Kano: Efficient Container Network Policy Verification

Yifan Li*, Chengjun Jia*, Xiaohe Hu*, Jun Li†
* The Department of Automation, Tsinghua University, Beijing, China
† Research Institute of Information Technology (RIIT), Tsinghua University, Beijing, China
{liyifan18, jcj18, hu-xh14}@mails.tsinghua.edu.cn,{junl}@tsinghua.edu.cn

***Abstract*—Container technology is a light weight back-end virtualization solution which copes with the growing demand for internet service concurrency. For security and availability, network isolation is essential in container network. Label based network access control policies are employed as a network isolation solution by the famous container orchestrator, Kubernetes. The large scale and flexibility of container networks implies a prohibitive complexity, whereas the constant changes of policies and containers demand a short response time, thus the policy verification needs to be efficient. However, there is no existing tool that solves the verification problem of label based network access control policies. *Kano* is proposed as the first system to cover container network policy verification, including incremental verification. It leverages on a prefiltration algorithm to reduce the time complexity of reachability matrix calculation from $O(n^2)$ to $O(n)$. With predefined and user-defined constraints which can be verified quickly with the reachability matrix, Kano removes potential risk from the container network. Based on the verification result, Kano provides advices to reinforce the security and availability of the container network.**

***Index Terms*—Container Networks; Network Verification; Attribute-Based Access Control**

I. Introduction

Containers are becoming more and more popular to deploy applications in a quick, cheap, and reliable way. Although containerization helps to achieve cost-efficient resources sharing, it also raises concerns about security and privacy. For example, a tenant of a container cloud may raise questions like: "Are my containers properly isolated from other tenants?" and a developer may doubt like: "Can my co-developers access my containers?" These concerns can be solved by network access control policies, which plays a critical role in ensuring network-wide security and availability. Network access control policies define the reachability between network elements, such as routers, VMs and in this work, containers.

Among various solutions, Kubernetes [1] label based network access control policies is specially designed for container networks. It attaches key-value pairs as labels to containers, and then uses policies to define the connectivity according to the labels. This mechanism is similar to ABAC [3], which has been proved of good expressivity and easy maintenance compared to classical solutions such as the 5-tuple based ACL.

To assure that the network policies are working properly, verification is necessary. In contrast to traditional networks whose verification has been studied by many researchers, there is no existing work on container network policy verification to the best of our knowledge. Container networks pose unique challenges to the verification of network policies.

- *The large scale*. The huge scale of container clusters implies a prohibitive complexity. According to the GKE document, the scale of a container cluster can reach 500k containers [21]. In addition, container networks need to be verified at container-level, instead of router-level like most existing control plane verification works.
- *Frequent update*. The labels in containers and network policies can be added, deleted, or migrated by any cluster user. Consequently, instead of verified only once and offline, the network polices need to be verified repeatedly or periodically at run-time. The frequent update also significantly shortens the lifespan of verification results.
- *Complex intention*. Different from simply isolation between different tenant in multi-tenant scenes, network policies are used to express more complicated intention. A user may expects that the database container can only be accessed by a few specific containers, but the web front-end should be accessible by everyone. This heterogeneous feature brings more complexity to the container network policies.

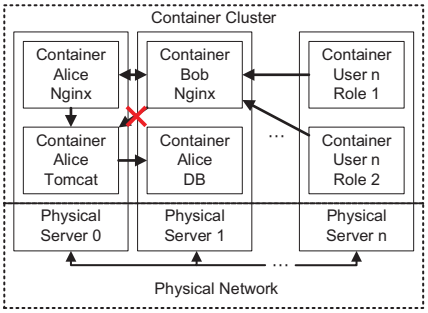


Fig. 1: The structure of a container cluster

Figure 1 shows a motivating example, which is a simplified view of a container cluster environment. The physical network insures that all physical servers are connected with each other. The owner and role are marked on each container.

- Suppose Alice would like her proxy container, Nginx, to be reachable by everyone, while her front-end container, Tomcat, to be reachable only by her Nginx container. Therefore, she activates a policy which indicates that only

2332-5569/20/$31.00 ©2020 IEEE
DOI 10.1109/HOTI51249.2020.00024

TABLE I: The comparison between existing network verification solutions and Kano

Network	Solutions	Methods	Features		Physical vs Virtual net.		Control vs Data plane		Size of input			Verif. Time(s)
			Incr.	All pairs Reach.	Phy.	Vir.	Ctr.	Data	VMs	Routers	Rules	
Non-Cloud	HSA [5]	Custom algorithms			•			•	-	26	756.5k	-
	NetPlumber [6]	Graph-theoretic	•	•		•		•	-	52	143k	60
	Anteater [7]	SAT solver			•			•	-	178	1,627	-
	Veriflow [8]	Graph-theoretic	•	•		•		•	-	172	5,000k	-
	AP verifier [9]	Custom algorithms	•			•		•	-	58	3,605	-
	ARC [10]	Graph-theoretic			•		•		-	few tens	-	-
	Batfish [11]	SMT Solver		•	•		•		-	21	-	86,400
	ERA [12]	Custom Algorithms			•		•		-	over 1,600	-	-
	Minesweeper [13]	SMT Solver			•		•		-	405	-	about 20
	CPC [14]	Custom Algorithms			•		•		-	2,000	-	946.4
	Jinjing [15]	SMT Solver	•		•			•	-	thousands	-	800
Cloud	NoD [16]	SMT Solver		•	•			•	100k	-	820k	471,600
	Plotkin et al. [17]	SMT Solver		•	•			•	100k	-	820k	7,200
	Cloud Radar [18]	Graph-theoretic				•			30k	-	-	-
	Probst et al. [19]	Graph-theoretic				•	•		23	-	-	-
	TenentGuard [20]	Custom Algorithms	•	•		•		•	100k	1200	850k	1,055.88
	Kano	Custom Algorithms	•	•		•	•		10k	-	5k	53.51

Nginx containers can reach Tomcat containers. However, Bob has a Nginx container, too. Alice becomes aware of that, and sets another role which isolates all her containers from others. Unfortunately, the former policy still works, and Alice's Tomcat remains accessible by Bob's Nginx, while her Nginx becomes unreachable by others.

- Suppose the cluster has 50k containers. Despite the large scale, users still wants to monitor their containers and find out risks in real time. Then they need to verify the reachability between $50k*50k*2 = 5billion$ of container pairs every time. Since they knows the verification result becomes meaningless as soon as the next change is made to the container cluster, they schedule the verification to be performed at every configuration update and expects to see the results within a few seconds.

In this paper, we present *Kano*, an efficient system for verifying large scale, container-level network policies at runtime. To address the aforementioned challenges, our main ideas are as follows. First, Kano takes advantage of the sparse reachability relationship between containers to reduce the performance overhead of verification. Second, instead of SMT solver or binary tries, Kano uses bitmatrix to keep the verification time and space cost relatively constant and limited. Third, Kano supports incremental verification by examining only parts of the container network affected by the configuration change. Finally, Kano provides declarative language for network operators to express intention besides predefined constraints. The following summarizes our main contributions:

- *An efficient reachability matrix calculation approach*. We propose an approach which can reduce the time complexity of calculating all pairs reachability matrix to $O(n)$ from $O(n^2)$ of naive algorithm.
- *A data structure for incremental verification*. We design a data structure in which containers and policies are correlated. By this we can identify the portion of the container network affected by an update quickly.
- *A set of constraints expression and verification algorithm*. We propose a set of predefined constraints to represent risks in container network, and a declarative language which can express user-specific constraints. By reachability matrix, these constraints can be verified conveniently.
- *Implementation and integration to Kubernetes*. We implement Kano to find potential risks with container network configurations and provide advice on how to fix it. Kano takes Kubernetes configurations files as input, or works automatically on Kubernetes controllers.

The rest of this paper is organized as follows. Section II summarizes and analyzes related existing work, and establishes the value proposition of Kano. Section III proposes a bipartite graph model for container network, and transforms it into a reachability matrix to represent network policies and constrains. Section IV elaborates the algorithm to calculate the reachability matrix and carry out network verification. The implementation of Kano is presented in Section V. Section VI shows the evaluation results of Kano. Section VII concludes this paper and proposes the future work.

II. Related Works

The most popular network access control solution used by container network is label based policies advocated by Google Kubernetes. Kano is designed to work with the label-based configurations.

To our best knowledge, so far there is no existing network verification tool proper to verify Kubernetes label based policies. Table I summarizes the comparison between existing network verification solutions and Kano. Based on the target environments, existing works can be divided into two categories, i.e., either cloud-based or non-cloud networks. It demonstrates that Kano differs from the existing works as follows. First, Kano performs verification at a better granularity (i.e., all-pair container-level vs single-pair router-level). Second, Kano is

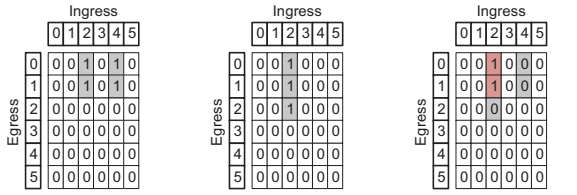
more scalable (e.g., it can verify 10k VMs within 53 seconds and has a linear time complexity). Finally, Kano applies the IBN philosophy. Besides pre-defined constraint, it provides a declarative language for network operators to express their intent. When error is found, Kano also provides fine-grained configuration modification advice instead of just alert.

III. Model

When doing container network verification, the problem we are going to solve is "In a container network, do the network policies violates the network constraints?" Therefore, We need to explain what are container network, network policies and network constraints, then model them.

A. Container Network

We model a container network as a bipartite graph. Suppose the two disjoint and independent sets U and V represent egress and ingress of all containers, and the edge sets $E1$ and $E2$ represent the connections allowed by network policies. $E1$ represents the connection set allowed by egress policies, and $E2$ represents ingress policies. The connections allowed by both policies can be actually allowed as represent by the joint connection set E, i.e., $E1 \cap E2$.



(a) Egress connections (b) Ingress connections (c) Final connections

Fig. 2: Container network reachability matrix model

We express this bipartite graph model by reachability bitmatrix. The egresses are transformed to rows, and the ingresses are transformed to columns. If a bit at row x and column y is set, it means container x is allowed to reach container y. A reachability matrix is shown in Figure 2 as an example.

We choose this reachability bitmatrix model due to the following advantages.

- *Low storage consumption*. The reachability bitmatrix can record an edge with a single bit, which costs much fewer space than using integer set [9] or string [2].
- *Fast calculation*. The matrix calculation and the verification procedure can be done by bitwise operation, which is much faster than integer and string calculation.
- *Easy verification*. The policies and constraints can be easily translated into matrix expression. We will define the constraints subsequently.

B. Network Policies

In the Kubernetes label based policies, a container has some labels and each label consists of a key and a value. A policy can be roughly divided into four parts - select part, allow part, direction part and protocol part.

- *Select part*. The select part uses label to identify which containers this policy is applied to. A policy only takes effect to the containers which meet all label restrictions, i.e., all keys appearing in the select part of a policy should appear in the selected containers, and their corresponding values should all match.
- *Allow part*. The allow part uses label to judge which containers are allowed to access. The identify mechanism is similar to the select part.
- *Direction part*. The direction part decides whether this policy is for ingress or egress control of the applied containers.
- *Protocol part*. The protocol part indicates the particular protocol to be allowed - UDP or TCP - and the allowed port of this policy.

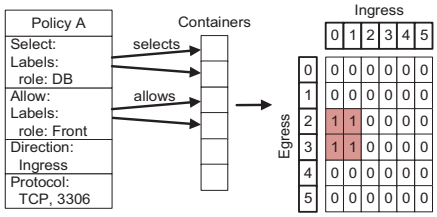


Fig. 3: An example of how policy works

An example is shown in Figure 3. If more than one policy select a same container, and the allow part is different, the final allowed containers are the union of all policies. It can be calculated by bitwise OR operation.

C. Network Constraints

We extract some constraints from the real production environment. Their violations is shown as follows. Each of them may cause serious risk, like data leak and improper isolation.

- *All reachable*. A container can be reached by all containers.
- *All isolated*. A container cannot be reached by any container.
- *User cross*. A container can reach other user's container in the container network.
- *System isolation*. A container is isolated with certain container, usually the kube-system container.
- *Policy shadow*. The connections built by a policy are completely covered by another policy, then this policy may be redundant.
- *Policy conflict*. The connections built by a policy are totally contradict the connections built by another.
- Other user-defined constraints which can be expressed by reachability.

The network constraints can be checked against the reachability matrix, and the exact procedure will be introduced in subsequent sections.

IV. Algorithm

In a typical container network, containers and network policies may be changed frequently, due to new service deployment, and dynamic traffic management, and so on. To ensure configuration update correctness, network service continuity, and quick trouble shooting, network verification needs to be conducted in a fast snapshot manner.

A. Reachability Bitmatrix Calculation

If reachability bitmatrix is calculated intuitively, the policies should be traversed. For each policy, we traverse every container to figure out whether it is selected and allowed by the policy. Suppose there are m policies and n containers, the time complexity of brute-force algorithm will be $O(mn)$. In practice, m and n usually have the same order of magnitude. As mentioned before, the scale of a container cluster can reach 500k containers [21], therefore a faster algorithm is demanded.

Despite the large number of containers and their policies in a cluster, the keys and values of the labels are scattered, thus the reachability matrix is usually sparse. The reachability bitmatrix calculation can be accelerated with a prefiltration to drop the impossible matching procedure between containers and policies. Therefore, the following prefiltration algorithm is proposed based on bitset mapping.

Firstly, hash the keys to a hashmap to associate every key to its bitset where each bit represents a container, as shown in Figure 4. Then, map keys of all the containers to there corresponding bitsets, so that the index of the containers are fully populated in the bitsets. Because each container has limited labels, the time complexity of this procedure is $O(n)$.

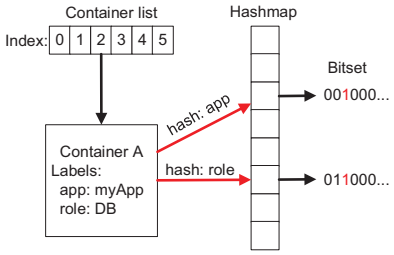


Fig. 4: Prefiltration of container labels

Secondly, hash the selector/allow keys of each policy to a bitset, then AND all resulting bitsets to obtain representation of containers, as shown in Figure 5. The time complexity of this procedure is $O(m)$.

Finally, traverse all set bit of the bitset. Determine whether the corresponding container is selected/allowed by the policy. This step is done by brutally comparing strings in labels and selectors. If matched, set the corresponding bit in the reachability matrix. Because the reachability matrix is sparse, there is few bits to be set, so the time complexity of this procedure is $O(1)$.

The complete algorithm is shown as Algorithm 1, with overall time complexity O(m + n).

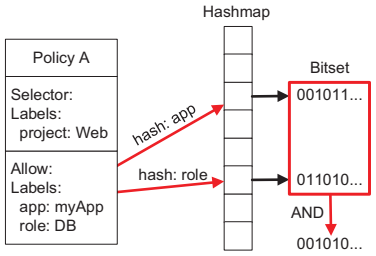


Fig. 5: Prefiltration of policies

Algorithm 1 Reachability bitmatrix calculation algorithm

Input: Container list C, policy list P
Output: Reachability bitmatrix M

```
1:  n = sizeof(C)
2:  m = sizeof(P)
3:  HashMap<String,BitSet> labelHash
4:  for i = 0 to n do
5:      for label in C[i].Labels do
6:          labelHash.hash(label.key).set(i)
7:  for i = 0 to m do
8:      BitSet SelectSet = new BitSet(1,n)
9:      for label in P[i].selectLabels do
10:         SelectSet = SelectSet&&labelHash.hash(label.key)
11:     BitSet AllowSet = new BitSet(1,n)
12:     for label in P[i].allowLabels do
13:         AllowSet = AllowSet&&labelHash.hash(label.key)
14:     for SetIndex in SelectSet do
15:         if !P[i] selects C[SetIndex] then
16:             SelectSet.clear(SetIndex)
17:     for SetIndex in AllowSet do
18:         if !P[i] allows C[SetIndex] then
19:             AllowSet.clear(SetIndex)
20:     for SetIndex in SelectSet do
21:         M.getRow(SetIndex)                    =
    M.getRow(SetIndex)||AllowSet
22: return M
```

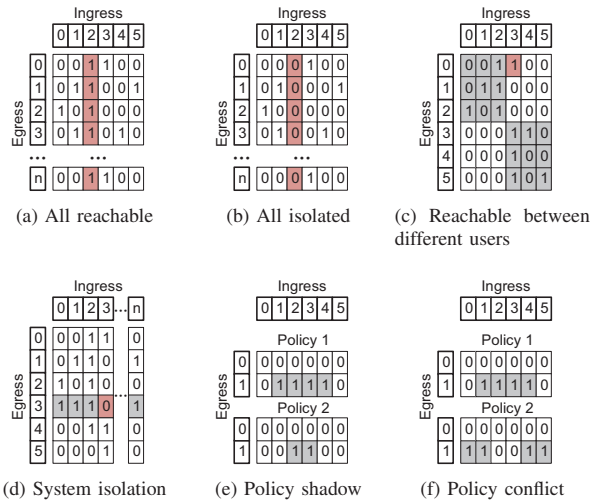
The space complexity of this algorithm is $O(n^2)$. Take the final bitmatrix as an example, the storage size it needs when the cluster size reaches the max - 500k containers - can be calculated as Equation 1.

$$S = 500000 * 500000(bit) = 250Gb = 31.25GB \quad (1)$$

It is a reasonable memory cost for clusters which can support 500k containers, for example, cloud data centers.

B. violation Check

With the reachability matrix, all of the constraint violations listed in in Section III C can be checked quickly. The violation check algorithm of each constrain violation, shown in Figure 6, is described respectively as follows.



(a) All reachable (b) All isolated (c) Reachable between different users

(d) System isolation (e) Policy shadow (f) Policy conflict

Fig. 6: Container network reachability bitmatrix model

- *All reachable*. This constraint violation can be found if there is a all-1-column in the reachability bitmatrix(Figure 6a). The algorithm is shown as Algorithm 2. Theoretically, the time complexity is $O(n^2)$. However, the bitwise operation can be done in batch mode, so in most cases, the time complexity is $O(n)$.

Algorithm 2 All reachable check

Input: Reachability matrix M
Output: All reachable container list C

```
1: n = sizeof(M)
2: for i = 0 to n do
3:     VeriSet = BitSet(1,n)//(0,n) for all isolated
4:     VeriSet = VeriSet∧M.getColumn(i)//|| for all isolated
5:     if VeriSet == ∅ then
6:         C.add(i)
7: return C
```

- *All isolated*. This constraint violation can be found if there is a all-0-column in the reachability matrix (Figure 6b). The algorithm is similar with the all reachable check procedure (As the annotation in Algorithm 2 shows), and has the same time complexity, $O(n)$.
- *User cross*. This constraint violation can be validated by comparing each row of the reachability bitmatrix with the user hash bitset with all bits corresponding to the containers belonging to the same user is set and the others clear. The user hash bitsets is calculated according to the user label. All containers should have the same user label, whose value differs according to the container owner. The user hash bitsets calculating algorithm is shown as Algorithm 3, and its time complexity is $O(n)$. The user cross check algorithm is shown as Algorithm 4, and its time complexity is $O(n)$.
- *System isolation*. This constraint violation can be found if a specified row has 0. The algorithm is shown as Algorithm 5, and its time complexity is $O(n)$.

Algorithm 3 User hash bitsets calculating

Input: Container list C, user label Key
Output: User hash bitsets map H

```
1: n = sizeof(C)
2: HashMap<String,BitSet> H
3: for i = 0 to n do
4:     H.hash(C[i].getValue(Key)).set(i)
5: return H
```

Algorithm 4 User cross check

Input: Reachability matrix M, User hash map H, Container List C
Output: User cross list U

```
1: n = sizeof(M)
2: for i = 0 to n do
3:     VeriSet = H.Hash(C.userLabel)
4:     VeriSet = !VeriSet
5:     VeriSet = VeriSet&M.getColumn(i)
6:     if VeriSet == ∅ then
7:         U.add(i)
8: return U
```

Algorithm 5 System isolation check

Input: Reachability matrix M, Container index i
Output: System isolate list C

```
1: n = sizeof(M)
2: VeriSet = BitSet(1,n)
3: VeriSet = VeriSet∧M.getRow(i)
4: if VeriSet != ∅ then
5:     for each set bit j in VeriSet do
6:         C.add(j)
7: return C
```

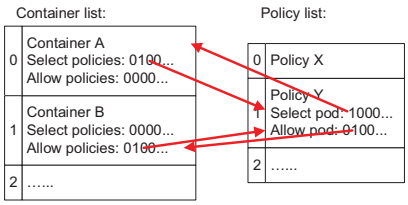


Fig. 7: Bidirectional container policy map

To find shadow and conflict among policies, a data structure named Bidirectional Container Policy Map (BCP map, Figure 7) is employed to record the corresponding relationship between containers and policies. The containers use bitsets to record ingress and egress allowed containers, and the policies select them. The policies use bitsets to record selected containers and allowed containers. The space complexity of

the bitsets is $O(n^2+mn)$. To store the reachability matrix, the space complexity is at least $O(n^2)$. So the space complexity of the bitsets has the same order of magnitudes with the reachability matrix. To generate this data structure, when generating reachability matrix, we store its selected containers and the allowed containers in each policy, and store the policies which select and allow itself in each container. The BCP map enables the system to handle incremental policy updates.

- *Policy shadow*. This constraint violation can be discovered by traversing all containers and comparing the selected and allowed containers among the policies which select the same container. If the coverage area of a policy can be covered by another one, then they are a policy shadow pair. The algorithm is shown in Algorithm 6, and its time complexity is $O(n)$.

Algorithm 6 Policy shadow check

Input: Reachability matrix M, Policy List P, Container List C, BCP map D
Output: Policy shadow list $PCList$

```
1: n = sizeof(M)
2: for i = 0 to n do
3:     polList = C.get(i).getSelectPolList()
4:     m = sizeof(polList)
5:     for j = 0 to m do
6:         for k = 0 to m do
7:             if j == k then
8:                 continue
9:             SetA = polList.get(j).getAllowC()
10:            SetA = SetA&polList.get(k).getAllowC()
11:            SetA = SetA∧polList.get(k).getAllowC()
12:            // For policy conflict check, replace the 3 lines
   of code above with
13:            //SetA = polList.get(j).getAllowC()
14:            //SetA = !SetA
15:            //SetA = SetA&polList.get(k).getAllowC()
16:            //SetA = SetA∧polList.get(k).getAllowC()
17:            if SetA == ∅ then
18:                C.add([j,k])
19: return C
```

- *Policy conflict*. The algorithm is similar with policy shadow. If the coverage area of a policy can be covered by the complementary set of another, then they are a policy conflict pair. The algorithm is shown as the annotations in Algorithm 6, and its time complexity is $O(n)$.

In conclusion, all predefined constraints can be verified with reasonable time complexity. Other user-defined constraints are all about reachability between limited number of specified containers, so the time complexity is also limited.

V. Implementation

We implemented a prototype of Kano based on Kubernetes. It consists of three components, namely matrix calculator, violation checker and fix advisor. The system framework is shown in Figure 8. The implementation details of each part are elaborated as follows.

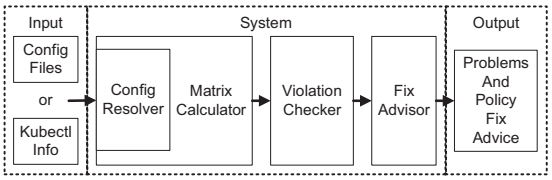


Fig. 8: System structure

The system is implemented by Java, and can be used as a library. A simple verification example is shown as follows.

```
Kano kano = new kano("/examples/test");
kano.generateMatrix();
kano.allVerify();
```

In this example, the program loads configuration files from a dictionary, then calculates the reachability bitmatrix and use it to check all constraint violations and provides fix advice.

A. Matrix Calculator

The matrix calculator takes the configuration files as input or get the configurations automatically by kubectl commands. After input, the configurations are resolved and transformed to system native data structures. Then the reachability bitmatrix will be calculated by the algorithm we mentioned before. Here are some details to deal with the Kubernetes features.

- *Kubernetes Namespaces*. The namespaces also have labels, and a namespace corresponds to many pods. The policies also select pods by namespaces. So we do the same prefiltration based on namespace selector first. Then cast namespace to pods to reduce computing time.
- *Default Reachable*. If a pod is not selected by any policy, it is all reachable. So we store every row of the reachability in the pod data structure. Set all bits to 1. At the first time a pod selected, we first set all bits to 0, and then do other operations.

B. Violation Checker

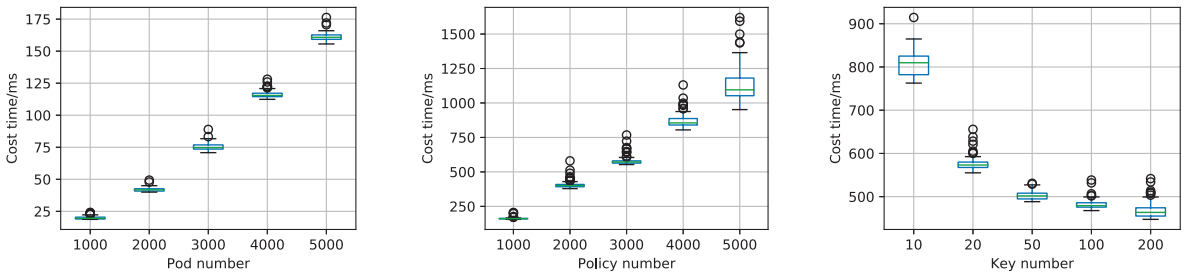
The violation checker uses the reachability matrix to check whether the constraints are all satisfied. If there is constraint violation, it will record the objects which cause the violation. Considering the production environment, Kano supports users to define new constraints by declarative language. A few possible options are shown as follows.

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

  ```
  kano.setUserLabel("User");
  ```

- *Link Constraint*. Users can add isolation/link constraint based on labels. If the link is built/not built, the system will warn about it. It can be added by the following code.

  ```
  kano.addLink("k1","v1","k2","v2");
  ```



(a) Pod number variable, other fixed (b) Policy number variable, other fixed (c) Key number variable, other fixed

Fig. 9: The relationship between reachability bitmatrix calculation time and pod number, policy number or key number

C. Fix Advisor

According to the violation check result, the fix advisor gives advices to cluster operators about how to correct the problems.

Although generate policy to fix potential risk can be simple, it may spoil the expressivity of the policies written manually. So the system does not directly modify the configurations, but give advice. The advices are shown as follows.

- Show all reachable and all isolated pods.
- Show policies which cause reachability between different users and isolation with kube-system.
- Advise deleting policies to solve policy shadow.
- Show conflicting policy pairs.
- Warn about other user defined constraints.

VI. Evaluation

The data set used for the evaluation of Kano is randomly generated by a Java program. The variable parameters are shown as follows.

- *Pod number*
- *Namespace number*.
- *Policy number*.
- *Pod label limit*. The maximum number of the labels attached to a pod.
- *Ns label limit*. The maximum number of labels which attached to a namespace.
- *Key limit*. The number of different keys.
- *Value limit*. The number of different values of each key.
- *Selected label limit*. The maximum number of select labels in a policy.
- *Allow ns label limit*. The maximum number of namespace allow labels in a policy.
- *Allow pod label limit*. The maximum number of pod allow labels in a policy.

To induce the relationship between pod/policy number and the cost time of reachability bitmatrix calculation, we first use the data sets whose policy number or pod number is variable and other parameters are fixed to do reachability bitmartix calculation. The result is shown in Figure 9a and Figure 9b. We can see that the calculating time grows linearly as the pod number or policy number grows. It differs with the theoretical analysis which indicates the time complexity is $O(m+n)$. However, when we keep the pod number and policy number fixed, and modify the key number, the cost time also changes (Figure 9c). The key number affects the probability of correlation between policies and pods. In practice the key number grows naturally when container cluster scales, and it significantly affects the calculation time.

TABLE II: The parameters of the dataset

Pod num	100	500	1000	5000	10000
Namespace num	5	10	20	50	100
Policy num	50	200	500	2000	5000
Pod label limit	5	5	5	5	5
Ns label limit	5	5	5	5	5
Key limit	5	10	20	50	100
Value limit	10	10	10	10	10
User limit	5	5	5	5	5
Selected label limit	3	3	3	3	3
Allow ns label limit	3	3	3	3	3
Allow pod label limit	3	3	3	3	3

So we generate experiment data sets with the parameters shown in the TABLE II. We keep the ratio of the number of policies and pods, and increases the number of possible key and value to keep the reachability matrix sparse. The time cost of reachability calculation is shown in Figure 10, and the space cost in Figure 11. All experiments are done on a CentOS server whose *CPU is Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz* with 20GB's RAM.

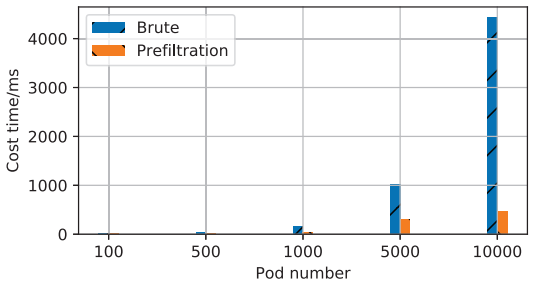


Fig. 10: Time cost of reachability matrix calculation

The time consumption of the naive algorithm has quadratic relationship with the cluster scale, and for the prefiltration algorithm is linear relationship, which proves the theoretical

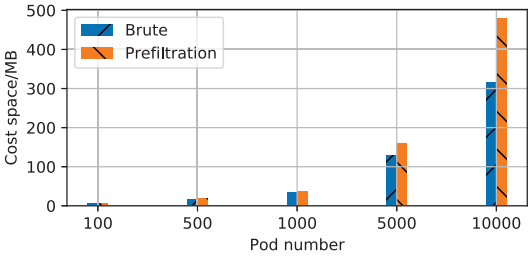


Fig. 11: Space cost of reachability matrix calculation

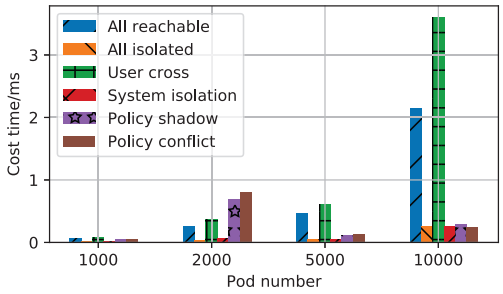


Fig. 12: Time cost of constraint verification

result. The prefiltration algorithm works much faster than the naive one, with a space complexity of the same order of magnitude. The reachability matrix of a 10,000-pod cluster can be calculated in a second with the prefiltration algorithm. Compared to the reachability bitmatrix calculation, the constraint verification procedure costs much less time (Figure 12).

VII. Conclusion

In this work, a label-based container network policy verification system, Kano, is proposed and evaluated for label based container network. Kano is implemented it for Kubernetes to establish as well as update reachability matrix in seconds, carry out network verification with predefined and user-defined constraints, and come up with fix advice.

In the future, we plan to design a distributed system based on this work to increase its speed and scalability. The distributed system will be integrated to the Kubernetes clusters and make Kano a pre-installed toolkit when deploying Kubernetes on new servers. Infrastructure verification can also be considered to make Kano a complete solution of container cluster network verification.

Acknowledgment

This work is supported by National Natural Science Foundation (No. 61872212) and National Key Research and Development Program (No.2016YFB1000101).

References

[1] Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Computing, 1(3), 81-84.

[2] Kubernetes. Network policies, 2020. Available at: https://kubernetes.io/docs/concepts/services-networking/network-policies/.

[3] Servos, D., & Osborn, S. L. (2017). Current research and open problems in attribute-based access control. ACM Computing Surveys (CSUR), 49(4), 1-45.

[4] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., ... & Turner, J. (2008). OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38(2), 69-74.

[5] Kazemian, P., Varghese, G., & McKeown, N. (2012). Header space analysis: Static checking for networks. In Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12) (pp. 113-126).

[6] Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., & Whyte, S. (2013). Real time network policy checking using header space analysis. In Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13) (pp. 99-111).

[7] Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P. B., & King, S. T. (2011). Debugging the data plane with anteater. ACM SIGCOMM Computer Communication Review, 41(4), 290-301.

[8] Khurshid, A., Zou, X., Zhou, W., Caesar, M., & Godfrey, P. B. (2013). Veriflow: Verifying network-wide invariants in real time. In Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13) (pp. 15-27).

[9] Yang, H., & Lam, S. S. (2015). Real-time verification of network properties using atomic predicates. IEEE/ACM Transactions on Networking, 24(2), 887-900.

[10] Gember-Jacobson, A., Viswanathan, R., Akella, A., & Mahajan, R. (2016, August). Fast control plane analysis using an abstract representation. In Proceedings of the 2016 ACM SIGCOMM Conference (pp. 300-313).

[11] Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., & Millstein, T. (2015). A general approach to network configuration analysis. In 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15) (pp. 469-483).

[12] Fayaz, S. K., Sharma, T., Fogel, A., Mahajan, R., Millstein, T., Sekar, V., & Varghese, G. (2016). Efficient network reachability analysis using a succinct control plane representation. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16) (pp. 217-232).

[13] Beckett, R., Gupta, A., Mahajan, R., & Walker, D. (2017, August). A general approach to network configuration verification. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication (pp. 155-168).

[14] Beckett, R., Gupta, A., Mahajan, R., & Walker, D. (2018, August). Control plane compression. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (pp. 476-489).

[15] Tian, B., Zhang, X., Zhai, E., Liu, H. H., Ye, Q., Wang, C., ... & Yu, D. (2019). Safely and automatically updating in-network ACL configurations with intent language. In Proceedings of the ACM Special Interest Group on Data Communication (pp. 214-226).

[16] Lopes, N. P., Bjørner, N., Godefroid, P., Jayaraman, K., & Varghese, G. (2015). Checking beliefs in dynamic networks. In 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15) (pp. 499-512).

[17] Plotkin, G. D., Bjørner, N., Lopes, N. P., Rybalchenko, A., & Varghese, G. (2016). Scaling network verification using symmetry and surgery. ACM SIGPLAN Notices, 51(1), 69-83.

[18] Bleikertz, S., Vogel, C., & Groß, T. (2014, December). Cloud radar: near real-time detection of security failures in dynamic virtualized infrastructures. In Proceedings of the 30th annual computer security applications conference (pp. 26-35).

[19] Probst, T., Alata, E., Kaâniche, M., Nicomette, V., & Deswarte, Y. (2013, September). An approach for security evaluation and analysis in cloud computing.

[20] Wang, Y. (2017). Tenantguard: Scalable runtime verification of cloud-wide vm-level network isolation (Doctoral dissertation, Concordia University).

[21] Cluster multi-tenancy | Kubernetes Engine Documentation | Google Cloud. Available at: https://cloud.google.com/kubernetes-engine/docs/concepts/multitenancy-overview.