Real-Time Verification of Network Properties Based on Header Space

YANG FANG AND YIQIN LU
School of Electronic and Information Engineering, South China University of Technology, Guangzhou 510641, China
Corresponding author: Yiqin Lu (eeyqlu@scut.edu.cn)

This work was supported in part by the Research and Development Program in Key Areas of Guangdong Province under Grant 2018B010113001 and Grant 2019B010137001.

ABSTRACT
The past ten years have seen increasingly rapid advances in the field of network verification. Data plane verification plays a crucial role in this field. Recent developments of SDN, which has been proposed for improving network flexibility and programmability, make the requirement of the dynamic verification. It also provides the ability to dynamically acquire the information of the network state and the rule updating behavior, which makes it possible to perform real-time data plane verification. Previous researches for the real-time data plane verification either perform the verification on the simplified forwarding rules or trace the symbolic headers cross the network behaviors on each switch. Inspired by the discussion of the header space in HSA, we focus on rules themselves and define the computation of transform functions in header space with BDD expressions to get the connection information of rules. The BDD expression makes the transform functions and the rules can be merged as the simplest form for reducing the run time. We propose an updating algorithm based on the matrix operation for incrementally updating rules. The typical network invariants are translated to the requirements of the matrix model. At last, we try to extend the model to verify the invariants in multi-domains networks. The prototype NetV has been compared with NetPlumber and APT on the rule sets of the Standford backbone network and the Internet 2 network. The experiments show that NetV performs better than these two tools on the two rule sets with OpenFlow format. A simple experiment of the real-time verification of two domains is conducted. The result indicates that NetV has advantages over the simulative method by tracking the symbolic headers.

INDEX TERMS
Computer network management, formal verification, software-defined network, header space, network verification.

I. INTRODUCTION
Recently, there has been renewed interest in network verification [4], [5]. Managing a large-scale network is one of the most significant challenges. It is error-prone to set switch rules manually with experience. It is also hard to manually reason the failure of the network because of the scale of large networks and the diversity of protocols. The network operators need to know which packets from one host can reach another host during the debugging process. The effect of the new adding configuration should be explored to keep the network stable and reliable. Inspired by the well-known technologies in the field of software and hardware, the entire network can be seen as a “program” to take packets from the input and delivery those transformed packets to the output.

In the field of network verification, data plane verification has attracted considerable attention since the data plane is more natural to be modeled with the well-understood semantics compared with the control plane. In conventional networks, the snapshot of the configuration and topology of the network can be collected for the off-line verification. Xie et al. [6] first proposed a method for static reachability analysis from a snapshot of the configuration state. Mai et al. [7] and NetSAT [8] perform the data plane verification based on SAT Solver. Al-Shaer et al. proposed ConfigCheck [9] to verify network reachability by using computation tree logic (CTL) and symbolic model. They extended their work and proposed FlowChecker [10], which uses the binary decision diagram (BDD) to encode the
FlowTables and uses the state machine representation to model the interconnection of switches. Al-Haj et al. further extended the work of FlowChecker for adjusting the structure of multiple FlowTables [11]. FLOWER [12] encodes the flow rule sets into Yices assertions. HSA [1] treats the behavior of the switch as a transfer function on packet headers. It uses a geometrical space with bit wildcard expression, which is called header space to represent the set of packets. The reachability is verified by tracing the symbolic packet headers (all wildcard) from all in-ports of the network. Zhang et al. designed two testing protocols for inter-domain loop tests. It utilizes HSA to verify the reachability between the in-ports and the out-ports of a single-domain [13].

Newly proposed “Software-defined Network (SDN)” decouples the control plane from the forwarding plane, which makes it possible for configuring computer networks to be programmable. However, it is naturally associated with increased risks for misconfiguration. By using OpenFlow [14], the centralized controller adds flow entries into flow tables on switches for handling network flows based on the deployed applications. Meanwhile, the centralized observation of the flow tables and the network topology provides the opportunity to verify network behaviors automatically and dynamically. Thus, it raises a new requirement to verify the data plane in real-time, which is more complicated. There are two main gaps in previous works after introducing SDN. Tools based on SAT Solver always provide a single counter-example when the violation happens. It is hard to reason the potential error by the counter-example. They all suffer the dynamic performance issue and the scalability problem.

For addressing these problems, VeriFlow [15], [16] is the first tool that can dynamically verify a rule in real-time before it is installed. The packet equivalence classes (ECs), which represent the packet set following the same network behavior, are computed from the rules stored with the trie structure. The paths are calculated for each EC. However, it is hard to introduce the rewrite action for changing the packet header. Li et al. introduced the field transition rules into VeriFlow for defending covert channel attacks [17]. Since the ECs, which are calculated by the match fields of rules, can not imply that they follow the same paths in the network with the modifying rules, it is hardly used to verify the reachability with the rewrite rules in SDN. NetPlumber [2] presents a method to perform incremental verification by running HSA checks incrementally. The wildcard expression causes the explosion of the expressions after the minus operation in the header space. Yang and Lam proposed Atomic Predicates (APs) Verifier [18] to compute reachability trees. The forwarding tables are merged into the port predicates for representing packet filters. They precompute the packet ECs with BDD expressions and then label each EC a unique integer identifier, which is called the atomic predicate. The reachability trees are obtained by traversing the identifiers which are injected into the in-ports of the network. Using identifiers reduces the computation of traversing the symbolic packet sets with BDD expressions. They improved their work and proposed APs for Transformers (APT) [3] for scaling to large networks with transformers. The APs are computed by iteratively using the new predicate, which is transformed by the transformers on the current APs to update the APs. The dynamic performance is limited by the incremental computation of the APs when updating a rule.

In this paper, inspired by HSA [1], we redefine the function of rules based on header space and separate the header transformation and the position transformation into two dimensions. The header transformation only changes packet headers. Then, we use the matrix structure to model the connection of rules. The position transformation is implied in the matrix structure. The element of the matrix is the header transformation on header space, while it also implies the position transformation from the input of a rule to the input of another rule. Different from the HSA, we define the computation of the composition between two rewrite actions rather than performing rewrite actions on the symbolic headers. For example, assuming a rewrite rule with parameter xxxx0010 and a next-hop rewrite rule with parameter xxxx01xx, they can be treated as one rewrite rule with parameter xxxx0110 if we only consider the reachability from the source to the destination. The definition and computation provides two advantages. If we change the constraint of the input after we complete the computation of a specific field, it only needs to do a small scale computation with the changes to get the reachability changes of the whole network rather than to do the simulation again on the symbolic packets of the changes across the field. Second, the equivalence of the header transformations makes some rules that can be merged as a single expression without losing their functionality since a flow entry may contain multiple changing header actions. Thus, it makes this method can be easily extended to the multi-layer controller for verifying the inter-domain invariants. We then use the BDD expression to represent the domain and the range of the transformation for eliminating the uncertainty and the explosion of expressions. Thus, two rules can be merged as an abstract one with a BDD domain if they have the same functionality. We also propose the incremental updating algorithm to update the connectivity matrix, which represents the complete connection information of the network. The common invariants are represented by using the model and can be incrementally verified when updating rules. The experiments show that it performs better than NetPlumber and APT.

The contributions of this paper are:

1) We redefine the rule function based on header space and present the algorithm of transformation and inverse transformation on the binary decision diagrams (BDDs) presentation. It also makes the rules can be merged based on its functionalities, which remarkably speeds up the updating process.

2) We further define the transform function and use the matrix structure to model the connection of the rules. We then propose an updating algorithm to dynamically update
rules and maintain the reachability information of the entire network.

3) We present the primary invariants verification based on this model and then extend it to the multi-layer controller for verifying the inter-domain invariants.

The remaining parts of this paper proceed as follows: Our basic network model, which including the definition of the rule function, the transform function on BDD expressions, and the fundamental adjacency matrix is presented in Section III. The dynamically updating algorithm is presented in Section IV. In Section V, we present the verification of some common invariants and the inter-domain loop verification. The performance evaluation of NetV is presented in Section VI. The conclusion is discussed in Section VII.

II. RELATED WORKS

This paper focuses on real-time data plane verification. Toward the off-line Verification, Xie et al. [6] first proposed a method that jointly reasons about how each behavior affects the reachability from a snapshot of the configuration state for each router for static reachability analysis. Mai et al. [7] and NetSAT [8] perform the data plane verification based on SAT Solver. Anteater collects the snapshot of the topology and the forwarding information (FIBs) and models the invariants as the instances of SAT. Then, the SAT solver is used to verify whether the instances are satisfied. Al-Shaer et al. proposed ConfigCheck [9] for reachability verification from the configuration. They use the state transition to model the network and construct a transition relation for each device. The BDD is used to represent the state and verify if the symbolic state satisfies the state of the transition relation. They also use CTL to represent security properties. They extended their work and proposed FlowChecker [10] for OpenFlow switches by encoding the FlowTable matching semantic with BDDs. For the intra-federated flow isolation, they make pairwise comparisons for each pair in the domain. Al-Haj and Tolone further extended the work of FlowChecker for adjusting the structure of multiple FlowTables by treating the table as the location of the state instead of the device to check the FlowTable pipeline misconfiguration [11]. FLOVER encodes the flow rule set into Yices assertions for verifying the property by checking if the property formula is unsatisfiable [12].

Their works meet two issues for practice. First, the verification time cost is high. Second, the SAT/SMT Solver based tools only return a single counter-example for the violation, which is hard to be used to analyze the reason for failures. HSA [1] uses transfer functions to represent the switches. They trace the symbolic packet headers which are injected from the source node as the transfer functions of the switches transform them. They also translate the invariants to the requirement of the reachability. They can get all the information on reachability and the change of the packet headers from the source node. However, the introduction of SDN provides the ability to dynamically get the network state changes and the updating of the data plane rules. It also makes a requirement of the real-time network verification.

Toward the real-time data plane verification, VeriFlow [15], [16] was designed for real-time verifying of network-wide invariants. The filters of rules which are stored in a trie tree are computed as the ECs by using a complicated method. When updating a rule, they compute the affected ECs and then generate a forwarding graph for each affected EC by connecting the involved forwarding rules. They speed up the real-time verification by computing the affected ECs for the forwarding rules. However, the ECs can not be used for the header change rules, since the headers may change through a path and the path can not be represented as the same filter on each switch. Recently, Li et al. [17] introduced the field transition rules into VeriFlow to defend the covert channel attacks. The generated ECs can be periodically updated when updating rules. The bi-directional forwarding graphs are constructed for each maintained EC. When adding a header change rule, they correlate the new generated ECs and the affected ECs after header change and then compare the forwarding graphs. However, the header change rules still can not take action in the forwarding graphs for verifying the reachability.

NetPlumber [2] performs incremental verification by running HSA checks incrementally. They use rules instead of switches to be the nodes of the graph. The pipes between rules reduce the computations when propagating flows from the pipes. The incremental update is done by maintaining the flows (symbolic headers) that are propagated from the source node on these pipes. When updating a rule, the flows before the rule are propagated through the newly updated pipes. Zhang et al. [13] designed two testing protocols for inter-domain loop tests. They use the HSA to get the reachability of the given headers between the in-ports and the out-ports of a single-domain and check if there are inter-domain loops. Following this concept, they can also use NetPlumber for the real-time inter-domain loop test. Compared with the NetPlumber, we use BDD expression to resolve the explosion of the wildcard expression after multiple union and difference operations. The merged rules reduce the redundancy of the rules that perform the same behavior with different wildcard expression filters. The composition operation of the transform function in header space and the matrix operation provides a more flexible way to acquire the desired end to end reachability associated with a transform function.

Yang and Lam proposed APs (Atomic Predicates) Verifier [18] to verify the reachability of the network. They translate the forwarding rules to the port predicates. They reduce the propagating time by precomputing the APs associated with an integer identifier and propagating these identifiers for the all-pairs reachability. They improved their work and proposed APT [3] for scaling to large networks with transformers. The APs with transformers are computed by iteratively getting the transformed predicates from the current APs and using these predicates to update the APs. Their experiments show that the time of traversing the identifier is short. However, the dynamic performance of the APT is limited by the per update run time of the APs, which is
strongly related to the number of the existed APs. The APT is suitable for the network with many rules which generate little APs. From our experiment in Section VI-D, the rule set generates many APs, and it takes non-negligible time for the per rule update.

III. NETWORK MODEL

We model an SDN network as a digraph of the symbol rules which contain the forwarding action and the header transforming action. Each abstract rule can be the SDN rule directly or the merging of the SDN rules. The basic rule function is defined based on the header space at first. Then we design an algorithm to do rewrite action on the BDD expression for the composition of functions in header space. At the end of the section, the adjacency matrix is build to represent the connection of rules. The operation on the matrix reflects the composition and the merging of the functions. The symbols used in this section are described in Appendix A.

A. FUNCTIONS OF RULES

HSA [1] has defined the header of a packet as $[0, 1]^L$, where $L$ is the length of the header. This space is called Header Space, $\mathcal{H}$. The position of a packet is identified by a directed link in the network. We extend their definition of Network Space and model the network as a set of links and the rules which connect these links. A packet is defined as $(h, l)$ in the network. $h$ is the header, and $l$ is the position of the packet. It represents a packet with a header traversing on a link (including the virtual link between flow tables). The identifiers of the switches ports are replaced by the identifiers of the links. Therefore the space $[0, 1]^L \times \{l_1, \ldots, l_N\}$ is called the network space, where $l_i$ is the integer that represents the unique directed link in the network.

In SDN, a rule is a tuple of Match fields and Action set. The action set is an ordered set and ends up with a forwarding action. In OpenFlow, there are two main types of actions: the action of modifying the headers of the packets and the action of forwarding the packets. The notations of the two actions are written as $a_m$ and $a_f$. They can be represented as functions. The two types of actions take action in different dimensions. The forwarding action only changes the packet’s position and can also copy the matched packet to different ports. Thus we introduce the “$+$” to represent the increase of the packets’ counts. The forwarding action $a_f$ is written as $a_f(l) = l_i + l_j + \ldots + l_k$ that means the $a_f$ copy and forward the matched packet to the specific links.

The modifying action $a_m$ only changes the header fields of packets. It is written as $a_m(h) = h^*. $ The encapsulation and de-encapsulation actions are defined in Appendix B. In the main body of this paper, we only consider the rewrite function $a_{rw}$. For the rewrite function, every bit $h^*_i = (m_2,i \land m_1,i) \lor v_i$ with the condition $m_1 \land v_i = 0$, where $m$ is the mask of the rewrite value, $v$ is the rewrite condition, and the condition means that if the specific bit of $m$ is 1, the corresponding bit of $v$ is 0. Therefore, the composition of two rewrite actions $a_{rw,c}(h) = a_{rw,2}a_{rw,1}(h) = h^*$, and for every bit $h^*_i = (m_2,i \land m_1,i) \lor v_i \lor (m_2,i \land v_1,i) \lor v_2,i$. The parameters of $a_{rw,c}$ are computed as $m_i = m_2,i \land m_1,i$ and $v_i = (m_2,i \land v_1,i) \lor v_2,i$. The composition of more modifying actions can be recorded as one. For example, assuming two rewrite actions $a_{rw,1}$ and $a_{rw,2}$ with parameters $(11000000, 00101101)$ and $(11110000, 00001101)$, which can be written as xx101101 and xxxx0110 in wildcard expression, the parameter of composition $a_{rw,c}$ is $(11000000, 01100110)$. If a header $h$ of a packet is 10000001, the $a_{rw,1}(h)$ is 10101101. The $a_{rw,2}a_{rw,1}(h)$ is the same as the $a_{rw,c}(h)$, and the result is 10100110.

A rule can be modeled using the composition of actions that maps a packet with header $h$ arriving on link $l$ to a packet set. A rule is defined as $r_i(h, l) = (a_m(h), a_f(l)), \text{ where } (h, l) \in X_i$ (1)

where $X_i$ is the domain of $r_i$, which is always the wildcard expression of the match fields of a rule. Therefore, the composition of the two rules is defined as $r_ir_j(h, l) = (a_m,i(a_m,j(h), a_f,i(a_f,j(l))), \text{ if } (h, l) \in X_i \land r_j(h, l) \in X_j$ (2)

If the conflict part of two rules is exiled, the merging of two rules in the same table is defined as $r_i + r_j(h, l) = \begin{cases} (a_m,i(h), a_f,i(l)), \text{ if } (h, l) \in X_i X_j(h, l) \end{cases}$ (3)

Two rules take action with disjoint domains. If the action sets of two rules are the same, the merging can be written as a single one with domain $X_i \cup X_j$.

B. THE MODIFYING ACTION ON BDD EXPRESSION

The data structure for representing the domain and the range is the wildcard expression in HSA [1]. If the given bit is the wildcard $*$, it represents a set that contains the values 0 and 1. However, the wildcard expressions will cause the explosion of expressions after union or minus operations even after Lazy Subtraction optimizing. The union and the minus operations may cause the explosion and the non-unique wildcard expressions. For example,

$10xxxxx - 1011xxxx = 100xxxx \cup 1010xxxx = 10x0xxxx \cup 1001xxxx$. The result can be written as two different wildcard expressions. Thus different wildcard expressions should be checked for the equivalence between two of them. The expression of the domain will be more and more complicated if the minus operation continues. AP [18] has discussed the desirable properties of BDDs [19] for representing header space. Therefore, we use the BDD expression to represent the header fields of the domain and the range. Thus, if the actions and the in-ports of $r_i$ and $r_j$ are the same in (3), the merging of $r_i$ and $r_j$ can be written as $(r_i + r_j)(h, l) = (a_m(h), a_f(l)), if \ (h, l) \in X_i \cup X_j$. The number of nodes in
Algorithm 1 Set the Given Fields to x(The Wildcard) for BDD (Set_x(BDD, mask_BDD))

Require: the processing BDD root node a, and the given mask BDD root node b.

1: if a < 2 or b < 2 then
2: return a {a or b is the terminal node}
3: end if
4: if a.level = b.level then
5: result = set_x(a.low, b.low) ∪ set_x(a.high, b.low)
6: else if a.level < b.level then
7: result = make_node(a.level, set_x(a.low, b), set_x(a.high, b))
8: else
9: result = set_x(a, b.low)
10: end if
11: return result

Algorithm 2 Compute the Domain or the Range of a Rule With Modifying Action

Require: domain X or range Y with BDD presentation

1: $T_{rw}(X) = B_{rw} \cap set_x(X, mask)$
2: $T_{rw}^{-1}(Y) = X \cap set_x(Y, mask)$

BDD $|X_{h,i} \cup X_{h,j}| \leq |X_{h,i}| + |X_{h,j}|$. This operation can reduce the redundancy of rules with the same actions.

The forwarding action does not do anything with the header space of packets. Since in the following sections, the forwarding action of a rule is implied in the matrix expression, we separate the modifying function from the rule function and only consider the composition of modifying actions and the domains and ranges on the header space. The parameters of the composition of modifying actions are computed with the wildcard expression for reducing the additional cost of BDD operations. We present the algorithm for computing the domain and the range of rules by the opposite ones with the BDD operations. We present the algorithm for computing the domain or the rewritten values. The rewrite action fills the modified fields with the given values. It is equal to the expression in APT [3]. We extend it to the inverse process for computing the domain by the range of the action. The inverse process fills the modified fields with the values of the corresponding fields of the domain. For example, if a rewrite action $(11110000, 00000101)$ modifies a domain 10xxxxx, the result of setting wildcard is 10xxxxx. The $B_{rw}$ is the BDD expression of xxxx0101, which represents the modified value. The final result is 10xxxxx ∩ xxxx0101 = 10x0101.

If we narrow the range to 10x10101, the result of setting wildcards of the range is 10x1xxxx. The result of the reverse is 10x1xxxx ∩ 10xxxxx = 10x1xx10.

Algorithm 1, which is used in Algorithm 2, is the BDD operation to set the given variable to wildcard. Although the basic idea is equal to the expression $∃x.P = P|_{x=true} \lor P|_{x=false}$ in APT [3], the Algorithm 1 do the computation in a basic iteration process of a BDD operation rather than using the “restrict” and the “or” operations a couple of times. The experiment in Section VI-A shows that Algorithm 1 is eight times faster. The required mask BDD of the above example is xxxx0000. The modified bit is set to 0, and the unmodified bit is set to the wildcard. If a node is required to be set to the wildcard, it will be replaced by the union of its child nodes. The reduce operation is done in the make node function. The structure of transform function with the BDD domain can also support the encapsulation and the de-encapsulation. The experiments do not include the encapsulation of a new instance of the protocol header. The details of the encapsulation and the de-encapsulation are described in Appendix B.

C. GRAPH AND MATRIX EXPRESSION

In SDN, as shown in Fig.1, rules are connected by the physical links between devices and the virtual links between flow tables. If two rules are connected by a link, and the range of the rule on the input side intersects the domain of the rule on the output side, the two rules are adjacent. A rule graph $G_R$ is a digraph that is constructed by $(V, E)$, such that $V$ is the set of rules, and $E$ is the set of connections of the adjacent rules. Therefore, a directed edge between two rules represents the filter of the match fields and the action on the filtered packets between the inputs of the two adjacent rules. These directed edges can locate the links.
As shown in Fig. 2, we give a simple example of 5 rules. The rules are located in 5 tables. The edge \( e_{12} \), which is located at the link \( l_{12} \), represents the function of the connection between rules \( r_1 \) and \( r_2 \) in the example. We establish the adjacency matrix \( R = (r_{ij}) \).

\[
r_{ij}(h) = \begin{cases} 
a_{ij}(h) = a_{m,j}(h), & \text{if } h \in X_{ij} \\
\emptyset, & \text{otherwise} \end{cases} \tag{4}
\]

The pair of two adjacent rules \( r_{ij} \) means the directed edge from \( r_j \) to \( r_i \). Thus, the forwarding actions are implied in the structure of the matrix. If the conflicts of rules are removed, the elements \( r_{ij} \) of the adjacency matrix are the functions in header space. The function of initial \( r_{ij} \) in header space is the same as the modifying action of the rule \( r_j \). The cols of \( R \) represent the connection from the corresponding rules to the next-hop rules. The domain and the range of \( r_{ij} \) are narrowed by \( r_1 \) and \( r_j \). It is computed by \( Y_{r_j} = Y_{h,r_j} \cap X_{h,r_j} \) and \( X_{r_j} = a_{m,j}^{-1}(Y_{r_j}) \), where \( Y_{h,r_j} \) is the header space range of \( r_j \) and \( X_{h,r_j} \) is the header space domain of \( r_j \). Thus each element of \( R \) represents the one-hop connectivity of two rules. The constructed adjacency matrix of Fig. 2 is

\[
\begin{bmatrix}
0 & f_{12} & 0 & 0 & 0 \\
f_{21} & 0 & 0 & 0 & 0 \\
f_{31} & 0 & 0 & 0 & 0 \\
f_{41} & f_{42} & 0 & 0 & 0 \\
0 & 0 & f_{53} & f_{54} & 0
\end{bmatrix}
\]

The rewrite parameter of \( f_{12} \) is xxxxxxx00, and the domain \( X_{f_{12}} \) is 00xxxx10. The rewrite parameter of \( f_{21} \) is xxxxxxx, and the domain \( X_{f_{21}} \) is 00xxxx10. The others can be computed in the same way.

The product of two elements \( r_{jk} \) and \( r_{ij} \) is almost the same as the composition of two rules.

\[
r_{ij}r_{jk}(h) = \begin{cases} 
a_{ij}a_{jk}(h), & \text{if } h \in a_{jk}^{-1}(Y_{jk} \cap X_{ik}) \\
\emptyset, & \text{otherwise} \end{cases} \tag{5}
\]

The product represents the connectivity from \( r_k \) to \( r_i \), and the path goes through \( r_j \). The plus of two elements is defined as

\[
(r_{ik} + r_{jk})(h) = \begin{cases} 
a_{ik}(h), & \text{if } h \in (X_{ik} - X_{jk} \cap X_{ik}) \\
a_{jk}(h), & \text{if } h \in (X_{ik} - X_{ij} \cap X_{jk}) \\
a_{ik}(h) + a_{jk}(h), & \text{if } h \in (X_{ik} \cap X_{jk}) \\
\emptyset, & \text{otherwise} \end{cases} \tag{6}
\]

If \( h \in X_{ik} \cap X_{jk} \), the plus operation implies that there exists a copy operation of the packet with the header \( h \).

Therefore the multiplication of two adjacency matrices \( R^2 \) represents the 2-hops connection of rules. The \( R^n \) is the \( n \)-hops connection matrix. An abstract element “1”, which does nothing when multiplied by a matrix element, is defined as \( 1 \circ r_{ij} = r_{ij} \circ 1 = r_{ij} \). The identity matrix \( E \) is a diagonal matrix in which the elements on the main diagonal are equal to “1”. The connectivity of entire rules is represented as a connectivity matrix

\[
R_c = E + R + R^2 + \ldots + R^N = E + \sum_{n=1}^{N} R^n \tag{7}
\]

where the \( N \) is the limit of max hops in the given network. The elements of \( R_c \) represent connectivities of any corresponding two rules. The connectivity matrix of Figure 2 is

\[
\begin{bmatrix}
g_{11} + 1 & g_{12} & 0 & 0 & 0 \\
g_{21} & 1 & 0 & 0 & 0 \\
g_{31} & g_{32} & 1 & 0 & 0 \\
g_{41} & g_{42} & 0 & 1 & 0 \\
g_{51} & g_{52} & g_{53} & g_{54} & 1
\end{bmatrix}
\]

where \( g_{11} = f_{12}f_{21} \), the rewrite parameter of \( g_{11} \) is xxxxxxx00, and the domain \( X_{g_{11}} \) is 00xxxx10. The other elements can also be computed using (7). The basic reachability of the two rules can be acquired by checking the corresponding element. For example, \( g_{41} \) represents all the flows that go through from the input of \( r_1 \) to the input of \( r_4 \) and \( g_{41} = f_{41} + f_{21}f_{31} + f_{41}f_{12}f_{21} \).

\[
g_{41}(h) = \begin{cases} 
(a_1 + 2a_2)(h), & \text{if } h \in 0010xx00 \\
\emptyset, & \text{otherwise}
\end{cases}
\]

where the rewrite parameter of \( a_1 \) is xxxxxxxx, and the rewrite parameter of \( a_2 \) is xxxxxxx00. It means that if a packet whose header belongs to 0010xx00 is delivered to the input of rule \( r_1 \), the three copied packets will be delivered to the input of rule \( r_4 \) through the network.

IV. INCREMENTAL UPDATING

A. PREPROCESSING DURING UPDATING

The purpose of the preprocessing during updating is to get the difference of the adjacency matrix after updating a rule. We use the method in our early work [20] to eliminate the conflicts and get the dominant part of the new adding rule and the covered part of the existing rules. As shown in Fig. 4 for each link, we construct a link MTBDD on the link-out side. Each terminal node of the MTBDD represents the dominant rule on the link-out side. When a new rule is added, we hash the rule to a merged rule by the action set and the matched links of the rule. The generated MTBDD of the new rule is added to the link MTBDD by an MTBDD operation. When two terminal nodes meet, in addition to the basic operation, we also need to update the dominant BDDs on the link and the covered rule records of the two rules and update the dominant BDDs of the corresponding merged rules. Then the changes of the merged rule will be connected to the previous-hop rule and the next-hop rule. The result is the difference of the adjacency matrix after adding a rule.

Deleting a rule is similar to adding one. When two terminal nodes meet, if two nodes are the same node, which represents the removing rule, return the “false” node. Otherwise, update the two rules and the corresponding merged rules. Besides, the covered parts of the lower priority rules that are covered by the delete rule should be re-added to the MTBDD.
Adding a new link creates the connection of the existing merged rules. Therefore, we should connect the merged rules between the adding link to get the difference of the adjacency matrix. The MTBDDs of the matched in-ports (the rules that match all in-ports in of a switch are preserved as an abstract in-port MTBDD) of rules are precomputed. When a new link arises, the link MTBDD can be got by the corresponding in-port. Thus, we only need to compute the connection of the merged rules between the link and generate the difference of the adjacency matrix.

**B. UPDATING THE MATRIX**

For formal description, we construct two matrices $L_d$ and $L_s$ to represent the outputs and the inputs of selected rules. For example, the in-link of the network in Figure 3 is constructed as $l_s = \begin{bmatrix} f_{s,11} & 0 & 0 & 0 & 0 \end{bmatrix}^T$, where the rewrite parameter of $f_{s,11}$ is xxxxxxxx and the domain of $f_{s,11}$ is 0010xxxx. The out-link of the same network is constructed as $l_d = \begin{bmatrix} 0 & 0 & 0 & 0 & f_{d,15} \end{bmatrix}$, where the rewrite parameter of $f_{d,15}$ is xxx00000 and the domain of $f_{s,11}$ is 0010xxxx. The elements of $L_s$ construct an abstract rule which delivers given packets to the selected rules. In $L_d$, these elements are initialed by the modifying action of the selected $r_i$, which represents the connection between $r_i$ and $l_j$. Therefore, the desired reachability checking is $L_s \rightarrow \mathcal{R} L_d$. The result of the example network in Figure 2 with the $l_{s,1}$ and $l_{d,1}$ is a single element

$$f_{s \rightarrow d}(h) = \begin{cases} (2a_3 + 2a_4)(h), & \text{if } h \in 0010xx10 \\ (a_3 + a_4)(h), & \text{if } h \in 0010xx00 \\ a_4(h), & \text{if } h \in 0010xx11 \\ \emptyset, & \text{otherwise} \end{cases}$$

where the rewrite parameter of $a_3$ is xxx00000, and the rewrite parameter of $a_4$ is xx110000.

The updating problem of rules is defined as computing $L_s^* R_s^* L_d^*$ from $R^*$, where $R^*$ is the updated adjacency matrix, and it is represented as:

$$R^* = \begin{bmatrix} R & \Delta R \\ x_1^T & 0 \end{bmatrix}$$

where $x_1$ and $x_2$ are $n$-dimension vectors. When a new rule is adding to the network, the covering part of the new rule is found by using the MTBDD based method. The $\Delta R$ represents the change parts of existing rules after a new rule is added. Figure 3 shows an example of adding a new rule $r_6$. The $\Delta R$ is

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -f_{\Delta,53} & 0 & 0 \end{bmatrix}$$
where the rewrite parameter of \( f_{\Delta,53} \) is xx11xxxx, and the domain of \( f_{\Delta,53} \) is also 000100xx. The \( x_1 \) is \([0 0 0 0 f_{56}]\), and the \( x_2 \) is \([f_{60} 0 0 0 0]\), where the rewrite parameter of \( f_{56} \) is 00xxxx, and the rewrite parameter of \( f_{60} \) is xxxxxxxx. The domain of \( f_{56} \) is xx0100xx, and the domain of \( f_{60} \) is also 000100xx.

Similarly, we get the updated \( L^*_s = [L_s l_s]^T \) and \( L^*_d = [L_d l_d]^T \). If the new adding rule does not directly connect the \( L_s \) and \( L_d \), the \( l_s \) and \( l_d \) are zero. When a new rule is adding into the network, we compute the changed part of the adjacency matrix. It is written as

\[
R^* = R + R_{\Delta,1} = \begin{bmatrix} R & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} \Delta R & x_1^T \\ x_2 \end{bmatrix} \tag{10}
\]

The product of two updated adjacency matrix is \( R^2 = R^2 + RR_{\Delta,1} + R_{\Delta,1}R + R_{\Delta,1}^2 \). Thus, the exponentiation of the adjacency matrix is computed by

\[
R^n = R^* + R_{\Delta,n} = R^* + R^n - 1R_{\Delta,1} + R + R_{\Delta,n-1}R_{\Delta,1}. \tag{11}
\]

The updated \( R^*_c \) is got by \( R^*_c = E + \sum_{n=1}^{N} R^n \) and the result is

\[
R^*_c = R_c + R_{\Delta,c} = R_c + \sum_{n=0}^{N} R_{\Delta,n}. \tag{12}
\]

where the \( R_{\Delta,0} \) is

\[
\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}
\]

The final checking is computed by \( L^*_d R^*_c L^*_s \), and the result is

\[
L^*_s \rightarrow d = L_s \rightarrow d + \Delta L^*_s \rightarrow d = L_s \rightarrow d + L^*_d R_{\Delta,c} L^*_s. \tag{13}
\]

Therefore, Algorithm 3 shows the process of updating new rules. In Fig 3, the \( R_{\Delta,1} \) of new adding \( r_6 \) is

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & f_{\Delta,53} & 0 & 0 & f_{\Delta,56} \\
f_{\Delta,61} & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

Therefore, after the loop, the \( R_{\Delta,c} \) is

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
f_{\Delta,51} & f_{\Delta,52} & -f_{\Delta,53} & 0 & 0 & f_{\Delta,56} \\
f_{\Delta,61} & f_{\Delta,62} & 0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

The difference of the final checking result is the \( l^*_d,1 R_{\Delta,c} l^*_s,1 \), and the element is

\[
f_{\Delta,s \rightarrow d}(h) = \begin{cases} 2(a_5 - a_4)(h), & \text{if } h \in 00100010 \\
(a_5 - a_4)(h), & \text{if } h \in 001000xx-00100010 \\
\emptyset, & \text{otherwise} \end{cases}
\]

\[
\text{Algorithm 3 Update New Rules}
\]

\textbf{Require:} The new adding rule \( r \), the maintained connection matrix \( R_c \), and the maintained checking result \( L_{s \rightarrow d} \).

1: Compute \( \Delta R, x_1^T, \) and \( x_2 \) by using the MTBDD based method.
2: \( R_{\Delta,1} \leftarrow \) using (9)
3: \( R \leftarrow \) using (9)
4: \textbf{for} \( i = 2 \text{ to } N \) \textbf{do}
5: \( R_{\Delta,1} R^i \leftarrow R_{\Delta,1} R^{i-1} R \)
6: \textbf{end for}
7: \( R^*_c \leftarrow \) using (12)
8: \( L_{s \rightarrow d} \leftarrow \) using (13)
9: Detect the desired invariants.
10: \textbf{return} \( R^*_c \)

where the rewrite parameter of \( a_5 \) is 00xx0000. Deleting a rule is almost the same as adding it except that

\[
R_{\Delta,1} = \begin{bmatrix} -\Delta R & -x_1^T \\ -x_2 & 0 \end{bmatrix}
\]

\textbf{Link Up and Link Down:} Similarly, when a new link rises, the MTBDD of this link should be established for computing the increment of the adjacency matrix. It is represented as \( \Delta R \) in (9). It can also be written as \( R_{\Delta,1} \) since the new link does not add a new rule. The updated network is computed by Algorithm 3. When a link goes down, the process is the same as adding a new link. The MTBDD of this link is frozen for a fixed time for dealing with the fast recovery.

\section{V. CHECKING INVARIANTS}

The invariants are checking during the updating. In this section, we mainly describe the intra-domain problems and the inter-domain forwarding loop.

\subsection{A. INTRA-DOMAIN INVARIANTS}

1) \textbf{REACHABILITY AND REACHABILITY VIA A WAY-POINT}

The reachability is the fundamental property in network verification. The connection matrix reflexes the reachability from a rule to another rule. We check for the reachability constraints by constructing the proper \( L_s \) and \( L_d \). The production \( L_d R_c L_s \) represents the reachability between the given position with the desired constraints.

If we want to know the reachability from \( l_{12} \) to \( l_d \) with the limited headers 00xxxxx, the corresponding

\[
l_{s,2} = \begin{bmatrix} 0 & f_{s,23} & 0 & 0 & 0 & 0 \end{bmatrix}^T \\
l_{d,1} = \begin{bmatrix} 0 & 0 & 0 & 0 & f_{d,15} & 0 \end{bmatrix}
\]

where the rewrite parameter of \( f_{s,23} \) is xxxxxxx and the domain is 00xxxxx. The \( l_{d,1} \) is presented in Section IV.
The result of $l_{d,1}R_{n}l_{n,2}$ is an element with the function

$$f_{s\rightarrow d,12}(h) = \begin{cases} (a_5 + 2a_3)(h), & h \in 00100010 \\ (2a_3 + a_4)(h), & h \in 0010xx10 \\ a_4(h), & h \in 00xxxx10 \\ \emptyset, & \text{otherwise} \end{cases}$$

where the rewrite parameters are described in Appendix A. This function represents the reachability from $l_{12}$ to $l_d$. If a packet with header 00100010 is sent to $l_{12}$, the $l_d$ will receive three packets with the header 00100000.

Reachability via a way-point is split into two successive checks of reachability. Suppose we wish to ensure that all traffic from $l_i$ to $l_d$ should go through the $S_d$ where the rule $r_4$ locates in. We compute the reachability $f_{s\rightarrow r_4}$ from $l_i$ to $r_4$ and the reachability $f_{r_4\rightarrow l_d}$ from $r_4$ to $l_d$. Then the composition $f_{r_4\rightarrow l_i}f_{l_i\rightarrow l_d}$ should be compared with the reachability from $l_i$ to $l_d$ that is $f_{s\rightarrow d}$. If they are equal, all the traffic from $l_i$ to $l_d$ do pass the $S_d$.

If the path of the packets is needed, every rule associated with the in-port of a switch will be attached with a push action to push the switch index to the path. They can also be computed as an additional composition. As an example, the $l_{d,1}R_{n}l_{n,2}$ will become

$$f_{s\rightarrow d,12}(h) = \begin{cases} (a_{1,1} + a_{1,2} + a_{1,3})(h), & h \in 00100010 \\ (a_{1,2} + a_{1,3} + a_{1,4})(h), & h \in 0010xx10 - 00100010 \\ a_{n,4}(h), & h \in 00xxxx10 - 0010xx10 \\ \emptyset, & \text{otherwise} \end{cases}$$

where the rewrite parameters of $a_{1,2}$ and $a_{1,3}$ are the same as the parameter of $a_3$. However, the path adding the parameter of $a_{1,2}$ is ($S_1, S_2, S_1, S_4, S_5$), and the path adding the parameter of $a_{1,3}$ is ($S_1, S_4, S_3$). If a packet with header 00100010 is sent to $l_{12}$, they are delivered to $l_d$ through 3 different paths and finally have the same header. In this way, the reachability via a way-point can be checked by the paths. However, the processing with paths decreases the performance of the composition of the two elements.

2) LOOPS AND THE LIMIT OF MAX HOPS

When a packet returns to a rule twice, the loop occurs. The loop in the network can be checked during the updating processing. The main diagonal of $R_n$ represents the reachability from a rule to itself by $n$-hops. Therefore, if the element of the main diagonal is not equal to “0”, there are some packets that go through this rule to be transmitted to this rule again, which is called the loop.

**Definition 1 (Generic Loop):** Suppose $R$ is the adjacency matrix of rules of a given network. $r_{n,ii}$ is the $i$-th element of the main diagonal of $R^n$. The $n$-hop generic loop of $r_i$ exists is defined as $\exists h \in X_{n,ii}, r_{n,ii}(h) \neq \emptyset$, where the $X_{n,ii}$ is the domain of $r_{n,ii}$.

If the packets that a rule receives twice will go through the path again, it will be the infinite loop, which is defined in Definition 2.

**Definition 2 (Infinite Loop):** Suppose

$$r_{n,ii} = \begin{cases} a_{n,ii,1}(h), & (h) \in X_{n,ii,1} \\ \ldots & \ldots \\ a_{n,ii,M}(h), & (h) \in X_{n,ii,M} \\ \emptyset, & \text{otherwise} \end{cases}$$

is in the simplest form. The $n$-hop infinite loop of $r_i$ exists is defined as $\exists h \in X_{n,ii,k} k \in (1, 2, \ldots, M)$ let $a_{n,ii,k}(h) \in X_{n,ii,k}$.

Therefore, after we get $R_c$, the loop under $N$-hop can be detected by Theorem 3.

**Theorem 3:** Suppose the connection matrix $R_c$ and $r_{c,ii}$ is the main diagonal element of $R_c$, $r_{c,ii} - 1 \neq 0$ implies that there exists at least a GenericLoop from rule $r_i$ to itself. In addition for the simplest form of the $r_{c,ii}, \exists h \in X_{c,ii,k} k \in (1, 2, \ldots, M)$ let $a_{c,ii,k}(h) \in X_{c,ii,k}$ implies that there exists at least an Infinite Loop.

The details and the proof are presented in Appendix C. In (8), $c_{11} - 1$ is $g_{11}$ and $g_{11} \neq 0$.

Thus, there is a loop from $r_1$ to $r_1$. However, the range of $g_{11}$ is $00xxxx00$ and $00xxxx00 \cap 00xxxx10 = \emptyset$. This loop is not an infinite loop.

During the processing of adding a rule to the network, we get $R_{N,c}$. The loop then can be checked by the main diagonal element of $R_{N,c}$. The loop over $N$-hop implies max-hops failure. The max-hops failure is identified by checking whether the $R^{N+1}$ has a non-zero element. During the Algorithm 3, we get the $R_{N,c}$. Suppose the existing rules have been checked already, $R^{N+1} = 0$. By using (11), the $R_{N,c}+1$ can be computed and used to checking the max-hops failure.

3) BLACK HOLES

A black hole is that a set of packets are dropped due to no entry matched. The basic leakage of a link can be detected during the preprocessing. If the sum of the domains of rules on the same in-port does not fulfill the all wildcards expression, there is a leakage on this in-port. We construct a fake rule which represents the “leakages” of every link. A vector $l_{d,3}$, which represents the abstract “leakage port,” is built to extract the reachability from the rules or the links to the “leakages.” The reachability from the in-port of the network to the “leakage port” is called the black hole.

In Fig 3, there is not a rule with “drop action.” $l_{d,3} = [0 0 0 0 0 0 1]$ The adjacency matrix with the “leakage” rule is

$$R^* = \begin{bmatrix} R & 0 \\ l_k & 0 \end{bmatrix}$$
where $lk = [f_{lk,1} f_{lk,2} f_{lk,3} 0 0 0]$. 

$$f_{lk,1}(h) = \begin{cases} 
2a_1(h), & \text{if } h \in 000xxxx - 00xxx10-0010xx00 \\
a_1(h), & \text{if } h \in 00xxx10+0010xx00 \\
\emptyset, & \text{otherwise}
\end{cases}$$

The $l_{s,1}$ with the “leakage” rule is $[f_{s,1} 0 \ldots f_{s,17}]$. The element of $l_{d,3R^cA_{l,s}}$ is

$$f_{s\rightarrow d,13}(h) = \begin{cases} 
(a_2 + a_6 + a_1)(h), & \text{if } h \in 0010xx10 \\
2a_1(h), & \text{if } h \in 0010xx10 \\
a_1(h), & \text{if } h \in xx10xxx \\
-0010xxx1-0010xx11 & \text{otherwise}
\end{cases}$$

It means that the traffic from $l_s$ within xx10xxxx meets the black hole in this network. When a rule is adding to the network, the black hole is detected by the computation of $l_{d,3R^cA_{l,s}}$. If the specific black hole position is desired, the more “leakage” rules should be constructed for each switch or each link.

### B. INTER-DOMAIN FORWARDING LOOP

Loops can raise in both intra-domain and inter-domain routing. In Figure 5, there are two connected domains. The reachability from $l_1$ to $l_3$ and $l_4$ to $l_2$ of domain B and the reachability from $l_3$ to $l_1$ and $l_2$ to $l_4$ of domain A is computed by using the described intra-domain method. They are written as $f_{B,31}$, $f_{B,24}$, $f_{A,13}$, and $f_{A,42}$. The links are treated as the vertexes, and the reachability between these links is treated as the edge. We establish the adjacency matrix of these links and the result is

$$
\begin{pmatrix}
0 & 0 & f_{A,13} & 0 \\
0 & 0 & 0 & f_{B,24} \\
f_{B,31} & 0 & 0 & 0 \\
0 & f_{A,42} & 0 & 0
\end{pmatrix}
$$

The square of this matrix is

$$
\begin{pmatrix}
f_{A,13} f_{B,31} & 0 & 0 & 0 \\
0 & f_{B,24} f_{A,42} & 0 & 0 \\
0 & 0 & f_{B,31} f_{A,13} & 0 \\
0 & 0 & 0 & f_{A,42} f_{B,24}
\end{pmatrix}
$$

Therefore if the main diagonal elements are not 0, there is a loop from $l_1$ to $l_1$.

We extend this situation to the multi-domains. As shown in Figure 6, there are more than two connected domains. A super controller of the topology of domains has been set to maintain the entire network. All the domains give the information of the reachability function from the in-domain links to the out-domain links to the super controller. The intra-domain information is only visible for the intra-domain controller. We establish the adjacency matrix $L$ of the inter-domain links in the super controller. The $L$ is computed by using the described intra-domain method.

Therefore, the definition of the inter-domain loop is the same as Definition 1 and Definition 2. The max cross-domain hop has been set to $N$. The $i$-th element $l_{m,ii}$ of main diagonal of $L^n$ represents the reachability from $l_i$ to itself. $L = E + \sum_{n=1}^{N} L^n$. The inter-domain loops are detected by Theorem 3.

When the rules are updating in a domain, if this updating changes the state of the inter-domain links, the domain sends the changing information to the super controller. The changing of the $L$ is got by this message and written as $\Delta L$. The connected matrix of inter-domain links $L_{c} = L_c + L_{\Delta,c}$. Computing is the same as Algorithm 3. The dynamical detection of inter-domain loops is done by computing the $L_{\Delta,c}$ and using Theorem 3 to check the main diagonal of $L_{\Delta,c}$.

If there is not a super controller, all the computation will be done in the controllers of domains to get a one-time loop check from an inter-domain link. For example in Figure 6, if we want to know if there are loops from the link $l_1$ to itself under the max cross-domain hop $N$, the domain A sends the index of $l_1$ and the forwarding function with the domain and the range on $l_1$ to the domain B. For the controller of each domain except domain A, they do the computation in two steps. First, if the hop counter is not exhausted, the domain $M$ compute the connection between the received function $f_j$ on the received link $l_r$ and the out-domain links by $l_{r\rightarrow o} = MLf_r$, where $f_r = [f_0 0 0 \ldots]^T$. Then, if the $l_{r\rightarrow o}$ is not 0, the controller sends the $l_{r\rightarrow o}$ and the index of $l_r$ to the next domain on $l_r$ and updates the hop counter. Finally, the domain A compute $l_{1\rightarrow 1} = l_{1,s}LAl_l$ where $l_{1,s} = [1 0 0 \ldots]$. The result
\( l_1 \rightarrow l_1 \) is checked by the Theorem 3 to identify a loop. However, it will take a long time to get the result.

**VI. IMPLEMENTATION AND EVALUATION**

In this experiment, we test all the functionalities and compare our model with NetPlumber [2] and APT [3] in the same environment. The performance of inter-domain loop detection has been compared with the method (using HSA) in [13].

**A. IMPLEMENTATION**

The NetV locates at the bottom of the controller. It extracts the rules which are generated by the applications and ready for installation. Then it merges those rules into a single policy. The defined invariants are checked during the updating processing. If the new adding rule violates the invariants, the error will be reported.

The prototype of NetV has been written in C programming language. We use BDDs to present the domain and the range of a rewrite function. The BDD environment is using the BuDDy library [21]. The preprocessing that includes getting the initial connection of rules, removing the conflicts, and merging the rules with the same action has been done by using the MTBDD method in [20]. We rebuild some parts of the BuDDy library to make the library adjust to the MTBDD structure and the described algorithm on BDD headers. We use the compressed sparse row or col (CSR/CSC) data structure to store the matrices for the computational performance since the valid elements of the matrices are less than 2%.

We evaluate the performance and functionality of our prototype on Internet 2 network dataset and Stanford's backbone network dataset [1] from real networks. The Stanford dataset has 14 operational zones Cisco routers and 2 backbone Cisco routers. They are connected via 10 Ethernet switches. There are more than 757,000 forwarding rules, 100+ VLANs, and 1,500 ACL rules. The Internet 2 network has 9 routers with more than 120,000 IPv4 forwarding rules and 300+ MPLS rules. The basic OpenFlow rule sets are generated from these two rule sets by the parsing part of NetPlumber [2]. The number of OpenFlow rules is shown in Table 1.

**B. PREPROCESSING OF UPDATING**

First, we respectively use Algorithm 1 and the existential quantification operation by using restrictions in APT [3] for the whole process and record the processing time for each call. The result is shown in Table 2 and Table 3. On average, using Algorithm 1 to set the wildcards on BDDs is 8-10 times faster than using restriction operation on the two network rule sets. Since the “or” operation for lower level part of the BDD is faster than the “or” operation for the entire BDDs \( P|_x=\text{true} \lor P|_x=\text{false} \). Then we record the per-call processing time for the rewrite function. The result in Table 4 shows that the average of the processing time for the rewrite function is 2-3 times than the set-wildcard function. The rewrite function contains a set-wildcard function and a BDD “and” operation. Thus, the set-wildcard function is almost the same order of magnitude as the BDD “and” operation.

Then we evaluate the performance of the preprocessing of the dynamical updating for getting the difference of the adjacency matrix by separately removing and adding back each rule in the rule sets. The updating process for NetPlumber can be divided into preprocessing part that crates pipes and deals with the cover relations when adding a rule and the propagating part for the previous hop rules. So, we use the preprocessing of NetPlumber to be the baseline. The time cost of preprocessing of the NetPlumber mainly depends on the amount of the flow table and the amount of the previous hop and next-hop flow tables. Thus for Fig. 7 and Fig. 8, per update preprocessing run time of the NetPlumber on Stanford network rules is even shorter than the run time on the Stanford network forwarding rules. The per update preprocessing run time of NetV depends on the number of the amount of the merged rules in previous-hop and next-hop tables and

| TABLE 1. The number of rules in the 3 rule sets. |
|-----------------------------------------------|
|                   | Forwarding Only | Stanford | Internet 2 |
| OpenFlow Rules    | 3844            | 8792     | 77841      |
| Merged Rules      | 231             | 1752     | 456        |

| TABLE 2. Per-call Run Time for setting wildcard on BDDs on stanford network. |
|-----------------------------------------------|
|                        | Average | Median | Maximum |
| Algorithm 1(us)        | 11.2    | 5      | 5133     |
| Using Restriction(us)  | 84.8    | 41     | 15424    |

| TABLE 3. Per-call run time for setting wildcard on BDDs on internet 2 network. |
|-----------------------------------------------|
|                        | Average | Median | Maximum |
| Algorithm 1(us)        | 7       | 3      | 11347    |
| Using Restriction(us)  | 97.7    | 31     | 27210    |

| TABLE 4. Per-call run time for rewriting on BDDs. |
|-----------------------------------------------|
|                        | Average | Median | Maximum |
| Stanford Network(us)   | 38.4    | 18     | 18158    |
| Internet 2 Network(us) | 14.8    | 9      | 13625    |

**FIGURE 7. The CDF of the per update run time of preprocessing on Stanford network forwarding rules.**
the efficiency of BDD operation. Thus for the rule sets, the average per update preprocessing run time of the NetV on Stanford network forwarding rules is the shortest. As shown in Fig. 7 Fig. 8 and Fig. 9, the time cost of the per update preprocessing of NetV is no more than the preprocessing of the NetPlumber. Therefore, the preprocessing of NetV only increases the appropriate cost for the entire process, while it computes the precise connection changes of the merged rules when updating rules.

C. EVALUATION WITH NETPLUMBER

First, we generate the merged rules from the OpenFlow rule sets to evaluate the redundancy of the rules with the wildcard expressions. As shown in Table 1, we use the parsed data sets in NetPlumber [2]. The first one is the forwarding rule set extracted from the Stanford backbone network. The rules in the same table are merged based on their actions, which means the plus operation of the rule functions in Section III. For the forwarding rule set, the number of the merged rules is almost 6% of the entire OpenFlow rules. It is even 0.5% for the Internet 2 rule set.

The NetV took 2.5s to generate the initial adjacency matrix $R$ for Stanford’s backbone network. It took 8.2s to get the result of $R_c$. We run NetPlumber in the same environment.

| TABLE 5. Per update run time on stanford network forwarding rules only. |
|---------------------------------------------------------------|
| NETV(ms)     | Average | Median | Maximum |
|---------------|---------|--------|---------|
| NetV without pre | 0.124   | 0.109  | 8.4     |
| NetPlumber    | 7.9     | 0.917  | 304     |

It took 0.8s to create the initial plumbing graph and 131s to generate checking results with 16 source nodes connected to each router. For NetV with the same 16 ports, it took 702ms by iteratively multiplying the adjacency matrix to find all results, and it took 108 ms by directly using the $R_c$. NetV took 23.7s to generate the initial adjacency matrix for Internet 2 network. The $R_c$ was computed in 1.1s. It took 88s to create the initial plumbing graph and 653s to generate checking results with 12 source nodes connected to the routers by using NetPlumber. By using NetV, it took 184ms by iteratively multiplying $R$ and 27ms by multiplying $R_c$. The result shows that the redundancy of the rules does increase the time cost of the reachability verification. NetV performs better even the reachability of the entire rules $R_c$ is computed.

We connect 16 source nodes for Stanford’s backbone network and 12 source nodes for i2 to each router for NetPlumber. These in-ports are also used to generate the $L_s$ for computing the final result $L_s \rightarrow r$. We test the per update run time by separately removing and adding back each rule in the rule sets. In Figure 10, the rule set is the forwarding rules in Stanford’s backbone network. The solid represents the main process without the preprocessing. It is the propagation part for NetPlumber. The dashed represents the entire process of the rule updating. From the figure, NetV performs better than NetPlumber. The solid blue line starts with nearly 40% at the ordinate. It means that nearly 40% rules’ updating in this experiment does not affect the reachability, which is also described by AP verifier [18]. A part of the solid red line is higher than the blue one since some rules affect the little number of flows, or they locate at the end of the flows from the source nodes. NetV gets the reachability of every two rules. Thus it can get more information, including the potential loops. From the result in Table 5, NetV is 70 times faster than NetPlumber on average. For the
main process, NetV is 48 times faster than NetPlumber on average.

The following two experiments on Stanford’s backbone network and the Internet 2 network also indicate that the redundancy of rules would reduce the updating performance. The time cost of the preprocessing is negligible compared with the main process in the two experiments. In Figure 11, the length of the header is raised from 32 bits in forwarding rules to 128 bits. From the result in Table 6, the NetV is 13 times faster than the NetPlumber on average. For Internet 2 network, since the redundancy is vast, it is clear that the per update run of the NetV is faster than the NetPlumber in Figure 12. Table 7 shows that it is 75 times faster than the NetPlumber on average.

For testing the link update, we remove and add back each connected link for Stanford’s backbone network and Internet 2 network. As shown in Table 8 and Table 9, the link per update run of NetV is hundreds of times faster than NetPlumber. For NetPlumber, the number of the rules associated with each link is always more than a hundred in Stanford’s backbone network, and it is even more on Internet 2 network. Adding a new link equals adding the pipes between all the rules associated with the link and then propagating all the flows in front of these pipes. Therefore, it takes a long time to process the link update. For NetV, the numbers of the merged rules associated with each link are quite small. Thus, the link update of NetV is faster by incrementally updating the $R_c$, which takes the connection of the merged rules as the difference $\Delta R$ of the adjacency matrix. The loop detection is done during the updating process by checking the element of the main diagonal. Previous experiments also check the reachability from the 16 in-ports in Stanford’s backbone network and 12 in-ports on Internet 2 network to the out-ports. Since the $l_d R_{\Delta,c,l_s}$ is always small and can be ignored compared with updating $R_c$, the time cost of the incremental verification depends on the run speed of updating $R_c$.

Above all, merging the rules with the same behavior in the same table does reduce the redundancy and speed up the rule updating process. The composition of the transform function in header space makes that the reachability information can be sustained to reason the reachability of the specific packets or the set of packets. For example, if the reachability from in-port 1 to out-port 2 has already calculated and we want to know the reachability of a specified set of packets from in-port 1 to out-port 2, the symbol headers will be sent to the source node to be propagated again by using NetPlumber. By using NetV, we use the transform function from in-port 1 to out-port 2 to do one-time computation on the symbol headers. The matrix operation is an efficient way to operate and locate the desired functions and separate the location space and the header space in networks. However, since the composition only focuses on the substitution in header space, the paths are not recorded. If the path record is required, the transform function will be defined as the function of two exclusive spaces by adding actions to push the current path associated

### TABLE 6. Per update run time on stanford network.

|          | Average | Median | Maximum |
|----------|---------|--------|---------|
| NetV(ms) | 3.355   | 0.720  | 113.308 |
| NetPlumber(ms) | 30.914 | 1.651  | 8793    |

### TABLE 7. Per update run time on internet 2 network.

|          | Average | Median | Maximum |
|----------|---------|--------|---------|
| NetV(ms) | 1.305   | 1.351  | 54.113  |
| NetPlumber(ms) | 98.263 | 93.173 | 64295   |

### TABLE 8. Link update run time on stanford network.

|          | Average | Median | Maximum |
|----------|---------|--------|---------|
| NetV(ms) | 19.54   | 12.8   | 94.7    |
| NetPlumber(ms) | 4569 | 1103   | 51561   |

### TABLE 9. Link update run time on internet 2 network.

|          | Average | Median | Maximum |
|----------|---------|--------|---------|
| NetV(ms) | 4.29    | 1.89   | 27.76   |
| NetPlumber(ms) | 11431 | 30     | 101170  |
with the function to the path record. The composition of two functions contains the operation of jointing two path records. It will add the extra process time and sometimes increase the number of the functions if there are multiple paths between two points.

### D. EVALUATION WITH APT

The key idea of APT is to precompute the atomic predicates and give each of them an integer identifier. The propagating time of the symbol header is strongly reduced by using these identifiers instead. Therefore, the computation of the APs takes the significant time of the entire verification process, especially for the incremental verification. We do not get the source code of the APT. AP Verifier [18] uses several optimization techniques to reduce verification time. Since the computation of the APs is the limit of the performance, we only write a program with C programming language to perform the APs generation based on the algorithms in their papers [3], [18]. We also have done the optimization to improve the performance of this program. APT converts the forwarding tables to the predicates of out-ports. In SDN, the rule provides the flexibility to perform the action for the specific in-port. Thus, the forwarding rules cannot be strictly translated to the predicates of the out-ports. Since the merged rules also remove the redundancy and represent the same behaviors, we use the domains of the merged rule functions to calculate the APs.

Table 10 shows the number of the basic APs generated by the fillers of the merged rules without iteratively computing the transformed predicates. If the basic data set is in the OpenFlow format, the number of the APs is far more significant than the traditional model in their works. If we iteratively compute the transformed predicates and use the transformed predicates to compute the extended APs, the numbers of entire APs are 150,000+ for Standford network and 38900+ for Internet 2 network. The performance of the updating strongly depends on the numbers of the existed APs. For each update of a predicate $P$, the algorithm first computes the APs $\{P, \neg P\}$ and then computes the updated APs by using the “AND” operation for every pair between $\{P, \neg P\}$ and the maintained APs. Therefore, the time complexity of the one-time update is $O(N \times K)$, where $N$ is the number of the existed APs, and $K$ is the number of the entire transformed predicates by the iterative algorithm.

Figure 13 presents the average run time for different numbers of the existed APs for each update of a predicate $P$. The updating run time rises with the growth of the number of the existed APs. The average per update time of a predicate is 504ms after the number rises over the 20000 for the Internet 2 network. It is 1.3s after the number goes over the 50000 for the Stanford network. If we take the iterative algorithm for the transformer into consideration, in the last 50% updating of the merged rules, half of the updates will be more than 2s for the Internet 2 network since the number of iterations is more than two and the number of transformed predicates is more than five. The number of transformed predicates of some updates is 700+, which leads to more than 200s to complete an update. For the Stanford network, some updates get 1000-3000 transformed predicates in the last 50% updating, which leads to more than 800s to update some merged rules.

We then test the per rule update on the 273 APs generated by the merged rules for the forwarding rules of the Stanford network. As shown in Figure 14, NetV performs better than the AP verifier. The mean per rule update time is 3ms for APT, which is bigger than the 0.124ms for NetV. Thus, the APT is not suitable for the rule set of this experiment with OpenFlow format.

Furthermore, APT only design an algorithm to updates the set of atomic predicates and do not keep the number of atomic predicates to be smallest for incremental updating, which causes the instability of the number of atomic predicates when updating rules. For example, we assume the existed two predicates are 1x1x and xxx1 and the APs
are \{1x11, 1x10, 0xx1 + 1x01, 0xx0 + 1x00\}. If adding a rule changes the predicate 1x1x to 1xxx, the updated APs are \{1x11, 1x10, 1x01, 1x00, 0xx1, 0xx0\} which contain 6 APs. If we use 1xxx at first as one of the two predicates, the computed APs are \{1xx1, 1xx0, 0xx1, 0xx0\}, which only contains 4 APs. We also do a test to compute the APs for forwarding rules of the Stanford network by updating each rule other than the merged rules. The result of the number of the final APs is 1515, which is much bigger than the 273 by using the merged rules.

APT performs very well in traditional networks. However, it has gaps with the rules in SDN for incremental verification unless the number of the generated APs is much lower than the rules. Overall, these results indicate that NetV performs better than APT in SDN for incremental verification.

E. EVALUATION OF TWO DOMAINS
In [13], they use HSA to check the inter-domain loop. For domains $A$ and $B$, if the set of the headers on the connected link is $H$, the checking is to verify $F_A(F_B(H)) \neq \emptyset$, which is computed by using HSA, where $F_A$ and $F_B$ are the transfer function of the domains. Therefore, the incremental verification can be done by using NetPlumber to verify $F_A(F_B(\Delta H)) \neq \emptyset$, where the $\Delta H$ represents the change part on the connected links by updating. We split the Stanford network into two parts. Each one has eight routers, and they are connected with two links. We select the filters of the rules in the switch, which is connected to the connecting link between the two domains to be the $\Delta H$. Then, we compute the $F_A(F_B(\Delta H))$ for each filter by using NetPlumber. For NetV, the change $\Delta L$ is generated from each filter, and then it is used to update the connectivity matrix for the two domains.

Table 11 shows the experiment on the forwarding rules. NetV took 76us to perform the verification for each change on average with the forwarding rules, while NetPlumber took 1.56ms. If we only use the forwarding rules, the transform function between every two inter-domain links only has one expression with a domain and a range by using NetV, since they do not change the headers. It can be represented as

$$f(h) = \begin{cases} a_I(h), & \text{if } h \in X \\ \emptyset, & \text{otherwise} \end{cases}$$

and the adjacency matrix is represented as

$$L = \begin{bmatrix} 0 & f_A \\ f_B & 0 \end{bmatrix}$$

If $f_A$ changes with $f_{\Delta A}$ for a update and the max number of the domain hops is two, the final connectivity matrix is

$$L_c = \begin{bmatrix} f_{AB} & f_A \\ f_B & f_{fA} \end{bmatrix}$$

and the updated connectivity matrix is

$$L^{u}_c = \begin{bmatrix} f_{AB} & f_A \\ f_B & f_{fA} \end{bmatrix} + \begin{bmatrix} f_{\Delta A} & 0 \\ 0 & f_{f\Delta A} \end{bmatrix}$$

Since $f_{AB} = f_{BA}$ for forwarding rules, it only needs to compute one composition with one operation on BDD to get the result. For computing the $F_A(F_B(\Delta H))$, NetPlumber needs to trace the symbolic headers $H$ in both domains. Therefore, NetV performs better than NetPlumber for inter-domain verification.

Table 12 and Table 13 also show a better performance of NetV. The maintained reachability of the single domain is the $L_{X \rightarrow X}$. For each element in $L_{X \rightarrow X}$, it is the transform function with the simplest form, which eliminates the detail information of the paths inside the domain and reduces the number of computations. The inter-domain reachability is achieved by computing the composition of these transform functions rather than retracing the symbolic headers. Overall, these results and the analysis indicate that the dynamic performance and the scalability of NetV are better than NetPlumber for the verification with multiple layer controllers (multiple domains) in SDN.

VII. CONCLUSION
The main goal of this paper is to design a scheme for dynamic data plane verification in SDN. In this paper, we define the function in header space with the BDD domain and the operations which make the substitution behaviors of the rules can be merged or connected as an expression of the transform function. We also propose the updating method based on graph and matrix operation to maintain the reachability information of the entire network. The common network invariants are translated to the requirements on the matrix model and can be incrementally verified when updating rules. Finally, we attempt to extend the model to the multi-domains network for incrementally verifying inter-domain invariants. These experiments confirmed that the prototype NetV performs better than NetPlumber both on the static verification and incremental verification. NetV also performs better than APT on the rule sets with the OpenFlow format, which generates a large number of APs. At last, it is more flexible than the
previous tools to scale to the large network and the multi-domains network.

The results of this work indicate that treating the header change behavior as functions that can be calculated by the defined operations is reasonable. This research provides a framework for the exploration of the rule relations in SDN. A limitation of this study is that some network behaviors (like the TTL minus operation) have not been defined into the function operations. Another limitation is that we only test the computation part for the multi-domains network without the cost of the communication between controllers. What is now needed is to design the protocol and the compressed structure for transporting the BDDs between controllers. Further research might explore the features of the network reachability stored in the matrix by using the decomposition operation of the matrix.

APPENDIXES
APPENDIX A
SYMBOLS AND PARAMETERS
Table 14 shows the meaning of the symbols used in this paper. Table 15 shows the rewrite parameter of the transform function used in this paper. For example, the rewrite parameter is xxxxxxx00 for \( a_1 \). Thus the rewrite mask is 11111100, and the rewrite value is 00000000. If the symbolic header is 110xxxx1, the rewritten symbolic header is 110xxx00.

APPENDIX B
ENCAPSULATION AND DE-ENCAPSULATION
We define a new function in header space and write it as \( a_r \) that means to replace a given field by another field of the packet. The parameters of the \( a_r \) are the set of relations for each bit, and two masks \( mask_f \) and \( mask_t \) can be acquired from the relation set. \( mask_f \) indicates the position of the value, and \( mask_t \) indicates the replaced position. The encapsulation operation is defined as \( a_e = a_r w a_r \). It first moves the values to the blank bits and then fills the original bits with the new given values. The de-encapsulation operation only moves the values back to the original bits. Thus it is represented as the \( a_r \). For example, if we want to perform the encapsulation (like IPv4, and for MPLS, it is more like the rewrite action) for the first four bits on symbolic header 0010xx00 with 1111, the header space will be extended to 12 bits, and the header becomes 0010xx00xxxx. It first moves the 0010 to the extended bit, and the header will be the xxxxxxx0000. Then the rewrite function with the parameter 1111xxxxxxx takes action on the header, and the header will be the 1111xx000010. For de-encapsulation, we only need to recover the stored bits and let the header be 0010xx00xxxx by using the replace function.

For the composition of the transform functions, we can define that a transform function \( a_t \) which always does the replace first has three parameters \( (rl, m_r, v_r) \) that include the replace relations which means to replace the desired bits by the values of the given bits, the mask and the value of the rewrite. Thus, to compute the \( a_{t,c} = a_{t,1} a_{t,2} \) is to compute the \( (r_l, m_v, v_v) \) from \( (r_l, m_1, v_1) \) and \( (r_l, m_2, v_2) \). For each bit of the vector \( h \)

\[
rl(h_j) = h_j, \quad \text{if} \quad j \to i
\]

associated with the condition that \( (\neg mask_f)(\neg mask_t) \neq 0 \), which means the replacing bit is not replaced by another bit in a replace function. The result is \((rl_1 \circ rl_2 . m_2 \circ rl_2 (m_1), (m_2 \& \neg rl_2 (v_1))) v_2\), where \( \circ \) represents the composition of two replace functions, “\( \& \)” is the bitwise AND and “\( \circ \)” is the bitwise OR.

\[
rl_1 \circ rl_2 (h_j) = h_k, \quad \text{if} \quad (j \to i, k \to j)
\]  

(14)

Algorithm 4 is proposed to perform the replace on BDD expression to compute the domain and the range of the transform function. If we want to perform the replace on a BDD domain, we first extract the values of the desired positions and then relabel them as the new positions by the given replace relations. Then the values in other positions should be filled by the BDD AND operation. The masks can be gotten from the relation parameter \( rl \). The completed transform function

| Symbol | Representation |
|--------|----------------|
| \( h \) | The header of a packet. |
| \( h_i \) | The \( i \)-th bit in header \( h \). |
| \( l_i \) | In Section III, it means the \( i \)-th link. In the later Section, it means the constructed vector. |
| \( r_i \) | A SDN rule or a merging rule. |
| \( a_f \) | Forwarding action. |
| \( a_m \) | Modifying action for changing the headers. |
| \( X \) | The domain. |
| \( Y \) | The range. |
| \( f \) and \( g \) | The header space functions. |
| \( R \) | The basic adjacency matrix of rules. |
| \( R^* \) | The updated basic adjacency matrix of rules. |
| \( \tau_{ij} \) | The element of \( R \). |
| \( R_e \) | The basic connection matrix of rules. |
| \( L_s \) and \( L_d \) | The constructed matrix of the source points and the destination points. |
| \( L_{s+d} \) | The connection matrix from the source points to the destination points. |
| \( L_M \) | The outside connection matrix of domain \( M \). |
| \( L_C \) | The inter-domain connection matrix. |

| Symbol | Rewrite parameter |
|--------|-------------------|
| \( a_1 \) | xxxxxxxx |
| \( a_2 \) | xxxxxxx0 |
| \( a_3 \) | xxx0000 |
| \( a_4 \) | xx10000 |
| \( a_5 \) | 00xx0000 |

TABLE 14. Symbol used in this paper.

TABLE 15. Parameter of example action used in this paper.
first performs the replace operation on the domain and then performs the rewrite operation.

**APPENDIX C**

**THE LOOPS**

In Section V-A2, we explain the Generic Loop, which is defined in Definition 1. In Definition 2, the simplest form equals if \( i \neq j \Rightarrow a_{n,i,j} \neq a_{m,i,j} \) and \( i, j \in [1, M] \) which means that there are no two same transform function expressions in \( r_{n,i,j} \). The Infinite Loops are the subset of the Generic Loops in a network. The difference between the Generic Loop and the Infinite Loop is that if the packet is transformed by the same transform behavior again. Since \( m&v = 0 \) for a rewrite function, \( a_{m} = (m&v) = (m, v) = a_{m} \). Thus, if there exists an \( a_{n,i,j} \) satisfy the Definition 2, \( a_{n,i,j} = a_{n,i,j}(h) = X_{n,i,j}(h) \), and the packet \( h \) is infinitely transformed by this transform function.

If we use the transform function definition in Appendix B for Encapsulation and De-Encapsulation, the Definition 2 should be modified to the statement that the \( n \)-hop infinite loop of \( r_{i} \) exists as defined as \( \exists h \in X_{n,i,j} \) \( k \in (1, 2, \ldots, M) \) let \( a_{n,i,j}(h) = X_{n,i,j} \) and \( a_{n,i,j}(h) = X_{n,i,j} \). From (14), \( r_{i} \circ r(h_{i}) = h_{j} \), if \( i \Rightarrow j \). Since there is a condition that \( ((\text{mask}_i)(\text{mask}_j)) \neq 0 \) for a replace function. Thus, \( j \Rightarrow i \neq i \Rightarrow j \Rightarrow j \) in a replace function. Therefore, \( r_{i} \circ r(h_{i}) = h_{j} \), if \( i \Rightarrow j \in r(h_{i}) \).

\[ a_{i}^{2} = (r_{i} \circ r_{l}(m))(m \& r_{l}(v)) \]
\[ a_{i}^{3} = (r_{i} \circ r_{l}, (m \& r_{l})(m))(m \& r_{l}(v)) \]
\[ = (r_{i} \circ r_{l}(m))(m \& r_{l}(v)) \]
\[ = (r_{i} \circ r_{l}(m), (m \& r_{l}(v)) \]
\[ = a_{i}^{2} \]

Thus, if \( \exists h \in X_{n,i,j} \) \( k \in (1, 2, \ldots, M) \) let \( a_{n,i,j}(h) = X_{n,i,j} \) and \( a_{n,i,j}(h) = X_{n,i,j} \), the packet \( h \) is infinitely transformed by this transform function.

**Proof (Theorem 3):** Because of (7), \( r_{c,i} \neq 1 + \sum_{n=1}^{N} r_{n,i} \). In \( R_{i} \) of a network, every element is positive, and only in the process of removing some part of the existed rules, there exists the negative (minus operation) for the moment. Then \( r_{c,i} \neq 1 \neq 0 \Rightarrow \exists h \in [1, N] \). \( r_{n,i} \neq 0 \). Therefore \( \exists h \in X_{n,i,j}, r_{n,i}(h) \neq \emptyset \), and by Definition 1, there exists at least a Generic Loop.

Also, if \( r_{c,i} \) is the simplest form, \( \exists h \in X_{c,i,j} \) \( k \in (1, 2, \ldots, M) \) let \( a_{c,i,j}(h) = X_{c,i,j} \) equals that \( \exists h \in X_{c,i,j} \) \( k \in (1, 2, \ldots, M) \) let \( a_{c,i,j}(h) = X_{c,i,j} \). Since \( a_{c,i,j}(h) = X_{c,i,j} \), there exist at least a \( n \)-hop Infinite Loop of \( r_{i} \) by Definition 2.

**REFERENCES**

[1] P. Kazemian, G. Varghese, and N. McKeown, “Header space analysis: Static checking for networks,” in Proc. 9th USENIX Conf. Netw. Syst. Des. Implement. (NSDI). Berkeley, CA, USA: USENIX Association, 2012, p. 9.

[2] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, “Real-time network policy checking using header space analysis,” in Proc. 10th USENIX Conf. Netw. Syst. Des. Implement. (NSDI). Berkeley, CA, USA: USENIX Association, 2013, pp. 99–112.

[3] H. Yang and S. S. Lam, “Scalable verification of networks with packet transformers using atomic predicates,” IEEE/ACM Trans. Netw., vol. 25, no. 5, pp. 2900–2915, Oct. 2017.

[4] Y. Li, X. Yin, Z. Wang, J. Yao, X. Shi, J. Wu, H. Zhang, and Q. Wang, “A survey on network verification and testing with formal methods: Approaches and challenges,” IEEE Commun. Surveys Tuts., vol. 21, no. 1, pp. 940–969, 1st Quart., 2019.

[5] J. Qudir and O. Hasan, “Applying formal methods to networking: Theory, techniques, and applications,” IEEE Commun. Surveys Tuts., vol. 17, no. 1, pp. 256–291, 1st Quart., 2015.

[6] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjaltmysson, and J. Rexford, “On static reachability analysis of IP networks,” in Proc. IEEE 24th Ann. Joint Conf. IEEE Comput. Commun. Soc., vol. 3, Mar. 2005, pp. 2170–2183.

[7] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, “Debugging the data plane with anteater,” ACM SIGCOMM Comput. Commun. Rev., vol. 41, no. 4, p. 290, Oct. 2011.

[8] S. Zhang and S. Malik, “SAT based verification of network data planes,” in Automated Technology for Verification and Analysis, D. Van Hung and M. Ogawa, Eds. Cham, Switzerland: Springer, 2013, pp. 496–505.

[9] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. ElBadawi, “Network configuration in a box: Towards end-to-end verification of network reachability and security,” in Proc. 17th IEEE Int. Conf. Netw. Protocols, Oct. 2009, pp. 123–132.

[10] E. Al-Shaer and S. Al-Haj, “FlowChecker: Configuration analysis and verification of federated openflow infrastructures,” in Proc. 3rd ACM Workshop Assurable Usable Secure Configuration (SafeConfig), New York, NY, USA: Association for Computing Machinery, 2010, pp. 37–44.

[11] S. Al-Haj and W. J. Tolone, “FlowTable pipeline misconfigurations in software defined networks,” in Proc. IEEE Int. Conf. Comput. Commun. Workshops (INFOCOM WKSHPS), May 2017, pp. 247–252.

[12] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, “Model checking invariant security properties in OpenFlow,” in Proc. IEEE Int. Conf. Commun. (ICC), Jun. 2013, pp. 1974–1979.

[13] Y. Zhang, B. Zhu, Y. Fang, S. Guo, A. Zhang, and S. Zhong, “Secure inter-domain forwarding loop test in software defined networks,” IEEE Trans. Depend. Secur. Comput., vol. 17, no. 1, pp. 162–178, Jan. 2020.

[14] 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. 39, no. 2, pp. 69–74, Mar. 2008.

[15] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, “VeriFlow: Verifying network-wide invariants in real time,” ACM SIGCOMM Comput. Commun. Rev., vol. 42, no. 4, pp. 467–472, Sep. 2012.

[16] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, “VeriFlow: Verifying network-wide invariants in real time,” presented at the 10th USENIX Symp. Netw. Syst. Des. Implement. (NSDI). Lombard, IL, USA: USENIX, 2013, pp. 15–27.

[17] Q. Li, Y. Chen, P. P. C. Lee, M. Xu, and K. Ren, “Security policy violations in SDN data plane,” IEEE/ACM Trans. Netw., vol. 26, no. 4, pp. 1715–1727, Aug. 2018.

[18] H. Yang and S. S. Lam, “Real-time verification of network properties using atomic predicates,” IEEE/ACM Trans. Netw., vol. 24, no. 2, pp. 887–900, Apr. 2016.

[19] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation,” IEEE Trans. Comput., vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[20] Y. Fang and Y. Lu, “Checking intra-switch conflicts of rules during preprocessing of network verification in SDN,” IEEE Commun. Lett., vol. 23, no. 9, pp. 1547–1550, Sep. 2019.

[21] J. Lind-Nielsen. (2003). BuDDe—A Binary Decision Diagram Package. [Online]. Available: http://vlsicad.eecs.umich.edu/BK/Slots/cache/ www.itu.dk/research/buddy/
YANG FANG was born in Yichang, Hubei, China, in 1988. He received the B.S. degree from the School of Electronic and Information Engineering, Tianjin Polytechnic University, China, in 2010. He is currently pursuing the Ph.D. degree with the South China University of Technology, China. His research concerns software-defined networks, network verification, network testing, and formal methods.

YIQIN LU received the B.S., M.S., and Ph.D. degrees from the South China University of Technology University in 1990, 1993, and 1996, respectively. From 1994 to 1996, he was a Visiting Student with the City University of Hong Kong. Since 1996, he has been with the South China University of Technology, where he is currently a Professor and the Director of the Network and Data Center. His research interests are in various applied topics including software-defined networks, network function virtualization, the Internet of things, and network security.