Kano: Efficient Cloud Native Network Policy Verification

Yifan Li, Xiaohe Hu, Chengjun Jia, *Graduate Student Member, IEEE*, Kai Wang, and Jun Li

***Abstract*—Cloud-native computing has become a prevailing paradigm with lightweight runtime-level isolation and fast delivery for scalable applications. Cloud-native network policies (CNNPs) are used to realize network isolation with respect to security and availability. Due to the dynamic environment, CNNPs are label-based instead of IP-based and take the form of attribute-based access control (ABAC) to obtain good expressivity. To ensure the correctness of network isolation, CNNP verification is an essential but challenging problem given the large scale and frequent updates of cloud-native environments and the operation automation demand. Thus, we design Kano, an efficient, *i.e.,* easy-to-use and fast-to-execute, system for verifying large scale CNNPs at runtime. Kano is operation-friendly, with a proposed intent-based verification language. A bit matrix model with a prefiltration algorithm and a partial-update method is proposed to support fast complete and incremental verification. Kano also generates fix plans for violations to assist operators. Kano is implemented as a CNNP verification system that is used in ABAC cloud-native platforms and is integrated into the popular Kubernetes orchestrator. An evaluation on a large scale network of 100k nodes and about 68k policies shows the efficiency of Kano, with 12.51 seconds for all reachable invariant verification and 0.299 milliseconds for policy addition verification.**

***Index Terms*—Cloud native, network verification, attribute-based access control, overlay networks.**

I. Introduction

CLOUD-NATIVE computing is a prevailing paradigm for building and running scalable applications in modern dynamic clouds [1]. The base virtualization techniques used in cloud-native environments include virtual machine (VM) and container, providing lightweight system-level and runtime-level isolation. Orchestration solutions, *e.g.,* OpenStack [2], Mesos [3] and Kubernetes [4], have been adopted to manage and schedule VMs and containers. Cloud-native applications are segmented into microservices to enable rapid and frequent delivery. New techniques are continually adopted by the cloud-native paradigm, and the persistent design principle is to realize resilient, manageable and observable systems that facilitate application agility in large-scale and dynamic clouds.

Network isolation is an essential pillar of the cloud-native paradigm with respect to availability and security. For example, a user in a containerized cloud may ask the question "Are my containers properly isolated from other users?" A developer may ask "Can my codevelopers access my VMs?" *Cloud-native network policies (CNNPs) define the connectivity between nodes, i.e., containers, VMs or other infrastructure units in which applications and network functions run, ensuring network segmentation*. Considering the ephemeral nature of IP addresses and the loosely coupled relationship between applications and hosts in modern dynamic clouds, *CNNPs choose to describe match fields according to labels that are key-value-pair attributes attached to nodes by the orchestrator* [5], [6]. A label-based CNNP is a form of attribute-based access control (ABAC) [7] that has been proven to have good expressivity and easy maintenance compared to classical solutions, such as the 5-tuple rules. CNNPs are a type of application-friendly control plane semantics enforced on the overlay networks. On the other hand, the underlay network is assumed to be fault-free and no abnormal blocks are caused by it, which is guaranteed by abundant existing verification work [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37].

To ensure that the CNNPs are working properly and to increase the operation correctness in fluid cloud-native environments, CNNP verification is necessary. *CNNP verification checks the reachability implication of the aggregated policies against network invariants*. We use a simplified view of a cloud-native cluster shown in Fig. 1, and a set of ABAC policies shown in Table I as an example to illustrate CNNP verification. Cloud-native clusters often have many users and our example is no exception. There are two users Alice and Bob in the example. The owner and role are marked on each node. Consider the following two anomalies:

- *Policy shadow:* Consider CNNPs 0, 1 and 2. CNNP 1 allows Bob-tomcat to access Alice-Redis and CNNP 2 allows Bob-tomcat to access Alice-MySQL. However, CNNP 0 allows Bob-tomcat to access both. Therefore, CNNPs 1 and 2 are shadowed by CNNP 0. This anomaly yields redundant policies, and makes policies difficult to manage.

Manuscript received 2 June 2022; revised 29 September 2022 and 8 December 2022; accepted 13 December 2022. Date of publication 16 December 2022; date of current version 9 October 2023. This work is supported by the National Natural Science Foundation (No. 61872212) and National Key Research and Development Program (No. 2016YFB1000101). The associate editor coordinating the review of this article and approving it for publication was S. Scott-Hayward. *(Corresponding author: Yifan Li.)*

Yifan Li, Chengjun Jia, and Jun Li are with the Department of Automation, Tsinghua University, Beijing 100084, China (e-mail: liyifan18@mails.tsinghua.edu.cn; jcj18@mails.tsinghua.edu.cn; junl@tsinghua.edu.cn).

Xiaohe Hu is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: hxhe@mail.tsinghua.edu.cn).

Kai Wang is with Yunshan Networks, Beijing 100084, China (e-mail: wangkai@yunshan.net).

Digital Object Identifier 10.1109/TNSM.2022.3229675

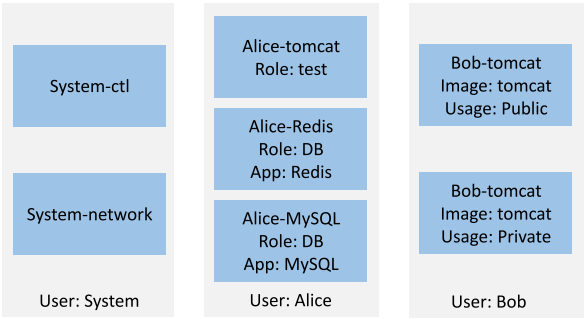

Fig. 1. A cluster in cloud-native infrastructure.

TABLE I
CLOUD-NATIVE NETWORK POLICIES (CNNPS)

ID	Src node	Dst node	Action
0	User=Bob, Image=tomcat	User=Alice, Role=DB	Allow
1	User=Bob, Image=tomcat	User=Alice, Role=DB, App=Redis	Allow
2	User=Bob, Image=tomcat	User=Alice, Role=DB, App=MySQL	Allow
3	User=Alice, Role=test	User=Bob, Role=DB	Allow
4	User=System	User=Alice	Allow
5	User=System	User=Bob	Allow
6	Default	Default	Block

- *Policy Irrelevance:* No node in Fig. 1 meets the destination node requirement of CNNP 3. Therefore, CNNP 3 is useless. This anomaly, which typically occurs when nodes are removed but policies are forgotten and remain, causes security issues when new nodes with the same labels but different roles and functions are created.

Unfortunately, verification studies on traditional networks are not suitable for CNNPs. Network verification research can be categorized into two main types: (1) data plane verification, which checks the correctness of a snapshot of data plane forwarding configurations [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22] or the correctness of the network functions running in the data plane [23], [24], [25], [26], [27], [28], [29], [30], [31], and (2) control plane verification, which checks the correctness of control plane configurations typically used as the inputs of routing protocols, such as open shortest path first (OSPF) and border gateway protocol (BGP) [9], [32], [33], [34], [35], [36], [37]. Although CNNP verification has the same goal as previous data plane forwarding verification, *i.e.,* reachability, the number and length of CNNP labels can be arbitrary, which means the data plane verification works designed for packet header fields with limited field number and fixed field length cannot be used to verify CNNPs. Meanwhile, CNNPs are within the control plane scope and are different from routing protocol configurations. *CNNPs define end-to-end node connectivity in an ABAC manner, the verification of which is a new but not trivial topic of control plane verification*. The cloud-native paradigm brings the following challenges to CNNP verification:

- *Large Scale:* The large scale of cloud-native clusters implies prohibitive complexity. According to the Google Kubernetes Engine documentation, the scale of a cluster can reach 100k nodes [38]. In addition, cloud-native networks must be verified at the node level, *i.e.,* runtime level, instead of at the router level or host level like most existing verification methods.
- *Frequent Updates:* CNNPs and labels of nodes can be added, deleted, or migrated by any cluster user according to application changes. Consequently, instead of a single offline verification, CNNPs must be verified periodically at runtime. These frequent and periodic updates significantly shorten the lifespan of the verification results.
- *Automatic Operation:* The cloud-native paradigm emphasizes agility and automation, such as continuous integration, continuous delivery, and applications using declarative programming rather than imperative. Hence, the CNNP verification should provide concise declarative interfaces to simplify the whole process, from information construction to result feedback. Moreover, advice is needed to assist operators in fixing sophisticated errors. Meanwhile, to maintain readability and manageability, operators prefer to use existing labels rather than new labels to adjust CNNPs, in which finding the minimal set is a hitting set problem, which is NP-complete.

In this paper, *we design and implement Kano, an efficient, i.e., easy-to-use and fast-to-execute, system for verifying large-scale CNNPs at runtime*. Kano was initially presented as the first preliminary system to cover container network policy verification [39]. We improve Kano to address a more general issue, *i.e.,* ABAC-formed CNNP verification, and provide an exhaustive declarative operation expression, more detailed algorithm designs, and more sufficient evaluation results.

To address the aforementioned challenges, our main ideas are as follows. With respect to the scalability challenge, instead of the data-plane-specific model or formal verification models used by other approaches, Kano uses the bit matrix as an internal data structure to model and represent reachability, maintaining a relatively constant and fixed verification time and space cost. Then, Kano takes advantage of the sparse reachability relationship between nodes to reduce the performance overhead of verification. Furthermore, Kano leverages a prefiltration algorithm to reduce the time complexity of the all-pair reachability calculation from the $O(n^2)$ of the naive algorithm to $O(n)$.

With respect to the update challenge, Kano supports incremental verification by examining only the parts of the network affected by configuration changes. A data structure is designed to correlate nodes and CNNPs, with which Kano can quickly identify the portion of the update-affected network. To address the operation automation challenge, Kano provides the intent-based Language for ABAC Verification Intents (LAVI) for network operators to express verification intentions. LAVI supports two primitives: *check* and *fix* corresponding to complete/incremental verification and fix plan generation. Moreover, Kano generates a fix plan to repair violated connectivity, including both adding links and deleting links, by leveraging a try-check-and-patch strategy and a greedy policy aggregation algorithm.

The following summarizes our main contributions:

- *Operation-friendly CNNP verification language:* We propose a high-level intent-based operation language, LAVI, to facilitate the use of the Kano verification system. The operations evolve from imperative to declarative, facilitating the automation of the cloud-native paradigm.
- *Fast complete and incremental CNNP verification:* We model the CNNP verification problem with a bit-matrix data structure, on which a sparsity utilization method, a prefiltration algorithm and a partially updated method are proposed to accelerate the complete and incremental verification.
- *Fix plan generation for verification violation:* We propose a try-check-and-patch method to repair connectivity that leads to anomalies. Our greedy fix plan updates CNNPs with as few CNNP and label changes as possible.
- *Integration into cloud-native orchestrator and scalability evaluation:* We implement Kano, which can be used as a library for any cloud-native platform and integrate it into Kubernetes to work automatically on the Kubernetes controller. The evaluation with large-scale cloud-native settings shows the efficiency of Kano, with 12.51 seconds for the verification of all pairs reachability and 2.99 milliseconds for 10-CNNP addition verification on a network of 100k nodes and approximately 68k CNNPs.

The rest of this paper is organized as follows. Section II explains the CNNP verification problem and summarizes and analyzes previous related work. Section III proposes the Kano design, including a workflow overview, an intent-based expression language and the CNNP-specific network model. Section IV elaborates the algorithms for calculating the reachability matrix and carrying out network verification. Section V shows the details of the incremental verification. Section VI explains how Kano provides fix plans for violations. The implementation of Kano is presented in Section VII. Section VIII shows the evaluation results of Kano. Section IX studies two cases extracted from real industrial problems. Section X concludes this paper.

II. Background and Related Work

In this section, the notation used in CNNP verification is first described to ensure consistent terminology throughout the paper. Then, the previous network verification work is reviewed. Finally, related operation automation work is introduced.

A. CNNP Verification Problem

With respect to the cloud-native paradigm, we use the term *node* to describe a virtualized unit where applications or network functions run. Each node has some attribute labels attached and expressed by key-value pairs, as shown in Def. 1.

Definition 1: Node: *List*[(*k*, *v*)]. *k* and *v* are strings.

As shown in the example in Fig. 2, the *ABAC-formed CNNP* is composed of four parts:

- *Source part:* The source part uses labels to identify which node a CNNP is applied to. A CNNP takes effect on only the nodes that meet all label restrictions; *i.e.,* all keys appearing in the source part of the CNNP appear in the source nodes, and the corresponding values of the keys also match.
- *Destination part:* The destination part uses labels to describe which nodes are allowed to be accessed. The identification mechanism is the same as that of the source part.
- *Action:* As Kano focuses on the reachability between nodes, actions in CNNPs contain *allow* and *block*, which indicate whether a connection will be established when the request matches a CNNP.
- *priority:* If a connection request is matched by multiple CNNPs, only the CNNP with the highest priority takes effect.

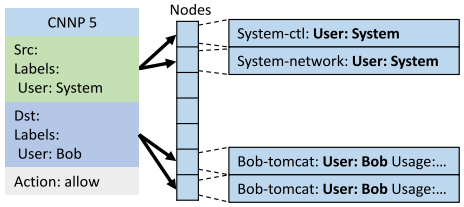

Fig. 2. A CNNP example.

Accordingly, the definition of a CNNP is shown as Def. 2.

Definition 2: CNNP: a tuple (*Src*, *Dst*, *Act*, *Prio*). *Src* and *Dst* are nodes. $Act \in (Allow, Block)$. *Prio* is an integer.

As in traditional firewall policies, when there are a large number of CNNPs, the possibility of writing conflicting or redundant rules is relatively high. In manually managed private clouds, although virtual nodes change frequently, their corresponding CNNPs seldom change unless necessary due to risk aversion. As a result, the shadowed and useless CNNPs are silently forgotten and may cause unexpected network errors when new nodes are created. Therefore, the first step to manage cloud network policies with Kano is to find the following anomalies.

- *Policy shadow:* Suppose there are two CNNPs P_a and P_b. The priority of P_a is higher than P_b. If the source and destination nodes of P_b are both subsets of the source and destination nodes of P_a, P_a shadows P_b and P_b is redundant.
- *Policy Irrelevance:* Suppose there is a CNNP P_a whose source part or destination part does not match any node, then P_a is redundant.

Meanwhile, in addition to policy-oriented anomalies, due to the multi-user application scenario of CNNPs, there are other anomalies caused by security concerns.

- *User cross:* In a multi-user cloud, different users' nodes are supposed to be isolated by default. A CNNP that allows a node to access other users' nodes may cause security issues.
- *System isolation:* There are system nodes in clouds. These system nodes provide infrastructure services such as network management or heartbeat to user nodes. Therefore, CNNPs should allow system nodes to access all nodes; otherwise, some services or functionalities may not work.

Formally, the definition of the anomaly is as follows.

Definition 3: Anomaly: $A(S,N) \in True, False$, $S = \cup CNNP_i, i = 1,2,\ldots,n, N = \cup node_i, i = 1,2,\ldots,n$.

Anomalies should be found and fixed when they occur. Otherwise, serious risks arise, such as service outages, data leakages or attacker invasions. Verification is the technology that finds anomalies and raises alerts to the network operator. The definition of the CNNP verification is as follows.

Definition 4: CNNP verification: given a set of CNNPs $S = \cup CNNP_i, i = 1,2,\ldots,n$ and nodes $N = \cup node_j, j = 1,2,\ldots,n$, check if the anomaly assertions $A_j(S,N), j = 1,2,\ldots,m$ are *True*.

B. Network Verification Approaches

As the network scale and configuration complexity substantially increase, ensuring the correctness of networks becomes a difficult task that must be addressed. Without a doubt, network verification has been a popular research topic in the past decade. We present a review of previous network verification approaches from three areas: data plane forwarding verification, data plane function verification, and control plane verification.

Data plane forwarding verification: These approaches [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22] check a snapshot of data plane forwarding configurations against network invariants. Anteater [10] models network data plane forwarding as a Boolean satisfiability problem (SAT) and uses an SAT solver to find counterexamples when violations are identified. Anteater finds forwarding loops and stale access control list (ACL) rules in a university network. Header space analysis (HSA) [11] is a network-specific protocol-agnostic model with header space and transfer functions. HSA solves the verification problem with set operations on rulesets and graph algorithms. Reachability analysis, loop detection, and slice isolation can be conducted with HSA. Netplumber [12] supports incremental verification for HSA. The Network-optimized Datalog (NoD) [15] models data plane forwarding using optimized Datalog to support network dynamics, such as protocol changes.

The following approaches improve the efficiency of data plane forwarding verification from different angles or in different scenarios. VeriFlow [13] refers to an equivalence class as a set of packets that are processed with identical forwarding actions. The equivalence class processing reduces the state space of the HSA model. AP Verifier [14] divides the original rulesets into nonoverlapping subsets as atomic predicates represented by bit vectors, accelerating set operations on rulesets in HSA processing. Delta-net [18] and APKeep [22] further improve AP Verifier by exploiting the similarity among packet behaviors through parts of the network and maintaining only the necessary equivalence classes.

Libra [16] verifies the forwarding tables of large-scale cloud data center networks using MapReduce to perform parallel processing on subnets. Plotkin et al. [17] exploit the surgical (slicing) and symmetrical (backup) properties of cloud data center networks and scale the verification using simplified networks. They also prove that the simplified transformed formula is equivalent to the original formula. Jayaraman et al. [21] describe their experiences deploying NoD in cloud data centers and propose an invariant decomposition technique to improve scalability. Cloud Radar [19] incrementally checks the cloud virtual machine (VM) isolation. TenantGuard [20] scales cloud VM reachability verification by leveraging the hierarchical structure and proposes efficient IP-prefix-related data structures.

Similar to the goal of data plane forwarding verification, CNNP verification checks the reachability between cloud-native nodes and analyzes anomaly cases, such as policy shadow and policy irrelevance. While different from the IP-based data plane forwarding rules, a CNNP is a high-level expression with ABAC semantics. The HSA model and formal verification models, such as SAT and NoD, are not suitable for CNNP verification. Kano proposes an efficient model and solves the scalability problem with customized algorithms.

Data plane function verification: These approaches [23], [24], [25], [26], [27] focus on the correctness of stateful network functions. Dobrescu and Argyraki [23] propose a symbolic execution tool with structure decomposition to check whether the data plane software satisfies certain properties, such as collisionless bounded execution and filtering. SymNet [24] models the data plane with a symbolic-execution friendly language that takes explicit execution paths. It uses a map data structure instead of real code with scalability issues or an HSA model with poor expressiveness. The verifying mutable networks (VMN) [25] method models stateful functions as a forwarding model, a set of abstract packet classes and a set of packet classification oracles and scales the verification by the same slicing as Plotkin et al.'s proposed approach [17]. Dumitrescu et al. [26] check the equivalence of two data planes based on SymNet. Yousefi et al. [27] use compact Boolean formulas to check the liveness property of stateful functions, *i.e.,* some status that eventually occurs.

The approaches in [28], [29], [30] target the correctness of programmable network functions that are expressed by P4 [40]. Vera [28] verifies P4 programs using SymNet and improves scalability with a match-action optimized data structure. Liu et al. [29] propose a verification tool, p4v, that expresses symbolic control-plane interfaces to define program constraints and leverages optimized symbolic techniques to enable scalability. Dumitrescu et al. [30] propose bf4 to free programmers from the specification process in P4 program verification by finding all possible bugs and presenting the corresponding mitigation predicates.

Data plane function verification is similar to software verification using methods such as symbolic execution. In contrast, Kano tackles the CNNP verification problem with a network-specific model.

Control plane verification: These approaches [9], [32], [33], [34], [35], [36], [37] check control plane configurations against network invariants. A control plane can generate multiple data planes according to network resources such as topology changes and route advertisements. Batfish [9] verifies a general control plane by transforming it into concrete data planes and using NoD to check the data planes. Abstract Representation for Control planes (ARC) [32] directly verifies the control plane routing protocol under link failures using a

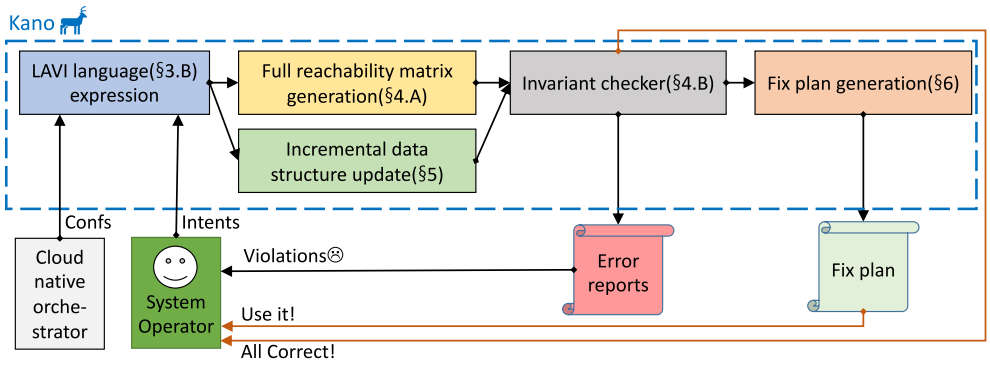

Fig. 3. Verification workflow overview of Kano and its operator and orchestrator interactions.

weighted direct graph model. Efficient Reachability Analysis (ERA) [33] models the control plane routing protocols using route and visibility functions similar to the header space and transfer functions in the HSA model. Minesweeper [34] improves the semantic expressiveness and coverage of control plane verification through the formal verification method of Satisfiability Modulo Theories (SMT). Beckett et al. [35] compresses a large network into smaller networks with control plane properties preserved to accelerate approaches such as Batfish and Minesweeper. The Back-Forward Algorithm (BFA) [36] improves ARC to support incremental updates. Campion [37] proposes a modular approach to debug control plane differences and identify the relevant configuration lines.

CNNPs are application-oriented and defined in the control plane. The Batfish and SMT approaches can be used to check CNNPs, while the scalability and frequent update challenges are hard to address.

C. Operation Automation

Network management has become an urgent problem in the cloud era, and self-driving networks with automatic life cycle management have attracted considerable attention [41]. Big cloud data center operators [21], [42], [43], [44] have presented their practices and experiences in automatic cloud network management. Their design principles include minimizing direct human interaction with network devices and providing high-level intent expression for operators that can be translated and deployed safely.

Certainly, verification is an essential part of the network management life cycle. To improve the usability of the verification process, NoD [15] proposes a general specification language to express data plane forwarding invariants, and the following work [21] automatically derives invariants from architectural metadata. Jinjing [45] designs an intent language that aids operators in automatically and safely updating the WAN ACL. Aquila [31] proposes a high-level language to facilitate specification expression of programmable data plane verification. Furthermore, control plane repair (CPR) [46] and Jinjing [45] generate fixing plans for violations of control plane routing and ACL update verification to alleviate the manual workload of operators. With respect to operation automation, Kano is akin to the above works with an intent-based language and fix advice for CNNP verification.

III. KANO DESIGN

This section first presents an overview of the design of Kano. Then, the intent-based language LAVI is elaborated. Finally, the CNNP-specific network model is described. The following sections describe the verification algorithms in detail.

A. Kano Overview

Kano runs as an independent verification system that interacts with the cloud-native orchestrator and the system operator. Kano automatically reads and analyzes configurations including nodes and CNNPs with label descriptions from the orchestrator. The operator expresses verification intents using the LAVI language provided by Kano. Configurations related to nodes and CNNPs are also input by LAVI.

According to different intents of the operator, Kano generates or updates the reachability matrix and other auxiliary data structures used to model the CNNP-specific network. Then, the anomalies are checked based on the generated model. Next, the operator receives an all-correct notification if the verification passes or an error report about the violations. Moreover, Kano supports fix plan generation using LAVI intent when violations occur. The system overview of Kano is shown in Fig. 3.

B. Intent Language: LAVI

With the cloud-native demand of continuous integration, continuous delivery, and declarative programming, we find it rational to define a high-level intent-based language. Therefore, the network operator can customize the verification scale and the anomalies to check. Moreover, when nodes and CNNPs in a cloud network change, the network operator can quickly check the changed cloud network by slightly modifying the verification program. Kano provides operators the high-level intent-based language LAVI to perform CNNP verification, simplifying the complex process.

From now on, when we mention policy, it means CNNP.

Fig. 4 shows LAVI's grammar, which includes the global variables *node*, *policy*, *link* and *unlink*, predefined *anomaly* and *property*, *configuration operation* and *primitive*. To provide sufficient expressivity, LAVI should allow the operator to define the verification scale and the verification tasks.

prog	::=	*target; conf; prim;*	LAVI program
target	::=	**Node**(*[key : value]) node_name*	Defining nodes
	|	**Policy**(*[key : value], [key : value], act, prio) policy_name*	Defining policies
	|	**add**(*node*|*policy*)	Add objects to network
	|	**del**(*node*|*policy*)	Del objects from network
conf	::=	**set**(*node*|*key, prop*)	Set properties
	|	**link**(*node, node*)	Defining links
	|	**unlink**(*node, node*)	Defining unlinks
prim	::=	**check**(*anomaly*|*link*|*unlink* *Public*|*Private*)	Check anomalies, links and node properties. If No reachability matrix, generate it.
	|	**fix**	Generate fixing plan
prop	::=	*Dense*|*User*	Label Properties
	|	*System*|*Public*|*Private*	Node Properties
anomaly	::=	*UserCross*	Anomalies to check
	|	*SystemIsolation*	
	|	*PolicyShadow*	
	|	*PolicyIrrelevance*	

Fig. 4. Intent language LAVI's programming.

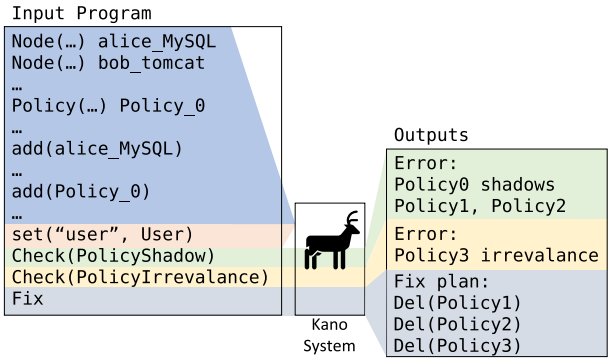

Fig. 5. A running example with LAVI.

TABLE II
COMMANDS FOR COMMON VERIFICATION TASKS

Verification tasks	Configuration operations	Primitives
complete verification	*set*, *add*	*check*
incremental verification	*set*, *add*, *del*	*check*
fix plan generation	*set*, *add*	*check*, *fix*

Additionally, considering the industrial reality, LAVI should allow network operators to specify environment configurations, such as how to confirm the owner of a given node. Therefore, LAVI must be able to express 3 components: *target*, *conf* and *prim*, as shown in the first line of Fig. 4, which is described as follows:

- *Target:* To express a verification task, the operator should first define the verification target. Typically, a verification target in a cloud-native network contains nodes and policies. The target can be declared by the operator or loaded from the orchestrator by Kano. The operator uses *node* and *policy* to define instances and then uses operations *add* or *del* to add instances to the target or remove instances from it.
- *Configuration Operation (conf):* We design *conf* to express the features that are difficult to express by labels as a complement of *target*. The operation *set* helps the operator attach properties to nodes and keys to express verification requirements, *e.g.,* which nodes are system nodes that should be allowed to access all other nodes and which key is used to distinguish the owner of nodes. Operations *link* and *unlink* are instances for zooming in to the reachability between a pair of nodes. According to Def. 3, whether a set of *links/unlinks* are satisfied can also be seen as an anomaly.
- *Primitive (prim):* LAVI provides the primitives *check* and *fix* to specify the verification operation under the given target and configurations. Some network operators prefer to write policies manually due to the concern of readability and manageability. Dividing the check and fix primitive gives them the freedom of using Kano to find bugs without the time cost of fix plan generation. Kano can *check* the predefined anomalies or user-defined links/unlinks and record missing or redundant links. If this is the first time *check* appears in a LAVI program, Kano generates the reachability matrix first and uses the reachability matrix to check the anomalies or links (complete verification). Then, the *check* primitive updates the data structures incrementally and checks the anomalies (incremental verification). When violations occur, the operator can use the *fix* primitive to generate fix plans that include attaching labels to nodes and adding or removing policies.
- *Property (prop):* Property provides optimizations and intentions that are hard to express by links. Some anomalies are also predefined as properties. With *prop*, Kano allows network operators to flexibly choose which anomalies to verify. The detailed properties and anomalies will be explained in Section VI.

We use the example in Fig. 1 and Table I to show how the operator uses Kano to find potential anomalies with LAVI. The operator first specifies the *target* that involves all the nodes and policies, which can be written manually or automatically generated by reading the orchestrator. The operator uses *set* to label the users with the property *User*, which means that in this cloud, the node owners are recorded by the value of the *User* label. The operator then uses the *check* primitive to find whether *PolicyShadow* and *PolicyIrrelevance* exist. Kano first calculates the reachability matrix and throws an error that *Policy 0* shadows *Policy 1* and *Policy 2*. After that, because the reachability matrix has already been generated, Kano can directly check *PolicyIrrelevance* and find *Policy 3* is irrelevant with all nodes. Finally the fix plan is output using the *fix* primitive. The overall LAVI program and outputs are shown in Fig. 5. This example illustrates how Kano handles the anomaly checking and fixing of the policy verification. The operator can express diverse ABAC-formed policy verification tasks by combining these commands, as shown in Table II.

C. CNNP-Specific Network Model

Kano models a CNNP-specific network as a bipartite graph. The edge set E represents the connections managed by the policy list L_{policy}. The bipartite graph model is expressed using a reachability bit matrix. If a bit is set/clear at row x and column y, it means that node x is allowed/blocked to reach node y, i.e., $e(x, y) = true/false$. An example reachability bit matrix

TABLE III
USEFUL SYMBOLS

Symbol	Type	Description
L_{node}	list	A list that contains all nodes.
L_{policy}	list	A list that contains all policies.
n_{node}	integer	The num of nodes.
n_{policy}	integer	The num of policies.
S_{key}	float	The average key sparsity.
S_{label}	float	The average label sparsity.
$e(x, y)$	bool	Edge between node x and y.
E	list	Edges allowed by L_{policy}.
R	bit matrix	The reachability bit matrix.
L	hashmap<str, bitvector>	Node label hash dict.
B	hashmap<str, bitvector>	The bidirectional node policy map (BNP map).

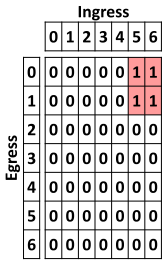

Fig. 6. A reachability bit matrix example.

using the example policy in Fig. 2 is shown in Fig. 6, in which $e(0, 5)$, $e(0, 6)$, $e(1, 5)$, $e(1, 6)$ are true. The advantages of the reachability bit matrix model are as follows:

- *Constant memory consumption:* The size of the reachability bit matrix is determined by the number of nodes. Thus, no additional memory allocation operations are required in the reachability bit matrix calculation, while additional memory allocation is necessary using a sparse matrix consisting of integer sets or strings.
- *Fast calculation:* The bit matrix calculation and the verification procedure can be performed via bitwise operations, which are much faster than integer and string calculations.
- *Efficient verification:* The policies and anomalies are translated into bit matrix expressions. The network anomalies can easily be found against the reachability bit matrix.

Moreover, basic definitions used in the verification procedure in this paper are introduced. Table III defines useful symbols. The items in Table III that are not intuitive are further explained as follows:

- *Sparsity* represents how sparse R is. A *pair* refers to a node and a policy. If all keys of the source or destination labels of a policy p can be found in a node n, we say n and p are a pair matched by keys. S_{key} represents the proportion of pairs matched by keys. If there are n_{key} pairs matched by keys, S_{key} is calculated with Equation (1).

$$S_{key} = n_{key}/(n_{node} * n_{policy}) \tag{1}$$

Like S_{key}, S_{label} represents the proportion of pairs matched by labels. A pair (n, p) matched by labels means all keys and values of the source or destination labels of p can be found in n. When there are n_{label} pairs matched by labels, S_{label} is calculated with Equation (2).

$$S_{label} = n_{label}/(n_{node} * n_{policy}) \tag{2}$$

S_{key} and S_{label} impact the time complexity of the full R calculation and incremental update. They are affected mainly by the number of node and policy labels and the number of keys and values of labels.

- B is the bidirectional node policy (BNP) map. It records which nodes are matched by a given policy and which policies match a given node. The BNP map is described in detail in Section IV-B.

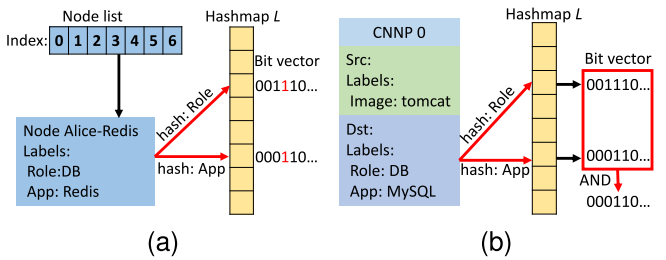

Fig. 7. Hash-based prefiltration. (a) Prefiltration of node labels. (b) Prefiltration of CNNP matching.

IV. COMPLETE CNNP VERIFICATION

This section explains how the reachability matrix R is generated and how anomalies are detected with R when using the primitive *check*. Then, the optimization and anomaly checks added by the *set* configuration operation are described.

A. Reachability Bit Matrix Calculation

All anomalies that Kano focuses on are reachability-related. Therefore Kano uses the reachability bit matrix R to perform verification and fixing. When a *check* primitive is called and R is not calculated and stored, Kano calculates R first. If R is calculated intuitively, L_{policy} is traversed. For each policy, we traverse every node to determine whether it is matched by the policy. The time complexity of this brute-force algorithm is $O(n_{node}n_{policy})$. In practice, n_{node} and n_{policy} usually have the same order of magnitude. As mentioned previously, the scale of a cloud-native cluster can reach 100k [38]. Therefore, the total compute complexity reaches 10 billion steps. Thus, the intuitive algorithm is inefficient, and a faster algorithm is needed.

Despite the large size of L_{node} and L_{policy} in a cluster, the keys and values of the labels are scattered; thus, R is usually sparse. The R calculation can be accelerated with a prefiltration process to drop the impossible matching conditions between nodes and policies. Therefore, the following **prefiltration algorithm** is proposed based on bit vector mapping:

- Hash the keys to the node label hash map L to associate every key to its bit vector. Each bit represents a node. Then, all the keys of nodes are mapped to these corresponding bit vectors so that the node indexes are fully populated in the bit vectors. As shown in Fig. 7(a), the labels "Role" and "App" are mapped to two different bit vectors. The node with index 3 has these two labels,

therefore the bits with index 3 in the corresponding bit vectors are set. As each node has very limited labels, the time complexity is $O(n_{node})$.

- Iterate the policies in ascending order of priority. Hash the source/destination keys of each policy to a bit vector in *L*; then, bitwise AND all resulting bit vectors to obtain the representation of nodes. As shown in Fig. 7(b), only nodes with index 3 and 4 have both label "Role" and "App", therefore in the prefiltration result bit vector only bits with index 3 and 4 are set. This means that only nodes 3 and 4 can be matched by this policy. The time complexity of this procedure is $O(n_{policy})$.
- Traverse all set bits of the bit vector. Determine whether the corresponding node matches the policy. This step is performed by comparing the strings in the labels. If a match is found, set/clear the corresponding bit in *R*. The time complexity of this procedure is $O(n_{policy} * n_{node} * S_{key})$.

The reason why Kano uses label keys instead of label key-value pairs as the hash key is the memory cost. The memory cost of using label key-value pairs is several times higher than that of using label keys, while the use of label keys as the hash key is already able to exclude most unrelated nodes. Therefore, the label key is the default hash key. Meanwhile, the label key-value pair hashing is kept as an option to deal with *Dense* labels which are widely used by nodes. A typical example is the *User* label which is attached to every node.

The pseudocode of the full *R* calculation algorithm is shown in Algorithm 1. The overall time complexity is $O(n_{node} + n_{policy} + n_{policy} * n_{node} * S_{key})$. When S_{key} is large and fixed, the overall time complexity is $O(n_{policy} * n_{node})$. However, in practice, as the cloud cluster scales, the number of nodes matched by each policy is relatively small, which means S_{key} decreases, and $O(n_{node} * S_{key}) \approx O(1)$. Therefore, the final overall time complexity is $O(n_{node} + n_{policy})$.

The space complexity of this algorithm is $O(n_{node}^2)$. Taking the largest mentioned scale [38] of 100k nodes as an example, the storage size for *R* can be calculated with Equation (3).

$$S = 100000 * 100000(bits) = 10Gb = 1.25GB \quad (3)$$

This is a reasonable memory cost for cloud-native clusters that can support 100k nodes.

Algorithm 1 *R* Calculation

Input: L_{node}, L_{policy}
Output: *R*

```
sort(L_policy, policy.prio)
for i = n_node − 1 to 0 do
    for label in L_node[i]. Labels do
        L.hash(label.key).set(i)
for i = n_policy − 1 to 0 do
    BitSet DstSet = new BitSet(1,n)
    for label in L_policy[i].dstLabels do
        DstSet = DstSet&&L.hash(label.key)
    BitSet SrcSet = new BitSet(1,n)
    for label in L_policy[i].srcLabels do
        SrcSet = SrcSet&&L.hash(label.key)
    for SetIdx in DstSet do
        if !L_policy[i].dst == L_node[SetIdx] then
            DstSet.clear(SetIdx)
    for SetIdx in SrcSet do
        if !L_policy[i].src == L_node[SetIdx] then
            SrcSet.clear(SetIdx)
    for SetIdx in DstSet do
        if L_policy[i].act == "allow" then
            R.getRow(SetIdx) | = SrcSet
        else if L_policy[i].act == "block" then
            R.getRow(SetIdx) &= ∼SrcSet
return R
```

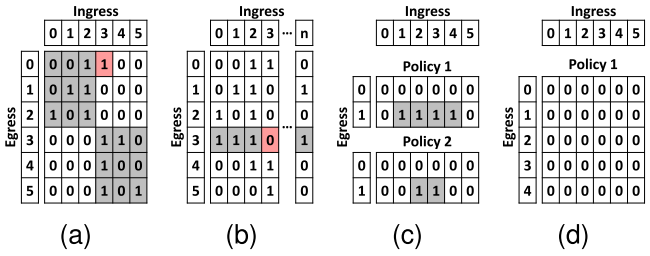

Fig. 8. Cloud-native network reachability bit matrix model. Each subfigure shows an anomaly checking algorithm applied to the bit matrix.(a) User cross. (b) System isolation. (c) Policy shadow. (d)Policy irrelevance.

B. Anomaly Checking

With *R*, all anomaly checking procedures listed in Section II-A are transferred into bit operations and therefore can be checked quickly. These procedures are described as follows.

- *User cross:* This anomaly can be found by comparing each row of *R* with the user hash bit vector, as shown in Fig. 8(a). Each bit of the user hash bit vector represents a node. Given a user, all bits belonging to the user are set, and the others are clear. The user hash bit vector is calculated by hashing the value of the user label of each node to associate every user value to a bit vector in *L* and setting the bit of the node index. The user hash bit vector algorithm is shown in Algorithm 5, and its time complexity is $O(n_{node})$. Given a node and its corresponding row in *R*, if there is a set bit in the row while the same index in the user hash bit vector of the node is clear, a user cross anomaly occurs. The algorithm is shown as Algorithm 6 (Algorithms with indexes larger than 4 are placed in the Appendix), and its time complexity is $O(n_{node})$.
- *System isolation:* This anomaly can be found if a system node row in *R* contains a 0, as shown in Fig. 8(b). The algorithm is shown in Algorithm 7, and its time complexity is $O(n_{node})$.

Because the policy shadow and policy irrelevance are related to the nodes which are selected by policies, a data structure recording the corresponding relationship between nodes and policies can accelerate the anomaly checking. Therefore, we design the data structure named bidirectional node policy

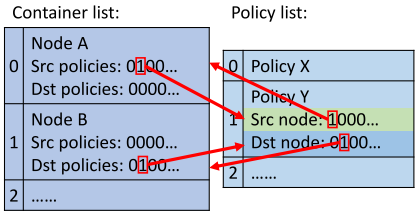

Fig. 9. Bidirectional node policy map structure.

(BNP) map, shown in Fig. 9. The nodes use bit vectors to record the policies that match them, and the policies also use bit vectors to record the matched nodes. The space complexity of the BNP map B is $O(n_{policy}n_{node})$, which has the same order of magnitude as the space complexity of R, $O(n_{node}^2)$. Thus, B does not cost vast additional memory space. To construct B, we store the matched nodes in each policy and the policies that match the nodes in each node during the R generation process. The BNP map B enables the system to handle anomaly checking between policies, and further supports the incremental R generation and fix plan generation which are introduced in Sections V and VI.

- *Policy shadow:* This anomaly can be checked by traversing all nodes and comparing the source and destination nodes for the policies that have one or more of the same source nodes. If the specification area of a policy can be covered by another one with higher priority, then they are a policy shadow pair, as shown in Fig. 8(c). The algorithm is shown in Algorithm 8, and its time complexity is $O(n_{node}(n_{policy}S_{label})^2)$, which approximately equals $O(n_{node})$.
- *Policy irrelevance:* This anomaly can be checked by traversing all policies, and for each policy check its source node or destination node bit vector. If either is empty, policy irrelevance occurs, as shown in Fig. 8(d). The algorithm is shown in Algorithm 9, and its time complexity is $O(n_{policy})$.

Therefore, all predefined anomaly checks can be completed with reasonable time complexity. Theoretically, Kano can verify all properties which can be expressed by reachability. To fully utilize the ability of Kano, besides predefined anomaly checking, we provided some user-defined constraints as follows.

- *Link and Unlink:* Operators can specify a link by the link configuration operation *link*(*node, node*) and then use *check* for this link, similar to other anomaly checks. Kano checks whether the corresponding link bit in R is true. The time complexity is $O(1)$. *unlink* works like *link*; the only difference is that the corresponding bit is false.
- *Public and Private:* If a node has a *public* or *private* property, Kano checks whether all nodes can access it or no node can access it by checking if the specific column in R is all true or all false. The time complexity is $O(n_{node})$.

These user-defined properties are all about reachability between a limited number of specified nodes, and their time complexity is also limited.

C. Property Set

In the anomaly checking we discussed before, some additional information is needed such as how to determine the user owning the node when verifying *user cross* and which nodes are system nodes when verifying *system isolation*. These inputs cannot be given by nodes and policies. Therefore, we design a property configuration operation *set* to input this additional information. Moreover, *set* can be used to specify some optimizations.

Label properties: The properties *User* and *Dense* can be set to a label key. *User* tells Kano which label is used to distinguish node owners. *Dense* is an optimization property we mentioned in Section IV-A, which means Kano uses the key-value pair of the label as a hash key instead of the key to generate L. Operators can manually specify the *Dense* labels or use Kano to automatically find *Dense* labels by giving a percentile threshold between 0% and 100%.

Node properties: The properties *System*, *Public* and *Private* can be assigned to nodes. *System* means that a node is owned by the cloud orchestrator system and authorized to access all nodes. *Public* means that a node is supposed to be accessible from all nodes, and Kano will not raise errors when finding the node *User Cross*. *Private* means that a node is compartmented and will not cause an error when Kano finds it *System Isolated*.

V. Incremental CNNP Verification

In cloud-native environments, nodes and policies change frequently due to new service updates and delivery. To ensure configuration correctness and service continuity, CNNP verification must be conducted in a rapid snapshot manner. Most operations in a cloud-native network are minor changes such as *add* or *del* a node or a policy. In these cases, the majority of the reachability bit matrix R remains unchanged. The R calculation can be accelerated by using an incremental algorithm to update the changed part instead of a full recalculation. Kano accelerates CNNP verification with incremental algorithms that update the main data structures such as the reachability bit matrix R and BNP map B. The incremental update algorithms are described according to the operations of adding/deleting nodes and policies.

A. Adding a Node

When adding a node n, reachability between other nodes will not be affected. Therefore, Kano needs to calculate only the nodes that can access or be accessed by n and update L and B. First, Kano initializes two new bit vectors *in* and *out*. Then, Kano iterates all labels of n and updates L. Next, for each policy, Kano iterates its destination and source labels to judge whether the new node matches the source or destination labels of the policy. If it matches the source labels, Kano bitwise ANDs the destination bit vector stored in B of this policy with *out*. If it matches the destination labels, Kano bitwise ANDs the source bit vector stored in B of this policy with *in*. Finally, we add *in* to R as a new row and add *out* to R as a new column. The algorithm is shown in Algorithm 2. The time complexity is $O(n_{policy})$.

Algorithm 2 Adding a Node

Input: L_{node}, L_{policy}, R, L, B, new node n
Output: R, L, B

```
1: inBitSet = bitSet(n_node)
2: outBitSet = bitSet(n_node + 1)
3: for key in n.labels do
4:     L.get(key).set(n_node + 1)
5: for i = 0 to n_policy do
6:     p = L_policies.get(i)
7:     if n.dst == p.dst then
8:         B[n].getDst().set(n_node + 1)
9:         if p.action == "allow" then
10:            outBitSet |= B[p].getSrc()
11:        else if p.action == "block" then
12:            outBitSet &= ~ B[p].getSrc()
13:    if n.src == pol.src then
14:        B[n].getSrc().set(n_node + 1)
15:        if p.action == "allow" then
16:            inBitSet |= B[n].getDst()
17:        else if p.action == "block" then
18:            inBitSet &= ~ B[n].getDst()
19:    R.addRow(inBitSet)
20:    R.addColumn(outBitSet)
21:    L_node.add(n)
22: return L_node, R, L, B
```

B. Deleting a Node

When deleting a node n, other nodes will not be affected. Kano deletes the corresponding column and row in R and updates B and L. The algorithm is shown in Algorithm 10. Because the deletion of B requires iterating L_{policy}, the time complexity is $O(n_{policy})$.

C. Adding a Policy

When adding a policy p, the reachability between nodes, as well as B must be updated. Kano initializes 2 new bit vectors s to record the source nodes and d to record the destination nodes. Then, Kano iterates the source and destination labels in p and uses L to find the nodes which match the source and destination labels of p. Then, Kano iterates d and applies bitwise OR or AND to s and the corresponding row. Finally, the policies affecting the same nodes by B are found, and policies with higher priority are applied. The algorithm is shown in Algorithm 11. The time complexity is $O(n_{node}S_{key})$.

D. Deleting a Policy

Deleting policies is the most complex incremental operation. When a policy p is deleted, the reachability between nodes changes, but the nodes matched by the deleted policy are also matched by other policies. Kano must use B to find them. First, Kano looks up all nodes that match its source and destination labels in B and records all $e(x, y)$ as affected links. Then, for each affected node, Kano searches in B to find other policies that match. Affected links not kept by other policies are deleted from R. Finally, Kano updates B by deleting the corresponding information of p. The algorithm is shown in Algorithm 12. The time complexity is $O(n_{node}n_{policy}S^2_{label})$.

After updating R and B, Kano runs the chosen anomaly checking algorithms. To demonstrate the performance benefit of incremental CNNP verification, we compare its time complexity with that of the full R calculation. As discussed in Section IV-A, the time complexity of the full R calculation is $O(n_{node} + n_{policy})$.

- Node addition and deletion have time complexities of the same order of magnitude, but the operations of node addition and deletion are simpler than full R calculation. Therefore, the calculation time on all CNNP networks is reduced without considering the scale and sparsity.
- The time complexity comparison between the full R calculation and policy addition/deletion depends on the sparsity. The sparser the network is, the greater the time reduction achieved by the incremental algorithm.

Therefore, the acceleration from incremental verification must be analyzed experimentally. Further discussion is presented in Section VIII.

VI. Fix Plan Generation

With anomaly checking, the redundant policies, missing and redundant links are identified and recorded. While the redundant policies can be easily deleted, how to fix the missing and redundant links is not obvious. To implement the fully automatic operation procedure, Kano needs to generate a fix plan based on these policies and links.

We design the *fix* primitive to transform the link additions and removals to label and policy additions and removals. The fix plan includes the removal of existing policies, modification of existing nodes and insertion of new policies. The basic operations of the *fix* primitive are adding and deleting links.

To keep the readability and manageability, the operator prefers to use existing labels rather than new random labels, and to use fewer but general policies to generate patch policies. Take adding missing links as an example, each missing link can be created naively by adding a policy directly using the labels of the source and destination node. However, to find the minimal set of source or destination labels covering all these policies is a hitting set problem, which is an NP-complete problem. Meanwhile, whether the added policy brings unneeded links or blocks necessary links makes the problem more complex. To solve this problem, Kano uses greedy algorithms to find a suboptimal solution, which does not add block policies and keeps the fewest allow policies.

A. Adding Links

When anomaly checking is complete, a list of links to be added is provided. Kano views this list in its entirety and tries to add them with the fewest policies. The fix steps for adding links are described as follows:

- *Generation:* For each link, Kano uses all labels of the source node and destination node to generate a new policy. This policy certainly creates the link and has the least

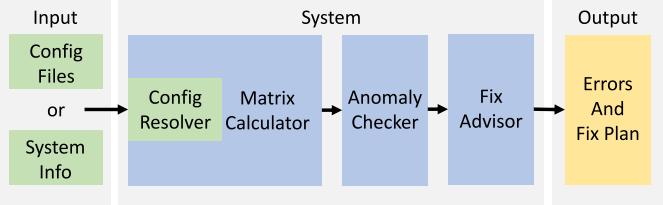

Fig. 10. System structure.

impact on other links, but we still need to check the exact ramifications.

- *Check:* Kano checks whether the links created by the policy are all needed links. Needed links include existing links and candidate links to be added. If the links added include unneeded links, go to the patch step.
- *Patch:* If checking cannot be passed, Kano adds a randomly generated label to the source or destination node. Add this label to the policy as well. This label guarantees that no other links are affected by the policy.
- *Aggregation:* Kano iterates all added policies, and uses a greedy algorithm to merge them. If two policies have labels whose key and value are both equal, check whether a new policy with these equal labels results in unneeded links. If not, replace these policies with the new policy.

In the first step *Generation*, if the link already exists, the following procedure is bypassed. The link may have already been created in the previous generation and aggregation procedure. The complete algorithm is shown in Algorithm 3.

B. Deleting Links

To avoid unexpected blocks, which bring functionality concerns, instead of adding new block policies, Kano generates a link deleting plan by removing allow policies. The fix steps for deleting links are described as follows:

- *Search:* Use *B* to find all policies creating the link. Search in *B* to find the policy list which matches the source or destination node as the removal policy list.
- *Removal:* Remove the generated policy list. For each link created by the removal policy list, use *B* to find whether other policies keep it. The removal algorithm works the same as the search of removal policy lists. If a link is kept by other policies, continue; otherwise, record the link in order to add it back.

The remove algorithm is shown in Algorithm 4. After removal, there is a list of links to add. Kano then goes through the adding procedure. Therefore, in a real verification task, Kano first deletes the unneeded links and then adds the links generated by both anomaly checking and link deletion.

Algorithm 3 Adding Links

Input: L_{policy}, R, B, L_{link}
Output: L_{new_p}

```
// Generation
L_new_p = []
for link in L_link do
    L_new_p.add(Policy(link.dst, link.src, allow, 0))
// Check
for p in L_new_p do
    flag_patch = False
    for src in B[p].getSrc() do
        for dst in B[p].getDst() do
            if R(src, dst) == False &
              Link(src, dst) not in L_link then
                flag_patch = True
    // Patch
    if flag_patch then
        p.addFixLabel()
        L_node[src].addFixLabel()
        L_node[dst].addFixLabel()
// Aggregation
for p in L_new_p do
    if p.isMerged then
        continue
    p_new = Policy()
    for p_other in L_new_p do
        for (key, value) in p_other.dst do
            if (key, value) in p.dst then
                p_new.dstLabel.add(key, value)
        for (key, value) in p_other.src do
            if (key, value) in p.src then
                p_new.srcLabel.add(key, value)
    flag_aggre = True
    for dst in B[p_new].getDst() do
        for src in B[p_new].getSrc() do
            if R[src, dst] == False |
              Link(src,dst) not in L_link then
                flag_aggre = False
    if flag_aggre then
        p_other.isMerged = True
        p = p_new
L_new_p.removeAllMerged()
return L_new_p
```

VII. Implementation

We choose to implement the prototype of Kano based on Kubernetes because Kubernetes uses an ABAC-based policy to manage its containers and is a popular large-scale cloud-native solution that supports up to 100K containers under one controller. The prototype of Kano consists of three components, namely, a matrix calculator, violation checker and fix advisor. The system framework is shown in Fig. 10.

The system is implemented in Java and can both handle LAVI program files and interactively work in the command line. We provide a script that automatically generates LAVI programs by accessing information from a Kubernetes controller. We provide a verification example as follows, which displays LAVI's ability to define a verification task with several lines of code, to show the usefulness of LAVI.

Algorithm 4 Deleting Links

Input: L_{policy}, R, B, L_{link}
Output: L_{del_p}, L_{add_link}

```
// Search
L_del_p = []
L_add_link = []
for link in L_link do
    for p in B[link.dst].getDst() do
        if p.src == (link.src) then
            L_del_p.add(p)
// Removal
for p in L_del_p do
    for dst in B[p].getDst() do
        for src in B[p].getSrc() do
            if ! Link(src,dst) in L_link then
                L_add_link.add(Link(src, dst))
for p in L_del_p do
    for dst in B[p].getDst() do
        for other_p in B[dst].getDst() do
            for src in B[p].getSrc() do
                if src in B[other_p].getSrc() then
                    L_add_link.remove(Link(src, dst))
return L_del_p, L_add_link
```

```
LoadFile("/configs")
set("namespace", User)
Node("namespace":"default") systemNode
set(systemNode, System)
check(UserCross)
check(SystemIsolation)
check(PolicyShadow)
check(PolicyIrrelevance)
fix
```

In this example, the program loads Kubernetes configuration files in YAML format from a dictionary, defines user label and system nodes, then calculates the reachability bit matrix and uses it to check all anomalies and provide fix advice.

The implementation details of each component are elaborated as follows.

A. Matrix Calculator

First, given Kubernetes configurations, Kano must resolve them and transform them into the native data structures of the Kano system including R, L and B. The Matrix calculator plays this role with the algorithms we propose in Section IV-A and V. If the reachability bit matrix has not been calculated and stored, it is calculated by the full R generation algorithm mentioned previously. Otherwise, if R has already been generated and there is some minor modification to the configurations, Kano uses incremental R generation to update R. We also provide some details to address the Kubernetes features.

- *Single action:* Policies only have the *allow* action, and therefore priority is not needed. A node can only access the nodes allowed by policies.
- *Kubernetes Namespaces*. The namespaces also have labels, and a namespace corresponds to many nodes. The policies also match nodes by namespaces. Therefore, Kano performs the same prefiltration based on the namespace first and casts the namespace to nodes to reduce the computing time.
- *Default Reachable:* If a node does not match the destination labels of any policy, it is reachable. Therefore, we store every row of the reachability in the node data structure and set all bits to 1. The first time the destination labels of a policy match it, we set all bits to 0 and then perform other operations.
- *Direction:* In Kubernetes, a policy must specify its direction, ingress or egress. We can support this by calculating the R of ingress and egress policies separately and using an bitwise AND action to obtain the final R, as shown in Fig. 11.
- *Port*. In Kubernetes, a policy must specify the port it allows. Because a container in Kubernetes either opens all of its ports or only several ports, if a container has only several ports, Kano sees each port as a node.

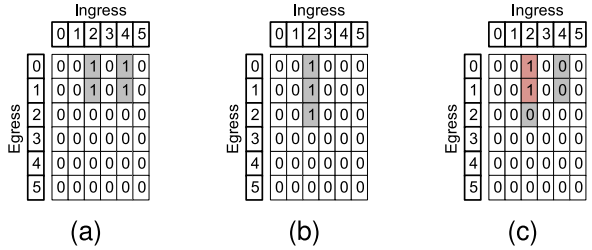

Fig. 11. Final R generated by the egress and ingress R. (a) Egress R. (b) Ingress R. (c) Final R.

B. Anomaly Checker

The anomaly checker uses the anomaly checking algorithms shown in Section IV-B to check whether anomalies exist in R. If there is an anomaly, it records the objects that cause the anomaly. Considering the production environment, Kano supports users in defining new constraints by declarative language. A few possible options are as follows.

- *User Label Definition:* By default, Kano views different namespaces as different users, but users can choose other labels to represent node users by the following code.

```
set("User", User)
```

- *Isolation and link:* Users can add isolation/link constraints based on labels. If the link is built/not built, Kano issues an anomaly. This can be added by the following code.

```
Node("User":"Alice","Role":"test") node1
Node("User":"Alice","Role":"DB") node2
Link(node1, node2) testLink
Check(testLink)
```

- *Public and private:* Users can assert which nodes are public/private nodes based on labels. If the specified nodes cannot be accessed by all other nodes/can be accessed by any other node, Kano issues an anomaly. This can be added by the following code.

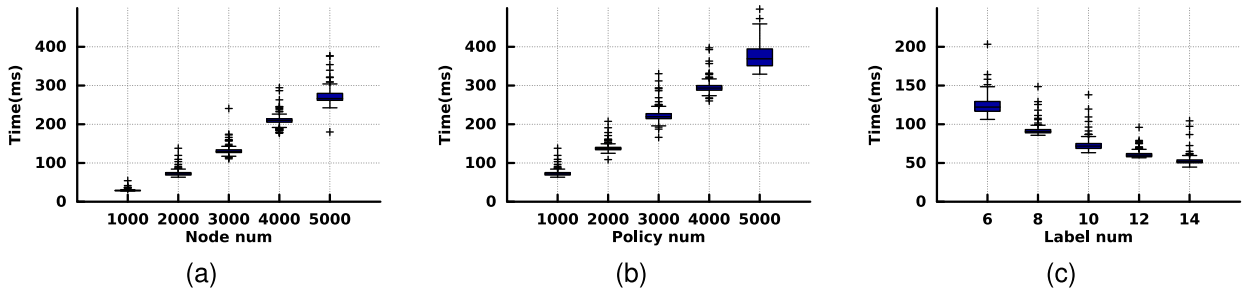

Fig. 12. The relationship between the reachability bit matrix calculation time and node number, policy number or key number. (a) Node number variable, others fixed. (b) Policy number variable, others fixed. (c) Key number variable, others fixed.

```
Node("User":"Bob","Usage":"Public") node1
Node("User":"Bob","Usage":"Private") node2
Set(node1, Public)
Set(node2, Private)
Check(Public)
Check(Private)
```

If there are anomalies, the anomaly checker outputs an error report about which anomalies occur, which policies should be removed and which links $e(x, y)$ should be removed or added to fix the problem.

C. Fix Advisor

According to the list of $e(x, y)$ violation checks, the fix advisor gives advice to network operators and generates fix plans to solve the problems. The advice is given to show the identified anomalies. The output performs as follows.

- Show all unsatisfied public and private nodes.
- Show policies that result in reachability between different users and isolation from system nodes.
- Advise the deletion of policies to eliminate policy shadows and policy irrelevances.
- Warn about other user-defined links.

If the network operator finds it difficult to manually fix the problems, the fix advisor can generate a fix plan for her/him with the algorithm shown in Section VI. The fix plan includes a list of operations, including adding labels to nodes and adding/deleting policies. The added labels and policies are marked as "fix labels" and "fix policies". The network operator can directly apply the generated operation to the network or modify the "fix" objectives to maintain readability.

VIII. Evaluation

In this section, we show the relationship between features of datasets and calculation time complexity, and then use simulated datasets to evaluate the scalability of reachability matrix generation, anomaly checking and fix plan generation. All experiments are performed on a CentOS server in which the CPU is an *Intel Xeon 6230R CPU @ 2.10 GHz* and DRAM size is 128 GB.

A. Time Complexity

First we use datasets generated by given parameters to introduce the relationship between n_{node}, n_{policy}, S_{key} and the time consumption of the R calculation. These parameters fully cover all features provided by the policies in Kubernetes. The parameters are as follows.

- *Node number*.
- *Namespace number*.
- *Policy number*.
- *Node label limit*. The maximum number of labels attached to a node.
- *Ns label limit*. The maximum number of labels attached to a namespace.
- *Key limit*. The number of different keys.
- *Value limit*. The number of different values of each key.
- *Destination label limit*. The maximum number of destination labels in a policy.
- *Source ns label limit*. The maximum number of namespace source labels in a policy.
- *Source node label limit*. The maximum number of node source labels in a policy.

Time complexity: We use the datasets in which the policy number or node number varies and other parameters are fixed to perform the reachability bit matrix calculation. The result is shown in Fig. 12a and 12b. When S_{key} is fixed, the calculation time increases proportionately as the node or policy number increases, *i.e.,* the time complexity is $O(n_{node}n_{policy})$. The experiment result is consistent with the theoretical analysis, which indicates the time complexity is $O(n_{node} + n_{policy} + n_{node}n_{policy}S_{key})$. When S_{key} is fixed, the $n_{node} + n_{policy}$ becomes negligible, and the time complexity becomes $O(n_{node}n_{policy})$. It is also proved by keeping the node and policy number fixed and decreasing S_{key} by increasing the key number, the time consumption falls inversely proportionally (Fig. 12c).

B. Scalability

To generate a persuasive dataset, instead of basing on parameters as in Section VIII-A, we collect popular container applications mentioned in B4 [47] like email and database. We also collect some applications based on containers mentioned in other papers [48], [49]. The collected applications are shown in Table IV and an example of the collected applications is shown in Fig. 13. We use these applications as templates and generate experimental datasets by massive replication. We vary the scales of datasets by changing the node number (node num) from 5k to 100k. To make the dataset closer to reality, we also randomly attach user-specific labels to containers and policies as shown in Fig. 14. The number of user-specific labels increases linearly as the node number grows. Features of

TABLE IV
COLLECTED CONTAINER APPLICATIONS

App name	Node num	Policy num
Bulletin Board	3	2
Waste Bin Sensor [48]	4	3
Surveillance Camera [48]	4	3
Anomaly Detection [49]	6	4
Mail Server	2	1
Photo Prism	1	0
MySQL	1	1
Elastic Search	1	1

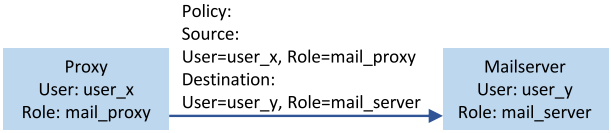

Fig. 13. Application: Mail Server.

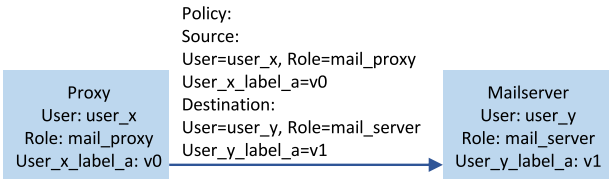

Fig. 14. User-specific labels added.

TABLE V
DATA SET FEATURES

n_{node}	n_{policy}	S_{key}	S_{label}
5,000	3390	$2.40 * 10^{-2}$	$1.55 * 10^{-3}$
10,000	6777	$1.21 * 10^{-2}$	$7.78 * 10^{-4}$
20,000	13569	$6.26 * 10^{-3}$	$3.98 * 10^{-4}$
50,000	34033	$2.45 * 10^{-3}$	$1.56 * 10^{-4}$
100,000	68111	$1.27 * 10^{-3}$	$8.07 * 10^{-5}$

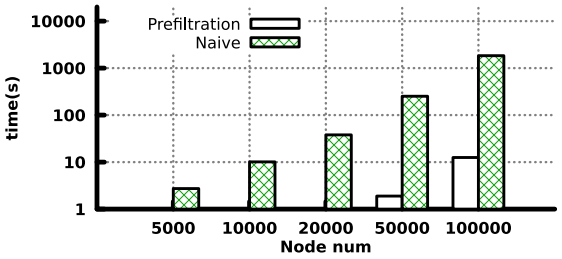

Fig. 15. Time consumption of the reachability matrix calculation.

the generated datasets are shown in Table V. We analyze the scalability of the proposed algorithms based on these datasets.

Complete R generation: The time consumption of the reachability calculation is shown in Fig. 15, and the space consumption is shown in Fig. 16. The time consumption of the naive algorithm has a quadratic relationship with the cluster scale, and the prefiltration algorithm has a linear relationship, which is consistent with the theoretical result. The prefiltration algorithm works much faster than the naive algorithm, with space complexity of the same order of magnitude. The reachability matrix of a 100,000-node network (with approximately 68,000 policies) is calculated in around 12.51 seconds with the prefiltration algorithm.

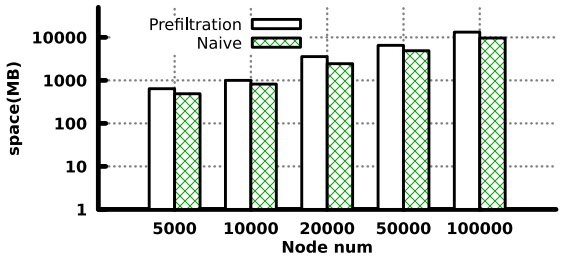

Fig. 16. Space consumption of the reachability matrix calculation.

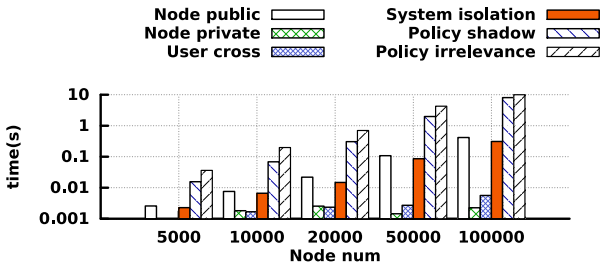

Fig. 17. Time consumption of the anomaly checking.

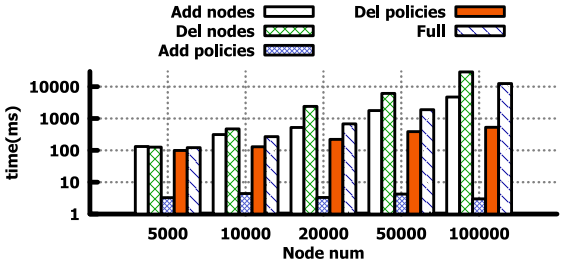

Fig. 18. Time consumption of incremental R generation.

Anomaly checking: We divide the anomaly checking into two classes. The first class is anomalies that cause reachability error, including *Node public*, *Node private*, *user cross* and *system isolation*. The second class is anomalies that have no effect on reachability but are not optimal or result in incorrect policy configurations, including *policy shadow* and *policy irrelevance*.

The experimental results of the anomaly checking are shown in Fig. 17. The first class takes much less time than *R* generation. Even in a 100,000-node network, the checking can be performed in a second, *i.e.,* reachability errors can be found in a very short time. The second class takes up to 10 seconds as the scale of the CNNP network grows to the maximum size of our experiments. As these anomalies do not directly result in incorrect reachability, the calculation time is reasonable for configuration optimization.

Incremental R generation: We compare the average time consumption of adding and deleting nodes and policies in Fig. 18. This shows that the incremental algorithm reduces the time consumption of adding or deleting a single node or policy by no less than one order of magnitude. The high time consumption of adding and deleting nodes is due to memory allocation and collection during *R* insertion and deletion. As the added nodes are always attached to the end of the arrays

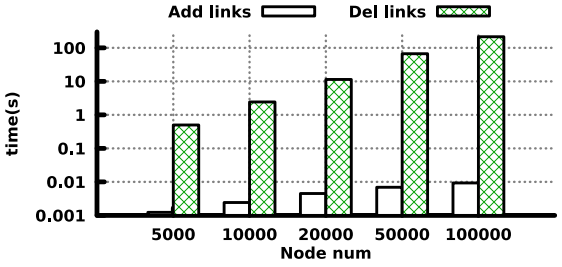

Fig. 19. Time consumption of the fix plan generation.

Algorithm 5 User Hash Bitset Calculation

Input: L_{node}, user label *Key*
Output: User hash bitset map *H*

```
HashMap<String,BitSet> H
for i = 0 to n_node do
    H.hash(L_node[i].getValue(Key)).set(i)
return H
```

Algorithm 6 User Cross Checking

Input: *R*, User hash map *H*, L_{node}
Output: User cross list *U*

```
for i = 0 to n_node do
    VeriSet = H.Hash(L_node.userLabel)
    VeriSet =!VeriSet
    VeriSet = VeriSet& R.getColumn(i)
    if VeriSet == ∅ then
        U.add(i)
return U
```

of which *R* and *L* consist, but the deleted nodes could be anywhere in the arrays, the node deletion costs more time than node addition. The time complexity is consistent with that of the theoretical analysis. The incremental *R* generation is a valuable alternative when the network modification is minor.

Fix plan generation: We randomly choose some links to add to or delete from the CNNP network and use Kano to generate a fix plan. The average generation time of each fix plan is shown in Fig. 19. We can see that deleting links costs much more time compared to adding links. The reason is that deleting links requires *B* to be searched an additional time, and deleting links entails additional adding link procedures. Even in a 100,000-node network, the fix plan can be generated in less than 2 minutes.

IX. Case Study

On the basis of a discussion with a cloud service company, we choose two application cases which actually occur in industrial environments to show how Kano finds and fixes CNNP anomalies.

A. Policy Stale

In industry, especially in a private cloud, cloud users must ask network operators to add network policies when they create new containers and VMs to make their cloud service work.

Algorithm 7 System Isolation Checking

Input: *R*, Node index *i*
Output: System isolation list *C*

```
VeriSet = BitSet(1,n_node)
VeriSet = VeriSet∧ R.getRow(i)
if VeriSet!= ∅ then
    for each set bit j in VeriSet do
        C.add(j)
return C
```

Algorithm 8 No Policy Shadow Checking

Input: *R*, L_{policy}, L_{node}, *B*
Output: Policy shadow list *S*

```
for node in L_node do
    pList = B[node].getDst()
    for p_A in pList do
        for p_B to pList do
            if p_A == p_B||p_A.prio > p_B.prio then
                continue
            SetSrc = B[p_A].getSrc()
            SetSrc &= B[p_B].getSrc()
            SetSrc = B[p_B].getSrc()
            SetDst = B[p_A].getDst()
            SetDst &= B[p_B].getDst()
            SetDst = B[p_B].getDst()
            if SetA | SetB == ∅ then
                S.add([p_A, p_B])
return S
```

Algorithm 9 No Policy Irrelevance Checking

Input: *R*, L_{policy}, L_{node}, *B*
Output: Policy irrelevance list *S*

```
for p in L_policy do
    SetS = B[p].getDst()
    SetA = B[p].getSrc()
    if SetS == ∅ || SetA == ∅ then
        S.add(p)
return S
```

However, when the service is deprecated, cloud users typically simply remove their containers and VMs and think everything is fine. Eventually, many useless policies are left in the cloud, and they may cause security issues when deploying new containers and VMs.

We built a Kubernetes cluster and deployed many pods and services with YAML files found in the Kubernetes community, and then removed only the containers and kept the network policies. We repeated the process many times, finally leaving many redundant policies, which we checked manually. We used Kano to check policy irrelevance and successfully found all policies that did not match any containers. After removing these policies, no redundant policies remained, and no policies were removed by mistake.

Algorithm 10 Deleting a Node

Input: L_{node}, L_{policy}, R, L, B, node index n
Output: L_{node}, R, L, B

```
R.removeRow(n)
R.removeColumn(n)
for p in L_policy do
    B[p].clearDst(n)
    B[p].clearSrc(n)
for key in L do
    L[key].remove(n)
L_node.remove(n)
return L_node, R, L, B
```

Algorithm 11 Adding a Policy

Input: L_{node}, L_{policy}, R, L, B, new policy p
Output: L_{policy}, R, B

```
B[p].setSrc(new BitSet(1,n))
B[p].setDst(new BitSet(1,n))
for label in p.srcLabels do
    B[p].getSrc() &= L.hash(label.key)
for label in p.dstLabels do
    B[p].getDst() &= L.hash(label.key)
for node in B[p].getSrc() do
    if !p.src == L_node[node] then
        B[p].getSrc().clear(node)
for node in B[p].getDst() do
    if !p.dst == L_node[node] then
        B[p].getDst().clear(node)
for node in B[p].getDst() do
    if p.act == "allow" then
        R.getRow(node) |= B[p].getSrc()
    else if p.act == "block" then
        R.getRow(node) &= ~ B[p].getSrc()
    for pol in B[L_node[node]].getSrc() do
        if L_policy[pol].prio < p.prio then
            if L_policy[pol].act == "allow" then
                R.getRow(node) |= B[pol].getSrc()
            else if L_policy[pol].act == "block" then
                R.getRow(node) &= ~ B[pol].getSrc()
L_policy.add(p)
return L_policy, R, B
```

B. Service Update

In this case, we switch the backend of a Web service from a server container(C_0) to a proxy container(C_p) and a cluster of server containers(C_1). After container deployment, we use Kano to update the CNNP policies automatically.

The first step is removing C_0, and the policies related to it. We delete C_0 in Kano and update R incrementally, and check policy irrelevance to remove unused policies. Then, we add C_p and C_1, and update R again. Next, we add two links. The first is the frontend container to C_p, and the second is C_p to C_1. We use Kano to check these two links. If they do not exist, Kano can give fix advice to build these policies. All needed links are built with no redundant policies retained.

Algorithm 12 Deleting a Policy

Input: L_{policy}, R, B, policy p
Output: L_{policy}, R, B

```
inBitSet = bitSet(n_node)
outBitSet = bitSet(n_node) + 1
L_other_p = []
L_links = [] //if p.act == "block", it is L_unlinks here
for i in B[p].getDst() do
    for j in B[p].getSrc() do
        L_links.add((i, j))
for node in B[p].getDst() do
    L_other_p.addAll(B[node].getDst())
    B[node].getDst().remove(L_policy.index(p))
for node in B[p].getSrc() do
    L_other_p.addAll(B[node].getSrc())
    B[node].getSrc().remove(L_policy.index(p)
for p_other in L_other_p do
    for link in L_links do
        if p_other allows link then
            L_links.del(link)
for link in L_links do
    R.clear(link)
return L_policy, R, B
```

X. Conclusion

As cloud-native computing becomes a prevailing paradigm, CNNP verification becomes an urgent problem in terms of ensuring service connectivity correctness. However, CNNP verification is challenging due to the large-scale and frequent updates of cloud-native networks and the demand for operation automation. In this work, an efficient system for verifying large-scale CNNPs at runtime, named Kano, is proposed. Kano is easy to use with a high-level intent-based language LAVI. Moreover, Kano is fast to execute for full and incremental verification using a CNNP-specific bit matrix model with a prefiltration algorithm and a partial-update method. Fix plans for violations are also generated by Kano to facilitate user operation. Kano is implemented as an independent CNNP verification library for any cloud-native platform and has been integrated into Kubernetes. An evaluation on a large-scale network of 100k nodes and approximately 68k policies shows the efficiency of Kano, with execution times of 12.51 seconds for all reachable invariant verification and 0.299 milliseconds for policy addition verification.

Appendix
Algorithms

See Algorithms 5–12.

References

[1] "CNCF cloud native definition v1.0." 2022. [Online]. Available: https://github.com/cncf/toc/blob/main/DEFINITION.md

[2] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "OpenStack: Toward an open-source solution for cloud computing," *Int. J. Comput. Appl.*, vol. 55, no. 3, pp. 38–42, Oct. 2012.

[3] B. Hindman et al., "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. 8th USENIX Conf. Netw. Syst. Design Implement.*, 2011, pp. 295–308.

[4] D. Bernstein, "Containers and cloud: From LXC to docker to kubernetes," *IEEE Cloud Comput.*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[5] "Network policies." Kubernetes. 2020. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/network-policies/

[6] "Deep dive into network policy." 2021. [Online]. Available: https://networkpolicy.io

[7] D. Servos and S. L. Osborn, "Current research and open problems in attribute-based access control," *ACM Comput. Surveys*, vol. 49, no. 4, pp. 1–45, Dec. 2017.

[8] N. McKeown et al., "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[9] A. Fogel et al., "A general approach to network configuration analysis," in *Proc. 12th USENIX Sympo. Netw. Syst. Design Implement. (NSDI)*, Apr. 2015, pp. 1–15. [Online]. Available: https://www.microsoft.com/en-us/research/publication/a-general-approach-to-network-configuration-analysis/

[10] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proc. ACM SIGCOMM Conf. SIGCOMM*, 2011, pp. 290–301.

[11] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. 9th USENIX Conf. Netw. Syst. Design Implement.*, 2012, p. 9.

[12] P. Kazemanian, M. Chang, H. Zheng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Jan. 2013, pp. 1–13. [Online]. Available: https://www.microsoft.com/en-us/research/publication/real-time-network-policy-checking-using-header-space-analysis/

[13] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proc. 10th USENIX Conf. Netw. Syst. Design Implement.*, 2013, pp. 15–28.

[14] 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.

[15] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *Proc. 12th USENIX Conf. Netw. Syst. Design Implement.*, 2015, pp. 499–512.

[16] H. Zeng et al., "Libra: Divide and conquer to verify forwarding tables in huge networks," in *Proc. 11th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Apr. 2014, pp. 87–99.

[17] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," in *Proc. 43rd Annu. ACM SIGPLAN-SIGACT Symp. Principles Programming Lang.*, Jan. 2016, pp. 69–83.

[18] A. Horn, A. Kheradmand, and M. Prasad, "Delta-Net: Real-time network verification using atoms," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Mar. 2017, pp. 735–749. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/horn-alex

[19] S. Bleikertz, C. Vogel, and T. Groß, "Cloud radar: Near real-time detection of security failures in dynamic virtualized infrastructures," in *Proc. 30th Annu. Comput. Security Appl. Conf. (ACM)*, Dec. 2014, pp. 26–35.

[20] Y. Wang et al., "Tenantguard: Scalable runtime verification of cloud-wide vm-level network isolation," in *Proc. NDSS*, 2017, pp. 1–15. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/tenantguard-scalable-runtime-verification-cloud-wide-vm-level-network-isolation/

[21] K. Jayaraman et al., "Validating datacenters at scale," in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2019, pp. 200–213.

[22] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li, "APKeep: Realtime verification for real networks," in *Proc. 17th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Feb. 2020, pp. 241–255. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/zhang-peng

[23] M. Dobrescu and K. Argyraki, "Software dataplane verification," in *Proc. 11th USENIX Conf. Netw. Syst. Design Implement.*, 2014, pp. 101–114.

[24] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "SymNet: Scalable symbolic execution for modern networks," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 314–327.

[25] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying reachability in networks with mutable datapaths," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Mar. 2017, pp. 699–718. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/panda-mutable-datapaths

[26] D. Dumitrescu, R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Dataplane equivalence and its applications," in *Proc. 16th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Feb. 2019, pp. 683–698. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/dumitrescu

[27] F. Yousefi, A. Abhashkumar, K. Subramanian, K. Hans, S. Ghorbani, and A. Akella, "Liveness verification of stateful network functions," in *Proc. 17th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Feb. 2020, pp. 257–272. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/yousefi

[28] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging P4 programs with vera," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 518–532.

[29] J. Liu et al., "P4V: Practical verification for programmable data planes," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 490–503.

[30] D. Dumitrescu, R. Stoenescu, L. Negreanu, and C. Raiciu, "BF4: Towards bug-free P4 programs," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 571–585.

[31] B. Tian et al., "Aquila: A practically usable verification system for production-scale programmable data planes," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 17–32.

[32] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 300–313.

[33] S. K. Fayaz et al., "Efficient network reachability analysis using a succinct control plane representation," in *Proc. 12th USENIX Conf. Oper. Syst. Design Implement.*, 2016, pp. 217–232.

[34] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 155–168.

[35] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Control plane compression," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 476–489.

[36] Y. Li, J. Jia, X. Hu, and J. Li, "Real time control plane verification," in *Proc. ACM SIGCOMM Workshop Netw. Programming Lang.*, 2019, p. 2.

[37] A. Tang et al., "Campion: Debugging router configuration differences," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 748–761.

[38] "Cluster multi-tenancy | Kubernetes engine documentation | Google cloud." 2007. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/multitenancy-overview

[39] Y. Li, C. Jia, X. Hu, and J. Li, "Kano: Efficient container network policy verification," in *Proc. IEEE Symp. High-Perform. Interconnects (HOTI)*, 2020, pp. 63–70.

[40] P. Bosshart et al., "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul 2014.

[41] N. Feamster and J. Rexford, "Why (and how) networks should run themselves," Oct. 2017, *arXiv:1710.11583*.

[42] A. Singh et al., "Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 183–197, Sep. 2015.

[43] Y.-W. E. Sung, X. Tie, S. H. Wong, and H. Zeng, "Robotron: Top-down network management at Facebook scale," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 426–439.

[44] H. H. Liu et al., "Automatic life cycle management of network configurations," in *Proc. Afternoon Workshop Self-Driving Netw.*, Aug. 2018, pp. 29–35.

[45] B. Tian et al., "Safely and automatically updating in-network ACL configurations with intent language," in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2019, pp. 214–226.

[46] A. Gember-Jacobson, A. Akella, R. Mahajan, and H. H. Liu, "Automatically repairing network control planes using an abstract representation," in *Proc. 26th Symp. Oper. Syst. Principles*, Oct. 2017, pp. 359–373.

[47] C. Y. Hong et al., " B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN," in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 74–87.

[48] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards delay-aware container-based service function chaining in fog computing," in *Proc. IEEE/IFIP Netw. Oper. Manage. Symp.*, Apr. 2020, pp. 1–9.
[49] J. Santos, P. Leroux, T. Wauters, B. Volckaert, and F. De Turck, "Anomaly detection for smart city applications over 5G low power wide area networks," in *Proc. IEEE/IFIP Netw. Oper. Manage. Symp.*, Apr. 2018, pp. 1–9.

Yifan Li received the B.Sc. degree from the Department of Automation, Tsinghua University, China, where he is currently pursuing the Ph.D. degree under the supervision of Prof. J. Li. His main research interests include network verification and programmable networks.

Xiaohe Hu received the B.Sc. and Ph.D. degrees from the Department of Automation, Tsinghua University, China, where he is currently working as a Postdoctoral Research Fellow with the Department of Computer Science and Technology. His main research interests include packet classification and network security.

Chengjun Jia (Graduate Student Member, IEEE) received the B.Sc. degree from the Department of Automation, Tsinghua University, China, where he is currently pursuing the Ph.D. degree under the supervision of Prof. J. Li. His main research interests include packet classification and congestion control.

Kai Wang received the B.S. degree from the Department of Electronic Science and Engineering, Nanjing University, China, in 2009, and the Ph.D. degree from the Department of Automation, Tsinghua University, China, in 2015. He is currently a Software Engineer with YunShan Networks. His research interests include network security and software-defined networking.

Jun Li received the B.S. and M.S. degrees from the Department of Automation, Tsinghua University, Beijing, China, in 1985 and 1988, respectively, and the Ph.D. degree from the Department of Computer Science, New Jersey Institute of Technology in 1997. He is currently a Professor with the Department of Automation, Tsinghua University. His research interests include network security and network automation.