IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 31 August 2023, accepted 23 September 2023, date of publication 27 September 2023, date of current version 4 October 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3320050

RESEARCH ARTICLE

Improved Formal Verification of SDN-Based Firewalls by Using TLA+

TATJANA KAPUS, (Member, IEEE)
Faculty of Electrical Engineering and Computer Science, University of Maribor, 2000 Maribor, Slovenia
e-mail: tatjana.kapus@um.si

This work was supported in part by the Slovenian Research Agency (ARRS) under Grant P2-0069.

ABSTRACT In an article published in IEEE Access in 2020, researchers present an approach to using TLA+ for the formal verification of whether a network of SDN (Software-Defined Networking) switches implements the filtering rules of a given monolithic firewall. The distributed as well as monolithic firewalls are specified with TLA+. It is shown that the correctness of the former with respect to the latter can be verified automatically by using the TLC model checker. The main contributions of this paper are the following improvements of that approach. Firstly, by specifying switches without using any variables, the time needed for the model checking is reduced significantly. For example, the verification of the same networks takes a few seconds with the new approach and does not end after several hours with the previous one. Secondly, the following problem is solved. With the latter, if a monolithic firewall allows a packet to pass through, all the paths in the distributed firewall which the packet is routed on must allow the same. Otherwise, the model checker proclaims the distributed firewall to be in error. We present an additional approach to the verification, which gives a positive answer if at least one of the paths allows the packet to pass through.

INDEX TERMS Firewalls, formal specification, formal verification, logic, model checking, software defined networking.

I. INTRODUCTION

In [1], the authors present an approach to using TLA+ [2] (a formal language based on the Temporal Logic of Actions (TLA) [3]) for the formal verification of whether a network of SDN (Software-Defined Networking) switches implements the filtering rules of a given monolithic firewall. It is shown how to perform the verification automatically with the TLC model checker, available in the TLA+ Toolbox integrated development environment [4]. Besides the network and the monolithic firewall, a special invariant has to be specified with TLA+, which defines what it means for the network to implement the firewall rules. The truth of this invariant is checked with the TLC model checker. The TLA+ specification of the network is written in such a way that the path followed through the network is recorded for each packet sent. As soon as the model checker detects a violation of the invariant, it can, thus, print an error trace indicating the packet and the switch(es) causing a violation of the firewall rules.

The main contributions of this paper are two improvements of the approach presented in [1]. In the latter, the TLA+ specification of switches contains variables for storing their current processing state and the packets currently residing in them. We do not use the variables, but rather specify a switch, as well as a network of them, as a function which either immediately maps a packet from an input port to some output ports or drops it, without any intermediate storing of packets. The verification consists in verifying whether the network passes or, respectively, drops the same packets as the rules of the monolithic firewall. The model checking of the new specifications is much more efficient. For example, it takes a few seconds to verify a network for which the verification does not end after several hours with the previous approach. In spite of the stateless network specification, the complete specification can be written in such a way that the model checker still returns an error trace indicating the reason for the network not implementing the required firewall rules.

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

Another improvement is related to the interpretation of the correctness of an SDN-based firewall. With the approach presented in [1], if a monolithic firewall allows a packet to pass through, then all the paths in the distributed firewall which the packet is routed on must allow the same. Otherwise, the model checker reports that the distributed firewall is in error and prints an error trace. This interpretation of the correctness is useful if the different paths in the network are meant to be a guarantee that the packet will pass through even if some of them fail. Otherwise, it is only important that (at least one copy of) the packet comes through the network. Our approach allows to write two kinds of invariants to verify the correctness: one similar to the invariant in [1], following the first interpretation, and a different one, incorporating the second interpretation.

This paper proceeds as follows. In Section II, TLA+ is presented briefly. In Section III, we describe the improved approach to the specification of SDN-based firewalls. In Section IV, we explain how to perform the verification with the new kinds of specifications. Basically, we illustrate and test the new approach by using the same networks and monolithic firewall rules as in [1]. Section V comments on the time needed for the model checking when using the old and new approaches. Related work and the results of this paper are discussed in Section VI. Section VII contains concluding remarks.

II. A BRIEF PRESENTATION OF TLA+

In TLA+ [2], a system (model) specification is written in terms of one or more modules. At the beginning of a module, the modules used by it, its parameters or constants, and its variables are usually declared, using the keywords EXTENDS, CONSTANT, and, respectively, VARIABLE. The main module is the one which contains a TLA [3] formula actually specifying the system behavior. As in [1], we deal only with the specification of safety properties, i.e., of what may happen in the system. In that case, the usual form of the formula is $Init \wedge \Box[Next]_v$, where $Init$ is (i.e., denotes) an initial predicate, $Next$ an action or a disjunction of actions, and v the tuple of variables (from among those declared) the system may change. A behavior is an infinite sequence of states. The variables can occur unprimed (for example, x) or primed (x') in the formula. A predicate contains only unprimed variables, and an action can contain both kinds. Suppose that $v \triangleq \langle p, d \rangle$ (the $\triangleq$ symbol means "by definition"), $Init \triangleq p = \text{“i0”} \wedge d = 2$, and $Next \triangleq Act1 \vee Act2$, where $Act1 \triangleq p = \text{“i0”} \wedge p' = \text{“i1”} \wedge d' = d$ and $Act2 \triangleq p = \text{“i1”} \wedge d' = 3 \wedge p' = p$, in the specification formula. Then, the formula says that, in every system behavior, in the initial state, variable p is equal to the string "i0" and variable d to 2, and that in each state of the behavior (this is expressed with the linear-time temporal operator $\Box$), either action $Act1$ or $Act2$ may execute, or the value of both variables may remain the same in the next state. An execution of $Act1$ in a state means that $p = \text{“i0”}$ in it and $p = \text{“i1”}$ in the next state, and that d is equal in both states, and analogous for action $Act2$. Additional information about TLA+ will be provided as necessary when explaining the specifications given in the succeeding sections.

III. SPECIFICATION OF SDN-BASED FIREWALLS

We treat exactly the same kinds of networks as [1] and specify the network topology as well as the rules of the switches and the monolithic firewall with TLA+ in the same way. Let us mention that a software-defined network consists of SDN switches, hosts connected to them, and at least one SDN controller [5]. The controller can control the switches by setting the rules in their flow tables on what they should do with the received data packets sent by the hosts. For each incoming data packet, the switch tries to find a rule in the table which matches the packet header, and, if found, executes an action on the packet, which is specified in the rule. As in [1], we assume that the rules are given and do not change, and, consequently, there is no need to model an SDN controller in the networks.

The basic difference between this paper and [1] is in the TLA+ specification of the behavior of switches and a network of them. The new specification is given in Fig. 1. As in [1], the specification uses the operations on sequences from a predefined module $Sequences$ and similar. The constants are the same as in [1]. The values of the constants are defined when preparing a concrete model of a network for model checking. For ease of further explanation, in Fig. 3 we provide in turn the set of switches and the description of the topology for the network from [1], shown in Fig. 2, and an example of rules for the switches. As in [1], for the verification of a network, a set of IP packets has to be sent into it. In fact, abstract packets are used. A packet consists of some of the following typical five header elements: the source and destination IP addresses, the source and destination port, and the name of the transport protocol used. The definition of an appropriate packet set ($Packet$) for the rules from Fig. 3 is in Fig. 4. (For this paper, it does not matter how such a set is generated – an interested reader can see [1]. Also, let us not discuss the decision there to include packets with the same source and destination IP address.) The definition of $IngressSwitch$ in Fig. 3 says that the packets will be sent into the network at port 1 of switch "s1". The elements of a $Topology$ set (please see Fig. 3) are TLA+ records. Each one represents a unidirectional link from a source port (given in the sp field) on a source switch (s) to a target port (tp) on a target switch (t). Notice that s (or, respectively, t) for a link from (respectively, to) the network environment is equal to "NW" and that the corresponding port value (i.e., sp or, respectively, tp) is 0.

In the definition of $SwitchRule$ in Fig. 3 it can be seen that a sequence of rules is given for a switch. Each rule is a record starting with a match field (mf), which specifies the requirements a packet must fulfill in order for the rule to match it. The a field specifies the action of the rule, which can be either "drop" or "outport" from the $Action$ set in Fig. 1.

```
MODULE SDNfirewall
EXTENDS Naturals, Sequences, FiniteSets
CONSTANTS Packet, Switch, Topology, SwitchRule, IngressSwitch
CONSTANT Match(_, _)
CONSTANT FR
VARIABLES pktQ, pendingPkt

vars ≜ ⟨pktQ, pendingPkt⟩

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

NoAction ≜ CHOOSE v : v ∉ Action
NoTopology ≜ CHOOSE v : v ∉ Topology
----------
Init ≜ ∧ pktQ = ⟨⟩
       ∧ pendingPkt = GetTuple(Packet)
PacketType ≜ [p : Packet, i : PortNRange, a : Action ∪ NoAction, o : PortNRange, r : Seq(Switch)]

TypeInvariant ≜ ∧ pktQ ∈ Seq(ProPacket)
                ∧ pendingPkt ∈ Seq(Packet)
NextSw(x, y) ≜ IF ∃ to ∈ Topology : to.s = x ∧ to.sp = y THEN CHOOSE to ∈ Topology : to.s = x ∧ to.sp = y
                                                        ELSE NoTopology
NewPkt(x, y) ≜ [p ↦ x, i ↦ y, a ↦ NoAction, o ↦ 0, r ↦ ⟨⟩]

PacketRule(pkt, r) ≜
  LET f[i ∈ 1..Len(r) + 1] ≜ IF i = Len(r) + 1 THEN [a ↦ "norule", o ↦ {0}]
                               ELSE IF Match(pkt, r[i]) THEN [a ↦ r[i].a, o ↦ r[i].o]
                                                        ELSE f[i + 1]
  IN f[1]
AddPktPath(x, y) ≜ [p ↦ x.p, i ↦ x.i, a ↦ x.a, o ↦ x.o, r ↦ Append(x.r, y)]

SendPkt(x, y) ≜ [p ↦ x.p, i ↦ y.tp, a ↦ NoAction, o ↦ 0, r ↦ x.r]

Transport(sw, pkt) ≜
  LET F[swPktQ ∈ Seq(Switch × PacketType), pktQacc ∈ Seq(ProPacket)] ≜
      IF swPktQ = ⟨⟩
        THEN pktQacc
        ELSE LET s ≜ Head(swPktQ)[1]
                 k ≜ PacketRule(Head(swPktQ)[2], SwitchRule[s])
             IN IF k.a = "drop" ∨ k.a = "norule"
                  THEN LET pr ≜ [Head(swPktQ)[2] EXCEPT !.a = k.a, !.o = CHOOSE v ∈ k.o : TRUE]
                       IN F[Tail(swPktQ), Append(pktQacc, AddPktPath(pr, s))]
                  ELSE LET t ≜ GetTuple(k.o)
                           q[i ∈ 0..Len(t)] ≜
                             IF i = 0
                               THEN [sQ ↦ ⟨⟩, pQ ↦ ⟨⟩]
                               ELSE LET pr ≜ [Head(swPktQ)[2] 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 = Append(@, ⟨ns.t, SendPkt(p, ns)⟩)]
                                         ELSE [q[i − 1] EXCEPT !.pQ = Append(@, p)]
                       IN LET rslt ≜ q[Len(t)]
                          IN F[Tail(swPktQ) ∘ rslt.sQ, pktQacc ∘ rslt.pQ]
  IN F[⟨⟨sw, pkt⟩⟩, ⟨⟩]
EnPacket(s, n) ≜ LET ns ≜ NextSw("NW", 0)
                 IN ∧ pendingPkt ≠ ⟨⟩
                    ∧ ns ≠ NoTopology ∧ ns.t = s ∧ ns.tp = n
                    ∧ ⟨s, n⟩ ∈ IngressSwitch
                    ∧ pendingPkt′ = Tail(pendingPkt)
                    ∧ pktQ′ = Transport(s, NewPkt(Head(pendingPkt), n))
Next ≜ ∃ s ∈ Switch : ∃ n ∈ PortNRange : EnPacket(s, n)
Spec ≜ Init ∧ □[Next]_vars
```

FIGURE 1. TLA+ specification of behavior of SDN-based firewalls.

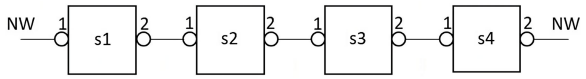

FIGURE 2. Network with a chain topology.

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

FIGURE 3. An example of the chain-topology network described in TLA+ [1].

The first one means that the matched packet is dropped, and the second one means that the packet is forwarded on all the output ports of the switch specified in the o field. The rules containing the word all in the match field match all the packets. As in [1], we assume that the first matched rule in the sequence is applied to a packet. For example, the first rule of switch “s1” would be applied to the first packet from the set $Packet$ in Fig. 4 as its incoming packet, although the other two rules also match it. Operator $PacketRule(pkt, r)$ defined in Fig. 1 finds the first matched rule for a packet pkt in a sequence of rules r and returns its action and the set of output ports. If there is no matched rule, it returns the action “norule” and the set of ports $\{0\}$. It uses a constant operator named $Match$ to check the matching. We have adapted its definition given in [1].

The FR constant in Fig. 1 is meant to define the rules of a monolithic firewall, as shown in [1] and Section IV.

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

FIGURE 4. An example of packets to be sent.

In the specification in [1] there are altogether five variables, three of them representing the state of switches. We retain only the other two. Similar to [1], variable $pendingPkt$ holds the packets that still have to be sent into the network for the purpose of verification, and in variable $pktQ$, packets

$$
\begin{aligned}
&GetTuple(s) \triangleq\\
&\quad \text{LET } F[set \in \text{SUBSET } s, t \in Seq(s)] \triangleq\\
&\qquad \text{IF } set = \{\} \text{ THEN } t\\
&\qquad\qquad \text{ELSE LET } p \triangleq \text{CHOOSE } e \in set : \text{TRUE}\\
&\qquad\qquad\quad \text{IN } F[set\backslash\{p\}, Append(t, p)]\\
&\quad \text{IN } F[s, \langle\,\rangle]
\end{aligned}
$$

FIGURE 5. Conversion of a set to a sequence.

that end their journey through the network are stored for the purpose of verification and diagnostics, together with additional information. The exact format of the latter can be seen in the definition of the symbol $ProPacket$ (please also see the definition of $TypeInvariant$, i.e., the expected types of the two variables). The additional information consists of the number of the input port last visited by the packet, the action last executed on the packet, the number of the last visited output port, and the sequence of switches visited by the packet. The maximal possible port number for switches is set to 5 as in [1], but it can be adapted freely, or could be declared as a constant and set in a model.

From the definition of the predicate $Init$ on, the main differences in the specification (Fig. 1) compared to [1] are in the part starting with the definition of operator $Transport(sw, pkt)$. The $Init$ and $TypeInvariant$ predicates refer only to the variables we have retained. In [1], variable $pendingPkt$ stores a set of packets. As we have found this to be an important cause of the state-space explosion, we made it a sequence, starting by assigning it a sequence generated from the $Packet$ set with the $GetTuple$ operator in the $Init$ predicate. The definition of that operator should be available in the module in Fig. 1. For the lack of space, it is given in Fig. 5 (it is not given in [1]). Please note that the action $EnPacket$, which changes the $pendingPkt$ variable, differs from the one in [1] accordingly. The definition of the $PacketRule$ operator is changed, so that it is called directly for a packet for which a matched rule is sought, and not for a switch in which the packet resides.

The specification formula $Spec$ in Fig. 1 says that, initially, variable $pktQ$ contains an empty sequence and that variable $pendingPkt$ stores all the packets to be sent into the network for the verification purpose. The definition of possible actions of the SDN-based firewall, $Next$, differs from the one in [1] in that it contains only one kind of action, $EnPacket(s, n)$. Like the equally named action in [1], action $EnPacket(s, n)$ for a switch s and a port n can execute if the set (in our case, in fact, the sequence) of packets ($pendingPkt$) to be sent into the network is not yet empty and if switch s is an ingress switch and port n is an input port on it connected to the network environment. If enabled, its effect, however, essentially differs from the one in [1] and contributes to the improved efficiency of the verification. It takes one packet from the set, “sends” the packet into the network at that port, forwards it through the network in accordance with the rules in every switch until any forwarding is possible, and stores the information on each copy of the packet that ended its journey in variable $pktQ$ in the form $ProPacket$ already explained.

In [1], the action $EnPacket(s, n)$ only puts the packet into a waiting queue of the ingress switch and changes the state of the switch so that it is next ready to take the packet from the queue. For the latter, it has to execute another kind of action, which only puts the packet into another variable of the switch. Only now, after two actions, can the switch execute the third kind of action, which tries to find a matched rule and possibly forwards the packet by putting (copies of) it into the waiting queue(s) of the next switch(es) reachable via the output port(s) specified in the rule. Whereas the first kind of action is executed only in the ingress switch, the other two kinds are necessary for forwarding a packet in every switch reached.

Next, we explain in more detail how the forwarding of a packet through the complete network is specified in order to be carried out by executing only one (TLA) action. As can be seen from the last line of the definition of the action $EnPacket(s, n)$ in Fig. 1, this is done by the operator named $Transport$. As the first argument it takes the name of the ingress switch and as the second one the packet accompanied with the information (please see the definition of operator $NewPkt$) on the port of this switch on which it is sent (field i), placeholders for information on the kind of the last executed action (a) and last visited output port (o), as well as the empty sequence (r), where the path of the packet is going to be recorded. Notice that a packet accompanied with information of this kind is of type $PacketType$. As can be seen from the definition of operator $Transport(sw, pkt)$, it is, actually, the recursive function F, called by it, that calculates the result of sending the packet (pkt) entering at the switch (sw) through the entire network.

As in [1], we assume that there are no cycles in the network. Consequently, the network is, in fact, a directed acyclic graph, where the switches are its nodes and the connections the directed edges. Function F has two arguments, which are sequences, and is similar to a breadth-first search of reachable nodes in a directed acyclic graph [6].

It starts with the ingress switch and the packet located at it (we call such a pair a located packet in the sequel) in the first argument, and with the empty sequence in the second one. If there is no matched rule (see "norule" in Fig. 1) for the packet in that switch or the action in the matched rule is "drop", it removes the located packet from the first argument and appends the packet with the information on the last action, output port (in fact 0), and switch visited to the second argument. If, on the other hand, it finds a matched rule with an "outport" action, it first converts its set of output ports into a sequence. Next, by calling the recursive function q, it tries to find for each output port of this sequence the switch and the input port on it connected to the output port. This function returns a record consisting of two sequences (denoted sQ and pQ in Fig. 1). The first one contains the new located packets, i.e., the pairs consisting of a newly reached switch and the packet forwarded to it. The second sequence contains the packets that finished their journey because either the output port was connected to the environment or nowhere (as in [1], we treat these cases equivalently, i.e., that the packet passed through the network successfully). The new located packets are accompanied with the information on the newly reached input port, the placeholders for an action and an output port, and the record of switches visited so far. The packets in the second sequence are also accompanied with such a path record, as well as with the information on the last visited input port, the last executed action, and the last reached output port. The sequence of the new located packets is appended to the first argument of function F, and the sequence of the packets that finished their journey to the second one.

Now, function F is called recursively with the new sequences in the arguments. It tries to forward the located packet at the head of the first one, and again adds the newly located packets to the first argument and the packets that could not proceed, either because of no matched rule, action "drop", or no further connection to a switch, to the second one, and so on, until the sequence of newly located packets is empty. When this is the case (see the comparison $swPktQ = \langle\rangle$ at the beginning of the definition of F in Fig. 1), it returns the second argument as the result, which is also the result of the $Transport$ operator and is, finally, assigned to variable $pktQ$.

Please note that the forwarding of a packet in a switch is specified in a similar way to [1], with a difference being that we have had to find out how to refer to the arguments of the recursive function F instead of the variables applied in [1].

IV. VERIFICATION

For the presented specification approach, the verification whether an SDN-based firewall implements a monolithic one can be performed in the same way as in [1]. Given a specification of the SDN-based firewall, consisting of concrete values of the constants presented in Section III, a specification of the monolithic firewall rules as a value of the FR constant, and the behavior specification $Spec$ as part of module $SDNfirewall$ (Fig. 1), the implementation relation holds if the formula $Spec$ satisfies the invariant $PRuleConsistency$ defined in [1] and shown in Fig. 6, i.e., if the implication $Spec \Rightarrow \Box PRuleConsistency$ is true [3].

$$
\begin{array}{l}
GetPktRule(p, r) \triangleq \\
\quad \text{LET } f[i \in 1..Len(r)+1] \triangleq \\
\qquad \text{IF } i = Len(r)+1 \text{ THEN } [a \mapsto \text{"drop"}] \\
\qquad\qquad \text{ELSE IF } Match(p, r[i]) \text{ THEN } [a \mapsto r[i].a] \\
\qquad\qquad\qquad \text{ELSE } f[i+1] \\
\quad \text{IN } \ f[1] \\
PRuleConsistency \triangleq pktQ \neq \langle\rangle \Rightarrow \\
\qquad \forall\, i \in 1..Len(pktQ) : \\
\qquad\quad \text{LET } pr \triangleq GetPktRule(pktQ[i], FR) \\
\qquad\qquad\ pa \triangleq \text{IF } pktQ[i].a = \text{"outport"} \text{ THEN "allow"} \\
\qquad\qquad\qquad\qquad \text{ELSE "drop"} \\
\qquad\quad \text{IN } \ pa = pr.a
\end{array}
$$

FIGURE 6. Invariant *PRuleConsistency* for verification from [1].

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

FIGURE 7. An example of firewall rules.

Suppose that the chain-topology network from Fig. 2 specified with the constants in Fig. 3 and the *Spec* formula, using the *Packet* set from Fig. 4, is verified against the monolithic firewall rules from [1] given in Fig. 7. The latter are interpreted as saying that every incoming packet must obey the first matched rule from the sequence, i.e., be dropped if the action of the rule is “drop”, or pass through the monolithic firewall if the action is “allow”. If there is no matched rule, the packet should be dropped (please see the definition of operator *GetPktRule* in Fig. 6), as is done in a switch of an SDN-based firewall with no matched rule (Section III).

Formula *PRuleConsistency* maps these requirements to the SDN-based firewall by referring to the packets gathered in variable *pktQ*. We can use this formula despite the fact that we gather the packets in a different way than in [1]. There, as in our specification, the initial value of variable *pktQ* is the empty sequence. Afterwards, for each packet from the *Packet* set sent into the network, in every state, at most one copy of the packet ends its journey, and it is appended to the sequence, accompanied with the additional information explained in Section III. Consequently, the length of the sequence increases constantly until all the packets are sent, and variable *pktQ*, generally, contains copies of different packets.

With our approach, whenever a packet is sent into the network, the sequence of all the copies of it ending its journey is obtained in the same state with the *Transport* operator and assigned to variable *PktQ* in the next state (please see the definition of action *EnPacket* in Fig. 1). Consequently, in our case, except initially, when empty, variable *PktQ* always contains only the copies of one packet from the *Packet* set, but, of course, with different additional information (e.g., different paths recorded).

Not regarding which of the two kinds of recording in variable *pktQ* is used, formula $\Box PRuleConsistency$ expresses the requirement that every packet sent into the network must have passed through the network regardless of the path followed if and only if there is a matched rule in *FR* with action “allow”, and must have not if and only if there is a matched rule with action “drop” or no matched rule.

As in [1], with our specification *Spec* the TLC model checker finds that the chain-topology network implements the *FR* rules. Likewise, it shows that the invariant is violated for the diamond-topology network from [1] (Fig. 8) if the set of output ports in the first two rules of switch $s1$ in the *SwitchRule* constant in Fig. 3 is changed to the value $\{2, 3\}$. Fig. 9 shows the error trace provided by the TLC model checker. It shows that, first, the TCP packet from the *Packet*

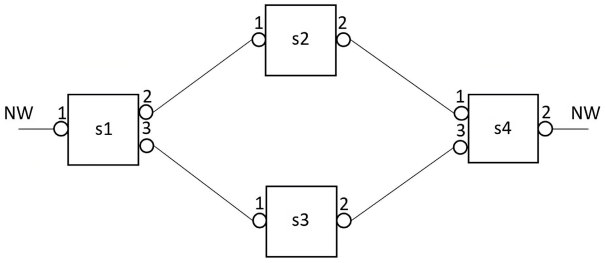

FIGURE 8. Network with a diamond topology.

set with IP addresses equal to 1 was sent into the network by the *EnPacket* action, and that it does not obey the *FR* rules. One can see that a copy of this packet in variable *pktQ* was not dropped as required by the first *FR* rule, but passed through the network along the path $< s1, s3, s4 >$.

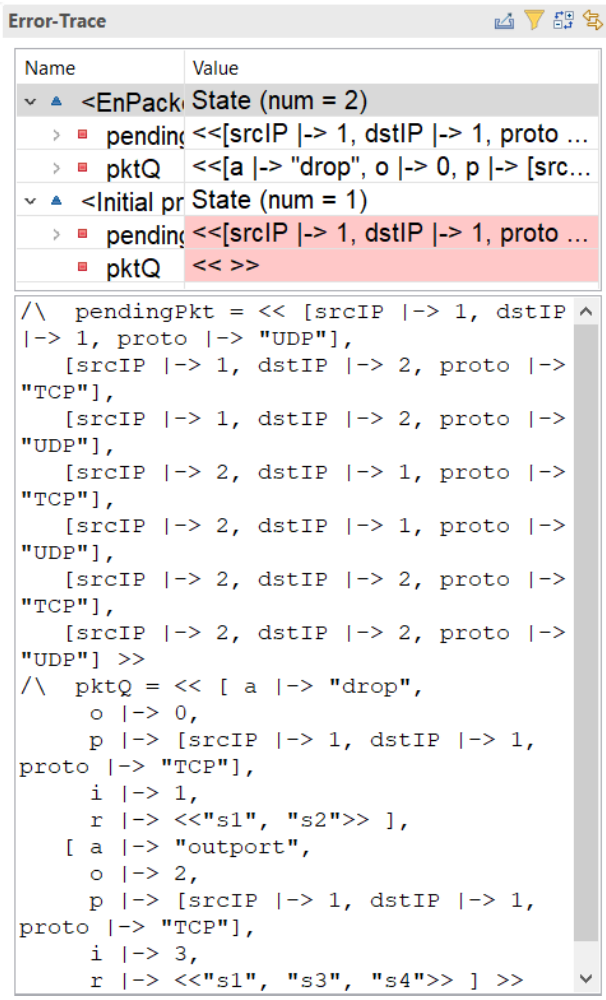

FIGURE 9. Error trace for the diamond-topology network.

Suppose that the action of the first rule of the *FR* constant changed to the value “allow”. The TLC model checker shows that both the chain- and diamond-topology networks violate the invariant when using our specification *Spec*

(and, of course, when using the one from [1]). In the chain-topology network, already the first packet sent (the TCP packet with the IP addresses equal to 1) violates it because it is dropped due to the first rule of switch $s2$. However, as can be seen from the error trace for the diamond-topology network, which happens to be equal to the one in Fig. 9, the copy of this packet that followed the path $< s1, s3, s4 >$ passed through the network and the one forwarded from $s1$ to $s2$ did not. So, in fact, the packet sent into the network passed through, but the interpretation of the correctness of the SDN-based firewall in [1], and thus the invariant, requires that all the copies of the packet, following different paths, have to pass. This interpreation is useful if the different paths are meant to ensure that the packet will pass even if some paths fail in the future.

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

FIGURE 10. Invariants for verification assuming different interpretations of the requirement for packets to pass through.

Another possibility is to interpret the SDN-based firewall to be correct if at least one copy of the packet comes through. Assuming this interpretaion, the verification using our specification approach can be performed by checking whether formula $Spec$ satisfies invariant $PRuleConsistencyE$, shown in Fig. 10. Of course, in this case, the TLC model checker gives a positive answer for the diamond-topology network and still a negative one for the chain-topology one. Formula $\Box PRuleConsistencyE$ still requires that no packet required to be dropped in the monolithic firewall must pass through the SDN-based one, but only requires that at least one copy of a packet allowed to pass through the monolithic firewall must come through the SDN-based one. Because this formula relies on the fact that variable $pktQ$ always contains at most copies of the same packet, it is not applicable to the $Spec$ formula from [1]. For the latter, we do not know how to write an invariant assuming the new interpretation.

On the other hand, assuming the interpretation used in [1], instead of formula $PRuleConsistency$ we could use formula $PRuleConsistencyA$ (Fig. 10), analogous to formula $PRuleConsistencyE$, for the verification of our $Spec$ formula.

Please note that we could have gathered the copies of all the packets sent in variable $pktQ$ by using concatenation to change it in action $EnPacket$. However, in that case, we could only perform the verification with the $PRuleConsistency$ invariant, and it could be less time-efficient than without the gathering, because, similar to [1], for every new packet sent, the copies of all the previous packets should also be checked.

V. PERFORMANCE

For the verification, we used the TLC model checker within TLA+ Toolbox Version 1.7.0 installed on a personal computer with the Windows 10 operating system, 16 GB of RAM and an Intel Core i7-8700 3.20 GHz processor with 6 cores and 12 threads in total. Unless otherwise stated, the latter was also the number of threads configured in the TLC options. As soon as the TLC model checker finds that a model violates an invariant, it ends execution and prints an error trace. The evaluation of the proposed specification approach was, therefore, carried out by verifying networks with invariants being true in them, so that the complete state space would have been checked by the model checker.

At first, we retained the same kind of variable $pendingPkt$ as in [1], i.e., storing a set of packets. Consequently, the choice of the next packet to be sent into the network in the $EnPacket$ action was also specified as in [1], i.e., with the existential quantification $\exists\, p \in pendingPkt : \ldots$ First, we verified the chain- and diamond-topology networks as described in Section IV for the cases with positive answers (i.e., for the FR in Fig. 7 and invariants $PRuleConsistency$, $PRuleConsistencyA$ and $PRuleConsistencyE$ in the case of the chain topology, and $PRuleConsistencyE$ in the case of the diamond one). In all the cases, model checking took from 1 to 2 seconds. With the specification from [1], the verification of the validity of $PRuleConsistency$ in the chain-topology network took more than 4 minutes, and 7,463,413 distinct states were generated. Note that, with our specification approach, the number of distinct states is determined solely by the possible values of the variable $pendingPkt$. The use of existential quantification for choosing the next packet from the latter caused that the model checker generated states for all possible orderings of sending the packets from the $Packet$ set, thus giving 1,025 distinct states for eight possible packets for our SDN-based firewall specification.

By making variable $pendingPkt$ a sequence and taking the next packet from its head (Fig. 1), we achieved that the number of states for our specification is equal to the number of packets in the $Packet$ set plus one. With the new specification, the model checking for the same cases as above still took from 1 to 2 seconds, but reported only 9 distinct states, as expected. For the $Packet$ set extended with packets from the source IP address 1 or 2 to the destination IP address equal to 3 and vice versa, for the TCP and UDP protocols, i.e., to 16 packets, the model checking of the new specification (Fig. 1) still took between 1 and 2 seconds, generating 17 distinct states, whereas with the first version of our specification, it took around half a minute even without the profiling (i.e., the model checker collecting certain metrics about the model checking [7]), generating 524,289 distinct states. The model checking of the specification from [1], with the profiling off, did not finish

successfully even after five hours for the $Packet$ set with 16 packets.

In order to evaluate the impact of the stateless specification of switches itself better, we subsequently also made variable $pendingPkt$ a sequence in the specification from [1]. For the changed specification, the model checking of the above chain-topology network took around 2 seconds for 8 packets, generating 2,713 distinct states, and for 16 packets (with the profiling off) around 45 seconds, generating 1,490,080 distinct states.

The stateless specification made more difference for larger SDN-based firewalls. We obtained the following results for the adapted specification from [1] and ours from Fig. 1, with the profiling off. We have built several larger networks, such that all of them satisfied all the consistency invariants for the $Packet$ set with 8 and 16 packets. First, we prolonged the chain to 10 switches by adding 6 switches with the rules equal to those of switch $s1$. With 8 packets, the model checking of our specification took 1 second, and of the adapted specification from [1] around 5.3 seconds. Next, we made a chain of 100 switches by adding nine ''copies'' of the chain with 10 switches to the latter. The model checking of our specification took around 1.5 seconds, and of the other one did not finish even after 4 hours, generating almost 24 million distinct states during that time.

We also made a ''diamond'' network by making 100 successors of switch $s1$ in Fig. 8 instead of only two. We only changed the rules of switch $s1$, so that it forwarded the packets in the first two rules to 100 successors, and used the rules of switch $s2$ for the latter. For our specification, the model checking for the $Packet$ set with 8 packets succeeded in 18.5 s with the number of threads set to 1, but reported a Java stack overflow error in the case of a greater number of threads after processing one packet. For the adapted specification from [1], it reported such an error already when the first packet had come to the waiting queue of the $s1$ successors, not regarding the number of threads. (This could be seen from the graphical representation of the state graph, which can be obtained optionally after the completion of model checking.)

The model checking of our specification, however, succeeded for any number of threads and took less than 2 seconds for a ''diamond'' with 40 successors of switch $s1$ with 12 threads set in the TLC options. For the adapted specification from [1], it stopped after 14 hours, due to the lack of disk space, generating more than 24 million distinct states during that time. The verification of our specification, for example, also succeeded for a network obtained from this ''diamond'' by continuing the diamond with a chain of 7 switches with the same rules as set in the ''final'' switch of the diamond, analogous to $s4$ in Fig. 8 (thus making the longest path in the network containing 10 switches). It took less than 3 seconds in the case of 8 packets with the number of threads set to 1 or 2, and less than 5 seconds in the case of 16 packets with one thread and around 3 seconds with two ones. With three threads, the stack overflow error was reported in both cases after checking the network for three packets.

Please note that, in all the larger networks, the switch rules were set in such a way that a half of the packets travelled along all the longest possible paths, that some were dropped in the first switch, and that in the diamond-topology networks, some reached at least all the neighbors of the ingress switch.

In order to see the impact of the size of the rule lists, we changed the chain-topology networks consisting of 10 and, respectively, 100 switches by adding rules of the form $[mf \mapsto [dstIP \mapsto n], a \mapsto action, o \mapsto \{2\}]$ at the beginning of the rule list for each switch, for $n = 3, \ldots, 80$ and with $action$ equal to ''outport'' in all the switches except the last one, where the action was set to ''drop''. We added TCP and UDP packets with the source IP address 2 and the destination addresses ranging from 3 to 80, i.e., altogether 156 packets, to the $Packet$ set with 8 packets. The rules were added with the intention that the matching should have been carried out for each of the new packets throughout the chain. The purpose of placing the new rules before the old ones in a switch was also to make it necessary for the model checker to check all the new rules before finding a matched one for every packet from the original $Packet$ set reaching the switch.

At the beginning of the FR list, the same 78 dropping rules for the destination addresses from 3 to 80 were added as in the last switch in the chain, thus ensuring the validity of the consistency invariants for the changed networks.

For the new network with 10 switches and the $Packet$ set with 8 packets, the model checking of the adapted SDN-based firewall specification from [1] finished in around 45 s (compared with 5.3 s for the original chain) and of ours from Fig. 1 it took around 1.3 s (compared to 1 s above). For the enlarged $Packet$ set, the model checking of the former did not finish even in 14 hours, and of the latter it took around 2.3 s. For the new network with 100 switches with the enlarged $Packet$ set, the model checking of our specification took around 1.5 min (compared with 1.5 s for the original 100-node chain with 8 packets).

VI. RELATED WORK AND DISCUSSION

The idea of specifying the behavior of switches as instantaneous mappings of packets from input ports to output ports is similar to the idea used for the purpose of Header Space Analysis (HSA) [8]. HSA is a method for network verification in which packet processing in networking boxes, such as Ethernet switches, routers, or stateless firewalls, is modeled with the so-called switch transfer functions.

The results of experiments reported in Section V indicate that the proposed stateless specification of SDN switches renders the use of TLA+ introduced in [1] appropriate for the verification of practical-sized software-defined networks, consisting of tens of switches with a moderate number of ports and tens of rules, accepting tens of different packets at an ingress port. The network could, for example, be (a part of) an enterprise or wide-area network which accepts

packets from an IP subnetwork at an ingress switch and should transfer them selectively to an egress switch in accordance with the given ''monolithic firewall'' rules. The TLA+ specification of the network with the switch rules to implement the latter could be verified before they are actually placed in the switches by the SDN controller.

The presented TLA+ approach could be extended to the specification of the kinds of networks and the verification of properties similar to those supported by HSA, such as, for example, the verification of reachability of a destination and absence of forwarding loops. However, it could not achieve the efficiency of the HSA method, because the latter uses an effective representation of packet headers in an abstract binary form, which allows very time-efficient automated verification of large real-world networks.

VII. CONCLUSION

We proposed improvements of the approach to the specification and verification of SDN-based firewalls using TLA+ presented in [1]. First, we reduced the possibility of state-space explosion significantly by changing the specification so that it prevented sending packets into the network in all the possible orders. Next, by the stateless specification of switches, we achieved that SDN-based firewalls consisting of tens of switches, for which model checking did not end even after several hours or ended unsuccessfully using the specification with variables, were model-checked in seconds. Additionally, the new specification approach enables one to verify whether an SDN-based firewall allows a packet to pass through the network at least along one of its possible paths.

REFERENCES

[1] Y.-M. Kim and M. Kang, ''Formal verification of SDN-based firewalls by using TLA+,'' *IEEE Access*, vol. 8, pp. 52100–52112, 2020, doi: 10.1109/ACCESS.2020.2979894.
[2] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley, 2002.
[3] L. Lamport, ''The temporal logic of actions,'' *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 872–943, May 1994, doi: 10.1145/177492.177726.
[4] L. Lamport. *The TLA+ Toolbox*. Accessed: Jan. 31, 2023. [Online]. Available: https://lamport.azurewebsites.net/tla/toolbox.html
[5] Y. Jarraya, T. Madi, and M. Debbabi, ''A survey and a layered taxonomy of software-defined networking,'' *IEEE Commun. Surveys Tuts.*, vol. 16, no. 4, pp. 1955–1980, 4th Quart., 2014, doi: 10.1109/COMST.2014.2320094.
[6] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. New York, NY, USA: Kluwer, 1996.
[7] *Profiling*. Accessed: Jul. 5, 2023. [Online]. Available: https://tla.msr-inria.inria.fr/tlatoolbox/doc/model/profiling.html
[8] P. Kazemian, ''Header space analysis,'' Ph.D. dissertation, Dept. Elect. Eng., Stanford Univ., Stanford, CA, USA, 2013.

TATJANA KAPUS (Member, IEEE) received the M.Sc. and Ph.D. degrees in electrical engineering from the Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia, in 1991 and 1994, respectively. She is currently a Full Professor with the Faculty of Electrical Engineering and Computer Science, Institute of Electronics and Telecommunications, University of Maribor. She teaches mainly courses on communications networks and protocols. Her research interests include formal methods for the specification and verification of reactive systems.