IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 30, 2020, accepted February 19, 2020, date of publication March 10, 2020, date of current version March 25, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2979894

Formal Verification of SDN-Based Firewalls by Using TLA+

YOUNG-MI KIM[1] **AND MIYOUNG KANG**[2]
[1]Department of Computer and Radio Communication Engineering, Korea University, Seoul 02841, South Korea
[2]Graduate School of Information Security, Korea University, Seoul 02841, South Korea

Corresponding author: Miyoung Kang (mykang@formal.korea.ac.kr)

This work was supported in part by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT under Grant 2017M3C4A7083676, and in part by NRF Grant funded by the Korean Government (MSIT) under Grant NRF-2018R1A2B6009122.

ABSTRACT Software-defined networking (SDN) has generated increased interest due to the rapid growth in the amount of data generated by the development of the Internet and communications, the commercialization of 5G, and increasingly complex networks. While SDN is more advantageous than traditional networks in terms of efficient network management, rapid deployment, and dynamic scalability, the correctness of a network configuration must be ensured in advance. In other words, SDN components such as network devices, SDN controllers, and applications need to be deployed correctly and must be free of rule conflicts, particularly between various application policies; otherwise, it may result in network paralysis in the worst case. This paper assumes that the SDN network is free of rule conflicts when the rules in the SDN switches correctly obey firewall application or policies. To solve this problem, this paper proposes a verification framework for SDN using TLA+. We show that the firewall rule behavior of switches can be formalized using TLA+, and this is verified with the TLC model checker that uses TLA+ as the model description language. We check two different types of topology models through our verification framework to ensure that the same firewall rules are maintained even if the topology changes. The findings show that the firewall rules may be inconsistent as the topology changes.

INDEX TERMS Firewall, formal methods, software-defined networking, TLA+.

I. INTRODUCTION

Software-defined networking (SDN) has been proposed to address problems associated with traditional physical network devices, such as difficult manageability, low configurability, and limited scalability. Because SDN technology has several advantages over physical networks, many technology companies such as Google, Facebook, and Amazon have adopted SDN to manage their networks [1]. SDN technology separates physical network devices (the data plane) from the control of network operations using an SDN controller situated between the network devices and network applications. Network policies can be flexibly controlled and network configuration can be quickly modified because software governs network management in SDN.

However, SDN also has some disadvantages. For example, because the applications for the application plane can issue different packet-processing rules to the SDN controller, any inconsistencies among the rules may impair efficient SDN operations. If conflicting rules transfer from the firewall application to the SDN controller, the controller can become stuck, paralyzing the entire network. Thus, a significant volume of academic and industry-based research has focused on solving this rule conflict problem in SDN networks. Formal methods have been suggested to mathematically prove that the application rules are conflict-free. NICE [2], HAS [3], Kuai [4], Veriflow [5], SDNRacer [6], and VeriSDN [7]–[9] have leveraged formal methods to verify the correctness of an SDN network. Our work uses formal methods to verify that there are no rule conflicts in the SDN network in terms of SDN switch rules that correctly implement firewall applications or policies.

In this paper, we propose a novel approach to prove the correctness of an SDN network that there are no rule conflicts using TLA+ [10]. TLA+ is a formal language based on temporal logic and ordinary mathematics that can be used to uncover design flaws in network systems. TLA+ was initially developed to specify and verify concurrent systems, but it was

The associate editor coordinating the review of this manuscript and approving it for publication was Giacomo Verticale.

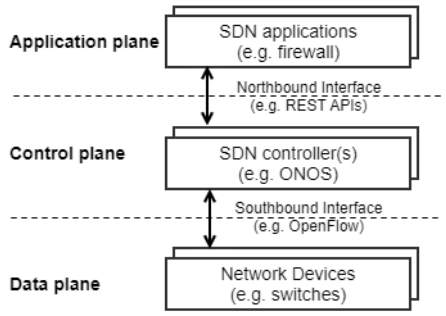

FIGURE 1. SDN architecture. SDN separates the data plane from the control plane so the network can change quickly and dynamically. Modified from [14].

later demonstrated that it can be employed in the verification of a wide class of systems, from program interfaces (APIs) to distributed systems [10]. Many companies, including Amazon, have used TLA+ to ensure the correctness of their network [11], [12].

The main contribution of this study is that we propose detailed rules for the specification of SDN network configurations using TLA+. This includes the specification of network hardware components, packet switch rules, and SDN firewall rules with TLA+. We also demonstrate the effectiveness of our approach to verify the correctness of an SDN network by specifying an example network into TLA+. The results prove that the rules in the SDN do not conflict with the rules of the firewall to be implemented by using the TLC tool [13], which is a model checker incorporated into the TLA+ Toolbox.

This paper is structured as follows. Section 2 provides background information on SDN and TLA+ and summarizes past work related to our research. Section 3 proposes a verification framework presenting the rules on how to describe SDN networks with TLA+, while Section 4 demonstrates the verification process using the TLC model checker. Finally, we provide conclusions in Section 5.

II. BACKGROUND

We briefly introduce SDN and TLA+ in this section. We also describe previous work related to the formal analysis of SDN.

A. SDN

SDN is a technology that separates the data plane and the control plane in a network and controls the network using software (Fig. 1). In contrast, in a traditional network, network control and the data transfer are conducted together in one network device. Because network control is programmable in SDN, complicated network configurations are easily configurable and network topology can be modified effectively.

As shown in Fig. 1, the core of an SDN network is the SDN controller, which acts as a type of operating system (OS). One side of the SDN controller is connected to the network devices and the other side to the applications. The controller uses a protocol such as OpenFlow to communicate with the data plane. OpenFlow is an implementation of SDN technology [15], [16]. An OpenFlow switch classifies a packet by flow and transmits it to an adjacent switch that maintains flow tables containing flow entries. The flow entries include information about the packet delivery path and method.

TABLE 1. Main components of a flow entry in a flow table (Source: OpenFlow Switch Specification, Version 1.5.1 [15]).

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags

When a packet arrives, the OpenFlow switch processes the flow according to the flow entry if there is a matching flow in the flow table, such as the match fields and instructions presented in Table 1; if not, it sends the packet to the controller. The controller passes the packet control information to the OpenFlow switch. This information is stored in the flow table, and the packet is processed according to the flow. The controller determines the control information for the packet in accordance with the policies of various applications. At this time, conflicts between the policies of different applications may occur. When multiple switches have rules based on the network topology, rule conflicts may mean that packets are no longer transferred to the next switch. As a basis for solving this problem, we focus on rule conflicts as to determine whether the rules on the SDN switch implement a single application correctly.

B. TLA+

TLA+ is a formal specification language that employs ordinary mathematics, such as propositional logic, sets, and predicate logic, to design and model concurrent systems. Most TLA+ specifications describing what systems may do have the following simple form:

$$Spec \triangleq Init \wedge \Box[Next]_{vars} \tag{1}$$

where *Spec* denotes a specification of the system to be described, *Init* is the initial condition, *Next* is the next-state relation, and *vars* is the tuple of all variables [17]. *Init* is represented by a state formula describing the initial state(s). A state is an assignment of values to the variables. *Next* is represented by an action formula describing the allowed transitions of a system. *Next* is usually the disjunction $A_1 \vee A_2 \vee \ldots \vee A_n$, where each A_i is a possible action. In (1), $[Next]_{vars}$ means that *Next* has stuttering transitions that do not change the values of the variables defined in tuple *vars*. The tuple $\langle e_1, e_2, \ldots, e_n \rangle$ represents the n-tuple whose i-th element is e_i, and $e[i]$ denotes the i-th element of tuple e. $\Box F$, where F is a state predicate, is a temporal formula, and the temporal operator $\Box$ means that F is always true.

For any variable v, v has different meanings depending on whether it is primed or not. A primed variable represents its value in the successor state of the transition, whereas an unprimed variable represents its value in the state before the transition. As a result, the expression $v' = v$ indicates that the value of v in the old state equals the value of v in the new

state. UNCHANGED v in TLA+ is shorthand for the expression $v' = v$.

For denotation of the power set of a set, the keyword SUBSET is a built-in operator of TLA+. For example, SUBSET $\{1, 2\}$ is $\{\{\}, \{1\}, \{2\}, \{1, 2\}\}$. The unary operator UNION, a built-in operator of TLA+, denotes the union of the elements of a set. For example, UNION $\{\{1\}, \{2\}, \{1, 2\}\}$ is $\{1, 2\}$.

TLA+ provides miscellaneous constructs for the selection of a value according to the conditions and for the definitions to be used in a local context. The construct IF p THEN e_1 ELSE e_2 means choosing e_1 if p, otherwise e_2, where p, e_1 and e_2 are expressions. The construct LET $d_1 \triangleq e_1 \ldots d_n \triangleq e_n$ IN e means e in the context of the definitions, where each d_i is a definition identifier and each e_i is an expression. For example, the operator

$$
\begin{array}{l}
ExOp(x, y) \triangleq \\
\quad \text{LET} \quad Rule \triangleq \langle \langle \text{“}a\text{”}, \text{“}r\text{”} \rangle, \langle \text{“}b\text{”}, \text{“}s\text{”} \rangle, \langle \text{“}c\text{”}, \text{“}u\text{”} \rangle \rangle \\
\qquad op(a1, a2) \triangleq \text{IF } a1 = Rule[a2][1] \\
\qquad\qquad\qquad\quad \text{THEN } \langle Rule[a2][2] \rangle \\
\qquad\qquad\qquad\quad \text{ELSE } \langle\ \rangle \\
\quad \text{IN } op(x, y)
\end{array}
$$

defines a tuple *Rule* and an operator *op* with arguments $a1$ and $a2$ in the LET/IN construct to return the second element of an y-th pair in *Rule* if the first element of the y-th pair in *Rule* equals argument x, and an empty tuple, otherwise.

When we specify a system, multiple values occasionally need to be grouped as a single expression. In this case, a record expression is appropriate. In TLA+, the expression $[h_1 : S_1, \ldots, h_n : S_n]$ refers to the set of all records with the h_i field in S_i. Mathematically, a record is a function whose domain is a set of strings. In TLA+, the function expression is $f[x]$ but not $f(x)$. It expresses the value that the function f assigns to each element x of its domain. Thus, the h-field of record e is represented by $e[\text{“}h\text{”}]$, which has the same meaning as $e.h$. The expression used to assign a value to each field of a record is $[field_1 \mapsto value_1, \ldots, field_n \mapsto value_n]$ where $field_i$ is the identifier of the i-th field and $value_i$ is its value. The EXCEPT construct in TLA+ describes a function that is almost the same as another function [10]. For any function f, the expression $[f$ EXCEPT $![c] = e]$ equals the expression

$$[x \in \text{DOMAIN} f \mapsto \text{IF } x = c \text{ THEN } e \text{ ELSE } f[x]]$$

where c is any field of f, e is an expression and DOMAIN f denotes the domain of f. The keyword DOMAIN is provided in the TLA+ standard module. For any sets D and R, the set of all functions whose domain equals D and whose range is any subset of R is written as $[D \to R]$.

More details on the syntax and semantics are described in the TLA+ book [10].

C. RELATED WORK

A number of previous studies have applied formal methods to SDN.

The Nox OpenFlow controller developed by NICE Work [2] is an error detection tool for programs uploaded to the OpenFlow controller. This tool combines model checking and symbolic execution techniques to reduce the number of states. This tool can check forward loops, the presence of black holes, path reachability, and packet loss related to the problems that occur when forwarding packets via devices in SDN networks. Our work uses model checking to verify rule conflicts between an SDN application and its implemented switch rules.

As a static analysis tool, Header Space Analysis (HAS) [3] uses the content of the packet header when analyzing network operations. The validation attributes are forward loop, reachability, and traffic segmentation. Our approach is also based on an analysis of a packet header.

Veriflow [5] monitors the status of the network environment and data plane. When a forwarding rule is inserted, it checks for invariance violations throughout the network and tracks all forwarding state changes.

Kuai [4] monitors the given SDN properties in real time, reducing state spaces while model checking uses a partially ordered set. This tool uses Murphi [18] as an input language. SDNRacer [6] is a dynamic analyzer that operates on actual traces and can quickly detect concurrency issues such as the root cause of many bugs.

VeriSDN [7], [9] is a framework verifying the firewall application from among SDN applications by using pACSR [9], an extension of the algebra of communicating shared resources (ACSR) [19], [20]. It verifies the rule conflicts that occur due to the multiple applications executed within the SDN controller. In VeriSDN, pACSR is used for formal modeling, and its correctness is verified with the VeriFM model checker. It can be used to formally model and verify design sketches for OpenFlow in the early stages.

Our work uses the TLA+ specification language, which can be used to specify network protocols for various tasks, such as Paxos for solving consensus in distributed systems [21]. We also use the TLC model-checking tool for TLA+ specifications to verify whether the rules in switches are consistent with the application policies by changing only the topology and the rules in TLA+.

Our work here is an extension of our previous research [22]. Specifying SDN-based firewalls using TLA+ was first attempted in our previous work where we specified the behavior of processing packets according to rules on a single switch using TLA+ and verified that the behavior is in accordance with the firewall policy.

In this study, we propose a framework for specifying and verifying the behavior of rule-based processing of packets coming into multiple connected switches using TLA+. We also use the proposed framework to verify whether the same rules work consistently with the firewall policy even if the topology changes; the rules in the SDN switches

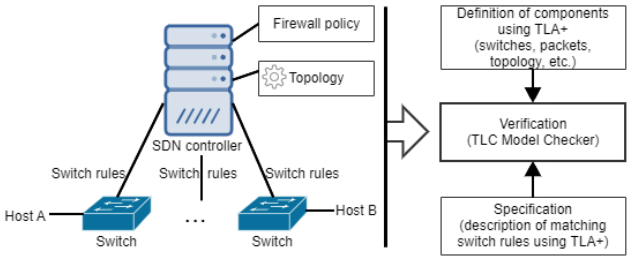

FIGURE 2. Verification framework. This verification framework consists of the definition of the main components of SDN, the specification describing the matching switch rules, and the verification of consistent policies and rules.

can change only to branch the forwarding path by the topology.

III. VERIFICATION FRAMEWORK FOR SDN USING TLA+

A. OVERVIEW

The SDN controller acts as a mediator for various applications and creates a flow table reflecting the policies established by each application. The SDN controller sets rules based on these policies. If a switch receives a packet whose flow entry does not appear in the flow table, it sends a request to the controller for the corresponding flow. The controller then updates the flow table for that switch. In this paper, we describe the operation of the flow table used by the SDN's southbound protocol OpenFlow. We assume that the flow for all packets exists in the flow table, so flow requests from the switch to the controller are not covered here. This paper also assumes that there is a single flow table for each switch.

In Fig. 2, each switch has its own rules created by the SDN controller according to its firewall policy and topology. Packets from Host A pass through the switches according to their rules and arrive at Host B outside the network. We propose a verification framework to check for consistency of the rules with the policies using the TLC model checker after defining the network components and specifying the handling of the packets by switch rules using TLA+.

B. COMPONENTS OF THE FRAMEWORK

The main network components we are interested in are the switches, topology, and packets. Each switch has its own rules, and the decision to allow or drop a packet coming into the switch is in accordance with these rules. Therefore, we define the switches, packets, topology, and switch rules to specify SDN-based firewalls behavior. We usually use a record type that can contain various values and a tuple type for the expression of the order to express the network information. We have defined the framework's components as much as possible using the built-in operators of TLA+, but we have also used some operators defined in TLA+ standard modules, such as *Naturals*. Standard module *Naturals* defines operator . . and the other usual operators on natural numbers. $0..n$ denotes the set of natural numbers from 0 to n. For example, $1..5$ equals $\{1, 2, 3, 4, 5\}$.

In this section, we define the components of the framework generally, so we do not describe all of the defined information in a concrete specification, depending on the component, such as the fields of a packet.

1) SWITCHES

Each switch is represented by a unique identifier, and all switches within the network are defined as a set of switch identifiers. When the entire switch set S is composed of n switch identifiers, the set S is defined as

$$S \triangleq \{swId_1, swId_2, \ldots, swId_n\}$$

2) TOPOLOGY

The network topology represents the connections between the switches in a network. Switches are connected to physical ports. We use a record type for the connections between the switches constituting a particular topology, and we call each connection of a record type a connection record. A connection record will contain information about the two target switches (s and t) and their respective ports (sp and tp). The direction of the switch connection is from s to t, which means that sp is an output port on the s switch side and tp is an input port on the t switch side. The physical ports (sp and tp) for the switches are natural numbers ranging from 0 to the maximum number of ports (P_{max}) on the switches. The switches in the topology are defined using the elements of switches set S. Some switches in the topology are connected to devices outside the network. In our study, we use NW instead of a switch identifier and port number 0 to represent a port on a device outside the SDN network.

We define C as the set of all possible connection records for a set of switches as follows:

$$C \triangleq [s : S \cup \{\text{“}NW\text{”}\}, sp : 0..P_{max}, \\ t : S \cup \{\text{“}NW\text{”}\}, tp : 0..P_{max}]$$

We define a valid topology T as a subset of C that satisfies the following conditions:

- $T \subseteq C$, where $T \neq \{\}$.
- at least one ingress switch exists: $\exists c \in T : c.s = \text{“}NW\text{”} \wedge c.sp = 0 \wedge c.t \in S \wedge c.tp \in 1..P_{max}$.
- at least one egress switch exists: $\exists c \in T : c.s \in S \wedge c.sp \in 1..P_{max} \wedge c.t = \text{“}NW\text{”} \wedge c.tp = 0$.
- no connection forms a loop: $\forall c \in T : c.s \neq c.t$
- there are no cycles: $\neg(\exists c_1, c_2, \ldots, c_n \in T : c_1.s \neq \text{“}NW\text{”} \wedge c_1.t = c_2.s \wedge c_2.t = c_3.s \wedge \cdots \wedge c_{n-1}.t = c_n.s \wedge c_n.t = c_1.s)$, where $c_i \neq c_j$ and $c_i.s, c_i.t \in S$ for each connection c_i and c_j $(1 \leq i, j \leq n)$.
- at least one path from an ingress switch to an egress switch exists: $\exists c_1, c_2, \ldots, c_n \in T : c_1.s = \text{“}NW\text{”} \wedge c_1.t = c_2.s \wedge c_2.t = c_3.s \wedge \cdots \wedge c_{n-1}.t = c_n.s \wedge c_n.t = \text{“}NW\text{”}$, where $c_i \neq c_j$ and $c_i.s, c_i.t \in S$ for each connection c_i and c_j $(1 < i, j < n)$.

For example, let's suppose that there are switches $swId1$ and $swId2$ whose maximum port number is 2. The set of all

connection records with these switches is the following:

$$[s : \{\text{“}swId1\text{”}, \text{“}swId2\text{”}, \text{“}NW\text{”}\}, sp : 0..2,$$
$$t : \{\text{“}swId1\text{”}, \text{“}swId2\text{”}, \text{“}NW\text{”}\}, tp : 0..2]$$

A valid topology for the above connections is shown in Fig. 3.

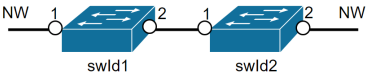

FIGURE 3. An example of a topology that has connections between two switches and to the outside of the network.

Fig. 3 presents a topology in which Ports 1 and 2 of the switch $swId1$ are connected to NW and Port 1 of $swId2$, respectively, and Port 2 of the switch $swId2$ is connected to NW (an external network connection), so its topology ExT is defined as follows:

$$ExT \triangleq \{[s \mapsto \text{“}NW\text{”}, sp \mapsto 0, t \mapsto \text{“}swId1\text{”}, tp \mapsto 1],$$
$$[s \mapsto \text{“}swId1\text{”}, sp \mapsto 2, t \mapsto \text{“}swId2\text{”}, tp \mapsto 1],$$
$$[s \mapsto \text{“}swId2\text{”}, sp \mapsto 2, t \mapsto \text{“}NW\text{”}, tp \mapsto 0]\}$$

For the topology, we also define information on the ingress switches to simplify our specification even though the ingress switches can be extracted from a given topology T. We define each ingress switch as a tuple consisting of a switch identifier and an input port number. The definition of the ingress switches information I is as follows:

$$I \triangleq \{\langle c.t, c.tp\rangle : c \in T \wedge c.s = \text{“}NW\text{”} \wedge c.sp = 0\}$$

3) PACKETS

A basic firewall decides whether to allow or drop a packet based on the 5-tuple information of the packet, including the source IP address, destination IP address, protocol, source port, and destination port. To define a packet, we use the 5-tuple information of a packet and a record as the appropriate data structure to contain the information.

Let PF be the set of all fields that we can use for a packet record. The definition of PF is as follows:

$$PF \triangleq \{\text{“}srcIP\text{”}, \text{“}dstIP\text{”}, \text{“}srcPort\text{”}, \text{“}dstPort\text{”},$$
$$\text{“}proto\text{”}\}$$

where “$srcIP$” represents the source IP address, “$dstIP$” is the destination IP address, “$srcPort$” is the source port number, “$dstPort$” is the destination port number, and “$proto$” is the protocol.

Let P be the set of possible packet records created with elements in PF. P is the union of the sets of records containing different numbers of fields. The definition of P is

$$P \triangleq \{p \in \text{UNION}\{[FS \to \text{UNION}\{V_{srcIP}, \ldots, V_{proto}\}] :$$
$$FS \in \text{SUBSET}PF \setminus \{\}\} :$$
$$\forall d \in \text{DOMAIN}\, p :$$
$$(d = \text{“}srcIP\text{”} \wedge p[d] \in V_{srcIP})$$
$$\vee \cdots \vee$$
$$(d = \text{“}proto\text{”} \wedge p[d] \in V_{proto})\}$$

where FS is each domain of possible packet records, d is an element of the domain of each packet record p, and each V_d is a set of values corresponding to each element of PF. This paper does not explicitly define the types of values for fields. As an example, the packet field $srcIP$ is defined not by an actual IP address but by a number such as $srcIP$ mapping to 1. As shown in the P definition, it is not mandatory to use all the elements defined in the set PF when we define a packet record. In other words, we can define a packet record with more than one field, if necessary. Our concrete specification also shows that we are not using all packet fields. For example, if the source IP address and the destination IP address of a packet are 1 and 2, respectively, and the protocol is TCP, the expression is

$$[srcIP \mapsto 1, dstIP \mapsto 2, proto \mapsto \text{“}TCP\text{”}]$$

where there are no definitions for ports $srcPort$ and $dstPort$.

4) SWITCH RULES

An OpenFlow switch has an OpenFlow table that stores the rules defined by the SDN controller. The flow entries in the flow table consist of match fields, priorities, counters, instructions, timeouts, cookies, and flags. In this paper, we describe the match fields, priorities, and instructions directly related to the handling of a packet according to the rules.

A rule can be defined by expressing it as a record, which we call a rule record. Let RF be the set of possible fields of rule records. The definition of RF is

$$RF \triangleq \{\text{“}mf\text{”}, \text{“}a\text{”}, \text{“}o\text{”}\}$$

where “mf” is the match field, “a” is the instruction (i.e., an action), and “o” is a set of output ports. In this study, we abstract the rule priority to ordered rules using a tuple in which the preceding elements have higher priority than the elements that follow. Therefore, even if we do not define the priority value directly, the priority is indirectly reflected when we express the rules as a tuple.

For match field mf, we use a record type, and the fields of a defined packet are used to configure the matching field for the rules. Let M be the set of records for a matching field that we can create with elements in PF defined in the previous section. The definition of M is

$$M \triangleq \{m \in \text{UNION}\{[FS \to \text{UNION}\{V_{srcIP}, \ldots, V_{proto}\}] :$$
$$FS \in \text{SUBSET}PF \setminus \{\}\} :$$
$$\forall d \in \text{DOMAIN}\, m :$$
$$(d = \text{“}srcIP\text{”} \wedge m[d] \in V_{srcIP})$$
$$\vee \cdots \vee$$
$$(d = \text{“}proto\text{”} \wedge m[d] \in V_{proto})\}$$

where FS is each domain of possible matching field records, d is an element of the domain of each matching field record m,

and each V_d is a set of values corresponding to each element of PF. Additionally, when we set a rule, an action occasionally needs to be applied to all packets. Therefore, we add the field *all*, meaning all packets, to the matching field. The value of the field *all* is fixed as " $*$ " and the expression is $[all \mapsto \text{“}*\text{”}]$.

For action field a, we abstract the actions as "*outport*" and "*drop*" actions in this paper; "*outport*" and "*drop*" actions mean forwarding packets to all next switch(es) connected to the output ports in o and dropping packets at the current switch, respectively. Field o is represented as a set because there may be more than one output port. When the instruction for a packet is an output port action, field o is defined as the set of ports on the current switch to which a packet is forwarded. For a drop action, field o is defined as $\{0\}$.

Let *Rules* be a set of all possible rule records we can create. The definition of *Rules* is as follows:

$$\begin{aligned} Rules \triangleq\ & [mf : M \cup \{[all \mapsto \text{“}*\text{”}]\}, \\ & a : \{\text{“}outport\text{”}, \text{“}drop\text{”}\}, o : \text{SUBSET } 0..P_{max} \setminus \{\}] \end{aligned}$$

where P_{max} represents the maximum port number of switches as in the definition of topology.

Let R_s be the ordered rules for switch s. The priority is the order in which rules are applied to packets when they are processed. As we noted above, we use a tuple type as the priority for the rules. The definition of R_s is as follows:

$$R_s \triangleq \langle r_1, r_2, \ldots, r_m \rangle$$

where each r_i denotes i-th element (rule) of the m-tuple (rules) and m represents the number of rules in switch s. Each rule is an element of *Rules*.

Lastly, let *SR* be ordered rules for switches $\{s_1, s_2, \ldots, s_n\}$. All switches used in the topology to analyze must define their rules. The definition of *SR* is as follows:

$$SR \triangleq [s_1 \mapsto R_{s_1}, s_2 \mapsto R_{s_2}, \ldots, s_n \mapsto R_{s_n}]$$

where each R_{s_i} denotes the ordered rules for switch s_i.

The *ExSR* represents an example of defining the rules for switches *swId*1 and *swId*2:

$$\begin{aligned} ExSR \triangleq\ & [swId1 \mapsto \langle [mf \mapsto [srcIP \mapsto 1], \\ & \quad a \mapsto \text{“}outport\text{”}, o \mapsto \{1\}], \\ & \quad [mf \mapsto [all \mapsto \text{“}*\text{”}], \\ & \quad a \mapsto \text{“}drop\text{”}, o \mapsto \{0\}]\rangle \\ & swId2 \mapsto \langle [mf \mapsto [dstIP \mapsto 2], \\ & \quad a \mapsto \text{“}outport\text{”}, o \mapsto \{1, 2\}], \\ & \quad [mf \mapsto [all \mapsto \text{“}*\text{”}], \\ & \quad a \mapsto \text{“}drop\text{”}, o \mapsto \{0\}]\rangle] \end{aligned}$$

C. SPECIFICATION OF FIREWALL RULES

For convenience, this section describes our long specification of an SDN with firewall rules in two subsections.

1) CONSTANTS AND VARIABLES FOR SDN BEHAVIORS

The first requirement when specifying the behavior of a switch is to identify which elements can be declared as constants and which ones are variables. Fig. 4 presents the declaration of the constants and variables required to describe the SDN behaviors. We provide the specification for a network with switches with a maximum port number of 5.

```
EXTENDS Naturals, Sequences, FiniteSets

CONSTANTS Packet, Switch, Topology, SwitchRule, IngressSwitch
CONSTANTS Match(_, _)
VARIABLES sw, swPkt, swPktQ, pktQ, pendingPkt

vars ≜ ⟨sw, swPkt, swPktQ, pktQ, pendingPkt⟩

PortNRange ≜ 0 .. 5
Action ≜ {"norule", "outport", "drop"}
SwStatus ≜ {"rdy", "rdyA", "busy"}
ProPacket ≜ [p : Packet, i : PortNRange, a : Action,
             o : PortNRange, r : Seq(Switch)]

NoPacket ≜ CHOOSE v : v ∉ ProPacket
NoAction ≜ CHOOSE v : v ∉ Action
NoTopology ≜ CHOOSE v : v ∉ Topology
```

FIGURE 4. Declaration of constants and variables. Constants and variables are declared before they are used.

The first line in Fig. 4 indicates that we use TLA+ standard modules *Naturals*, *Sequences*, and *FiniteSets* in our specification. As mentioned in the previous section, the operator .. is defined in the *Naturals* module. The *Sequences* module defines $Seq(S)$, $Append(s, e)$, and $Head(s)$ for a set S, a sequence s, and an element e. $Seq(S)$ denotes the set of all finite sequences of elements in S. $Append(s, e)$ denotes the sequence obtained by appending e to the end of sequence s. $Head(s)$ denotes the first element of sequence s. The *FiniteSets* module defines the $Cardinality(S)$ operator denoting the cardinality of a given finite set S.

During the operation of an SDN-based firewall, the switch receives the firewall rules for packet processing from the SDN controller and stores them in the flow table. In this study, it is assumed that the rules for all packet inputs are stored at each switch. In other words, situations in which the flow table information at a switch changes dynamically is beyond the scope of this paper. Therefore, the required information in this research is the packets to be processed, the switches transmitting the packets, the topology depicting the connection of the switches, and the rules for each switch. This information is declared as constants *Packet*, *Switch*, *Topology*, and *SwitchRule*, respectively, because they are considered to be unchanging when specifying the switch operation. Additionally, we define the constant *IngressSwitch* in relation to the topology, which denotes information about the ingress switches.

Packets entering the network are matched with the rules defined at each switch and processed according to these rules. This matching function indicates whether the flow is defined

in a rule. We declare it as the constant operator *Match*(_, _), which matches two arguments and returns TRUE or FALSE.

In addition to the constants, several variable declarations are needed to describe the packet handling behavior. The variables used in the specification are *sw* for the status information of the switch, *swPkt* for the packet to be processed at each switch, *swPkQ* for the packets waiting for processing at each switch, *pendingPkt* for packets waiting at the ingress switch, and *pktQ* containing the processing result information for all input packets. Because the packets are processed in order, *swPkQ* and *pktQ* are tuples. Each processed packet in *pktQ* is a record containing five fields:

- p as an incoming packet of a switch.
- i as the input port number for a switch.
- a as the action (i.e., the instruction for the packet).
- o as the output port number for the execution of action a.
- r as the tuple that stores identifiers of the switches through which the packet passes.

Except for p, these fields represent information that changes every time a packet passes through a switch, with the processed packet information contained in *pktQ* representing the last information to change. The type of variables in the TLA+ specification is determined by *Init* described in the next section.

We define the information in the declaration. To limit the range of values assigned to variables according to our needs, we define the sets *PortNRange*, *Action*, *SwStatus* and *ProPacket*, which represent the range of ports, the action set of rules, the status set of a switch and the form of packets being processed, respectively. *ProPacket* is a record type and contains information about the switches a packet passes through so we can trace the packet after checking the model.

At the end of the declaration, arbitrary values that do not belong to a specific value set, *NoPacket*, *NoAction*, and *NoTopology*, are explicitly defined.

2) SPECIFICATION OF SDN BEHAVIORS

Packet processing in the switch consists of an action indicating the initial state and an action in which the state changes over time or a stuttering situation occurs. The specification *Spec* of packet processing in the switches is defined as follows:

$$Spec \triangleq Init \wedge \Box [Next]_{vars}$$

In TLA+, constants, variables, and operators must be declared before they are used. The constants or variables, and some operators that appear in this section are declared in the previous section.

Init defines the initial states of the declared variables (Fig. 5). The initial state of each switch is set to "rdy." The variable *swPkt* for the packet to be processed in each switch is defined as *NoPacket* because there is no packet being processed at each switch in the initial state. In addition, both the variables *swPktQ* for packets waiting for processing in each switch and *pktQ* containing the processing result

$$\begin{aligned} Init \triangleq\ & \wedge sw = [s \in Switch \mapsto \text{“rdy”}] \\ & \wedge swPkt = [s \in Switch \mapsto NoPacket] \\ & \wedge swPktQ\ = [s \in Switch \mapsto \langle\rangle] \\ & \wedge pktQ = \langle\rangle \\ & \wedge pendingPkt = Packet \end{aligned}$$

$$\begin{aligned} &PacketType \triangleq [p : Packet,\ i : PortNRange, \\ &\quad a : Action \cup \{NoAction\},\ o : PortNRange,\ r : Seq(Switch)] \\ &TypeInvariant \triangleq \\ &\quad \wedge sw \in [Switch \to SwStatus] \\ &\quad \wedge swPkt \in [Switch \to PacketType \cup \{NoPacket\}] \\ &\quad \wedge swPktQ \in [Switch \to Seq(PacketType)] \\ &\quad \wedge pktQ \in Seq(ProPacket) \\ &\quad \wedge pendingPkt \in \text{SUBSET}\ Packet \end{aligned}$$

FIGURE 5. Definitions of *Init* and *TypeInvariant*.

information for all input packets are defined as empty tuples because there are no packets stored in the initial state.

Like *ProPacket*, *PacketType* is also the form of packets being processed but it can represent the initial state. We define *PacketType* simply to represent a type invariant.

The values of the declared variables are altered by the *Init* and *Next* actions. We define *TypeInvariant* as a type invariant to describe the values that the variables can assume in a behavior that satisfies the specification. The state of the switch must be an element of the defined *SwStatus*. The *swPkt* of each switch should be an element of the union of *PacketType* and {*NoPacket*} because the type of packet handled by the switch is represented by *ProPacket* (which is the same as *PactketType*, except for the set of values of the action field), but the initial state is *NoPacket*. The queue of packets to be processed in each switch and the processed packet queue should be tuples composed of *ProPacket*(*PactketType*), i.e., an element of *Seq*(*ProPacket*).

The *Next* action is composed of actions *EnPacket*, identifying the packet entering the ingress switch, *ExPacket*, which extracts packets waiting to be processed at the switch, and *SwAct*, which processes each packet according to the rules (Fig. 6). The *Next* action is formally expressed as follows:

$$\begin{aligned} Next \triangleq\ & \exists s \in Switch : \\ & \vee \exists n \in PortNRange : EnPacket(s, n) \\ & \vee ExPacket(s) \vee SwAct(s) \end{aligned}$$

The *Next* action will work if switch s meets the conditions for some of the actions. However, to enable the *EnPacket* action, input port number n on switch s into which the packet comes is required.

The *EnPacket* action is enabled when the switch is in state "rdy" and is an ingress switch. Of course, information for this switch and the input port number should be defined as an ingress switch which exists in the topology. The ingress switch is searched using our *NextSW* operator as follows:

$$\begin{aligned} &NextSW(x, y) \triangleq \\ &\quad \text{IF } \exists to \in Topology : to.s = x \wedge to.sp = y \text{ THEN} \end{aligned}$$

```
EnPacket(s, n) ≜
  LET ns ≜ NextSW("NW", 0)
  IN  ∧ sw[s] = "rdy"
      ∧ pendingPkt ≠ {}
      ∧ ns ≠ NoTopology ∧ ns.t = s ∧ ns.tp = n
      ∧ ⟨s, n⟩ ∈ IngressSwitch
      ∧ ∃ p ∈ pendingPkt :
          ∧ swPktQ' =
              [swPktQ EXCEPT ![s] = Append(@, NewPkt(p, n))]
          ∧ pendingPkt' = pendingPkt \ {p}
      ∧ sw' = [sw EXCEPT ![s] = "rdyA"]
      ∧ UNCHANGED ⟨swPkt, pktQ⟩

ExPacket(s) ≜
  ∧ sw[s] ∈ {"rdy", "rdyA"}
  ∧ swPktQ[s] ≠ ⟨⟩
  ∧ swPkt' = [swPkt EXCEPT ![s] = Head(swPktQ[s])]
  ∧ swPktQ' = [swPktQ EXCEPT ![s] = Tail(@)]
  ∧ sw' = [sw EXCEPT ![s] = "busy"]
  ∧ UNCHANGED ⟨pktQ, pendingPkt⟩

SwAct(s) ≜
  LET k ≜ PacketRule(s, SwitchRule[s])
  IN  ∧ sw[s] = "busy"
      ∧ IF k.a = "drop" ∨ k.a = "norule" THEN
          LET pr ≜ [swPkt[s] EXCEPT !.a = k.a,
                                    !.o = CHOOSE v ∈ k.o : TRUE]
          IN  ∧ pktQ' = Append(pktQ, AddPktPath(pr, s))
              ∧ UNCHANGED swPktQ
        ELSE
          LET t ≜ GetTuple(k.o, Cardinality(k.o))
              q[i ∈ 0 .. Len(t)] ≜
                IF i = 0 THEN
                  [sQ ↦ swPktQ, pQ ↦ pktQ]
                ELSE
                  LET pr ≜ [swPkt[s] EXCEPT !.a = k.a,
                                            !.o = t[i]]
                      ns ≜ NextSW(s, t[i])
                      p  ≜ AddPktPath(pr, s)
                  IN
                    IF ns ≠ NoTopology ∧ ns.t ∈ Switch THEN
                      [q[i − 1] EXCEPT !.sQ[ns.t] =
                                       Append(@, SendPkt(p, ns))]
                    ELSE
                      [q[i − 1] EXCEPT !.pQ = Append(@, p)]
          IN  LET rslt ≜ q[Len(t)]
              IN  ∧ swPktQ' = rslt.sQ
                  ∧ pktQ' = rslt.pQ
      ∧ swPkt' = [swPkt EXCEPT ![s] = NoPacket]
      ∧ sw' = [sw EXCEPT ![s] = "rdy"]
      ∧ UNCHANGED pendingPkt
```

FIGURE 6. Definitions of actions *EnPacket*, *Expacket*, and *SwAct*.

```
    CHOOSE to ∈ Topology : to.s = x ∧ to.sp = y
  ELSE
    NoTopology
```

The $NextSW(x, y)$ operator selects the switch information connected to the port number y of switch x from the defined topology; otherwise, i.e. if there is no switch connected to switch x in the defined topology, it is *NoToplogy* denoting an arbitrary value that does not belong to *Topology* (Fig. 5). When the *EnPacket* action is enabled, a packet arriving at the ingress switch is stored in the processing packet queue of the ingress switch. Following is the expression that belongs to *EnPacket* in Fig. 6:

$$\exists p \in pendingPkt : swPktQ' = [swPktQ \text{ EXCEPT } ![s] = Append(@, NewPkt(p, n))]$$

We define the operator *NewPkt* setting values for a new packet as follows:

$$NewPkt(x, y) \triangleq [p \mapsto x, i \mapsto y, a \mapsto NoAction, o \mapsto 0, r \mapsto \langle\rangle]$$

For $NewPkt(x, y)$, fields p and i of the newly generated packet of type *ProPacket* in the switch are set to input packet information x and input port information y, respectively. The other fields of the packet have default values *NoAction*, 0, and $\langle\rangle$. The state of the switch satisfying the *EnPacket* action becomes "rdyA" so the next action can be performed. To distinguish where a packet is forwarded, we have defined a different state for the switch to which the packet is forwarded. The "rdyA" state in action *EnPacket* refers to the state of the switch that queues packets coming from the outside of SDN. The "rdy" state in action *SwAct* refers to the state of the switch that queues packets coming from the inside of SDN. As long as the switch is active, the packets continue to flow into the wait queue, and the switch processes the packets in the queued order (FIFO). Our specification abstracts the behavior of the switch as the switch ready state "rdy" and "rdyA" and the operation state "busy."

The *ExPacket* action that extracts the packets to be processed is enabled when there are packets to be processed at the corresponding switch. Because the switch processes packets in the order they are received, the first packet is extracted and stored in variable *swPkt*. We specify that the switch processes one packet at a time. Thus, the next state of the switch becomes "busy" and *pendingPkt* and *pktQ* do not change.

The *SwAct* action, which describes the processing behavior of packets coming into the switch, is enabled when the switch is in the "busy" state. Depending on the definition of $PacketRule(s, r)$, which finds the rule (from rules r of switch s) corresponding to the packet to be processed at switch s (Fig. 7), the packet is delivered to another switch or dropped from the current switch according to the action defined in the corresponding rule. Our specification does not describe the communication with the SDN controller, so if there is no rule, we define the action as "norule" and treat it like a "drop."

In the *SwAct* action, the *AddPktPath* operator is used to modify the variable *pktQ* regardless of the action of the rule applied to a packet. Except when a packet is forwarded to the next switches, the packet is stored in *pktQ* because the process is complete for the current switch. If the current switch is connected to an external network, where $ns.t \notin Switch$ for the next switch $ns.t$ (Fig. 6), the packet is also considered to have been processed. The *AddPktPath* operator shows that

$$
\begin{aligned}
&PacketRule(s,\ r)\ \triangleq\\
&\quad \text{LET } f[i \in 1\ ..\ Len(r)+1]\ \triangleq\\
&\qquad \text{IF } i = Len(r)+1 \text{ THEN}\\
&\qquad\quad [a \mapsto \text{“norule”},\ o \mapsto \{0\}]\\
&\qquad \text{ELSE}\\
&\qquad\quad \text{IF } Match(swPkt[s],\ r[i]) \text{ THEN}\\
&\qquad\qquad [a \mapsto r[i].a,\ o \mapsto r[i].o]\\
&\qquad\quad \text{ELSE } f[i+1]\\
&\quad \text{IN } f[1]
\end{aligned}
$$

FIGURE 7. Definition of the *PacketRule* operator used in the *SwAct* action.

the packet has been processed at the current switch and the definition is

$$AddPktPath(x, y) \triangleq [p \mapsto x.p, i \mapsto x.i, a \mapsto x.a, o \mapsto x.o, r \mapsto Append(x.r, y)]$$

where packet information x is an element of *ProPacket* and switch y is an element of *Switches*. The $AddPktPath(x, y)$ operator modifies only the value of field r, for packet information x, and the result is an element of *ProPacket*.

The variable *swPktQ* in the *SwAct* action does not change when an action is “drop” or “norule”, but it changes when an action is “outport” because a packet at the current switch is forwarded to all next switches connected to its ports in o at the current switch; the next switches are searched with field o of the rule processing the packet. Local function q in the *SwAct* action computes the modified information of the switches connected to the current switch. Function q requires a tuple as the input, so we change the set of output port numbers to a tuple by using operator $GetTuple(s, c)$ with arguments s and c representing a set and the number of elements of the set. This paper does not describe the details of the *GetTuple* operator we defined, because that operator can be defined in different ways by different authors.

In function q, the packet information forwarded to each switch is modified. Input port i of the packet information stores the port number of the switch connected to the current switch, and both action a and output port o are initialized. Operator *SendPkt* does this operation and the definition is

$$SendPkt(x, y) \triangleq [p \mapsto x.p, i \mapsto y.tp, a \mapsto NoAction, o \mapsto 0, r \mapsto x.r]$$

where packet information x is an element of *ProPacket* and next switch information y is an element of *Topology*. The modified packet information is added to the packet to be processed by the corresponding switch. Following is the expression that belongs to *SwAct* in Fig. 6:

$$[q[i-1] \text{ EXCEPT } !.sQ[ns.t] = Append(@, SendPkt(p, ns))]$$

Finally, the *SwAct* action initializes the packet information to be processed by the current switch and changes the next switch state to the ready state “rdy” to process the next packet.

As a result, all behaviors from this specification *Spec* defined with the *Init* action and *Next* action always satisfy the *TypeInvariant* defined above. We also define this as a theorem that is verified using model checking in the verification section:

$$\text{THEOREM } Spec \Rightarrow \Box\, TypeInvariant$$

TABLE 2. Firewall rules.

Flow entry name	Rule	Action
R1	srcIP=1, proto=TCP	Drop
R2	dstIP=2	Allow
R3	proto=TCP	Allow
R4	*	Drop

IV. VERIFICATION OF SDN-BASED FIREWALL RULES

In this section, we present an example of formal verification using the TLA+ specification for SDN firewall rules. We use the TLC model checker, which is a model-checking tool for TLA+ specifications. More detailed information about TLC can be found in [13].

For the example, we simplified the process to verify whether the results for the packets passing through a switch match the results when the SDN controller rules that were created according to the firewall policy are applied to the switches. This study does not cover how to apply the rules at each switch. We assume that the SDN controller has firewall rules and each switch has its own rules. Firewall rules must survive any change in topology, and our goal is to check whether this is the case. We verify chain topology and diamond topology models for the same firewall rules.

A. SPECIFICATION OF SWITCH RULES

To check the specification for delivering a packet based on the rules reflecting the firewall policy, we define the firewall rules for an SDN controller. Then a model of the specification must be given to TLC. To define the model, all declared constants in the specification are set for TLC, and then map the constants *Switch*, *Topology*, *SwitchRule*, *IngressSwitch*, and *Match*(_, _), to the values.

1) FIREWALL POLICY FOR AN SDN CONTROLLER

We use the simplified rules determining whether to forward a packet with the IP information as in [8], [9], [22]. Rules are applied to each packet sequentially using the example of firewall rules described in [8], [9], [22], which is modified by adding the protocol information to rules, as summarized in Table 2.

In Table 2, flow entries are identified with $R1$, $R2$, $R3$, and $R4$, and packets are compared with the rules in the order of $R1$, $R2$, $R3$, and $R4$ to determine whether to allow or drop the packet. According to the firewall rules, packets for which the source IP address is not 1, and the destination IP address

is 2 or the protocol is TCP; that is, $srcIP \neq 1 \land (dstIP = 2 \lor proto = TCP)$, must reach the destination through a switch connected outside the network and other packets must be dropped, even if the packets pass through multiple switches according to the topology.

We define the rules in Table 2 with TLA+ to determine if the firewall rules are followed when the topology changes. The firewall rules *FR* are defined as follows:

$$\begin{aligned} FR \triangleq \langle & [mf \mapsto [srcIP \mapsto 1, proto \mapsto \text{“}TCP\text{”}], \\ & a \mapsto \text{“}drop\text{”}], \\ & [mf \mapsto [dstIP \mapsto 2], a \mapsto \text{“}allow\text{”}], \\ & [mf \mapsto [proto \mapsto \text{“}TCP\text{”}], a \mapsto \text{“}allow\text{”}], \\ & [mf \mapsto [all \mapsto \text{“}*\text{”}], a \mapsto \text{“}drop\text{”}]\rangle \end{aligned}$$

2) SWITCHES

The network consists of four switches identified as $s1$, $s2$, $s3$, and $s4$:

$$Switch \triangleq \{\text{“}s1\text{”}, \text{“}s2\text{”}, \text{“}s3\text{”}, \text{“}s4\text{”}\}$$

3) TOPOLOGY

We employ the chain and diamond topology models to determine whether rule consistency is maintained when the topology changes, as in [8], [9]. These two models are the simplest ones to check for rule consistency while maintaining minimal changes to the components other than topology, such as switches and switch rules. These models are depicted in Fig. 8 and Fig. 9, respectively. Each topology is abstracted in a unidirectional flow, as reflected in the the definition of the topology.

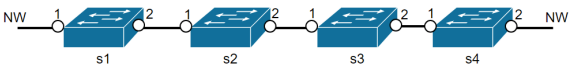

FIGURE 8. Switches connected in a chain topology. The ingress switch is *s1* and the egress switch is *s4*; thus, the packets will pass through *s1*, *s2*, *s3*, and *s4* in order.

Fig. 8 presents the chain topology (*CT*), in which switches are sequentially connected from $s1$ to $s4$. *CT* is expressed based on the topology definition as follows:

$$\begin{aligned} CT \triangleq \{ & [s \mapsto \text{“}NW\text{”}, sp \mapsto 0, t \mapsto \text{“}s1\text{”}, tp \mapsto 1], \\ & [s \mapsto \text{“}s1\text{”}, sp \mapsto 2, t \mapsto \text{“}s2\text{”}, tp \mapsto 1], \\ & [s \mapsto \text{“}s2\text{”}, sp \mapsto 2, t \mapsto \text{“}s3\text{”}, tp \mapsto 1], \\ & [s \mapsto \text{“}s3\text{”}, sp \mapsto 2, t \mapsto \text{“}s4\text{”}, tp \mapsto 1], \\ & [s \mapsto \text{“}s4\text{”}, sp \mapsto 2, t \mapsto \text{“}NW\text{”}, tp \mapsto 0]\} \end{aligned}$$

Fig. 9 displays the diamond topology (*DT*), in which $s1$ is connected to $s2$ and $s3$ and these two switches are connected to $s4$. Switch $s1$ is connected to $s2$ and $s3$ through output ports 2 and 3 of $s1$, respectively.

DT is defined as follows:

$$\begin{aligned} DT \triangleq \{ & [s \mapsto \text{“}NW\text{”}, sp \mapsto 0, t \mapsto \text{“}s1\text{”}, tp \mapsto 1], \\ & [s \mapsto \text{“}s1\text{”}, sp \mapsto 2, t \mapsto \text{“}s2\text{”}, tp \mapsto 1], \end{aligned}$$

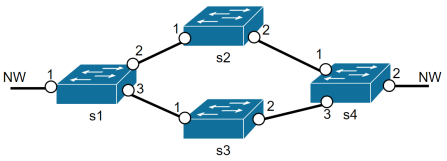

FIGURE 9. Switches connected in a diamond topology. The ingress switch is *s1* and the egress switch is *s4*; thus, the packets go through *s1*, *s2* or *s3*, and *s4* sequentially.

$$\begin{aligned} & [s \mapsto \text{“}s1\text{”}, sp \mapsto 3, t \mapsto \text{“}s3\text{”}, tp \mapsto 1], \\ & [s \mapsto \text{“}s2\text{”}, sp \mapsto 2, t \mapsto \text{“}s4\text{”}, tp \mapsto 1], \\ & [s \mapsto \text{“}s3\text{”}, sp \mapsto 2, t \mapsto \text{“}s4\text{”}, tp \mapsto 3], \\ & [s \mapsto \text{“}s4\text{”}, sp \mapsto 2, t \mapsto \text{“}NW\text{”}, tp \mapsto 0]\} \end{aligned}$$

Additionally, for both chain and diamond topology, we define the constant *IngressSwitch* as follows:

$$IngressSwitch \triangleq \{\langle \text{“}s1\text{”}, 1\rangle\}$$

4) PACKETS

For a concrete packet, we use only the packet fields necessary for switch rules from among the packet fields defined in the packet definition section. And we use the minimum set of packets that we can make using the fields and their values that appear in Table 2. That is, we abstract a packet to the source and destination IP address and the protocol, and define the constant *Packet* as the following eight packets:

$$\begin{aligned} Packet \triangleq \{ & [srcIP \mapsto 2, dstIP \mapsto 2, proto \mapsto \text{“}TCP\text{”}], \\ & [srcIP \mapsto 2, dstIP \mapsto 2, proto \mapsto \text{“}UDP\text{”}], \\ & [srcIP \mapsto 1, dstIP \mapsto 1, proto \mapsto \text{“}TCP\text{”}], \\ & [srcIP \mapsto 1, dstIP \mapsto 1, proto \mapsto \text{“}UDP\text{”}], \\ & [srcIP \mapsto 1, dstIP \mapsto 2, proto \mapsto \text{“}TCP\text{”}], \\ & [srcIP \mapsto 1, dstIP \mapsto 2, proto \mapsto \text{“}UDP\text{”}], \\ & [srcIP \mapsto 2, dstIP \mapsto 1, proto \mapsto \text{“}TCP\text{”}], \\ & [srcIP \mapsto 2, dstIP \mapsto 1, proto \mapsto \text{“}UDP\text{”}]\} \end{aligned}$$

When we define packets, the number of packets should be minimized as much as possible, because it affects the execution time of model checking. Defined packets should traverse all possible paths generated by rules. If the rules are very simple, we can easily create the minimum set of packets manually. However, rules are usually not so straightforward that we can create packets manually. Therefore, we need to automate the generation of packets.

For automation, we have written an algorithm that generates packets. This algorithm requires firewall rules *FR* as input. Then it gets match fields of rules used in *FR* except “$*$” denoting all packets and computes the set of possible values in the rules for each field. Finally, the algorithm outputs the Cartesian product of the sets of field-value pairs for each field. Our eight packets are the result of $\{srcIP \mapsto 1, srcIP \mapsto 2\} \times \{dstIP \mapsto 1, dstIP \mapsto 2\} \times \{proto \mapsto \text{“}TCP\text{”}, proto \mapsto \text{“}UDP\text{”}\}$.

5) SWITCH RULES

We modify the example of switch rules in [8], [9] to fit our firewall policy and define it according to TLA+. However, it is beyond the scope of this paper to describe how policies are decomposed into individual switch rules. The rules for each switch have a record structure and are distinguished by a switch identifier.

SwitchRule represents switch rules for the chain switch topology according to the definition in the previous section:

$$
\begin{aligned}
&SwitchRule \triangleq \\
&\quad [s1 \mapsto \langle [mf \mapsto [dstIP \mapsto 2], a \mapsto \text{“}outport\text{”}, o \mapsto \{2\}], \\
&\qquad [mf \mapsto [proto \mapsto \text{“}TCP\text{”}], \\
&\qquad\quad a \mapsto \text{“}outport\text{”}, o \mapsto \{2\}], \\
&\qquad [mf \mapsto [all \mapsto \text{“}*\text{”}], a \mapsto \text{“}drop\text{”}, o \mapsto \{0\}]\rangle, \\
&\quad s2 \mapsto \langle [mf \mapsto [srcIP \mapsto 1, proto \mapsto \text{“}TCP\text{”}], \\
&\qquad\quad a \mapsto \text{“}drop\text{”}, o \mapsto \{0\}], \\
&\qquad [mf \mapsto [all \mapsto \text{“}*\text{”}], a \mapsto \text{”}outport\text{”}, o \mapsto \{2\}]\rangle, \\
&\quad s3 \mapsto \langle [mf \mapsto [dstIP \mapsto 2], a \mapsto \text{“}outport\text{”}, o \mapsto \{2\}], \\
&\qquad [mf \mapsto [proto \mapsto \text{“}TCP\text{”}], \\
&\qquad\quad a \mapsto \text{“}outport\text{”}, o \mapsto \{2\}], \\
&\qquad [mf \mapsto [all \mapsto \text{“}*\text{”}], a \mapsto \text{“}drop\text{”}, o \mapsto \{0\}]\rangle, \\
&\quad s4 \mapsto \langle [mf \mapsto [dstIP \mapsto 2], a \mapsto \text{“}outport\text{”}, o \mapsto \{2\}], \\
&\qquad [mf \mapsto [proto \mapsto \text{“}TCP\text{”}], \\
&\qquad\quad a \mapsto \text{“}outport\text{”}, o \mapsto \{2\}], \\
&\qquad [mf \mapsto [all \mapsto \text{“}*\text{”}], a \mapsto \text{“}drop\text{”}, o \mapsto \{0\}]\rangle]
\end{aligned}
$$

The rules for the diamond switch topology are defined using the switch rules for the chain switch topology except that the outport of switch $s1$ is set to $\{2, 3\}$ to branch the forwarding path. Because the specification reflects both the chained and diamond switch topologies, it is possible to model a variety of topologies by modifying the definitions of the constants *Topology* and *SwitchRule* while maintaining the definitions of the other constants.

6) MATCHING A PACKET TO A RULE

We define the match operator in relation to the switch rules. According to our abstraction for the rules, packets coming into the switch are either forwarded to other switches or dropped at the current switch depending on the rules for the switch. To make this decision, the operator that matches the packet to the rules for each switch is defined as the constant *Match*(_, _), for which the names of the arguments are undefined while the count of the arguments is defined. The operation of *Match*(_, _) is defined when we define a model for TLC. We define it based on the packet and rule structure.

The constant *Match*(_, _) is as follows:

$$
\begin{aligned}
&Match(A, B) \triangleq \\
&\quad \text{LET } C \triangleq \text{DOMAIN } A.p \cap \text{DOMAIN } B.mf \\
&\quad \text{IN} \lor (\land \text{“}all\text{”} \in \text{DOMAIN } B.mf \\
&\qquad\qquad \land B.mf[\text{“}all\text{”}] = \text{“}*\text{”}) \\
&\qquad \lor (C \neq \{\} \land \forall x \in C : A.p[x] = B.mf[x])
\end{aligned}
$$

In *Match*(A, B), the first argument, A, of the match operator is the packet information that is dealt with at the switch and the second argument, B, is one of the ordered rules for the switch. The operator returns TRUE if the rule holds true for the packet, and FALSE otherwise.

B. VERIFICATION OF THE FIREWALL RULES

We have specified the handling of packets according to the rules for each switch, and we have defined a model for that specification. This section describes how we check the defined models using the TLC model checker included in the TLA+ Toolbox.

To check the model, we employed TLA+ Toolbox Version 1.5.7 for Windows using the Windows 10 operating system, an Intel i7-7700 CPU with four physical cores, and eight threads at 3.6 GHz with 16 GB of RAM. The execution time for our model depended on the number of packets and was around 5 seconds for 4 packets when checking the type invariants in this environment.

We first checked the models for the invariant *TypeInvariant*. Using TLC, we found some errors, corrected them, and ran TLC again. When an error was detected, TLC displayed the error and stopped operating. We then checked the models with the invariant *PRuleConsistency*. For the diamond model, errors were detected because the invariant was violated. In the next two sections, we describe the process of checking these invariants.

1) VERIFICATION OF THE TYPE INVARIANCE OF THE VARIABLES

In the TLA Toolbox model creation screen, the invariant *TypeInvariant* is registered as one of the invariants for which the formula must be true in every reachable state. When we checked the invariant, we found that we had initially incorrectly defined the invariant; thus, TLC reported that *TypeInvariant* was violated. When the error was fixed and TLC restarted, the model checking process was completed without error. We checked both the chain and the diamond topology models. We considered *TypeInvariant* to be satisfied if no violation was found. As a result, we were able to conclude that *Spec* implies $\Box$*TypeInvariant*.

2) VERIFICATION OF FIREWALL RULE CONSISTENCY

For verification, our work focuses on whether the packets that passed through the SDN switches were dropped or allowed according to the firewall policy. Even if there are multiple paths to forward packets within an SDN network, if there are paths that allow or drop packets that must be dropped or allowed according to the firewall policy, we assume that there is rule conflict between the firewall policy and the SDN switch rules. We do not consider rule conflicts for each forwarding path.

One of the goals of this study is to determine whether the firewall rules are maintained and do not come in conflict with the switch rules when the topology changes. In other words, when the topology changes, we must verify that the results generated when applying the rules to a packet are the same. To accomplish this, we define the invariant *PRuleConsistency*:

$$
\begin{aligned}
&PRuleConstistency \triangleq pktQ \neq \langle\rangle \Rightarrow \\
&\quad \forall i \in 1..Len(pktQ) : \\
&\quad\quad \text{LET } pr \triangleq GetPktRule(pktQ[i], FR) \\
&\quad\quad\quad pa \triangleq \text{IF } pktQ[i].a = \text{“}outport\text{”} \text{ THEN} \\
&\quad\quad\quad\quad\quad \text{“}allow\text{”} \\
&\quad\quad\quad\quad \text{ELSE “}drop\text{”} \\
&\quad\quad \text{IN} \quad pa = pr.a
\end{aligned}
$$

The invariant *PRuleConsistency* determines whether the results are the same when the previously defined firewall rules (*FR*) are applied to a packet and when the packet passes through each switch. In our specification, we store the processed packets in *pktQ* according to switch rules *SwitchRule* to compare these action results. If action *a* of a packet in the processed packets *pktQ* is “outport,” the action for the packet is “allow.” We ultimately compare the action results for *FR* and *SwitchRule*.

The *GetPktRule* operator used in *PRuleConsistency* is similar to the *PacketRule* operator defined in our specification except for finding an action corresponding to the target packet in *FR*:

$$
\begin{aligned}
&GetPktRule(p, r) \triangleq \\
&\quad \text{LET } f[i \in 1..Len(r)+1] \triangleq \\
&\quad\quad \text{IF } i = Len(r)+1 \text{ THEN} \\
&\quad\quad\quad [a \mapsto \text{“}drop\text{”}] \\
&\quad\quad \text{ELSE} \\
&\quad\quad\quad \text{IF } Match(p, r[i]) \text{ THEN } [a \mapsto r[i].a] \\
&\quad\quad\quad \text{ELSE } f[i+1] \\
&\quad \text{IN } f[1]
\end{aligned}
$$

This work uses the *GetPktRule* operator we defined in [22]. The *GetPktRule* operator returns action information of the rule matching a packet *p* in the ordered rules; if there is no rule matching packet *p*, we consider that the action for the packet is “drop.” The operator *Match* is defined in the specification section.

We check for consistency with the firewall rules using the invariant *PRuleConsistency* for the previously defined chain topology and diamond topology models. If *PRuleConsistency* is violated, an error is detected and displayed in the error trace section of the model-checking results.

Fig. 10 presents the results for the diamond topology model. As can be observed, the invariant *PRuleConsistency* was violated and the violated state was displayed in the error-trace area. The error trace shows the eight states created by actions, such as *EnPacket*, *ExPacket*, and *SwAct*, from the initial state to the error state. According to the firewall rules defined in the SDN controller, the rules unconditionally drop a packet whose source IP address is 1 and the protocol is TCP even if the destination IP address is 2. In the chained topology (Fig. 8), packets sequentially pass through connected switches $s1$, $s2$, $s3$, and $s4$. In the chained topology, a packet whose *srcIP* is 1 and protocol is TCP is filtered at $s2$, but a packet that should be dropped in accordance with the first switch rule at $s2$ is passed when the topology changes to branch from $s1$ to both $s2$ and $s3$ (Fig. 9). In the two topologies, switch rules at $s3$ do not filter a packet

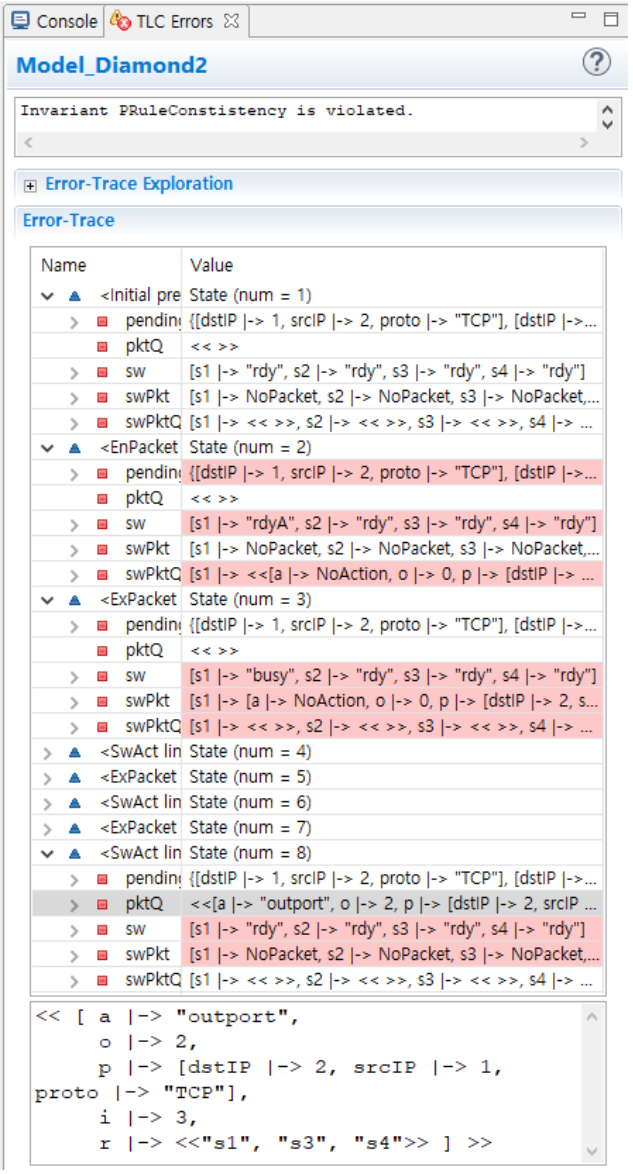

FIGURE 10. The model-checking result of the diamond topology. In the diamond topology, the invariant *PRuleConsistency* is violated. The error trace shows the state that violated the invariant. A packet for which the destination IP address is 2, the source IP address is 1, and the protocol is TCP reached the final switch $s4$ even though this packet should have been dropped earlier.

whose *srcIP* is 1 and protocol is TCP. Consequently, it passes through the network with the diamond topology in a way that does not conform to the firewall rules. As a result, the invariant is violated, and the violated state shows that the packet $[srcIP \mapsto 1, dstIP \mapsto 2, proto \mapsto \text{``}TCP\text{''}]$ passed through switches $s1$, $s3$, and $s4$, in turn.

This problem arises when the topology changes in a network that is currently operating successfully, such as one with a chain topology, and only the ''outport'' value of the switch rules changes according to the topology. In reality, this problem can occur when appropriate rules are created and applied to each switch managed with the SDN controller. The results could be disastrous for network administrators. Therefore, it is important to check whether consistency with the firewall rules is maintained when the topology changes before the rules are applied to the switches. The proposed method described in this paper can help solve this problem.

V. CONCLUSION

The correctness of the SDN network is a critical factor in guaranteeing the successful operation of a network. To address this, we proposed a novel approach to formally verify the correctness of an SDN firewall policy using TLA+. We first presented the rules for translation of SDN networks and firewall policies into a TLA+ model and then showed that the TLA+ specification could be formally verified by the TLC model checker. Our example demonstrated how type invariance and rule conflict between rules in SDN switches and SDN controller policies could be identified using our approach. It was also noted that TLA+ and the TLC model checker are well-suited to the specification and verification of SDN firewall policies.

In future research, we intend to write TLA+ specifications of flow rules that reflect security and QoS policies that are created by various SDN applications and automatically generate models to inspect SDN controller policies and rules for consistency. The automatically generated model can be checked directly by using TLC, so even non-experts in formal methods can benefit from the use of formal tools. Further, we will improve our research to work with firewall policies that take into account the forwarding paths in the SDN network.

REFERENCES

[1] L. Doyle. (2015). *The SDN Network as a Competitive Advantage*. Accessed: Oct. 30, 2018. [Online]. Available: https://www.networkworld.com/article/2977018/software-defined-networking/the-sdn-network-as-a-competitive-advantage.html
[2] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, ''A NICE way to test openflow applications,'' in *Proc. 9th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2012, pp. 127–140.
[3] P. Kazemian, G. Varghese, and N. McKeown, ''Header space analysis: Static checking for networks,'' in *Proc. NSDI*, vol. 12, 2012, pp. 113–126.
[4] R. Majumdar, S. Deep Tetali, and Z. Wang, ''Kuai: A model checker for software-defined networks,'' in *Proc. Formal Methods Comput.-Aided Design (FMCAD)*, Oct. 2014, pp. 163–170.
[5] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, ''Veriflow: Verifying network-wide invariants in real time,'' in *Proc. 1st Workshop Hot topics Softw. Defined Netw.*, 2012, pp. 49–54.
[6] A. El-Hassany, J. Miserez, P. Bielik, L. Vanbever, and M. Vechev, ''SDNRacer: Concurrency analysis for software-defined networks,'' *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 402–415, Jun. 2016.
[7] M.-K. Shin, K.-H. Nam, and J.-Y. Choi. (2012). *Formally Verifiable Networking Framework for SDN*. Accessed: Oct. 30, 2018. [Online]. Available: https://tools.ietf.org/html/draft-shin-sdn-formal-specification-00
[8] M. Kang, J.-Y. Choi, H. H. Kwak, I. Kang, M.-K. Shin, and J.-H. Yi, ''Formal modeling and verification for SDN firewall application using pACSR,'' in *Electronics, Communications and Networks IV*. Boca Raton, FL, USA: CRC Press, 2015, p. 155.
[9] M. Kang, J.-Y. Choi, I. Kang, H. H. Kwak, S. J. Ahn, and M.-K. Shin, ''A verification method of SDN firewall applications,'' *IEICE Trans. Commun.*, vol. 99, no. 7, pp. 1408–1415, 2016.
[10] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley, 2002.
[11] C. Newcombe, ''Why Amazon chose TLA$^+$,'' in *Proc. Int. Conf. Abstract State Mach., Alloy, B, TLA, VDM, Z*. Springer, 2014, pp. 25–39.
[12] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, ''How Amazon Web services uses formal methods,'' *Commun. ACM*, vol. 58, no. 4, pp. 66–73, Mar. 2015.
[13] Microsoft. (2010). *The TLA Toolbox*. Accessed: Oct. 30, 2018. [Online]. Available: https://lamport.azurewebsites.net/tla/toolbox.html
[14] *What is Software Defined Networking (SDN)?* Accessed: Jan. 7, 2019. [Online]. Available: https://www.sdxcentral.com/sdn/definitions/what-the-definition-of-software-defined-networking-sdn/
[15] ONF. (2015). *OpenFlow Switch Specification-Version 1.5.1*. Accessed: Oct. 30, 2018. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf
[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, ''OpenFlow: Enabling innovation in campus networks,'' *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.
[17] S. Merz, ''The specification language TLA$^+$,'' in *Logics of Specification Languages*. Springer, 2008, pp. 401–451.
[18] D. L. Dill, ''The Murϕ verification system,'' in *Proc. Int. Conf. Comput. Aided Verification*. Springer, pp. 390–393.
[19] I. Lee, P. Bremond-Gregoire, and R. Gerber, ''A process algebraic approach to the specification and analysis of resource-bound real-time systems,'' *Proc. IEEE*, vol. 82, no. 1, pp. 158–171, Jan. 1994.
[20] H. Ben-Abdallah, J.-Y. Choi, D. Clarke, Y. S. Kim, I. Lee, and H.-L. Xie, ''A process algebraic approach to the schedulability analysis of real-time systems,'' *Real-Time Syst.*, vol. 15, no. 3, pp. 189–219, 1998.
[21] L. Lamport, ''Fast paxos,'' *Distrib. Comput.*, vol. 19, no. 2, pp. 79–103, Oct. 2006.
[22] Y.-M. Kim, M. Kang, and J.-Y. Choi, ''Formal specification and verification of firewall using TLA+,'' in *Proc. Int. Conf. Secur. Manage. (SAM)*, 2017, pp. 247–251.

YOUNG-MI KIM received the M.S. degree from the Department of Computer, Korea University, Seoul, South Korea, in 2001, where she is currently pursuing the Ph.D. degree with the Department of Computer and Radio Communication Engineering. From 2001 to 2015, she worked as a Web Programmer in South Korea. Her research interests include formal methods, networks, SDN security, and software testing.

MIYOUNG KANG received the M.S. degree from the Department of Computer Science and Engineering, Dongguk University, and the Ph.D. degree from the Department of Computer Science and Engineering, Korea University, Seoul, South Korea. She is currently a Research Professor with the Graduate School of Information Security, Korea University, Seoul. Her research interests include formal methods, process algebras, SDN, and network security.