ZenLib Implementation of Kano

Abstract

This project uses Zen Library [8] to implement Kano [1] – a label-based network verifier based on network policy [5] in Kubernetes clusters. Besides algorithms introduced in the paper, this project also fixes bugs/flaws that omitted by it. Since Zen is a Library implemented mainly by reflections, the performance of this project drops significantly compared to the original paper. Besides some important flaws in the paper, the performance introduced is considered not appropriate for benchmark usage. This project is more of a proof of concept for Zen-version Kano.

I. Introduction

Over the last few years, containers have become a popular way of deploying applications. It provides light-weighted virtualization to different applications. Easy-to-deploy/update/roll-back advantages make applications more scalable and robust. In spite of the numerous benefits of containers, one common concern in general is with respect to its security and privacy [2]. Orchestration tools such as Kubernetes(K8s) become more and more popular when we deploy our cloud-based applications. As Sysdig report [6] points out, 77% of their survey respondents are using K8s as deploy framework meanwhile 58% of the images are running as root users. The security and privacy of the whole system becomes more and more important. In order to make sure containers are only accessing its own data and running in a secure environment, network verification check is necessary. While traditional network verifiers [3], [4] analyze headers, ACL rules, forwarding tables to do network verification, most of K8s clusters are not applicable to use them, because K8s clusters mainly use label-based network control mechanisms. K8s network policy resource allows maintainer to control network traffic between different pods using labels [5]. The control plan of container network is largely different from traditional networks. Plus the future cluster scale could reach up to 500k containers [7]. New technologies would be required to deliver quick and accurate network verification for container-based networks.

There already exists several tools to do network verification in traditional networks e.g. NoD, HSA [3], [4]etc. However, verification for container network remains almost infertile. Kano [1] introduces some algorithms last year of doing network verification based on network policy resource for K8s clusters. Unlike real-time verification, it uses some prefiltration algorithms to save calculation time. ZenLib [8] is a research library and verification toolbox that allows for creating models of functionality. The Zen library has a number of built-in tools for processing these models, including a compiler, model checker, and test input generator. This project combines these two tools together to prove the possibility of integrating Kano into Zen which could be used for future network verification.

The rest of the report is organized as follows. Section II will introduce some context including how Kano works, the flaws of it and how we can fix those flaws. Section III will present the implementation of Kano using ZenLib. Section IV will introduce limitations of Kano and potential future improvement of it. Section V will summarize the whole project.

II. Background

A. Network Policy in Kubernetes

As documented by Kubernetes [5], Network Policies are an application-centric construct which allow us to specify how a pod is allowed to communicate with various network "entities" over the network. The entities that a pod can communicate with are identified through a combination of the following 3 identifiers:

- Other pods that are allowed (exception: a pod cannot block access to itself)
- Namespaces that are allowed
- IP blocks that are allowed

Detailed yaml example is shown as fig1. Pod selector selects pods to receive/send traffic (based on policy type). From/To field defines pods allowed to send/receive traffic to/from selected pods. This is selected by allowed namespaces + allowed pods combination where if namespace is not defined, the namespace of the policy would be used.

By default, pods are non-isolated; they accept traffic from any source. Pods become isolated once a network policy has selected them. The way a network policy selects pods is usually by the label of the pods, e.g. as marked in fig1, this policy selects pods with label:key=role and value=db.

After selecting pods, a policy also has to specify which pods are allowed to send/receive traffic to/from selected pods. This process is also mostly done by label selection as listed above. The IP Block selector is more like a traditional network ACL which might be affected by middle boxes in the cluster. And in terms of Kano, we only deal with label-based selectors (the IP selector could be implemented by traditional verifiers).

Worth to mention, network policy is additive and should never have conflicts with one another (additive whitelist). Also, it has two types, ingress and egress as indicates the direction of traffic that allows. PodA can send traffic to podB if and only if ingress of podB allows podB to receive traffic from podA and egress of podA allows podA to send traffic to podB.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

Fig. 1. network policy

B. Brief introduction of Kano

Kano is an efficient system for verifying large scale, container-level network policies at run time [1]. It uses bitmatrix to save time and space. Kano is designed to work with the label-based configurations. The core idea of Kano is to use a 2-D bitmatrix M to model a container network connection. pod_i is allowed to send traffic to pod_j, if and only if M[i][j] is set. Based on the bitmatrix, Kano has proposed various violation checkers.

- It can check if a pod is all reachable or all isolated.
- It can check if a pod could be reached from other users' pods.
- It can check if a pod is isolated from certain pods.
- It may also check if current network policy is problematic which we will discuss in later section.

To create the 2-D reachability bitmatrix, Kano proposed a prefiltration algorithm to save time based on the fact that the total number of labels are limited. In short, Kano maps all labels to a bitvector which has the length equal to number of pods in total. It uses bit operations to get pods with selected labels and compare the values at the end to save time compared to brute-force comparison.

C. Flaws and fix of original Kano

There are some flaws with the original paper:

- Time and space complexities are not accurate.
- Policy violations are not accurate.
- Some minor problems.

Before going into details, let's take a glimpse at the core algorithm in Kano – reachability matrix creation. All violation checkers and following algorithms are based on it. The algorithm [1] creates a label-pod mapping to do prefiltration. It traverses all labels (keys) in all pods. For each label, it maps to a bitvector of size n where n is the number of pods. Pod with this label will be set in the bitvector. After the label-pod map created, it traverses all the policies. To get selected pods by the policy, it takes a bitwise AND operation for all labels in the selector. The result bitvector has all the pods indices that have all the selected labels. Then a label match for every pre-selected pods will produce the final pods that are selected. This applies to the allow pods calculation as well. Since the bitwise operation for bitvectors takes O(1) time, the overall time complexity is expected as O(n+m) where n is the number of pods and m is the number of policies.

1) Time and Space: As introduced above, an O(n+m) time complexity is expected for the matrix creation where n is the number of pods and m is the number of policies. However, the paper only takes one type of network policy into consideration and creates a bitmatrix based on it. As described before, a certain connection is allowed if and only if the ingress and egress are both allowed. Creating one matrix (either ingress or egress or even both) only takes O(n+m) time, but to get the combined one from both ingress and egress takes much more time.

In order to get the correct result, we have to do the algorithm twice, one for ingress and one for egress. And a connection bit M[i][j] can be set if and only if egress[i][j] and ingress[j][i] are all set as shown in fig2. This leads to an unavoidable $O(n^2)$ algorithm to either transpose one of the matrices or traverse the two matrices together to get the final one.

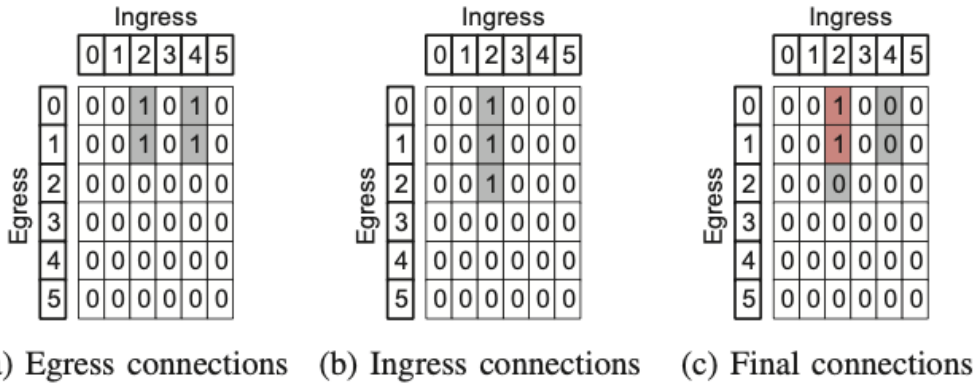

Fig. 2. Ingress and Egress matrices

Based on the fix above, it consumes at least 2 matrices to store necessary information. In the paper, it expects one matrix space consumption to store reachability information which for 500k pods is 31.25GB [1]. However, as described above, at least 2 matrices are required, which will lead to at least double the space as described in the paper. We are not certain if that would be a potential problem.

Another issue omitted by the original Kano is the bitvector batch operation for column of 2-D bitmatrix. It is not clear how they implemented the operation to get a bitvector from a spcific column out of a 2-D matrix. To achieve this goal, another transpose of the reachability matrix is required (which we can get a bitvector from a row rather than a column).

2) Policy violations: Kano wants to target policy conflict and shadow problems which are not accurately defined in the paper. As elaborated in the documentation [5], network policies could never be conflict with each other since they are all addictive. Hence the policy conflict detection is not implemented in our project.

In terms of policy shadow, the original algorithm doesn't produce correct results as expected. Take the following case as an example:

- PolicyA selects podA, allows podB.
- PolicyB selects podA and podC, allows podB.
- Correspondingly, PodA is selected by policyA and policyB.

With algorithm 6 introduced in [1], policyA and policyB will be considered shadowed by each other. However, by definition, only policyA should be considered shadowed by policyB because policyB's range covers policyA's, not the other way around.

To solve this problem, besides allowedPods in policy, selectedPods should also be used to calculate coverage of each policy. An improved version of the algorithm is used for Policy Shadow Checker as shown in Algorithm 1.

Algorithm 1 Improved Policy Shadow Check

Input: Container List C, Policy List P, BCP map D
Output: Policy shadow list PList

```
n = sizeof(C)
m = sizeof(P)
pairChecked = 2-D matrix(m × m)
PList = ListOfPolicyTuple()
for i = 0 to n do
    polList = C.get(i).getSelectPolicies();
    for j = 0 to m do
        if polList[j] not set then
            Continue
        end if
        for k = 0 to m do
            if pairChecked[j][k] is set then
                Continue
            end if
            if j == k then
                Continue
            end if
            if polList[k] not set then
                Continue
            end if
            SetA = polList.get(j).getAllowC();
            SetB = polList.get(j).getSelectC();
            SetA = SetA&polList.get(k).getAllowC()
            SetB = SetB&polList.get(k).getSelectC()
            SetA = SetA∧polList.get(k).getAllowC()
            SetB = SetB∧polList.get(k).getSelectC()
            if SetA == ∅ and SetB == ∅ then
                PList.Add([j,k])
            end if
        end for
    end for
end for
return PList
```

There is also another point worth attention. The policy shadow definition in Kano is not accurate as the ports in the network policies are ignored. For example:

- PolicyA targets podA and podB on port 8000
- PolicyB targets podA, podB and podC on port 8001

PolicyA is strictly not shadowed by policyB, whereas it is by Kano definition. But for now, the project will ignore the ports component as the original paper.

3) Other problems:

- Intra-traffic is not allowed in Kano if it is not explicitly allowed but in real K8s clusters intra-traffic will always be allowed. In other words, one pod can always access itself no matter what. This problem is fixed by always setting M[i][i] to true.
- Namespace is not explicitly shown in Kano. As elaborated in [5], namespace plays an important role in network policy selection. Kano doesn't explicitly show how namespace selector will be implemented. This part is implemented by the project using similar algorithms of pod selector.

III. Implementation

The full implementation is on Github. Only key parts are introduced here.

A. Terminologies

- **Reachability Matrix**: a two-dimensional matrix of n rows and n columns (n is the number of pods), it indicates connection status between each pod.
- **Ingress Reachability Matrix (or ingress matrix)**: one kind of Reachability Matrix that Matrix[i][j] indicates if pod_i can receive traffic from pod_j.
- **Egress Reachability Matrix (or egress matrix)**: one kind of Reachability Matrix that Matrix[i][j] indicates if pod_i can send traffic to pod_j.
- **BCP Matrices**: a set of matrices that maps bi-directional relations between pods and policies.

NOTE: matrix[i][i] will always be set for both ingress and egress (intra-traffic is always allowed).

B. Data Structures

To accurately simulate network policy resources in Kubernetes, three core data structures are created.

- Pod – pod resource in K8s cluster
- Namespace – namespace resource in K8s cluster
- Policy – policy resource in K8s cluster

Each data structure has their corresponding Zen extension methods to integrate with ZenLib. To iterate over labels, apart from $Zen < IDictionary < string, string >>$, an auxiliary $Zen < IList < string >>$ is used to store keys of labels. Since traversing a $Zen < IList >$ is much easier than $Zen < IDictionary >$.

However, the Zen Library has brought in an important defect that the number of labels can only be within the range of unsigned short (only ushort is supported).

C. Algorithms

As described above, reachability matrix creation is split into two parts. One matrix is created by ingress network policies and the other matrix is created by egress policies. For convenience, the first matrix is called ingress matrix and the second one is called egress matrix. So by definition, ingress matrix takes pods receiving traffic as row and pods sending traffic as column where egress matrix is the opposite. Detailed implementation is the same as Algorithm 1 introduced in [1]. Instead of returning one reachability matrix, our implementation returns five matrices: ingress and egress matrices plus three BCP matrices. The time complexity is $O(n^2 + m)$ where n is the number of pods and m is the number of policies. Space complexity is $O(n^2)$.

In Zen, no original bitvector structure is supported. Hence $Zen < IList < Bool >>$ is used as bitvector. And 2-D matrix is implemented by array of $Zen < IList >$. Also, Zen Library has brought in the limit of unsigned short range for matrix length which means only USHORT.MAX_VALUE(65535) number of pods are supported (less than 500k).

D. Violation Checkers

All reachable and isolated checks are implemented as Algorithm 2 introduced in [1]. One thing worth mentioning, in Kano, it uses getColumn to get a bitvector for ingress information where we use getRow instead ([1] uses egress matrix, getColumn of egress matrix equals getRow of ingress matrix).

User cross check is implemented as Algorithm 3&4 in [1]. The getColumn has been converted to getRow of ingress matrix as well.

System isolation check is implemented as Alogirithm 5 in [1]. Egress matrix is used to getRow.

Policy conflict as discussed above, is not implemented in the project. Policy shadow is implemented as improved algorithm shown above. There are some changes made to implement this checker. As ZenLib is hard to modify internal fields of objects (e.g. Zen objects of Pod and Policy are hard to modify their internal members), BCP maps are not stored in Pod and Policy as described in the paper. The bi-directional mapping is returned from reachability matrix creation function along with ingress and egress matrices.

E. Usage

All public methods of Kano has been implemented in this project (except the policy conflict check as discussed above) and can be found in KanoVerifier.cs.

- *CreateReachMatrix* takes all pods, policies and namespaces as input and returns 5 matrices including ingress and egress reachability matrices described above, three BCP matrices that map bi-directional relations between pods and policies (BCP matrices are used for policy shadow check).
- *AllReachableCheck* takes ingress reachability matrix and a flag (indicates if it's all reachable or all isolated check) as input and returns a list of indices of satisfied pods.
- *UserCrossCheck* takes ingress reachability matrix, user-pod matrix (this can be produced from *GetUserHash* function), all pods and key of user label as input and returns a list of indices of pods that can only be reached from the same users' pods.

- *SystemIsolationCheck* takes egress reachability matrix and a pod index as input and returns a list of indices of pods that can't be reached from the input pod.
- *PolicyShadowCheck* takes three BCP matrices as input and returns a list of tuples of indices where the second index is the policy shadowed by the first one.

All algorithms and violation checkers are covered with their corresponding unit tests where detailed usage and data generation can be found.

NOTE: All functions are Zen functions.

IV. Limitations and improvement

Kano uses bitmatrix to do prefiltration on network polices based on label which by nature could not solve selectors based on IP blocks. However, this part could be covered by traditional verifiers e.g. NoD, HSA, NetPlumber etc.

Currently, Kano doesn't get down to port granularity which leads to inaccurate policy shadow check as already discussed. Moreover, it can't accurately reflect real network connections. One potential fix is to store allowed ports in pods structures. While doing policy shadow checking, besides current work, we go into the ports stored in the data and confirm a shadow if the ports are also shadowed.

Currently, Kano can not be used to do dynamic verification although it was mentioned by the original paper to use BCP matrices to track dynamic updates. A rough idea is to track existing pods and policies in the bitvectors. This should require extra meta-data storage. But how to dynamically update the bitvectors with different scenarios requires more work. This will significantly bring more work to this Zen-version Kano as ZenLib's design does not support convenient data writes.

This project proves the concept of Kano with ZenLib. However, it can only work with simulated data structures as described above so far. In future, integrated into Kubernetes clusters would be an ultimate goal. One intermediate step could be read yaml configuration through interactive tools e.g. kubectl and convert the yaml file to the internal data structures in this project.

V. Summary

This project implements a Zen Library version of Kano which is a label-based network verifier specifically for Kubernetes clusters. It proves the possibility of integrating this verifier into Zen (which could be used to do traditional network verification). Besides implementing Kano, this project also fixes bugs/problems in original version. However, limited by Zen Library, this project has brought in some extra limitations i.e. number of labels and pods. Also, fixing bugs of the original one brings extra overhead to Kano which can't reach an O(n+m) time complexity.

In future, Kano and this project could be further improved from following aspects:

- Dynamic verification.
- Fix Zen limitations.
- Integrate with K8s clusters.

Acknowledgement

Thank Prof. George Varghese and Siva Kakarla for bringing up this project. Thank Siva Kakarla for the guidance and support during the whole project. Thank Rob Scott and Satish Matti for providing the reference of network policy rules in Kubernetes. Thank Kevin Yin and his teammates for dicussions about different versions of Kano.

References

[1] Yifan Li, Chengjun Jia, Xiaohe Hu, Jun Li, "Kano: Efficient Container Network Policy Verification", 2020.
[2] Harish Sekar, "Network verification in Kubernetes and Istio".
[3] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, George Varghese, "Checking Beliefs in Dynamic Networks". 2015
[4] Peyman Kazemian , George Varghese, Nick McKeown, "Header Space Analysis: Static Checking For Networks".
[5] Network Policies, URL: https://kubernetes.io/docs/concepts/services-networking/network-policies/
[6] Sysdig Report Spotlights Runtime Security Placeholder Image, URL: https://www.sdxcentral.com/articles/news/sysdig-report-spotlights-runtime-security/2021/01/?utm_campaign=websiteutm_source=sendgridutm_medium=email
[7] Cluster multi-tenancy, Kubernetes Engine Documentation, Google Cloud. URL: https://cloud.google.com/kubernetes-engine/docs/concepts/multitenancy-overview
[8] Zen Library. URL: https://github.com/microsoft/Zen