Misconfiguration Checking for SDN: Data Structure, Theory and Algorithms

Heng Pan*†, Zhenyu Li*†, Penghao Zhang*, Kave Salamatian‡. Gaogang Xie§
*ICT, CAS, China, †Purple Mountain Laboratories, Nanjing, China, ‡ University of Savioe, France, §CNIC, CAS, China
{panheng, zyli, zhangpenghao}@ict.ac.cn, kave.salamatian@univ-savoie.fr, xie@cnic.ac.cn

***Abstract*—Software-Defined Networking (SDN) facilitates network innovations with programmability. However, programming the network is error-prone no matter using low-level APIs or high-level programming languages. That said, SDN policies deployed in networks may contain misconfigurations. Prior studies focus on either traditional access control policies or network-wide states, and thus are unable to effectively detect potential misconfigurations in SDN policies with bitmask patterns and complex action behaviors.**

To address this gap, this paper first presents a new data structure, minimal interval set, to represent the match patterns of rulesets. This representation serves the basis for composition algebra construction and fast misconfiguration checking. We then propose the principles and algorithms for fast and accurate configuration verification. We finally implement a misconfiguration checking tool in Covisor with optimisations to further reduce the overhead. Experiments with synthetic and random rulesets show its fitness for purpose.

I. Introduction

Software-defined Networking (SDN) decouples network control logic from the data plane via an *open* interface (e.g., OpenFlow [26]). It simplifies network management and enables a network to customize its behaviors [25]. This separation benefits from a good abstraction of data plane - a series of "Match-Action" flow tables so that network administrators can put more efforts to build network applications. The vision of SDN is to construct a SDN "APP Store" for network management services [23], [29]. And network administrators can download one or more third-party applications suited to their needs and deploy them into the network. Furthermore, SDN allows applications written by different languages and platforms to cooperatively manage the network [17], [11]. Overall, SDN makes it easy to program the network and facilitates network innovation.

Early SDN provides a few low-level APIs built into controllers [14], [1], [2] for network administrators to program the network. They have to write SDN applications by directly manipulating the flow tables of underlying devices. This is tedious and error-prone. To reduce the complexity of programming, recent interests in SDN move to high-level programming languages and frameworks [12], [35], [27]. Usually, they provide a programming model to write applications and design a runtime system for compiling network applications into switch-level rules. However, this compiler system is unable to check all policy misconfigurations (e.g. *function conflict* and *deployment inefficiency*) that are introduced by network administrators [34], [24]. Worse, the emerging compositional SDN allows diverse policies to cooperatively manage the *same* network traffic [11], [17], [13]. Composing rulesets from different components in SDN will inevitably increase the risk of errors.

978-1-7281-6992-7/20/$31.00

Although a large amount of research effort has been devoted to detecting anomalies in access control polices (e.g. firewall) [6], [36], [15], these solutions fail to detect potential misconfiguration in SDN policies. This is because they focus on the rulesets with prefix-based patterns and simple actions (e.g., $drop$ and $forward$). However, SDN policy rules support arbitrary bitmask patterns and complex behaviors [3]. Recently, some verification tools [21], [20], [33] have been proposed for the purpose of detecting whether one policy will violate their predefined network-wide invariants (e.g., black hole). They are unable to detect the misconfigurations that do not affect the states of network. Furthermore, compositional SDN increases the possibility of misconfigurations.

This paper aims at addressing this gap by developing a misconfiguration tool for accurate and fast checking of both individual policy rulesets and the compositional rulesets. This is indeed challenging. SDN policies reflect the intent of network administrators, but it is difficult to surmise the intent so as to identify the misconfigurations. To illustrate this, consider two rules r_1 and r_2: r_1 drops some packets that are interested by r_2. We cannot hastily conclude that this behavior is a misconfiguration (a.k.a violating the intent of network administrators). Rather, we need formal theoretical models and methodologies for this purpose. Another challenge lies in the fact that checking misconfigurations should introduce relatively small overhead. Misconfiguration detection in SDN policies will compute match patterns of rules. However, SDN rules support arbitrary bitmask patterns that will add the computation complexity. The complex action behavior of SDN rules further increases the difficulty for accurate checking.

To address these challenges, we develop a Policy Misconfiguration Management tool (PMM) that works at the data plane. It transforms arbitrary match patterns of SDN rules into a set of intervals via numerical calculation, and uses a new data structure—minimal interval set, to represent rulesets. This numerical representation accelerates misconfiguration checking. We develop compositional algebra and misconfiguration checking theory that PMM leverages off to design accurate and fast checking algorithms. PMM also adopts optimisations for

reduced overhead. We finally evaluate PMM with the implementation in a SDN compositional hypervisor (CoVisor [17]).

To sum up, the contributions of this paper are three-fold:

We present a new data structure—minimal interval set, to represent the match patterns of individual and compositional SDN rulesets. This representation accelerates misconfiguration checking.

We formalize the misconfiguration checking problem and propose two checking principles (*Rule-Usefulness* and *Rule-Minimalism*). We develop methodologies and theorems to implement these two rules for accurate misconfiguration checking.

We design a misconfiguration checking tool and implement it in CoVisor. The implementation adopts optimisations for reducing overhead. Experiments with synthetic and random rulesets demonstrate its efficiency.

II. Background and Motivation

SDN policies can be expressed as a set of rules to be deployed to underlying network devices. These rules might be defined through high-level SDN languages (*e.g.*, Frenetic [12], Pyretic [28] and PGA [30]), or low-level programming interfaces supported by controllers (*e.g.*, NOX [14] and Onix [22]). These rules classify packets into flows, and apply them to the associated actions. As an example to illustrate our presentation, and without loss of generality, we will use OpenFlow [3] in this section, and describe the general case in Section III. We will assume that each SDN policy contains one flow table, as a pipeline composed of a sequence flow tables is similar to a sequential composition.

A rule in a flow table equates to a flow entry, a 3-tuple $r = (p, m, al)$, with $r.p > 0$ being the priority of r that defines precedence of this rule; $r.m$ represents the patterns to match for packets in the flow; and $r.al$ represents the action to apply to the specified packets. The patterns $r.m$ define a flow, and consist of one or more matching patterns that are defined over different fields. OpenFlow defines a few different fields that can be used in defining flows. The matching patterns might be *Bitmask-based* with wildcards (*e.g.*, $0*1*$ or $0*10$), or logical (*e.g.*, $TCP.ACK = true$). It is noteworthy that bitmasks have not to be prefixed (to begin with a bit mask).The rule also contains an action part that defines the set of predetermined atomic actions, like *drop*, *fwd*, *push/pop tag*, *etc.*, to apply to packets in the flow. For example, adding a VLAN tag for packets matching $r.m$, and then forwarding them to port 1.

The combination of matching patterns and action semantics enables a large variety of policies that are leveraged by most packet processing systems like firewalls, intrusion detection systems, switches and routers. SDN introduces a new dimension that increases flexibility: *composition* [11], [17]. Composition consists of combining several packet processing components and their related policies into more complex processing pipelines. SDN introduces an algebra of simple composition operators [13]: the *parallel operator* ($+$), the *sequential operator* ($>>$) and the *override operator* ($\triangleright$). Parallel operator assumes that a copy of the input traffic is forwarded to both components, and therefore rules relative to each component are applied independently. For sequential operator, the traffic is forwarded from the first to the second stage, and rules of the second stage are applied on the output of first stage that has been applied to traffic matching the first stage's rules. The override operator defines an intermediate operation where both components are running but when a packet matches the two components' rule, only the first one is applied.

Composition increases notably the flexibility of SDNs. In particular, it enables to combine components developed by different providers, and to deploy them together to address complex network scenarios. However, this comes with a cost in term of complexity, and in term of possible misconfiguration. We illustrates this through one example shown in Fig. 1. We consider three SDNs components: SP1, a NAT component that is masquerading input IP address; SP2, a load balancer; and SP3, a monitoring device. These three components are composed through a sequential operator and a parallel operator: SP2 and SP3 being in parallel after the sequential SP1 component. SP1 implements three rules, while SP2 have four rules and SP3 consists only of a single rule. Note that rules in each component are ordered by decreasing precedence.

In SP1, r_1 hides the source IP addresses of some packets ($srcip = 10.1.1.16$) that also are interested by SP2. Unfortunately, due to the sequential composition, SP2 cannot utilize its r_1 to block the packets with $srcip = 10.1.1.16$ since their source IP addresses have already been translated by SP1. Similarly, SP3 cannot monitor what it cares about. Obviously, compositional operations are error-prone and easily lead to *function conflict*. Furthermore, in SP2, r_2 are logically redundant because packets it cares about can be operated by r_4 equivalently. This redundancy that is very common in individual components, is exacerbated by the composition that will combine them with other rules. As resources are scarce in the data plane it is important to detect these redundant rules in order to optimize resource usage.

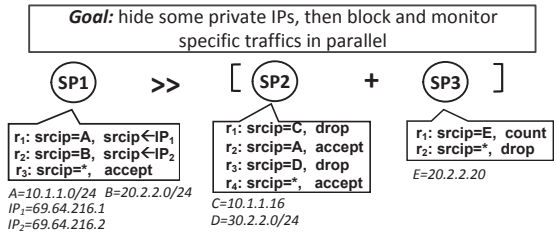

Fig. 1. Illustration of SDN policy anomalies.

The above example demonstrates different types of misconfigurations that can happen in SDN with composition. Some of misconfigurations have already been pointed out in [32], [15]. The goal of this paper is to provide a global framework for representing composed SDN rules. This framework will enable the detection of existing misconfigurations and help in optimizing SDN rules in a composition framework. The framework works at the data plane for detecting misconfigurations. This is because all relevant misconfigurations happening

in the SDN, whether resulting from composition (inter-policy) or being only related to a single component (intra-policy), will show up at the data plane and translates into abnormal behaviors in packet forwarding. Specially, our approach consists of *i*) translating policies into a data plane representation that applies to a composition algebra, *ii*) detecting misconfigurations over the composed data plane representation.

III. Data Structure for Ruleset Representation and Composition Algebra

In this section, we will assume that we have a ruleset $\mathcal{R} = \{r_i\}$ consisting of rules r_i ordered by their precedence. Each rule r_i consists of a matching patterns $r_i.m$ and an action part $r_i.al$. This ruleset might come from a single SDN component or might result from composed components through a composition algebra that we will describe later.

A. Minimal interval set representation of ruleset

Matching patterns are applied to classification fields. These fields are bit-set of different lengths, *e.g.*, 32 bits for IPv4 addresses, 16 bits for TCP port numbers, 1 bit for logical values like TCP.ACK. The rule matching ignores the protocol-specific meanings associated with each field and match them with a flat bit pattern composed of 0,1 and *, *e.g.,* 0*0*. These patterns can be transformed into a set of non-overlapping intervals of contiguous values, where all values in intervals match the patterns. This results into a geometric representation of the SDNs' rules. For example, pattern 0*0* over a four bits field translates into 2 intervals of size 2, $[0000, 0001], [0100, 0101]$, that are generally represented using a decimal representation for the bit-fields, $[0, 1], [4, 5]$. We show in Fig. 2 one example illustrating how the rule 0*0* is translated into intervals for a 4 bits field.

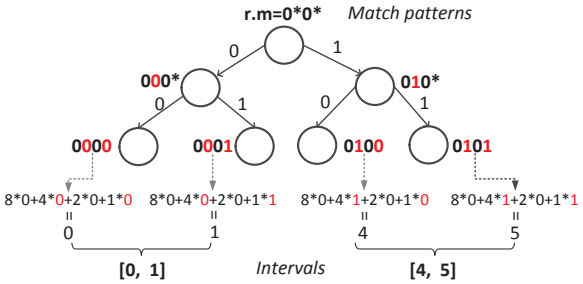

Fig. 2. Geometric representation of a SDN rule as interval

In most general terms, let's consider a matching pattern operating over a field with N bits, and with k of its bits being set explicitly in the pattern and l '*' being used in the pattern, *e.g.*, for 1*01 that is defined over an IP address field with $N = 32$, we have $k = 3$ and $l = 1$. The number of intervals needed to represent this pattern depends on its suffix. If the suffix is *, like in 1*0*, this interval will be represented with $L = 2^{k+l-1}$ intervals of length 2^{N-k}. If the suffix is a bit pattern of length $m \leq l$, *e.g.*, 1*10 has a suffix of size $m = 2$, the pattern will be represented by $L = 2^{k+l-m}$ intervals of length 1 each.

Therefore, the matching pattern used for a rule defines L intervals that partition the whole range of field values into at most $2L + 1$ disjoint intervals. We can label the L intervals matching the rule with the rule number or the rules actions. We call these intervals the "R1 matching intervals". This results in a geometrical representation of a rule R1 as a set of R1 matching intervals.

However, rules in a ruleset generate overlapping intervals. We thus define a data structure named *"minimal interval representation of the ruleset"*. The idea is to represent the whole ruleset with a segment tree [10] containing all disjoint intervals. Let's illustrate this data structure with a concrete example. In Fig. 3 nine overlapping intervals relative to a rule set with 3 rules. These rules induce a partition into 16 intervals of the fields over which the patterns are defined, *e.g.*, [0,3], [3,5], [5,9], *etc.* An interval is labelled with the rules that are matched in it. In order to differentiate these intervals with rule matching intervals defined above, we will call them, *partition intervals*.

We build the minimal interval representation over all rule matching interval set I. We first sort all endpoints of the intervals in I. This generates the elementary intervals. Then, a balanced binary tree is built on the elementary intervals, where each node v represents an interval Int(v). The tree is built by inserting interval in I one by one into the segment tree. An interval $[a, b]$ can be inserted into a subtree rooted at node T, using the following procedure. If Int(T) is contained in $[a, b]$, then we store $[a, b]$ and T and finish. Otherwise, if $[a, b]$ intersects the interval in left (right resp.) child of T, then we move to the sub-tree on the left (right resp.) and try recursively to insert the interval into it. The complete construction operation takes $\mathcal{O}(nlogn)$ time, n being the number of segments in I.

This partition is minimal as it is not possible to use less partition intervals to represent the ruleset, and it is also maximal in the sense that adding any partition interval will result in two contiguous interval with exactly the same rules in the label. This partition is therefore equivalent to the ruleset. It can easily be constructed by putting all extremities of all intervals in the ruleset and by ordering them in increasing order. The consecutive values in the resulting sorted list are intervals boundary in the minimal interval representation. Thereafter the interval can be labelled by finding all rules overlapping with it.

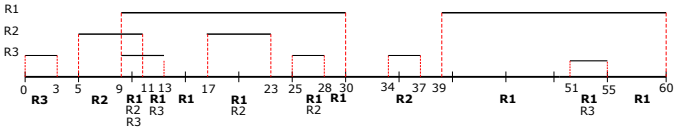

Fig. 3. Example of a minimal interval representation for a ruleset

The minimal interval representation has a fundamental property: only rules that are in the label set of an interval are overlapping, and other rules are completely independent of them.

Moreover, among all rules in the interval's label there is one that has the highest precedence. One can label the interval only with this label, because the action that is going to be applied to a packet in this interval will be the one with the

highest precedence among overlapping rules. This might make possible to merge some intervals in order to maintain the minimal property. Applying this approach to the example shown in Fig. 3 will result into 8 partition intervals for representing the ruleset, as for example the intervals [39,51], [52,55] and [56,60] will be merged into a single interval [39,60] that will be labeled with R1. In the forthcoming we will use this representation as the minimal interval representation of a component ruleset.

One important issue is relative to the size of the minimal interval representation. It is straightforward that adding a new rule interval into a ruleset will at most add two new partition intervals. Therefore the number of partition intervals cannot be larger than 2 times the rule partition intervals. For cases where the rules are prefix based, *i.e.*, rules have a bit value prefix and a * suffix like the rules used commonly as IP address mask, the number of rule's interval for each rule will be 1, and the the number of partition interval is at most $2N$, where N is the number of rules. That said, the number of partition's interval will be $\mathcal{O}(N)$, but will depend precisely on the features of the ruleset.

The minimal interval representation can be extended to multiple fields (or dimensions) directly. Let's suppose that a ruleset is defined over multiple fields. By choosing one of the fields we can build a minimal interval representation for this dimension, with each partition interval I containing the list of overlapping rules R^I. Now we can choose another field and build a new minimal interval representation over the second field but only with rules in R^I. This minimal representation is attached the previous interval in place of the set R^I. We can continue this operation recursively for each interval, until it remains in the partition interval only a single rule, or there is no more fields to define partition over it. This results into a nested interval representation, which can be represented by a multidimensional tree of intervals. Following this approach, a multiple fields ruleset is translated into a set of multi-dimension partition interval, we will call in the forthcoming d-intervals and note as $(I^1, I^2, ..., I^K)$ where the interval I^i is relative to dimension (or field) i. Each d-interval is labeled with a set of rules matching it. This set might be empty, or contains only the rule with highest precedence matching it, or contains all rule matching the partition interval. The overall space complexity of the minimal interval partition will be $\left(N \log N^{K-1}\right)$, where N is the number of rules and K is the number of fields used for the matching [9].

It is noteworthy that we can also use the minimal interval representation to characterize a packet flow over a link by assuming a rule that should match all packets on the link. Let's suppose that all packets crossing a link are following some common characteristics, *e.g.*, all packets coming out of a web server share the same source IP address and source port number 80, one can characterize the packets crossing this link by a rule matching the source IP address and the source port number. The minimal interval representation of this rule can be used as the link minimal interval representation. In order to separate the link minimal interval representation from the ruleset's representation, we will represent in the forthcoming a link's minimal interval representation as J, and a ruleset's representation as I. This enables us to define the action of a packet processing device as a mapping from the minimal interval representation of the ingress link (that might contain all the field space) into the minimal interval representation of the egress link opening the way for an algebraic characterization of packet processing systems. This is what we will be developing in the next section.

B. Composition algebra for minimal interval representation

Let's suppose we have translated the rulesets relative to different SDN components into minimal interval representations. The question is: how can we derive the final minimal interval representations after composing these components. For this purpose we will develop a composition algebra for minimal interval representations. But we need to first introduce 3 operations: the restriction operator, the union operator and the minimal interval representation mapping.

Restriction operator. Let's assume a minimal interval representation of the traffic on a link L to be J_L and that this traffic is processed by a ruleset with minimal interval representation I_R. We can define a restriction operator between J_L and I_R that generate a new minimal interval representation, noted $\mathsf{I}_R^{\mathsf{J}_L}$, that contains for each interval in I^R the sub-intervals where J^L is set. It is therefore a restriction of the ruleset to the specific traffic pattern in link L. The restriction operation is in fact transforming the ruleset R acting on the traffic on link L into an equivalent ruleset acting on a link without restriction.

Union operator Suppose we have two ruleset's minimal interval representation I_1 and I_2. The union minimal interval representation $\mathsf{I}_{1 \cup 2}$ is the minimal interval representation resulting from the union of the rules in each ruleset, with the restriction that attaching only the highest precedence rule to each interval. We will attach the two rules relative to the each of the initial rulesets to any resulting interval.

Minimal interval representation mapping. A flow might be transformed by going through a packet processing component. For example, the TCP/IP header of a flow going through a NAT components is changed by the masquerading process. In a more general term, one can see the effect of a packet processing component as a mapping between an input state space into an output state space, where the state being the fields that are used to represent the packet flow. As we described early, the action of a packet processing component is controlled by the minimal interval representation *i.e.*, the action relative to the rule with highest precedence is applied to all packet fitting in a partition interval. This action maps a given input partition interval into potentially several other intervals defined over possibly other fields. For example, a NAT component that takes a set of source IP addresses identified by an IP mask like 192.168.*/16 and maps it over a visible address like 12.34.45.78, is mapping an interval of length 2^{16} into an interval of size 1 both defined over the source IP address field. However, more complex mapping

might also be possible. For example, a de-multiplexer will map a single d-interval into several disjoint intervals. We thus able to represent the action of a packet processing component as a mapping that takes a given ingress link minimal interval representation J^i and mapping it into an egress link minimal interval representation $\mathsf{J}^o = (\mathsf{J}^i)$. The mapping depends completely on the restricted ruleset minimal interval representation $\mathsf{I}^{u\mathsf{J}^i}$. Therefore one can represent a packet processing component, *e.g.*, a SDN components, with its restricted ruleset minimal interval representation $\mathsf{I}^{u\mathsf{J}^i}$.

With the above 3 basic operators, we then develop a composition algebra (sequential operator, parallel operator and override operator) for minimal interval representation.

Sequential Operator. As explained in Sec. II, the sequential operator $\gg$ is one operator enabling sequential SDN composition. Let's assume that two SDN components C_1 and C_2, with unrestricted minimal representation I_1 and I_2, and mapping function $_1$ and $_2$, are sequentially composed. This composed component is equivalent to a SDN component with restricted ruleset minimal interval representation $\mathsf{I}_2^{u\Phi_1(\mathsf{J}_i)}$.

Parallel Operator. A second composition operator in SDN is the parallel operator +. A parallel SDN component is equivalent to a SDN component with ruleset minimal interval representation defined as the union of the two components minimal interval representation.

Override operator. The last composition operator in SDN is the override operator. The minimal representation of an overridden SDN component can easily be derived by looking at interval in minimal interval representation that have not any rules assigned to it, and to check if by default a rule is applicable to a subset of this interval.

IV. Misconfiguration Checking Theory

A. Definition of Misconfigurations

To restate, our aim is to detect misconfigurations in compositional SDNs. However, we need to clarify which type of misconfiguration we are checking. In this paper, we are mainly interested in functional misconfigurations arising because of SDN composition. This means that we are not validating the functions of each individual component, but rather seeing how these components interact. In this context the misconfigurations we target are the cases that will make any individual component not working as expected because of being composed with another component. We call these misconfigurations ***functions conflict***. In addition, we also consider redundancy in each individual component. This is because one component redundant rule may lead to exponential redundant rules after it is composed with other components. We call these misconfigurations ***deployment inefficiency***. It is noteworthy that we are not aiming at detecting implementation related misconfiguration like memory or CPU races, or to detect logical and semantic errors that might be caused by misconfiguration of individual components, *e.g*, missing to cover a port number range in a firewall component. In the forthcoming, we will describe in more details both misconfigurations.

Deployment inefficiency. In compositional SDN [28], [17], [11], multiple policies can cooperatively manage network traffic. These policies are compiled into a single one and then deployed to the underlying devices. The compilation operation is the Cartesian product of policy rules. Let us take two policies $sp_a = \{r_{a1}, r_{a2}, ..., r_{am}\}$ and $sp_b = \{r_{b1}, r_{b2}, ..., r_{bn}\}$ as an example. $sp_a \times sp_b = \{r_{a1} \oplus r_{b1}, r_{a1} \oplus r_{b2}, ..., r_{a1} \oplus r_{bn}, ..., r_{am} \oplus r_{bn}\}$, where operator $\oplus$ is used to compute the match patterns of two operands (rules) and decide whether to build a new composed rule or not. Thus, if there is a redundant rule r_{ak} in sp_a (r_{bk} in sp_b resp.), it will generate n (m resp.) redundant rules, $\{r_{ak} \oplus r_{b1}, r_{ak} \oplus r_{b2}, ..., r_{ak} \oplus r_{bn}\}$, in the worst case for the composed policy. We say *a rule is redundant* if the packets that it cares about can be operated by other rules equivalently. For instance, r_2 in SP2 in Fig. 1 is redundant because r_4 can operate the packets r_2 interested in equivalently. Through a sequential operator, r_2 in SP2 combines r_2 and r_4 in SP1 to generate composed but redundant rules because these rules can be completely replaced by the composed rules generated from r_4 in SP1. Redundant rules, hence, can lead to exponential redundancy in compositional SDN, which is unacceptable for the scarce resources in underlying devices at the data plane.

Function conflict. Network policies are volatile due to traffic load shifting, customer demand changes or network topology upgrading. SDN provides an opportunity for network administrators to deploy and update their customized policies[34], [19], [24]. However, frequent update by network administrators is error-prone [31] and easily leads to *function conflict*. There are two possible cases of function conflict. The first case is relevant to scenarios where two rules in a policy or two parallel component policies care about same packets (due to either match patterns overlap or containment[1]) but operate them differently (due to different actions). The second case is relevant to the sequential operator in compositional SDNs, where the rules in the earlier stages change the packet fields that the rules in the following stages will match. For instance, in Fig. 1, SP1 masquerades input IP addresses but a following SP2 wants to monitor traffic based on their original IP addresses. Thus such a configuration makes SP2 function failure. In addition, r_4 in SP2 and r_2 in SP3 operate default traffic in parallel. However, their behaviors are contradictory. Thus function conflicts may violate the intents of network administrators.

B. Misconfiguration checking principles

Indeed, misconfiguration is prone to happen among the rules whose match patterns are overlapped. This is because the overlapped rules naturally have common interest in the same packet sets. We thus having the first principle on the usefulness of any rule in a policy.

Principle 1. *(Rule-Usefulness). Each rule in a SDN policy should match and process packets. That said, for a SDN policy*

[1]In this case, the match patterns of one rule (e.g. r_i) is included in another's (e.g. r_j) entirely. In other words, r_i.m $\subseteq r_j$m.

$sp = \{r_1, r_2, ..., r_n\}$ *where* $r_1.p \geq r_2.p \geq ... \geq r_n.p$, $\forall r_i \in sp$, $\{PS - (r_1.m \cup r_2.m \cup ... \cup r_{i\ \ 1}.m)\} \cap r_i.m \neq \emptyset$, *where* PS *refers to the header space of packets.*

This principle can be easily implemented by our previously proposed minimal interval representation. It can be used for checking the above misconfigurations. In particular, it directly means that each rule in a SDN policy should not be redundant, tackling the *deployment inefficiency*. It deals with the *function conflict* within a policy by checking the existence of match patterns overlap and containment of rules. For the possible function conflict due to sequential composition, we can first compose the policies and then check for the composed policy the existence of match patterns overlap and containment

Another possible case of rule redundancy is not covered by the above principle—two rules "close" enough in a rule-set having equivalent action lists and with mergeable match pattern. We define it formally in the following principle.

Principle 2. *(Rule-Minimalism). Consider a SDN policy* $sp = \{r_1, r_2, ..., r_n\}$. *If the following three conditions are hold for* r_i *and* r_j, *then they can be merged into one rule* r_g: *i)* $r_i.m$ *and* $r_j.m$ *are mergeable (see Theorem 1); ii) the minimal distance between* r_i *and* r_j *is 1 (see Definition 1); iii)* $r_i.al \equiv r_j.al$ *(see Theorem 2).*

This principle focuses on the compactness and minimised quantity of rules. Indeed, the number of rules supported by underlying devices are limited given the cost and power consumption [18], [7]. In view of this, any rule that violates this principle can lead to *deployment inefficiency*. Next, we elaborate the details of the three conditions of this principle.

1) Mergeable match patterns: Mergeable match patterns means they can be merged into one single match pattern. Theorem 1 shows how to decide whether two match patterns are mergeable or not.

Theorem 1. *(Match Pattern Mergeable Theorem). For two rules* r_i *and* r_j, *their match patterns are mergeable as long as they satisfy one of the below three conditions: i)* $C_m^1 : r_i.m \subseteq r_j.m$; *ii)* $C_m^2 : r_j.m \subseteq r_i.m$; *iii)* $C_m^3 : r_i.m$ *and* $r_j.m$ *are identical except only one non-wildcard bit*[2].

Proof. We prove this by enumeration that $C_m = C_m^1 \vee C_m^2 \vee C_m^1$ covers all the cases, which enable $r_i.m$ and $r_j.m$ mergeable: (1) If the relationship between $r_i.m$ and $r_j.m$ are containment, it leads to $r_i.m \cup r_j.m = r_i.m$ or $r_j.m$, which is evidently mergeable. Thus both C_m^1 and C_m^2 are the sufficient conditions that enable $r_i.m$ and $r_j.m$ to be mergeable. (2) If $r_i.m$ and $r_j.m$ are intersecting, at least two bits are different. This is because if their only one bit is diverse, we can assume that this bit can be * for $r_i.m$ and 0/1 for $r_j.m$. As a result, $r_j.m \subset r_i.m$. Obviously, if $r_i.m$ and $r_j.m$ contain more than one different bit, they cannot be merged into one single rule. (3) If $r_i.m$ and $r_j.m$ are disjoint, we say they are mergeable when they contain only one different non-wildcard bit (e.g. $r_i.m$=*0*0 and $r_j.m$=*1*0). This is because if this bit of $r_i.m$ ($r_j.m$, resp.) is wildcard, it leads to $r_j.m$ ($r_i.m$, resp.) $\subset r_i.m$ ($r_j.m$, resp.). In addition, bit 0 and 1 can be merged into *. Therefore, C_m^3 is another sufficient condition that makes $r_i.m$ and $r_j.m$ mergeable. □

[2]This bit in both r_i:m and r_j:m must be non-wildcard.

It is noteworthy that we can easily check whether the three conditions in Theorem 1 hold or not with the minimal interval set representation described in Section III.

2) Rule distance computation: Next we discuss the *rule distance*, which is a measure of the closeness between two rules (see Definition 1).

Definition 1. *(Rule Distance). Consider a policy* $sp = \{r_1^{(1)}, r_2^{(2)}, ..., r_n^{(n)}\}$, *where* $r_1^{(1)}.p \geq r_2^{(2)}.p \geq ... \geq r_n^{(n)}.p$. *The superscripts of the rules refer to their locations in* sp, *such as* $r_i^{(j)}$ *locates the* j*-th position in* sp. *For rule* $r_g^{(k)} \in sp, r_h^{(t)} \in sp$, *we define the distance between them as dist*$(r_g^{(k)}, r_h^{(t)}) = \|k - t\|$.

Note that we can minimize the distance of two rules because of the *commutativity property* in rules: if there is no overlap between rules, they can exchange their position. For example, in the left sub figure of Fig. 4, because $r_3^{(3)}.m \cap r_4^{(4)}.m = \emptyset$, we can exchange them. As a result, $r_3^{(3)}$ becomes $r_3^{(4)}$ while $r_4^{(4)}$ is changed to $r_4^{(3)}$ (see the middle sub figure). Similarly, we exchange $r_4^{(3)}$ and $r_2^{(2)}$. Finally, sp_1 becomes sp_2 (see the right sub figure). sp_1 and sp_2 are equivalent as the traffic each rule can process is identical. Through this commutativity property, the distance between r_1 and r_4 has been reduced to 1.

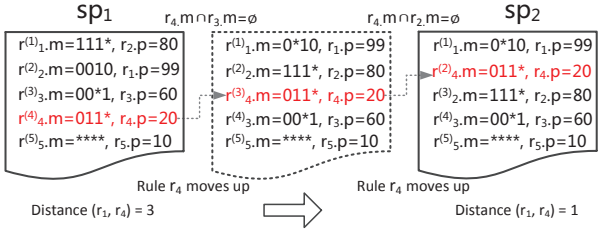

Fig. 4. An example of the commutativity property of rules.

Then, we define the *minimal distance* of two rules: we unitize the rule commutativity property to exchange the rule positions to reduce their distance until the commutativity property does not hold any more or their distance becomes 1. At this moment, the distance is their minimal distance. We say two rules are "close" enough if their minimum distance is one. In other words, they can become adjacent rules.

3) Action Equivalence Theory: The actions defined in OpenFlow [3] can be categorized into three categories: *modification* actions, *output* actions and *miscellaneous* actions. The modification actions are to change packet headers (e.g.*set_field* actions). The goal of output actions is to forward packets to some specific ports (e.g. *fwd* and *drop* actions). Miscellaneous actions do not operate packets directly. Instead, they either generate some statistics (e.g. *count*) or facilitate packet processing (e.g. *clear_action*). Different types of actions can introduce different "effects", making it possible to constitute a complex action list.

For a rule r, we view its action list $r.al$ as a *black box*. Logically, when one input packet, denoted by p_{in}, enters this *black box*, it will generate one or more output packets (denoted respectively by $\{p_{out1}, ..., p_{outn}\}$) to specific switch ports. Note that the headers of the output packets may be different due to *modification actions*. As a result, these output packets and ports together form a set of tuples, each of which can be denoted as PP=$\langle p_{out}, port\rangle$, where the packet PP.$p_{out}$ is transmitted via PP.$port$. We use $r.al^{\mathsf{PP}}$ to refer to the set of tuples. In addition, $r.al$ may generate a few statistics and modifies the status of flow tables, denoted as $r.al^{\mathsf{d}}$. The "effects" of the action list of r is then constituted by $r.al^{\mathsf{PP}}$ and $r.al^{\mathsf{d}}$. Fig. 5 shows an example. This action list will generate two PP tuples ($\langle p^1_{out}, 3\rangle$, $\langle p^2_{out}, 4\rangle$) with once counter update.

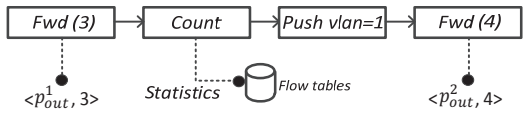

Fig. 5. Example of an action list. p^1_{out} and p^2_{out} represent two output packets.

Thus, we conclude that if two action lists can generate the same "effects" for any input packet, we say they are equivalent. We formalize this conclusion as Theorem 2.

Theorem 2. *(Action List Equivalence). Consider two rules, r_1 and r_2, their action lists are equivalent if and only if both $r_1.al^{PP} \equiv r_2.al^{PP}$ and $r_1.al^{d} \equiv r_2.al^{d}$ hold with the same input packet p_{in}.*

Proof. We prove this by analogy. For any action list $r.al$, it can be viewed as a function, whose input is packets that hit rule r and output is its generated effects ($r.al^{\mathsf{PP}}$ and $r.al^{\mathsf{d}}$). Based on function equivalence, we can say two functions ($r_1.al$ and $r_2.al$) are equivalent as long as their output ($r_1.al^{\mathsf{PP}} \equiv r_2.al^{\mathsf{PP}}$ and $r_1.al^{\mathsf{d}} \equiv r_2.al^{\mathsf{d}}$) are identical for any same input. □

This theorem leads to a simple checking about whether two action lists are equivalent or not. The core idea of this algorithm is to compare the PP tuples and statistics generated by two action lists. Note that we do not need to check all possible input packets. Instead, we only need one special packet whose packet header consists of only wildcards, use two action lists to operate on it and check whether their outputs (e.g. $r.al^{pp}$ and $r.al^{d}$) are identical or not. The computation complexity of this algorithm is $\mathcal{O}(n^2)$, where n is the number of the tuples. However, n is very small in practice, usually no more than 5. Therefore, the overhead of this algorithm is acceptable in practice.

V. Design and Implementation of PMM

This section presents the design and implementation of PMM , a tool for checking misconfiguration in compositional SDN polices based on the ruleset representation (see Section III) and misconfiguration checking theory foundation (see Section IV). To simplify our description, we first show how to check misconfiguration in a single policy, and then discuss the misconfiguration in compositional policies.

A. Intra-policy Misconfiguration Checking

Checking intra-policy misconfiguration consists of three steps as follows.

Rule transformation. A native SDN policy is represented as a list of prioritized rules where the match pattern of each rule is bitmask-based with wildcards. The target of the rule transformation is to transform each rule match pattern into the interval representation.

Minimal interval set generation. After the rule transformation for a policy ruleset, we group these intervals and generate an interval set and minimize it. An interval may associate with multiple rules.

Misconfiguration checking. Based on the interval sets, we check whether there are cases that violate the principles described in section IV.

This section focuses on the third step about how to check misconfiguration for a given ruleset. Rules in the ruleset are ordered by their priorities. We begin with the rule of the highest priority, and then add one rule each round from the second highest-priority rule to the lowest. The checking process described below is applied for each round until some misconfigurations are identified or we reach the lowest-priority rule.

Let us suppose a minimal interval set mis corresponding a set of rules R with no misconfigurations is generated after k rounds. Each interval (e.g. iv) in mis associates with a list of rules ranked by their priority. We use $iv.L$ to denote iv's associated list of rules. We now need to add r at the $(k+1)$-th round into R and check whether there will be misconfiguration after adding r. To this end, the match patterns of r, $r.m$, will be transformed into a set of intervals. Let us denote the intervals in mis that is overlapped with r's intervals as $r.iv$. We then check whether misconfigurations raise following the two principles described in Section IV.

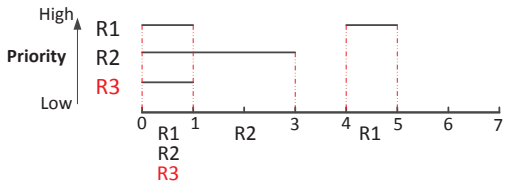

Fig. 6. Example of the redundant rules checking.

Checking against Principle 1 . We scan $r.iv$ to identify whether there are intervals whose associated rule list contains only r. If there are no such intervals, adding r will result in violation of *Principle 1*. Otherwise, we continue checking against Principle 2. Figure 6 shows an example, where $R3$ with the lowest priority needs to be added. $R3.iv$ contains only the interval $[0, 1]$, which already has two associated rules and thus violates emphPrinciple 1. Indeed, the interval $[0, 1]$ will associate with three rules after adding $R3$, but $R3$ is with the lowest priority and thus not useful.

Checking against Principle 2 . For each interval in $r.iv$, we identify its adjacent intervals, and then assemble the adjacent intervals of all intervals in $r.iv$ into a set, denoted

as $r.iv^{(aj)}$. We then check whether there is a rule $s \in R$ that is mergeable with r. Following Theory 1, s and r can be merged into one rule if the following three conditions are hold: (1) $s.iv \subseteq r.iv^{(aj)}$; (2) $\forall iv \in s.iv$, s is the lowest-priority rule in the list associated with iv; (3) $s.al = r.al$ (*i.e. the two action lists are equivalent.*). If the conditions hold, s and r can be merged into one rule, and this violates Principle 2. Otherwise, no misconfiguration will raise when adding r into R. We show in Figure 7 an example to illustrate the above checking process. For the new rule $R3$, its corresponding adjacent intervals set $R3.iv^{(aj)} = \{[0,1], [4,5]\}$. We can also see that $R2.iv \subseteq R3.iv^{(aj)}$ and $R2$ is the lowest priority rules in the associated lists of $R2.iv$. If $R2.al = R3.al$ (see Theory 2), $R2$ and $R3$ can be merged into one rule. That said, adding $R3$ will raise misconfigurations.

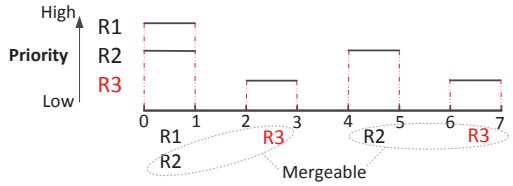

Fig. 7. Example of the mergeable rules checking.

Finally, if adding r does not incur misconfiguration, we update R by adding r and compute the updated mis with the minimal interval set representation of R (see Section III). We then continue to the next round of checking if needed.

The pseudo code of the above misconfiguration checking process within individual policies is listed in Algorithm 1.

Algorithm 1: CHECKINGPOLICY(sp)

Input: sp, a policy ruleset.
Output: Return 0 if there is no misconfiguration in sp, or the index of the first rule that causes misconfiguration.

```
1  mis ← ∅ and num ← 0;
2  foreach R ∈ sp do
3      num ← num + 1 and is_redundancy ← true;
4      mis ← computeMinSet(R, mis);
5      foreach iv ∈ R:iv do
6          if iv:L.head ≡ R then
7              is_redundancy ← false;
8              break;
9      if is_redundancy then
10         return num;
11     R:iv^(aj) ← computeAdjset(R:iv) and AdjRule ← ∅;
12     foreach iv^θ ∈ R:iv^(aj) do
13         AjdRule:insert(iv^θ.tail);
14     foreach R^θ ∈ AjdRule do
15         if R^θ:iv ⊆ R:iv^(aj) && ActionEquivalent(R^θ:al; R:al)
             then
16             return num;
17 return 0;
```

B. Misconfiguration Checking for Compositional Policies

Compositional SDN [28], [17], [11] defines operators that enable multiple polices cooperatively operate same traffic. However, different cooperative operations may introduce diverse types of misconfiguration. To this end, we utilize the composition algebra defined in Section III to obtain the minimal interval set representation, and then apply the detection theories and algorithms check the identify misconfigurations.

In what follows, we consider two policy sp_1=$\{r_1, r_2,..., r_n\}$ and sp_2=$\{r_1^{\theta}, r_2^{\theta},..., r_n^{\theta}\}$ that are composed by compositional operations. We assume that there is no intra-policy misconfiguration in sp_1 and sp_2. That said, both have passed the checking with Algorithm 1.

Parallel operation. In parallel operation, sp_1 and sp_2 operate the same traffic in parallel. We use mis_1 and mis_2 to respectively represent the minimal interval sets of sp_1 and sp_2. If $mis_1 \equiv mis_2$, this parallel operation is misconfiguration-free. Otherwise, it means that one packet can be processed in sp_1 (sp_2, resp.), but there is no rule in sp_2 (sp_1, resp.) to operate this packet. It violates the semantic of the parallel operation that enables sp_1 and sp_2 process the same traffic.

Sequential operation. In this operation, sp_1 and sp_2 process packets sequentially. We assume that packets are processed by sp_1 first, and then are operated by sp_2. Thus sp_1 can affect the packet proceeding of sp_2. In this case, we can use the compiling algorithm [17] to compose sp_1 and sp_2 into one ruleset sp_3 at first. And then, we use the approach of intra-policy misconfiguration checking (Algorithm 1) to check sp_3.

Override operation. The override operation makes one policy being the default policy for the other policy. We assume that sp_2 is assigned to the default policy for sp_1. As a result, packets will be processed by sp_1. If there is no rule in sp_1 that can operate packets, they will be processed by sp_2. We merge sp_1 and sp_2 into one single policy sp_3 by stacking sp_1 on top of sp_2 with higher priority. Next we detect whether there is misconfiguration in sp_3 with the intra-policy misconfiguration checking algorithm (Algorithm 1).

C. Optimisation

The match patterns of each rule will be converted into a set of intervals so that they can perform interval operations for misconfiguration detection. The performance of the misconfiguration checking is dependent on the number of intervals. As mentioned before, the wildcard bit in a rule r affects the number of intervals it can generates. For example, if $r.m = 001*$, its corresponding interval set is $\{[2,3]\}$. However, if $r.m$ is $00*1$, its interval set becomes $\{[1,1],[3,3]\}$.

For a match pattern, we use w to refer to the number of wildcard bits and sw to refer to the length of the longest suffix with wildcard. We compute the *interval indicator* as $\{w-sw\}$. Then the number of intervals is 2^n, where n is the *interval indicator*. For example, for a match pattern $r.m = 0**0*$, the number of wildcard bits is 3 while the length of the longest suffix with wildcard is one. Therefore, the number of intervals is $2^{3-1} = 2^2 = 4$: $\{[0,1],[4,5],[8,9],[12,13]\}$.

We propose an optimisation mechanism to reduce the complexity of set operations by decreasing n. Specifically, for a set of rules, we simultaneously move the wildcard bits of all rules

at the same locations backward by one 1 bit (see Figure 8). Through such a movement, R becomes $R^{'}$, where the number of intervals in $R^{'}$ is much less than that in R.

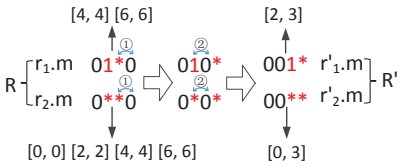

Fig. 8. Reducing the number of wildcard bits proceeding the first non-wildcard bit from the tail.

Note that such a simultaneous movement will not affect the misconfiguration checking results. This is because although simultaneously moving the wildcard bits will result in the change of the minimal interval set representation of individual rules, the relationship between the two sets for two rules remain unchanged. Here the set relationship includes intersection, containment and equivalence. Let us consider two rules r_1 and r_2. If we simultaneously exchange arbitrarily bits for r_1 and r_2, they become $r_1^{'}$ and $r_2^{'}$. Next we perform a set operation between $r_1.m$ ($r_1.m$) and $r_2.m$ ($r_2^{'}.m$), generating a result t ($t^{'}$). Finally, if we restore the positions of the exchanged bits for $t^{'}$, the result will be equal to t. For example, we assume $r_1.m = 0*1*$ and $r_2.m = *0*1$. We exchange the second bit and the third bit so that $r_1.m^{'} = 01**$ and $r_2.m^{'} = **01$. Next $t = r_1.m - r_2.m = \{0010, 0110, 0111\}$ and $t^{'} = r_1^{'}.m - r_2^{'}.m = \{0100, 0110, 0111\}$. Finally, if we exchange the second bit and the third bit for $t^{'}$ (a.k.a restore the positions), the result is $\{0010, 0110, 0111\}$ which is identical to t. Thus, we can use this optimisation to reduce the complexity of set operations.

VI. Experiments

A. Experimental Setup

We have implemented our PMM based on CoVisor [17], a compositional SDN hypervisor. We deployed the implementation on Intel®Core(TM) i7-8700CPU, clocked at 3.20GHz. In the experiments, we used two types of rules:

1) D1 (synthetic rulesets): we use ClassBench[3] [4], to generate synthetic rulesets. D1 contains all available types of rules, including Firewall, Access Control List and IP Chain with respectively 5, 5 and 2 seeds.
2) D2 (random rulesets): rules are randomly generated associated with different types of actions (e.g., modification, forwarding and miscellaneous actions).

Each rule in D1 is associated with one action, while in D2, each rule contains a sophisticate action list with different types of actions. Its action list can generate multiple $tuples$, which include both PPs (Packet port tuples) and statistics. Note that the detection accuracy of PMM is guaranteed by the theories in Section IV. Thus we focus on its overhead in terms of time for each step: rule transformation, minimal interval set generation and misconfiguration checking. We also evaluate the benefit of our optimisation.

[3]ClassBench has been widely used to evaluate SDNs mechanisms. Furthermore, it can generate complex rules to evaluate our approach.

B. Experimental Results

We first evaluate the overhead of rule transformation that converts rule match patterns into a set of intervals. To this end, we select different sizes (from 100 to 5K) of synthetic rules from D1 and random rules from D2 respectively, and evaluate the rule transformation completion time. Figure 9 shows the results. The cost of the rule transformation in $D2$ is always larger than that in $D1$ when tackling the same scale of rules. This is because the match patterns in $D1$ are prefix-based while those in $D2$ are random. Consequently, the number of the generated intervals in $D2$ is much larger, which results in longer time for rule transformation. Nevertheless, the time overhead is reasonable as for 5,000 D2 rules, the transformation time is only 380ms.

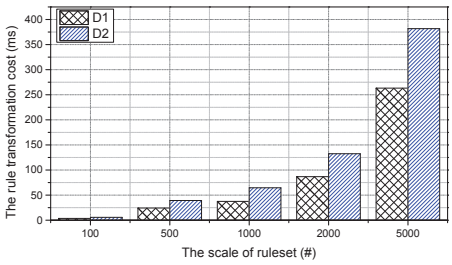

Fig. 9. The average cost of the rule transformation.

Next we measure the overhead introduced by the minimal interval set generation. Likewise, we use the synthetic rules and random rules to generate rulesets of different sizes, ranging from 100 to 5,000. We generate the minimal interval sets for each ruleset based on the intervals of each rule's match patterns. Figure 10 reports the time cost. Though the cost of the set generation in $D2$ is also larger than that $D1$, their performance gap is very small. For 5,000 D2 rules, the cost is only 36ms, showing the efficiency of the set generation.

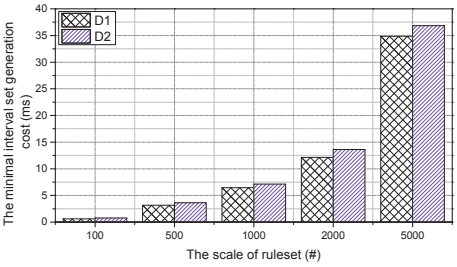

Fig. 10. The average cost of the minimal interval set generation.

Third, we evaluate the overhead introduced by the misconfiguration checking. We vary the sizes of rules, which are from $D1$ and $D2$ respectively. Note that the checking process will stop once it detects a misconfiguration. In this set of experiments, each rule associates with one different action for simplifying the action computation (the action computation overhead is examined in the next set of experiments). Figure 11 report this overhead, which is indeed very low (less then 15ms for 5,000 rules). This is because the minimal interval set representation enables fast checking.

We then evaluate the overhead of the equivalence computation for action lists. To this end, we select different sizes (from

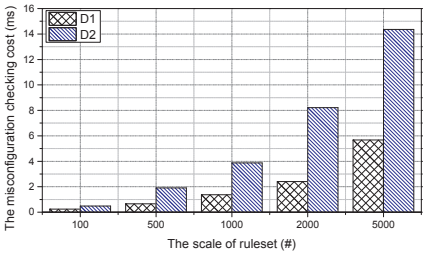

Fig. 11. The average cost of the misconfiguration checking.

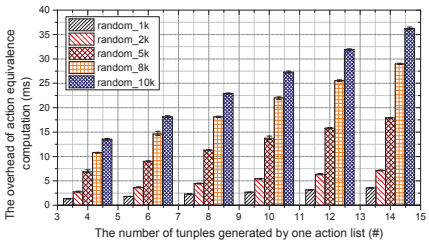

Fig. 12. Performance of the action equivalence computation.

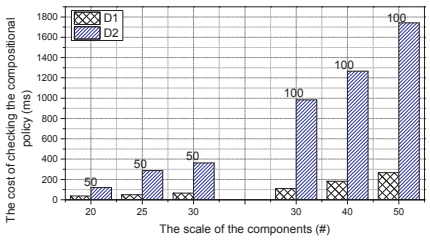

Fig. 13. The average overhead of the rule transformation. The top of the histograms represent the number of rules in each component.

1,000 to 10,000) of random rules from D2 and vary the number of $tuples$ (from 4 to 14) generated by each rule action list. Then we record the overhead due to the action equivalence computation for different sizes of rules under various $tuple$ configurations. Figure 12 reports the results. As expected, a larger number of $tuples$ introduces longer computation time. But even the number of $tuples$ reaches 14, the computation overhead for 10,000 rulesets is within 36ms.

Next we consider the overhead of checking the compositional policies with two experiments. Note that we assume that each component policy is misconfiguration-free. For the first set of experiments, we build a compositional policy with four components: $Monitor >> [Route, Load - balance] \triangleright Elephant\ \ flow$, which is very common in compositional SDN [17]. Each component is padding with several rules from $D1$. We run 5 rounds of experiments and the average time cost is only 2.45ms

In the second set of experiments, we evaluate the cost for checking compositional policies with dozens of components. Each component contains 50 or 100 rules from $D1$ and $D2$. We select randomly the three types of compositional operators. Figure 13 shows the overhead of misconfiguration checking for compositional policies. The checking overhead for composition of rules from $D2$ is much larger than that for rules from $D1$. This is because rules in $D2$ generate more intervals. Nevertheless, checking 50 compositional components requires less than 2 seconds.

Finally, we measure the benefit of the optimisation solution. Specifically, we randomly select 1,000 rules from $D2$, and count the number of generated intervals. We find that applying the optimisation can reduce the number of intervals by as many as 80%, which in turn greatly saves the time overhead for misconfiguration checking.

In summary, as our theoretical analysis has proved the detection accuracy and coverage, we conclude PMM is practical and efficient in misconfiguration checking for SDN policies.

VII. Related Work

SDN controllers such as NOX [14], Beacon [1] and Floodlight [12] provide low-level interfaces to write bitmask-based rules with complex action lists. High-level programming languages (e.g., Frenetic [12] and Pyretic [28]) and compositional hypervisor (e.g., CoVisor [17] and FlowBricks [11]) enable multiple policies manage the network cooperatively. However, both low-level APIs and high-level languages may introduce potential anomaly because of misconfigurations or error-updates.

Anomaly detection in access control polices have been studied in [6], [15], [36]. Some works [16], [5] present a set of algorithms to discover pairwise misconfiguration in Firewall. However, these work do not support bitmask-based rule detection, and the associated action is relatively simple.

Nice [8] adopts symbolic execution and model checking to explore the state space of an OpenFlow network. HSA [20] statically checks network configurations to verify network properties (e.g. reachability and forwarding loops). Nevertheless, it does not target at misconfiguration checking for compositional SDN, and thus cannot apply to this context. Veriflow [21] is able to verify network-wide invariant violations in real time. It considers the entire network, but cannot detect the misconfigurations that only affect the functions of one policy. The authors in [32] introduce a method to generate anomaly-free compositional SDN policies, but it does not support update.

VIII. Conclusion

SDN policies use bitmask-based match patterns and complex action behaviors to enable network administrators to build network applications. However, these features come at a cost: SDN programming are prone to error, regardless using low-level APIs or high-level programming languages. In this paper, we propose to represent rulesets using minimal interval sets for fast and accurate misconfiguration checking. We develop compositional algebra and misconfiguration checking theory using this representation. We further design a miconfiguration tool based on the algebra and theories. Experiments using the implementation in CoVisor with synthetic rulesets and random rulesets demonstrate the efficiency of our tool in checking misconfigurations for SDNs.

Acknowledgments

We thank our shepherd Diman Zad Tootaghaj and the anonymous reviewers for their insightful feedback. This work is supported in part by National Key R&D Program of China No.2019YFB1802800, the National Science Fund of China (Grant NO. 61725206) and the Youth Innovation Promotion Association CAS. Corresponding author: Zhenyu Li.

REFERENCES

[1] Beacon. http://www.beaconcontroller.net.

[2] Opendaylight. http://www.opendaylight.org/.

[3] Openflow specification. https://www.opennetworking.org/.

[4] The rules set of evaluation packet classification. http://www.arl.wustl.edu/hs1/PClassEval.html.

[5] Pedro Adão, Claudio Bozzato, G Dei Rossi, Riccardo Focardi, and Flaminia L Luccio. Mignis: A semantic based tool for firewall configuration. In *Computer Security Foundations Symposium (CSF), 2014 IEEE 27th*, pages 351–365. IEEE, 2014.

[6] Ehab S Al-Shaer and Hazem H Hamed. Discovery of policy anomalies in distributed firewalls. In *Ieee Infocom 2004*, volume 4, pages 2605–2616. IEEE, 2004.

[7] Samaresh Bera, Sudip Misra, and Abbas Jamalipour. Flowstat: Adaptive flow-rule placement for per-flow statistics in sdn. *IEEE Journal on Selected Areas in Communications*, 37(3):530–539, 2019.

[8] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A {NICE} way to test openflow applications. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 127–140, 2012.

[9] Bernard Chazelle. Lower bounds for orthogonal range searching: Part ii. the arithmetic model. *J. ACM*, 37(3):439–463, July 1990.

[10] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000.

[11] Advait Dixit, Kirill Kogan, and Patrick Eugster. Composing heterogeneous sdn controllers with flowbricks. In *2014 IEEE 22nd International Conference on Network Protocols*, pages 287–292. IEEE, 2014.

[12] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ICFP*, 2011.

[13] Franck Le Yeon-sup Lim Geng Li, Y. Richard Yang and Junqi Wang. Update algebra: Toward continuous, non-blocking composition of network updates in sdn. In *IEEE INFOCOM*, 2019.

[14] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[15] Hongxin Hu, Gail-Joon Ahn, and Ketan Kulkarni. Detecting and resolving firewall policy anomalies. *IEEE Transactions on dependable and secure computing*, 9(3):318–331, 2012.

[16] Rasool Jalili and Mohsen Rezvani. Specification and verification of security policies in firewalls. In *Eurasian Conference on Information and Communication Technology*, pages 154–163. Springer, 2002.

[17] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. Covisor: A compositional hypervisor for software-defined networks. In *NSDI*, volume 15, pages 87–101, 2015.

[18] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the Symposium on SDN Research*, page 6. ACM, 2016.

[19] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 49–54. ACM, 2013.

[20] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, volume 12, pages 113–126, 2012.

[21] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 15–27, 2013.

[22] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, 2010.

[23] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[24] Geng Li, Y Richard Yang, Franck Le, Yeon-sup Lim, and Junqi Wang. Update algebra: Toward continuous, non-blocking composition of network updates in sdn. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1081–1089. IEEE, 2019.

[25] Nick McKeown. Software-defined networking. *INFOCOM keynote talk*, 2009.

[26] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2), 2008.

[27] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. 2012.

[28] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. Composing software defined networks. In *NSDI*, 2013.

[29] Bruno Astuto A Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, 16(3):1617–1634, 2014.

[30] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. Pga: Using graphs to express and automatically reconcile network policies. In *ACM SIGCOMM*, 2015.

[31] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. *ACM SIGCOMM Computer Communication Review*, 42(4):323–334, 2012.

[32] Mohsen Rezvani, Aleksandar Ignjatovic, Maurice Pagnucco, and Sanjay Jha. Anomaly-free policy composition in software-defined networks. In *IFIP Networking Conference (IFIP Networking) and Workshops, 2016*, pages 28–36. IEEE, 2016.

[33] Ali Kheradmand P. Brighten Godfrey Santhosh Prabhu, Kuan-Yen Chou and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In *NSDI*, 2020.

[34] Stefano Vissicchio, Laurent Vanbever, Luca Cittadini, Geoffrey G Xie, and Olivier Bonaventure. Safe update of hybrid sdn networks. *IEEE/ACM Transactions on Networking (TON)*, 25(3):1649–1662, 2017.

[35] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying sdn programming using algorithmic policies. In *ACM SIGCOMM*, pages 87–98. ACM, 2013.

[36] Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su, and Prasant Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 15–pp. IEEE, 2006.