, a destructive tampering action to a packet is also considered as a `drop` operation because the tampered packets will be seen as illegal packets and ignored by other components.

$$\begin{cases} p := pop(Pend_{sw_i}), \\ Pend_{sw_i} := Pend_{sw_i} - \{p\}. \end{cases} \tag{11}$$

4) Transition: T can be described mathematically as Eq. (12), while the elements in the transition set T indicate that an operation takes place leading to a transfer of system state from one to another. For example, $(s_i, \texttt{send}, s_j)$ that $s_i, s_j \in S$, an element of T, represents that a `send` operation causes the state of the SDN system model to change from state s_i to state s_j.

$$T = S \times O \times S. \tag{12}$$

E. Detecting Attacks

As mentioned earlier, we specify the model in Promela language in order to execute its formal verification with the off-the-shelf formal verification tool Spin, which requires us to define a set of security properties to discover suspicious states that need to report corresponding counterexamples. We summarize the security properties that the model needs to satisfy based on the security issues revealed in previous work, the OpenFlow protocol specification and the specific controller implementation code. We classify the security properties into three categories, which is component capability limitation, topology incorrectness and packet forwarding abnormality.

1) Component Capability Limitation: In SDN, controllers and switches are common targets for DoS attacks [15]. As for the controller, which is the core of the whole SDN network, it is worthwhile to take it down by multiple methods and the centralized structure also makes the controller vulnerable. As we know, the switches are designed to send packet-in messages to the controller if table-miss happens. Furthermore, the switches send the complete packets if their buffers are full or the configurations are specified as no buffer due to the bugs in the old version OVSes [42] and many controllers' default settings, which compels a huge load for the controller to handle and causes DoS attacks on the controller. As for switches, constrained by the fact that the size of the TCAM is limited, the switch is only able to maintain a certain number of flow entries [14]. Moreover, the traffic jam in the data plane can also be a disaster for packet forwarding, which can be caused by improper routing and affect switches and links. The bandwidth of the data path through the choked link in the data plane will remain very low [14]. In summary, we define thresholds of these capability limitations of network components as the security properties of the model, which are $MAXCTRLCAPABILITY$ for the controller's processing capability, $FLOWTABLESIZE$ for switch's flow-table space and $MAXSWCAPABILITY$ for switch's processing capability. The processing capability of each component is represented by the the size of *Pend* list. Accordingly, we present the constraints of security properties about component capability limitation as Eq. (13).

$$\begin{aligned} \forall s \in S, &|Pend_{con}| <= MAXCTRLCAPABILITY, \\ &|FlowTable_{sw_j}| <= FLOWTABLESIZE, \\ &|Pend_{sw_j}| <= MAXSWCAPABILITY. \end{aligned} \tag{13}$$

2) Topology Incorrectness: A link discovered by the controller is defined as a tuple, (sw_i, p_x, sw_j, p_y) $(\in ViewTopo)$, which should be consistent with the link in *RealTopo*. Any discovered links that are not contained in *RealTopo*, that is, do not exist in actual topology, are considered illegal and violate the constraint of the security property. The constraints of security properties about link discovery correctness can be described formally as Eq. (14).

$$\begin{aligned} \forall (sw_i, p_x, sw_j, p_y) \in ViewTopo, \\ (sw_i, p_x, sw_j, p_y) \in RealTopo. \end{aligned} \tag{14}$$

Similarly, a host information is also defined as a tuple, $(h_i, 0, sw_j, p_y)$ $(\in ViewTopo)$. The discovered host information, including the identifier h_i like MAC address and the location. The host location is represented by the host identifier, the host port, the edge switch connected to the host and its port. We simplify the host's port to 0. Host information observed by

the controller should be identical to real host information in *RealTopo*, which is presented as Eq. (15).

$$\forall(h_i, 0, sw_j, p_y) \in ViewTopo, \\ (h_i, 0, sw_j, p_y) \in RealTopo. \tag{15}$$

3) Packet Forwarding Abnormality: A packet that transfers in the network is defined as a tuple with a source host, destination host and other information about the protocol or the OpenFlow message. Regularly, the controller updates the mac-to-port table with the source address of the packet in the data field of the packet-in message, the *IN_PORT* field of the packet-in message and the switch identifier $dpid$. Therefore, we check the rationality of the updates based on the correct predicted mac-to-port table *RealMacToPort* as Eq. (16).

$$\forall(h_s, h_d, ..., IN\_PORT, dpid) \in Pend, \\ (h_s, IN\_PORT, dpid) \in RealMacToPort. \tag{16}$$

We count the steps of components that a packet has been processed to detect potential illegal packet routings, such as the unprocessed loop topology or forwarding loops. Any packets that have been processed and passed to more than $MAXSTEP$ components are suspicious, as Eq. (17).

$$\forall(h_s, h_d, ..., cntstep) \in Pend, cntstep <= MAXSTEP. \tag{17}$$

We also check whether the packet ends at the correct destination when it leaves the network eventually. The output port should normally be an edge port, therefore the output port should not be connected to another switch. Further, the link formed by the destination host and the output port should exist in a real topology, in order to detect black holes and wrong routing strategy as the ultimate constraint in the whole process of packet transferring, presented as Eq. (18).

$$\forall(h_s, h_d, output, .., cntstep) \in Pend_{sw_i}, \\ (h_d, 0, sw_i, output) \in RealTopo. \tag{18}$$

IV. Implementation

The model is implemented using approximately 1400 lines of Promela code. We implement all the components in Fig. 4 with approximately 1,000 lines of Python codes. The main functions encompass model tuning based on configuration files, generating verifiers using Spin, creating readable attack reports and statistical analysis reports.

A. SDN System Model

Host. We describe hosts by their `hostid` representing their address. The `hostid`s, starting at 1, are assigned to each host in order and used as the indexes in topology descriptions. The hosts are conceptualized as stateless identifiers related to a specific port of a switch, which means we do not maintain the state of the pending or processing of packets on hosts, and only consider them as the source and destination of data packets when determining routing strategies on the controller and judge whether the packet reaches the correct destination. For malicious hosts, we use a variable `MaliciousHost` to specify the attacker's `hostid` and a set of bool variables to represent its attack abilities on forging data packets, forging LLDP packets and flooding with other variables that define the details about an exact attack such as the packet volume of flooding and addresses of forging.

Switch. We describe switches by their `swid`, also starting at one and assigned to each switch sequentially, which can be recognized as the DPID of the devices. Unlike hosts, the switches in the model are stateful and constitute part of the overall SDN network state. Each switch holds an array of pending list of packets containing the arrival packets waiting to process, variables as the pointer of the index of the next arrival packet and the index of the next packet to be processed, and also a variable to record the total number of current packets. Besides, each switch maintains an array of flow-table and maintaining-related variables similar to the array of the pending list to hold flow mod messages sent by the controller and to modify flow entries. As for malicious switches, we define a variable `MaliciousSwitch` to recognize the `swid` compromised by the attacker and a set of bool variables to represent its attack abilities on tampering with key fields and dropping packets, along with other variables that define the details about an exact attack such as the field and value to tamper.

Controller. Similar to switches, the controller maintains an array of the pending list of arrival packets and the related variables in order to sustain the packet processing. Besides, the controller also has an array of the mac-to-port table and an array of the topology view, which is updated while the packets process and affects the core function of the controller like routing.

Topology. The topology of the network is represented as a 2-D array of device information, in which the device's id and port id are the indexes. For switches, the device's id and port id are the `swid` and `portid`. For hosts, `hostid` starts at the next value from the last `swid` and 0 as the port id. The device information includes the type, the device id and the port id of the opposite device. On the one hand, if both the index and the device type indicate it as a switch, then this entry of the array can be seen as a link in the network. On the other hand, if the index or the device type indicates a host, then this entry of the array represents the host information about the connecting switch and port. Both the actual topology and the topology view of the controller, `Realtopo` and `Viewtopo` are stored in this structure so as for convenient comparison. Since we do not consider external network effects, the actual real topology does not change during the experiments. The real topology will be initialized during the very beginning of the model as the topology of our experiment configuration.

Operation. The implementations of operations mainly focus on the add, delete and modification of the arrays of pending lists. For example, a simple description of one data packet transition from one switch to another is deleting the packet in the pending list of the source switch and copying it into the pending list of the destination switch. The respective operation details are drawn from the abstraction of the theoretical logic and actual practical effects of operations.

B. Counterexample Analysis

We use assertions to describe the security properties presented in Section III-E. Specially, the assertions are defined into 8 kinds, which are controller capability limitation (*CtrlCapability*), switch capability limitation (*SwCapability*), flow-table space limitation (*FlowTableSize*), link discovery incorrectness (*LinkDiscovery*), host information incorrectness (*HostInfo*), mac-to-port table abnormality (*MacToPort*), packet process step abnormality (*PktSteps*) and packet transfer abnormality (*PktTransErr*).

Spin will automatically check the correctness of the assertions inserted in the model. Once it occurs to a state that violates the assertions during the verification running, Spin will report a counterexample and output the trail file about how the state transitions lead to the violation. After the execution of formal verification with Spin, we obtain the trails of counterexamples if any states violate the assertions of security properties and leverage them to derive the specific readable logs of the trails in order to study how the actions in the network cause the violation of security properties, that is, how the attacker succeeds.

C. Model Optimization

The looming challenge confronting model checking is the risk of state explosion. To address this issue, we employ a set of reasonable assumptions derived from our prior knowledge and goals of our detection. Model checking operates as a depth-first traversal search across the Finite State Machine (FSM) representing the target system, while the primary aim of model optimization is to minimize the number of states in the state machine or eliminate unreachable state branches. Common optimization techniques include abstraction, symbolic model checking, partial order reduction, state compression, among others.

Network topology optimization. Network topology optimization is a higher level of abstraction optimization. The SDN controller, as the central brain of the system, is engineered to be compatible with topologies of any size. Considering that security issues in SDN controllers generally do not depend on the size of the topology [43], we have chosen to utilize small and simple topologies for our attack discovery process. These topologies are depicted in Figure 6 and Figure 7, selected specifically for their efficiency in revealing attacks. To validate the adequacy of using smaller topologies in our research, we have carried out extensive experiments with various network topologies. The findings and justifications for this approach are detailed in Section V-E of our study.

Variable value space equivalence reduction. The value space of a variable can be symbolized, mapping the original concrete values to symbolic values. In our SDN model, the network operations encompass a wide array of values. Recognizing that some of these values may have equivalent effects on the SDN controller, we adopt an optimization approach akin to that used in NICE [21]. This method involves grouping values into equivalence classes, which significantly reduces the value space of variables in the network. For example, when modeling the port_number variable, we classify ports into

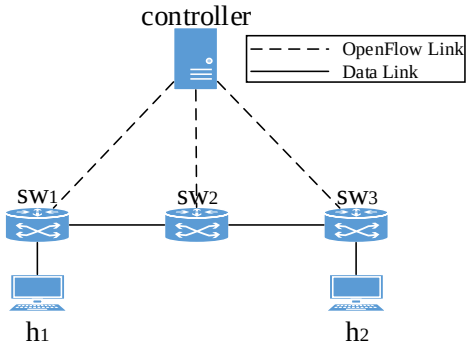


Fig. 6: The basic topology

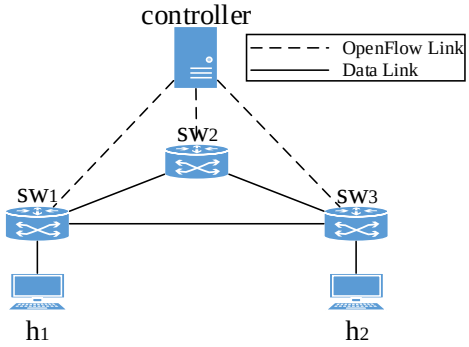


Fig. 7: The topology with a loop

open and closed states. Each open port is treated as a distinct equivalence class, whereas all closed ports are grouped into another class. This systematic approach to classifying values allows us to effectively streamline and condense the range of values associated with the port number variable during the modeling process.

Data plane simplification. In the data plane, concrete packet types can also be abstractly mapped to symbolic packet types. Our approach to data plane simplification emphasizes the reduction of complexities in switches and end hosts. In our model, we retain only those elements of the data plane that are pertinent to the controller's packet processing and topology discovery mechanisms. To achieve a more streamlined model, we exclude certain OpenFlow data plane structures, such as the Meter Table, as well as specific OpenFlow message types, like the `OFPT_FLOW_REMOVED` message, for switches. Similarly, for end hosts, we omit various network protocols, including ARP, TCP, UDP, and LLDP.

We further simplify our model by categorizing packets into two primary types: LLDP packets, identified by the LLDP type, and all other packets, labeled as the DataPkt type. This distinction is based on their respective handling mechanisms within the SDN environment. LLDP packets are specifically managed by the link discovery mechanism, while all other packets trigger the packet processing mechanism. This focused and strategic simplification ensures that our formal verification process is both efficient and effective, without compromising the integrity of the model.

Packet order optimization. The packet order optimization fully follows the idea of partial order reduction. In the SDN network state model, factors like packet transmission latency, packet processing latency at network devices, and the device responsible for processing at any given time contribute to a notable increase in the number of states. In our model optimization, we address this complexity by treating each internal processing or transmission within the various SDN planes and devices as an atomic operation, thereby simplifying the states. Furthermore, we preserve the stochastic nature of network processing by randomly selecting components in the model to execute operations. More specifically, our model maintains a separate circular queue for each network component that receives and forwards packets arriving at the network component following the FIFO (First-In, First-Out) principle. This ensures that packets arriving first at a component are typically prioritized and forwarded to the next-hop network component. This approach effectively simplifies the handling of order changes that might result from complex processing within the network components. Additionally, for the interaction between the controller and the switch, we integrate the NO-DELAY heuristic algorithm from NICE [21] into our model. This optimization mirrors the implementation of the direct OFPT_BARRIER function in OpenFlow, thus guaranteeing the sequential order of OpenFlow messages. For instance, we ensure that an OFPT_PACKET_OUT message always comes after the OFPT_FLOW_MOD message. It is also important to note that while out-of-order packet delivery is a common occurrence in networks, including SDNs, the security issues we focus on in this paper are generally not reliant on this aspect. Therefore, for simplicity, our model does not consider out-of-order packet delivery.

Component capability optimization.

The capacity constraints in the optimized model stem from the concept of state compression, which efficiently diminishes the number of loops in the state machine. Network component capabilities are usually relatively large in real networks. For example, in most controllers, MAXCTRLCAPABILITY reaches values in the millions [44], FLOWTABLESIZE often numbers in the thousands [45], while MAXSWCAPABILITY usually fall within the range of 1,000 to 9,000 [46]. However, replicating these real-world values in our experimental setup could result in state explosion. For example, setting FLOWTABLESIZE to 8,000 would create thousands of new states to represent the scenarios with 0 to 8,000 rules installed in the switches, most of which are irrelevant for attack discovery. Therefore, we opted for smaller values, adhering to the principle of selecting the smallest possible value for each variable under which our method would not falsely report attacks in non-attack scenarios. For example, in the case of the three-switch topology, as it requires two rules for bidirectional traffic flow, we set FLOWTABLESIZE to 2. Crucially, these variables are configurable. We utilize a configuration file to set these values, offering flexibility to adjust them according to different topology sizes and specific security concerns.

Other optimization. We count the number of operations and limit its maximum to avoid unnecessary transitions after the essential operation amount to detect counterexamples, which we derive according to multiple tests of model checking. This can also reduce duplicate types of counterexamples and lighten the workload of further analysis.

V. Evaluation

We translate the SDN model depicted in Section III in the way of Section IV and apply the Python script to run the whole process of verification with Spin and result analysis. Through reviewing the readable attack logs and the analysis report, we have found two kinds of novel attack methods and manually confirmed their feasibility with practical demonstrations.

A. Experiment

1) Experiment Setup: We performed the formal verification with Spin (6.5.2) [33] running with the machine with 8 GB of RAM. For attack demonstrations, we created an emulated experiment environment on a Ubuntu Linux 20.04 virtual machine. The simulated network was created by Mininet (2.3.0d6) [47] with Open vSwitch (2.13.1) [42]. We verified whether two new attacks can be implemented on different controllers and adopted POX(0.7.0) [48],Ryu(4.34) [49], Floodlight(v1.2) [50], ONOS(v2.5.0) [51], OpenDaylight(Carbon) [52] as the target SDN controller. We also verified whether the attacks can be implemented under different OpenFlow versions (see Table VI and Table VII).

The characteristic of model checking dictates that a small difference can derive an individual state, which means there can be a large number of states, even if there are only a few components defined in the SDN model. Though the automated formal verification process can iterate through all possible states theoretically, we set the search depth of Spin as 20,000 based on several times of attempts considering the overhead, which is adequate to derive satisfying results.

2) Effectiveness: We evaluate the model with an example of normal traffic, benign components and the topology without loops to verify the correctness of the SDN model and the rationality of the configuration of the parameters and constraints. Specifically, the example includes the LLDP packets of the initial link discovery and a data packet sent by each host to another respectively. With the current settings, the correct model should not output any counterexamples. It runs with the same settings of Spin as the experiments with malicious participants and outputs no counterexample after the verification of 104 s of time and 138 MB of memory usage. It shows that the generation of counterexamples is caused by abnormal traffic, operation of malicious components, etc. Further, it indicates the reliability of the later experimental results.

We test our SDN model by applying the Python script and successfully detect 39 errors with their trail files, which can be categorized into 8 types by the final assertions they triggered, and the number of errors of each assertion are shown in Table IV. The analysis report shows the statistics of different assertions and the summaries that refine the key contexts in trails of each counterexample, like assertion violations and malicious operations. The analysis can help the users identify the attack paths and their causes and guide the further specific check of each readable attack log.

TABLE IV: Summary of counterexamples

Assertion	Number
CtrlCapability	2
SwCapability	6
FlowTableSize	1
LinkDiscovery	10
HostInfo	4
MacToPort	11
PktSteps	4
PktTransErr	1

3) Performance: We measure the total time of the verification of the SDN model, the generation of readable counterexamples and the analysis of the results. The whole process takes about 375 *s*, of which 104 *s* are for running the verification process with Spin to generate counterexamples. Meanwhile, the report of Spin also shows that the total actual memory usage of the model checking is about 138 *MB*.

B. Case Study

We manually look into the attack logs guided by the analysis report and filter the unique attack paths summarized in Table V. There are 23 unique attack paths among all the counterexamples and 2 of them are new attacks detected by our method. Also, the detection of known attacks shows that our method can reach the goal of effectively helping users find practical attack paths against SDN network based on the formal verification. These known attacks were also not discovered by a single work, but were summarized in the previous research efforts.

CtrlCapability. The malicious participants can leverage the mechanism of processing new packets to trick SDN switches to send more packet-in messages to the controller leading to overload and resource exhaustion of controller. As for the malicious host, it can implement the attack by generating data packets with fake addresses which would be seen as packets with no matching flow entry and triggering the packet-in messages. As for the malicious switch, it can drop the flow-mod messages to keep data packets not matching and to continuously send packet-in messages.

SwCapability. Instead of normally sending a massive of data packets, the switch would be exhausted by the packet-out and flow-mod messages in a less amount of data packets. Both the packet-out messages and the flow-mod messages consume the CPU of the switch agent. The flow-mod messages have more overhead than the packet-out messages. Under saturation attack and the striking attack, the bandwidth of normal users will be significantly decreased [23]. Moreover, the attack can be successful even faster, if the switch bans the use of buffer, which will force the controller to send both packet-out and flow-mod messages. With the optimization of our method that limits the transitions, we can easily filter out the more advantageous approach that triggers the switch capability assertion earlier.

FlowTableSize. The switch's flow table is usually implemented as TCAM, so the flow table's are usually constrained and not too large (e.g. IBM RackSwitch G8264 with a TCAM of size 1K) [14]. Therefore, it is not difficult to add enough flow table entries to fill the flow table. In order to trick the controller to issue more new flow entries, the malicious host generates data packets with different source addresses based on the gullibility of the controller, which is also convenient to implement with only one compromised host.

LinkDiscovery. The malicious switch can launch attacks against link discovery by modifying the key fields of LLDP packets, such as IN_PORT, Chassis ID and Port ID. Simply dropping or invalidating the LLDP packets can also cause the controller to delete links, giving the data plane a different view of the network than the control plane. The malicious host can forge LLDP packets or indirectly interfere with the link discovery process by launching DoS attacks against the SDN network components.

HostInfo. The host information can be poisoned through the injection of data packets and packet-in messages containing tampered information. Additionally, we have uncovered two novel attacks with benign data packets. In SDN architecture, edge ports typically denote connections to hosts, while internal ports facilitate inter-switch connections. The controller only registers the host location when the IN_PORT of the packet-in message is an edge port. Consequently, a host information poisoning attack can be initiated through orchestrated link deletions, which can be accomplished by two conspiring neighbor switches or one malicious switch. After the link deletion of both ways, the edge ports and associated hosts of the attacker switch erroneously relocate to the internal ports of neighboring switches, thereby corrupting the host location information maintained by the controller.

MacToPort. The malicious participants can directly modify the data packets and packet-in messages to poison the mac-to-port table and further affect the routing. Also, the malicious switch can indirectly mess up the packet transfer causing another innocent switch to send abnormal packet-in messages. In addition, a network topology with a loop will cause MAC address flapping if the controller does not leverage any forwarding loop preventing solutions like spanning tree.

PktSteps. The modification of the IN_PORT field of packets by the malicious switch will make a message broadcast to the port it arrived at. The invalid broadcast might not lead the packet to the right destination, which causes more transitions and operations of the packet transfer and trigger the assertion of the limitation of process steps. Besides, modifying the output port of the packet-out message to the IN_PORT of the packets can make a routing loop, leading the packet to repeat transfer between two switches and finally violating the assertion. Moreover, a network topology with a loop can cause a forwarding loop which leads to a broadcast storm as mentioned earlier.

PktTransErr. Obviously, the packets might transfer to the wrong destinations if the malicious switch modifies the output port of packet-out messages, which violates the normality of packet forwarding when the data packets leave the network in the wrong ports. Packets that do not eventually reach their destination may result in black hole routing or denial of service in the data plane.

TABLE V: Summary of attacks

No	Assertion	Result	Attack Description	Count	Known
1	*CtrlCapability*	Controller resource exhaustion	Malicious host generates data packets causing switches to send packet-in messages to the controller.	1	Yes [14] [15] [17]
2		Controller resource exhaustion	Malicious switch drops flow-mod messages causing packet-in message flooding to the controller.	1	Yes [41]
3	*SwCapability*	Switch resource exhaustion	Malicious host generates data packets causing packet-out and flow-mod message flooding to the switch.	6	Yes [23]
4	*FlowTableSize*	Switch resource exhaustion	Malicious host generates data packets leading to flow-mod message flooding and new flow entries filling the flow-table.	1	Yes [14] [15]
5	*LinkDiscovery*	Link fabrication	Malicious host forges LLDP packets with fake Chassis ID and Port ID.	1	Yes [9] [10]
6		Link deletion	Malicious host generates data packets to DoS the controller and make it unable to process the LLDP packets in time.	1	Yes [9]
7		Link deletion	Malicious host generates data packets to DoS the switch and makes it unable to process the LLDP packets in time.	1	Yes [9]
8		Link fabrication	Malicious switch modifies IN_PORT fields of LLDP packets.	1	Yes [41]
9		Link fabrication	Malicious switch modifies Port ID fields of LLDP packets.	1	Yes [41]
10		Link fabrication	Malicious switch modifies Chassis ID fields of LLDP packets.	1	Yes [14]
11		Link deletion	Malicious switch drops LLDP packets.	4	Yes [9]
12	*HostInfo*	Fake host information	Malicious host sends packets with victim's address to convince the controller that victim is at malicious host's location.	1	Yes [9] [10] [14]
13		Fake host information	Two malicious switches conspire to drop LLDP packets from each other to make controller believe there are edge ports and record the host joining on those ports as processing packet-in messages.	1	No
14		Fake host information	Malicious switch drops both LLDP packets from a existing link and LLDP packets to the existing link to make controller believe there are edge ports and record the host joining on those ports as processing packet-in messages.	1	No
15		Fake host information	Malicious switch modifies IN_PORT fields of data packets with an edge port and makes controller record the host joining on those ports as processing packet-in messages.	1	Yes [41]
16	*MacToPort*	Mac-to-port table poison	Malicious host sends packets with victim's address to convince the controller relate the victim's address with wrong port related to malicious host.	1	Yes [14]
17		Mac-to-port table poison	A topology with loops can cause MAC address flapping.	1	Yes [21]
18		Mac-to-port table poison	Malicious switch modifies IN_PORT fields of data packets.	7	Yes [41]
19		Mac-to-port table poison	Malicious switch modifies output port fields of packet-out messages and returns the packets to the former switch.	2	Yes [41]
20	*PktSteps*	Broadcast storm	A topology with loops can cause data packet flooding.	1	Yes [21]
21		Broadcast error	Malicious switch modifies IN_PORT fields of packet-out messages and makes the packets broadcast to the received ports.	2	Yes [41]
22		Routing loop	Malicious switch modifies output port fields of packet-out messages with the received ports.	1	Yes [14] [41]
23	*PktTransErr*	Blackhole	Malicious switch modifies output port fields of packet-out messages.	1	Yes [14]

C. Attack Demonstration

We demonstrated the new attacks derived through our method, Attack No. 13 and No. 14. As for the other known attacks, they have been verified and described in detail in previous work. Therefore, our practical experiments focus more on the unknown new attacks. For the practical experiment, we uses the same network topology as Fig. 6 and tested different SDN controllers and different widely used OpenFlow versions. For the controllers, they are all able to support packet processing as well as link discovery and host tracking service based on the OFDP protocol.

1) Attack No. 13: Following the attack description, we define sw_1 and sw_2 as the malicious switches compromised by the attacker as shown in Fig. 8(a) and Fig. 8(b). There is a both-way link between sw_1's p_2 and sw_2's p_1. The two malicious switches drop LLDP packets from each other by installing flow entries to take drop actions on LLDP packets from the specific ports on their own causing the controller reports the `EventLinkDelete` events of both ways as Fig. 8(a). Furthermore, flow entry will be installed on sw_1 to directly pass its arrival packets to sw_2 to prevent matching the table-miss entry and sending packet-in messages to the controller. Later, a data packet sent by the benign host h_1 will trigger sw_2 to send a packet-in message as Fig. 8(b). Since the controller does not recognize the link between sw_1 and sw_2 and consider sw_2's p_1 as an edge port, the attacker can successfully trick the controller to believe that h_1 is joining in the network on sw_2's p_1 instead of sw_1's p_1. This attack aims at hijacking the location of benign hosts with less attention to the controller and the hosts, which will affect the routing or

implement an attack of bypassing the access control policies further.

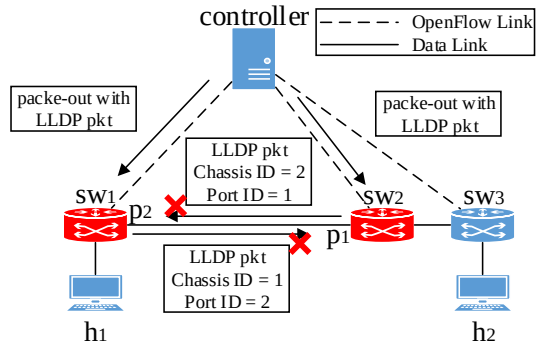
TABLE VI: Verify Attack No. 13 in different controllers support different OpenFlow version

Version	POX	Ryu	FL	ONOS	ODL
OF1.0	✓	✓	✓	✓	✓
OF1.1	N/A	✓	N/A	N/A	N/A
OF1.2	N/A	✓	N/A	N/A	N/A
OF1.3	N/A	✓	✓	✓	✓
OF1.4	N/A	✓	✓	✓	N/A
OF1.5	N/A	✓	N/A	N/A	N/A

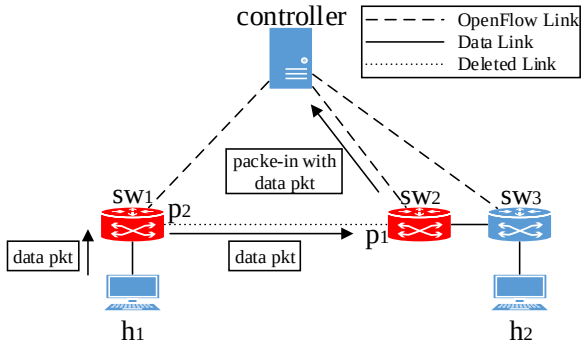
✓ indicates the attack can be validated. ✗ indicates the attack cannot be validated. *N/A* indicates OpenFlow channel cannot be established between controllers and switches. *FL* stands for Floodlight Controller. *ODL* stands for OpenDaylight Controller.

We further try to verify the attack in different OpenFlow versions with different controllers. Except for POX controllers that only support OpenFlow 1.0 and OpenDaylight controllers that do not support OpenFlow 1.4, the attack can be successful as shown in Table VI. In the process of our experiments, we noticed that although we had the malicious switches sw_1 and sw_2 perform the same operation of dropping LLDP packets, there were differences in the way they were implemented in different controllers. The ONOS controller requires the northbound APIs to install malicious flow entries to drop LLDP packets, while other controllers only require the Open vSwitch command for flow entries installation. The reason for this is that ONOS keeps flow entries consistent between the controller and the data plane. If the flow table installation is done only in the data plane, ONOS detects the inconsistency and removes the flow entries. Although the attack exists on different controllers, for ONOS controllers it may require an attacker with stronger attack capabilities to successfully perform the attack. Different OpenFlow versions also change flow entries installation commands, but eventually do not affect the location of benign hosts hijacked by attackers. Among the different controllers, we can observe the hijacking of benign hosts location through the controller logs or the controller Web GUI.

2) Attack No. 14: Attack No. 14 exploits the same vulnerability as No. 13 but with fewer requirements, making it less conspicuous as the attacker only needs to compromise a single switch within the SDN network. In this scenario, we assume sw_1 as the malicious switch trying to disrupt the bidirectional link between sw_1's p_2 and sw_2's p_1 as Fig. 9. The attacker drops the LLDP from sw_2 to sw_1 with self-installing flow entries and replacing LLDP packets containing Chassis ID 1 and Port ID 2 in packet-out messages with invalid information(e.g., Chassis ID 0 and Port ID 0). Although the modified LLDP packets still reach the controller, their invalid topology information prompts the controller to discard them after parsing, thereby tampering with the semantics of the link originally connecting from sw_1 to sw_2. It's worth noting that directly dropping LLDP packets might lead to the disconnection between the switch and the controller. Therefore, we consider processing LLDP packets on the malicious switch and disabling them from being accepted by the controller. Then we can see the `EventLinkDelete` events of both ways in the



(a) Malicious switches drop LLDP packets causing the link deletion



(b) The data packet sent by h_1 misleads the controller into believing that h_1 is on sw_2's p_1

Fig. 8: Attack No. 13

controller console logs or see both ways links being deleted in the controller topology. Then the data packets sent from h_1 to h_2 leading the innocent sw_2 to send a packet-in message similar as Fig. 8(b), which indirectly convince the controller that h_1 is on sw_2's p_1 as the `EventHostAdd` event. Taking Ryu as an example, the log of the whole attack process are shown in Fig. 10. Ryu performs link discovery and observes two bidirectional links between sw_1, sw_2 and sw_3. Further, the bidirectional link between sw_1 and sw_2 is successfully removed by the operation of the malicious switch sw_1. Finally, the benign packet sent by h_1 will cause the controller to assume that h_1 appears at the port p_1 of the switch sw_2.

TABLE VII: Verify Attack No. 14 in different controllers support different OpenFlow version

Version	POX	Ryu	FL	ONOS	ODL
OF1.0	✓	✓	✓	✓	✓
OF1.1	N/A	✓	N/A	N/A	N/A
OF1.2	N/A	✓	N/A	N/A	N/A
OF1.3	N/A	✓	✓	✓	✓
OF1.4	N/A	✓	✓	✓	N/A
OF1.5	N/A	✓	N/A	N/A	N/A

✓ indicates the attack can be validated. ✗ indicates the attack cannot be validated. *N/A* indicates OpenFlow channel cannot be established between controllers and switches. *FL* stands for Floodlight Controller. *ODL* stands for OpenDaylight Controller.

Comparing to attack No. 13, this attack might be more

difficult to identify the malicious component since sw_1 is not exposed in the host location and both the related participants h_1 and sw_2 are innocent. It is also easy to notice that the attack can also be reproduced in different OpenFlow versions of different controllers as shown in Table VII. Both new attack paths were eventually validated in the current mainstream SDN controllers and widely used version of OpenFlow. It illustrates that our approach finds out common vulnerabilities across different OpenFlow controllers, and different OpenFlow versions.

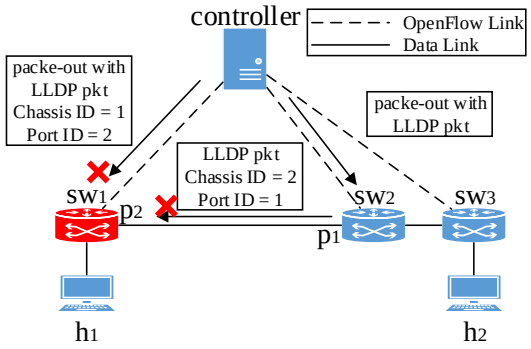


Fig. 9: Attack No. 14. Malicious switch sw_1 drops LLDP packets and invalidates packet-out messages with LLDP packets

```
EVENT ofp_event->switches EventOFPPacketIn
EVENT ofp_event->switches EventOFPPacketIn
EVENT switches->DiscoveryEventDumper EventLinkAdd
EVENT switches->DiscoveryEventDumper EventLinkAdd
EVENT ofp_event->switches EventOFPPacketIn
EventLinkAdd<Link: Port<dpid=2, port_no=1, LIVE> to Port<dpid=1, port_no=2, LIVE>>
EventLinkAdd<Link: Port<dpid=3, port_no=1, LIVE> to Port<dpid=2, port_no=2, LIVE>>
EVENT switches->DiscoveryEventDumper EventLinkAdd
EventLinkAdd<Link: Port<dpid=1, port_no=2, LIVE> to Port<dpid=2, port_no=1, LIVE>>
EVENT ofp_event->switches EventOFPPacketIn
EVENT switches->DiscoveryEventDumper EventLinkAdd
EventLinkAdd<Link: Port<dpid=2, port_no=2, LIVE> to Port<dpid=3, port_no=1, LIVE>>
......
EVENT ofp_event->switches EventOFPPacketIn
EVENT switches->DiscoveryEventDumper EventLinkDelete
EventLinkDelete<Link: Port<dpid=1, port_no=2, LIVE> to Port<dpid=2, port_no=1, LIVE>>
......
EVENT ofp_event->switches EventOFPPacketIn
EVENT switches->DiscoveryEventDumper EventLinkDelete
EventLinkDelete<Link: Port<dpid=2, port_no=1, LIVE> to Port<dpid=1, port_no=2, LIVE>>
......
EVENT ofp_event->switches EventOFPPacketIn
EVENT switches->DiscoveryEventDumper EventHostAdd
EventHostAdd<Host<mac=00:00:00:00:00:01, port=Port<dpid=2, port_no=1, LIVE>,10.0.0.1>>
```

Fig. 10: Ryu console log

D. Comparison with Other State-of-the-Art Works

Our analysis, as detailed in Table VIII, shows that some attacks uncovered by our approach have also been identified in other research. However, it is crucial to note that each of these tools only reveals a subset of the attack paths that our method has detected.

NICE [21] is a tool utilized for both modeling and formal verification of SDN applications specifically designed for NOX controllers. It employs symbolic execution, simplified modeling of hosts and switches, and incorporates four heuristic algorithms to pinpoint issues within NOX controller applications. However, as we discuss in Section II-D, NICE [21] is exclusively tailored to model controller applications and is restricted to NOX controllers. SPHINX [14] constructs an incremental flow graph by collecting FLOW_MOD, PACKET_IN, STATS_REPLY, and FEATURE_REPLY from the OpenFlow Proxy. SPHINX verifies whether the flow graph meets security policy constraints to find malicious switches and malicious hosts in the network. BEADS [41] is an automated attack discovery framework that focuses on testing controllers through protocol-aware fuzzing. It defines the OpenFlow protocol structure to ensure that inputs can be received by the controller without generating a large number of unparsable error messages.

In terms of attack discovery, our method has shown remarkable effectiveness by identifying 23 security vulnerabilities. In contrast, each of the other three tools we evaluated detected fewer than ten security flaws. It is noteworthy that the remaining known security flaws, which were identified prior to our detection, were typically discovered through manual methods [9].

Moreover, our method exhibits significant efficiency advantages over these three existing tools. A notable limitation of NICE [21] is its capacity to handle only a finite number of packets, leading to a dramatic increase in the number of states with a slight increase in ping packets. For example, as the number of ping packets in NICE increases from 4 to 5, the detection time escalates exponentially from 30 minutes to 30 hours, and processing more packets risks state explosion. BEADS [41] requires around 60 seconds to test each input, resulting in approximately 200 hours to complete testing on a controller. SPHINX [14] operates in dynamically running networks, employing network invariants to detect attacks. While it can respond to certain malicious attacks in just a few microseconds, its ability to identify unknown attack paths remains quite limited.

TABLE VIII: Compare our method with other vulnerability detection tools

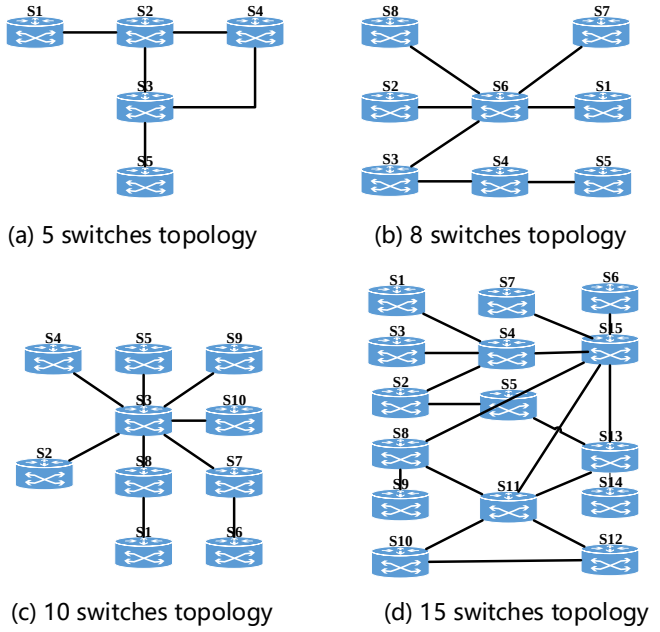
Attack No.	NCIE [21]	SPHINX [14]	BEADS [41]
1		✓	
2			✓
3			
4		✓	
5			
6			
7			
8			✓
9			✓
10		✓	
11			
12		✓	
13			
14			
15			✓
16		✓	
17	✓		
18			✓
19			✓
20	✓		
21			✓
22		✓	✓
23		✓	

In addition to the aforementioned works, it's worth noting that numerous other studies investigate security issues in SDN

controllers. However, their focus diverges from ours. For instance, Intender [53] focuses on security concerns within the network intent module of controllers. AudiSDN [54] aims to detect the inconsistencies of network rules across various layers of the SDN system. EVENTSCOPE [55] identifies missed event handlers in SDN applications. SVHunter [56] targets D2C2 vulnerabilities (data dependency creation and chaining) exploitable by attackers. SDN-fuzzer [57] detects storage vulnerabilities in NMDA-based controllers; and Evgenii Vinarskii et al. [58] have proposed a modeling approach for identifying concurrency issues (i.e., races) in controllers. Due to the distinct nature of these studies, they were not included in our direct comparison.

E. Performance Evaluation

In this subsection, we examine the influence of network topology and the execution time of our tool on the outcomes of attack discovery. To this end, as depicted in Figure 11, we selected 4 network topologies of varying sizes for our evaluation. These topologies were chosen based on existing research work and from the topology zoo dataset [59].



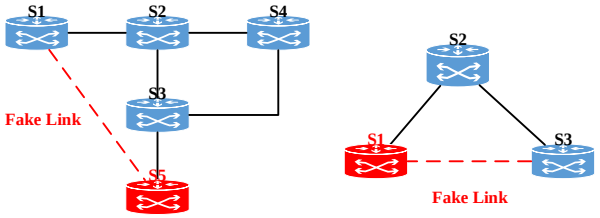
(a) 5 switches topology (b) 8 switches topology

(c) 10 switches topology (d) 15 switches topology

Fig. 11: Tested topologies (For simplicity, hosts were excluded from the topologies under examination).

Performance under different topologies. We conducted experiments where we varied network topologies and monitored the associated overheads, such as time consumption, needed to discover all attacks as listed in Table V. Additionally, we evaluated the effectiveness of our tool in identifying attacks across these different topologies. The results presented in Table IX shows an increase in overheads, including computational time and storage requirements, as the network size expands. This is in line with our expectations, given the increasing complexity of the state model in larger networks, necessitating more resources for Spin to analyze the model. Notably, while the number of reported attack paths by our tool increases with larger network sizes, our findings indicate that no new types of attacks (beyond those discovered with the three-switch topology, see Table V) were identified.

Outcomes of our tool with different execution time. We also explored whether our tool could uncover attacks that differ fundamentally from those previously identified when operated over extended time periods. For this purpose, we ran our tool for varying durations — 30 minutes, 1 hour, 2 hours, and 4 hours — and across different network topologies. The findings, as detailed in Table X, indicate that while the number of reported attack paths increases in larger networks with longer execution times, there are no new types of attacks identified beyond those already discovered.



(a) Link fabrication in 5 Switches topology (b) Link fabrication in 3 Switches topology

Fig. 12: Equivalent attacks in different topologies

Discussion. The analysis of industrial-scale topologies is crucial for validating network policies, primarily because the complexity of these policies and their validation challenges are directly linked to the scale of the topology [38]. However, when it comes to detecting security issues in SDN controllers, it is important to note that most of these issues are not inherently dependent on the size of the topology [43]. This assertion is supported by the results of our experiments.

Specifically, we found that choosing a larger topology does result in a higher number of reported attack paths as the topology scale increases, but it does not lead to the discovery of new types of attacks. The same attacks identified in smaller and simpler networks (e.g., a 3-switch topology) are also identifiable in larger networks, as the underlying vulnerabilities leading to these attacks remain constant regardless of network size and topology. In larger networks, the increased number of network nodes potentially exploitable by attackers results in more attack paths. For instance, as depicted in Figure 12, the link fabrication attack is detectable in both 3-switch and 5-switch topologies, albeit with varying attack paths. Considering the additional time required to discover all attacks in larger network topologies, we opted for a smaller and simpler network topology for attack discovery in our current research.

Conclusions for different scale topologies. Our proposed model checking tool has the capability to accommodate various scale topologies. However, scaling up to larger topologies inevitably results in considerable increases in both time and memory overhead. Additionally, when it comes to automated attack discovery for SDN controllers, expanding the scale of the topology does not necessarily translate into a qualitative

TABLE IX: Performance of our tool under different topologies

Network nodes [switches, hosts]	**Overheads**				**Capability of attack discovery**	
	Compile time	Total time	RAM used	State-vector size	Number of attack paths reported	New attacks beyond those in Table V
[3, 2]	155 s	375 s	138 MB	1044 B	39	N/A
[5, 6]	311 s	550 s	149 MB	2108 B	71	✗
[8, 8]	489 s	1318 s	375 MB	4588 B	128	✗
[10, 10]	652 s	2043 s	548 MB	7308 B	157	✗
[15, 15]	868 s	2915 s	1304 MB	14524 B	225	✗

TABLE X: Outcomes of our tool with different execution time

Network nodes [switches, hosts]	**30min**	**1h**	**2h**	**4h**
[3,2]	(39,✗)*	(39,✗)	(39,✗)	(39,✗)
[5, 6]	(490,✗)	(1056,✗)	(2214,✗)	(4471,✗)
[8, 8]	(218,✗)	(516,✗)	(1094,✗)	(2308,✗)
[10, 10]	(130,✗)	(331,✗)	(746,✗)	(1545,✗)
[15, 15]	(103,✗)	(312,✗)	(691,✗)	(1572,✗)

* For the tuple (X, Y), *X* represents the number of reported readable paths, and *Y* represents whether new attacks (compared to those listed in Table V) are discovered.

change in the number of detected security issues; instead, it often leads to redundant exploration of state transitions. Given these considerations, we opt for smaller topology sizes as an optimization strategy to expedite the discovery process of security vulnerabilities, prioritizing time efficiency and effective memory utilization.

VI. Discussion and Further Work

Our methodology is centered around modeling the components of an SDN system to uncover potential security risks in SDN controllers and generate comprehensible attack paths. These paths provide a roadmap that enables us to efficiently validate the reported attacks, confirming the presence of any security vulnerabilities within the controller. While our current approach has its limitations, we are actively considering enhancements and future developments to address these areas and augment our methodology.

Addressing dynamic network activities. Our current methodology focuses on attack discovery within static network environments. While the attacks we have reported are applicable to real-world, dynamic SDN systems, the introduction of dynamic network activities could potentially lead to the emergence of new attack vectors not accounted for in our existing framework. As such, a future objective is to expand our methodology to include dynamic scenarios. This expansion will involve integrating new operations into our model that reflect dynamic network activities, like adding or deleting switches and managing switch ports. The high-level design of our tool is poised to adapt to the dynamics of SDN systems. For example, we can add new operations involving the addition or deletion of switches to modify the *RealTopo* variable (see Table II) to reflect dynamic switch changes. However, the challenge extends beyond simply describing the network topology changes; it also involves developing a model that accurately describes how SDN controllers process dynamic events. This requires an abstraction of controller behaviors, which can be derived either from static analysis of controller source codes or from dynamic testing on the controllers. Given the substantial challenges this poses, we plan to address this aspect in our future work.

Manual works. We still need manual work to translate the mechanisms into formalized model language and refine the security properties, which needs ample knowledge of prior works and developer documentation. As for the enhancement, we can leverage NLP (Natural Language Processing) to extract the security properties with the keywords in the documentation to reduce the workload of manual reading, which is left to our future research. We can also consider some static analysis methods to analyze the code similarity among the specific implementations of different controllers, to extract the common mechanisms to help model the SDN system. How to translate controller implementation languages such as Java and Python into the formal language Promela is also worth exploring.

State explosion. We adjust and simplify the parameters of the model to avoid meaningless transitions and limit the states for more effective and valuable counterexamples and reduce the later manual effort to study the reported attack paths. But this solution also limits the coverage of transitions to discover more possible attack paths. We shall balance the verification coverage with the scale of the model when more operations are added based on experience from multiple tests and estimation from prior knowledge.

VII. Related Work

Attacks and detection in SDN. Many works have proposed attacks and defenses in SDN focusing on different attack surfaces. As for the attack against the control plane, the most common target is poisoning the topology view of the controller since it is easy to spoof the controller with forge packets because of the gullibility of the controller. There are researches [10]–[14] that proposed host location hijacking attacks and link fabrication attacks to affect the topology view and their corresponding defenses that check more conditions before accept the topology information. Besides, Marin et al. [9] proposed reverse loop attack which tricks the controller into re-compute the topology under unnecessary conditions in order to exhaust the CPU of the controller. The DoS attacks are also widely researched to affect the control plane, among which the packet-in messages are often abused to launch resource exhaustion attacks against switches and controllers [14], [15], [17]. The frequent occurrence of DOS attacks in SDN networks is mainly due to the controller's role as the brain of the network and its limited processing capabilities. Further, attacks on the controller can also stem from bugs present in the switch implementation. Cao et al. [19] proposed

buffered packet hijacking that can cause resource exhaustion on the SDN control channel, the controller and the switches, which leverages the recklessness of SDN switches that blindly process buffered packets with the buffer id of flow-mod messages and do not check the consistency of the buffered packets and the match fields. Cross-plane attacks are also an integral part of control plane attacks. Xie et al. [60] presented CrossPath attack which DoS the control channel by injecting low-rate traffic to the shared links. As for attack discovery, there are many works [14], [41] that detect attacks by testing the controllers in realistic network environments or by reading the code of the controller [9] while our method adopts the model checking which verifies the formalized model and only does further practical test when new attacks are reported, which makes the discovery process lighter and faster.

Model-based attack discovery. Prior researches demonstrate the effectiveness and practicality of employing formalized methods for uncovering attacks within network protocols. For instance, Richey et al. [61] utilized formal methods to assess traditional network vulnerabilities, while Yuan et al. [36] developed the VerioT tool based on model checking to verify IoT delegation systems. The formal methods have also been integrated into SDN security practices, with formal verification techniques being extensively applied in the SDN control plane. Canini et al. [21] examined NOX controller applications using symbolic execution, while Vinarskii et al. [58] and Lu et al. [62] explored the concurrency races problems between components in SDN network. While these studies focused on specific controller types, our work delves into universal issues arising from common design principles in the SDN paradigm. Consequently, our tool identifies attacks that are prevalent across the majority of controllers. In Section II-D and Table I, we discuss various works related to model checking in the context of SDN. It's important to note that the majority of these studies primarily focus on assessing the correctness of implementations within SDN applications. However, none of these works specifically concentrate on the formal verification of the universal mechanisms of SDN controllers. Compositional reasoning, another formal method suitable for SDN's distributed nature, enables the segmentation of model checking into simpler verification tasks for individual components [63]. Despite its relevance, this approach is more aligned with addressing network isolation issues in complex SDN networks rather than with universal SDN controller mechanisms.

VIII. Conclusion

We present an automatic method to discover attacks in SDN through formal verification with a formalized model describing SDN network process based on the universal design of SDN in order to find the exploitable attacks leveraging the common vulnerabilities shared by controllers regardless of vendors. We modeled the SDN system in formal language and performed the model checking with Spin with the assertions representing the security properties concluded from the general weaknesses of SDN. To reduce the state explosion. we adapted the model by rational assumptions and transition limitations from experience. With experiments, we found 23 unique attack paths based on our formal SDN model among which there are 2 new attacks and We verified the feasibility of the new attack in SDN systems with different controllers and different OpenFlow versions. Therefore, our method shows its lightness in the implementation of SDN model with effectiveness and efficiency in finding instructive attack paths leading to practical attacks, which proves that it can guide new insight of attacks with automated attack discovery based on the understanding of SDN universal mechanism.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No. 62372191), the National Key R&D Plan of China (No. 2022YFB3103403), the Hubei Province Key R&D Technology Special Innovation Project (No. 2021BAA032), and the Wuhan Applied Foundational Frontier Project (No. 2020010601012188).

References

[1] P. Goransson, C. Black, and T. Culver, Software Defined Networks: A Comprehensive Approach. Elsevier Science, 2016.
[2] N. McKeown, T. E. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. S. Turner, "Openflow: enabling innovation in campus networks," ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, pp. 69–74, 2008.
[3] D. Li, S. Wang, K. Zhu, and S. Xia, "A survey of network update in SDN," Frontiers of Computer Science, vol. 11, no. 1, pp. 4–12, 2017.
[4] Y. Gong, W. Huang, W. Wang, and Y. Lei, "A survey on software defined networking and its applications," Frontiers of Computer Science, vol. 9, no. 6, pp. 827–845, 2015.
[5] "OpenFlow Discovery Protocol," https://groups.geni.net/geni/wiki/OpenFlowDiscoveryProtocol, accessed: 2023-06.
[6] D. Kreutz, F. M. V. Ramos, P. J. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," Proceedings of the IEEE, vol. 103, no. 1, pp. 14–76, 2015.
[7] G. Huang and H. Y. Youn, "Proactive eviction of flow entry for SDN based on hidden markov model," Frontiers of Computer Science, vol. 14, no. 4, p. 144502, 2020.
[8] Y. Guo, F. Miao, L. Zhang, and Y. Wang, "CATH: an effective method for detecting denial-of-service attacks in software defined networks," SCIENCE CHINA Information Sciences, vol. 62, no. 3, pp. 1–15, 2019.
[9] E. Marin, N. Bucciol, and M. Conti, "An in-depth look into SDN topology discovery mechanisms: Novel attacks and practical countermeasures," in Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security, 2019, pp. 1101–1114.
[10] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in Proceedings of the 22nd Annual Network and Distributed System Security Symposium, 2015, pp. 8–11.
[11] R. Skowyra, L. Xu, G. Gu, V. Dedhia, T. Hobson, H. Okhravi, and J. Landry, "Effective topology tampering attacks and defenses in software-defined networks," in Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2018, pp. 374–385.
[12] S. Jero, W. Koch, R. Skowyra, H. Okhravi, C. Nita-Rotaru, and D. Bigelow, "Identifier binding attacks and defenses in software-defined networks," in Proceedings of the 26th USENIX Security Symposium, 2017, pp. 415–432.
[13] T. Alharbi, M. Portmann, and F. Pakzad, "The (in)security of topology discovery in software defined networks," in Proceedings of the 40th IEEE Conference on Local Computer Networks, 2015, pp. 502–505.
[14] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: detecting security attacks in software-defined networks," in Proceedings of the 22nd Annual Network and Distributed System Security Symposium, 2015, pp. 8–11.
[15] S. Shin and G. Gu, "Attacking software-defined networks: a first feasibility study," in Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking, 2013, pp. 165–166.

[16] J. Cao, Q. Li, R. Xie, K. Sun, G. Gu, M. Xu, and Y. Yang, "The crosspath attack: Disrupting the SDN control channel via shared links," in Proceedings of the 28th USENIX Security Symposium, 2019, pp. 19–36.

[17] H. Wang, L. Xu, and G. Gu, "Floodguard: A dos attack prevention extension in software-defined networks," in Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2015, pp. 239–250.

[18] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. A. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, "Rosemary: A robust, secure, and high-performance network operating system," in Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security, 2014, pp. 78–89.

[19] J. Cao, R. Xie, K. Sun, Q. Li, G. Gu, and M. Xu, "When match fields do not need to match: Buffered packets hijacking in SDN," in Proceedings of the 27th Annual Network and Distributed System Security Symposium, 2020.

[20] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. A. Porras, "DELTA: A security assessment framework for software-defined networks," in Proceedings of the 24th Annual Network and Distributed System Security Symposium, 2017.

[21] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test openflow applications," in Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, 2012, pp. 127–140.

[22] M. Kuzniar, M. Canini, and D. Kostic, "OFTEN testing openflow networks," in Proceedings of the 2012 European Workshop on Software Defined Networking, 2012, pp. 54–60.

[23] M. Zhang, G. Li, L. Xu, J. Bi, G. Gu, and J. Bai, "Control plane reflection attacks in sdns: New attacks and countermeasures," in Proceedings of the 21st International Symposium on Research in Attacks, Intrusions and Defenses, 2018, pp. 161–183.

[24] B. Yuan, F. Zhang, J. Wan, H. Zhao, S. Yu, D. Zou, Q. Hua, and H. Jin, "Resource investment for ddos attack resistant sdn: A practical assessment," SCIENCE CHINA Information Sciences, vol. 66, no. 7, pp. 172 103:1–172 103:18, 2023.

[25] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "A balance of power: Expressive, analyzable controller programming," in Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking, 2013, pp. 79–84.

[26] D. Sethi, S. Narayana, and S. Malik, "Abstractions for model checking sdn controllers," in Proceedings of the 13rd Formal Methods in Computer-Aided Design, 2013, pp. 145–148.

[27] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "Vericon: towards verifying controller programs in software-defined networks," in Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation, 2014, pp. 282–293.

[28] R. Majumdar, S. D. Tetali, and Z. Wang, "Kuai: A model checker for software-defined networks," in Proceedings of the 14th Formal Methods in Computer-Aided Design, 2014, pp. 163–170.

[29] "The nox controller," https://github.com/noxrepo/nox, accessed: 2023-12.

[30] "How to write an application in floodlight sdn controller," https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343513/How+to+Write+a+Module, accessed: 2023-12.

[31] "How to write an application in opendaylight sdn controller," https://docs.opendaylight.org/en/latest/developer-guides/developing-apps-on-the-opendaylight-controller.html, accessed: 2023-12.

[32] "Ryubook," https://book.ryu-sdn.org/en/Ryubook.pdf, accessed: 2023-12.

[33] "Spin," http://spinroot.com/spin/whatispin.html, accessed: 2023-06.

[34] "Promela," http://spinroot.com/spin/Man/promela.html, accessed: 2023-06.

[35] P. Fiterau-Brostean, B. Jonsson, K. Sagonas, and F. Tåquist, "Automata-based automated detection of state machine bugs in protocol implementations." in Proceedings of the 30th Annual Network and Distributed System Security Symposium, 2023.

[36] B. Yuan, Y. Jia, L. Xing, D. Zhao, X. Wang, D. Zou, H. Jin, and Y. Zhang, "Shattered chain of trust: Understanding security risks in cross-cloud iot access delegation," in Proceedings of the 29th USENIX Security Symposium, 2020, pp. 1183–1200.

[37] X. Bai, Z. Cheng, Z. Duan, and K. Hu, "Formal modeling and verification of smart contracts," in Proceedings of the 7th international conference on software and computer applications, 2018, pp. 322–326.

[38] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation, 2020, pp. 953–967.

[39] "The open vswitch database management protocol," https://www.ietf.org/rfc/rfc7047.txt, accessed: 2023-12.

[40] "Netconf configuration protocol," https://datatracker.ietf.org/doc/html/rfc4741, accessed: 2023-12.

[41] C. N. H. O. R. S. Samuel Jero, Xiangyu Bu and S. Fahm, "BEADS: automated attack discovery in openflow-based SDN systems," in Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses, 2017, pp. 311–333.

[42] "Open vSwitch," http://www.openvswitch.org/, accessed: 2023-06.

[43] J. Kim, M. Seo, S. Lee, J. Nam, V. Yegneswaran, P. Porras, G. Gu, and S. Shin, "Systematizing attacks and defenses in software-defined networking: A survey," Authorea Preprints, 2023.

[44] D. S. Manish Paliwal and O. Tembhurne, "Controllers in sdn: A review report," vol. 6, pp. 36 256–36 270, 2018.

[45] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," vol. 12, no. 3, pp. 14–76, 2014.

[46] C.-Y. W. Ying-Dar Lin, Yu-Kuen Lai and Y.-C. Lai, "Ofbench: Performance test suite on openflow switches," vol. 103, no. 1, pp. 2949—-2959, 2017.

[47] "Mininet," https://mininet.org/, accessed: 2023-06.

[48] "The pox network software platform," https://github.com/noxrepo/pox, accessed: 2023-06.

[49] "Ryu," https://ryu-sdn.org/, accessed: 2023-06.

[50] "Floodlight openflow controller (oss)," https://github.com/floodlight/floodlight, accessed: 2023-06.

[51] "Onos : Open network operating system," https://github.com/opennetworkinglab/onos, accessed: 2023-06.

[52] "Opendaylight controller project," https://github.com/opendaylight/controller, accessed: 2023-06.

[53] J. Kim, B. E. Ujcich, and D. J. Tian, "Intender: Fuzzing intent-based networking with intent-state transition guidance," in Proceedings of Proceedings of the 32nd USENIX Security Symposium, 2023, pp. 4463–4480.

[54] S. Lee, S. Woo, J. Kim, V. Yegneswaran, P. Porras, and S. Shin, "Audisdn: Automated detection of network policy inconsistencies in software-defined networks," in Proceedings of the 39th IEEE Conference on Computer Communications, 2020, p. 1788—1797.

[55] B. E. Ujcich, S. Jero, R. Skowyra, S. R. Gomez, A. Bates, W. H. Sanders, and H. Okhravi, "Automated discovery of cross-plane event-based vulnerabilities in software-defined networking," in Proceedings of the 27th Network and Distributed System Security Symposium, 2020.

[56] F. Xiao, J. Zhang, J. Huang, G. Gu, D. Wu, and P. Liu, "Unexpected data dependency creation and chaining: A new attack to sdn," in Proceedings of the 41st IEEE symposium on security and privacy, 2020, pp. 1512–1526.

[57] V. H. Dixit, A. Doupé, Y. Shoshitaishvili, Z. Zhao, and G.-J. Ahn, "Aim-sdn: attacking information mismanagement in sdn-datastores," in Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 664–676.

[58] E. Vinarskii, J. Lopez, N. Kushik, N. Yevtushenko, and D. Zeghlache, "A model checking based approach for detecting SDN races," in Proceedings of the 31st IFIP WG 6.1 International Conference on Testing Software and Systems, 2019, pp. 194–211.

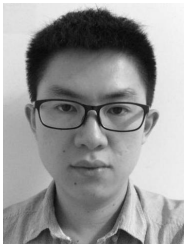
[59] "Topology zoo dataset," http://www.topology-zoo.org/dataset.html, accessed: 2023-12.

[60] R. Xie, J. Cao, Q. Li, K. Sun, G. Gu, M. Xu, and Y. Yang, "Disrupting the SDN control channel via shared links: Attacks and countermeasures," IEEE/ACM Transactions on Networking, vol. 30, no. 5, pp. 2158–2172, 2022.

[61] R. W. Ritchey and P. Ammann, "Using model checking to analyze network vulnerabilities," in Proceedings of the 21st IEEE Symposium on Security and Privacy, 2000, pp. 156–165.

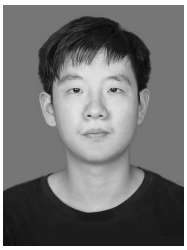
[62] G. Lu, L. Xu, Y. Yang, and B. Xu, "Predictive analysis for race detection in software-defined networks," SCIENCE CHINA Information Sciences, vol. 62, no. 6, pp. 1–20, 2019.

[63] A. Majith, O. Sankur, H. Marchand, and D. T. Bui, "Compositional model checking of an sdn platform," in Proceedings of the 17th International Conference on the Design of Reliable Communication Networks, 2021, pp. 1–8.



Bin Yuan is an Associate Professor at Huazhong University of Science and Technology (HUST), Wuhan, China. Bin received his B.S. and Ph.D degree in Computer Science and Technology from HUST in 2013 and 2018, respectively. His research interests include software-defined network security, network function virtualization, cloud security, privacy and IoT security. He has published several technical papers in top conferences/journals, such as USENIX Security, CCS, IEEE TSC, IEEE TNSM, IEEE TNSE, IEEE IoT Journal and FGCS.

Chi Zhang is currently a master student at Huazhong University of Science and Technology (HUST), Wuhan, China. Chi received his B.S. degree in Software Engineering from University of Electronic Science and Technology of China in 2021. His research interests include software-defined network and SDN security.



Jiajun Ren is currently master degree candidate in School of Cyber Science and Engineering at Huazhong University of Science and Technology(HUST), Wuhan, China. He received his B.S degree in Cyber Science and Engineering from Huazhong University of Science and Technology in 2021. His research interests include network protocol security and SDN security.

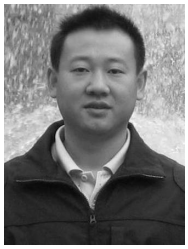
Qunjinming Chen is currently master degree candidate in School of Cyber Science and Engineering at Huazhong University of Science and Technology(HUST), Wuhan, China. He received his B.S degree in Cyber Science and Engineering from Huazhong University of Science and Technology in 2022. What he interests in include network protocol and IoT security.

Biang Xu is currently master degree candidate in School of Cyber Science and Engineering at Huazhong University of Science and Technology(HUST), Wuhan, China. He received his B.S degree in Cyber Science and Engineering from Huazhong University of Science and Technology in 2023. What he interests in include network protocol and IoT security.

Fan Zhang is currently a master student at Huazhong University of Science and Technology (HUST), Wuhan, China. Fan received her B.S. degree in Information Security from Hefei University of Technology in 2020. Her research interests include software-defined network and network security.

Qiankun Zhang is an associate researcher at Huazhong University of Science and Technology (HUST). He obtained his Ph.D. degree from the University of Hong Kong (HKU) in 2021, and before that he received his Bachelor's degree for Zhejiang University (ZJU) in 2017. He is broadly interested in theoretical computer science, and more specifically in design and analysis of approximation and online algorithms. He has published several papers in top theoretical computer science conferences including STOC and FOCS.

Zhen Li is an Associate Professor at Huazhong University of Science and Technology, Wuhan, China. She received the Ph.D. degree in Cyberspace Security at Huazhong University of Science and Technology in 2019. She was a Postdoctoral Fellow at the University of Texas at San Antonio, USA, from 2019 to 2021. She is a member of the IEEE and a member of the ACM. Her research interests mainly include software security and artificial intelligence security.



Deqing Zou is a Professor of Computer Science at HUST. He received his PH.D at HUST in 2004. His main research interests include system security, trusted computing, virtualization and cloud security. He has been the leader of one "863" project of China and three NSFC (National Natural Science Foundation of China) projects, and core member of several important national projects, such as National 973 Basic Research Program of China. He has applied almost 20 patents, published two books (one is entitled "Xen virtualization Technologies" and the other is entitled "Trusted Computing Technologies and Principles") and more than 50 High-quality papers, including papers published by IEEE Transactions on Dependable and Secure Computing, IEEE Symposium on Reliable Distributed Systems and so on. He has always served as a reviewer for several prestigious Journals, such as IEEE TPDS, IEEE TOC, IEEE TDSC, IEEE TCC, and so on. He is on the editorial boards of four international journals, and has served as PC chair/PC member of more than 40 international conferences.

Hai Jin is a Cheung Kung Scholars Chair Professor of computer science and engineering at HUST. Jin received his PhD in computer engineering from HUST in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. Jin worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security.

Jin is a Fellow of IEEE and CCF, and a member of the ACM. He has co-authored 22 books and published over 700 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.