LETTER

Verikube: Automatic and Efficient Verification for Container Network Policies

Haney KANG[†a)] ***and*** **Seungwon SHIN**[†b)], ***Nonmembers***

SUMMARY Recently, Linux Container has been the de-facto standard for a cloud system, enabling cloud providers to create a virtual environment in a much more scaled manner. However, configuring container networks remains immature and requires automatic verification for efficient cloud management. We propose *Verikube*, which utilizes a novel graph structure representing policies to reduce memory consumption and accelerate verification. Moreover, unlike existing works, *Verikube* is compatible with the complex semantics of Cilium Policy which a cloud adopts from its advantage of performance. Our evaluation results show that *Verikube* performs at least seven times better for memory efficiency, at least 1.5 times faster for data structure management, and 20K times better for verification.

key words: *cloud, Linux Container, network verification, first-order logic, virtualization*

1. Introduction

Linux Container's [1] ability to provide better virtual environment capacity and resource efficiency let Linux Container be the de-facto standard for cloud systems. With the paradigm of application shifting from monolithic to microservice architecture, service provider initiate their services over clouds with containerized applications. In practice, Google, one of the most well-known IT vendors, newly creates billions of containers for its services, such as Youtube, Gmail, and the search engine [2].

Meanwhile, configuring cloud networks remains challenging due to the potentially large number of tenants and the complexity. Cloud providers are responsible for manually writing available and secure network policies that are thus error-prone. Existing works such as NoD [3], Plotkin et al. [4], Cloud Radar [5], Probst et al., [6] and TenantGuard [7] proposed several methods for verifying policies formatted routing rules.

However, policies of container-based cloud environments introduce unique challenges due to the unique ecology of containers. Containers are faster and scale better than VMs; the policy verification for containers also has to be a run-time and scale better than those for VMs [2]. Moreover, instead of the network policy operating within routers, policy enforcement shifts to direct packet processing [8], [9]. Kano [10] proposes outperforming matrix-based policy verification for containers compared to existing tools. However, (i) its memory consumption to manage the matrix representing connectivity between every pair of containers still burden host machines, and (ii) it only focuses on container-level connectivity without considering detailed packet types, including L4 and L7 protocols.

To overcome these challenges of verifying container cloud networks, we propose *Verikube*, a prototype tool for verifying network policy. *Verikube* introduces a graph to manage data of the cloud environment efficiently. Each node of the graph holds a group of formalized policies; the formalized policies are later used as an input of Yices2 [11], an SMT solver, to figure out cloud network-wide conflicts.

In summary, our contribution is twofold:

- We design an effective and fine-grained policy verification tool for containers that is scalable in system resource consumption and verification time while supporting a wide range of network policies (L3 to L7).
- We implement a prototype system of *Verikube* and demonstrate its effectiveness in real-world cloud environments.

2. Background and Motivating Example

2.1 Cilium Network Policy

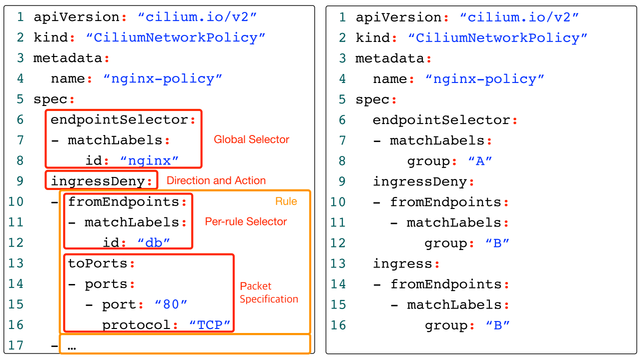
Cilium is a tool that controls network connectivity implemented with an extended Berkeley Packet Filter (eBPF) that runs sandboxed programs in the Linux kernel [12]. On the one hand, Cilium modules running in the kernel as an eBPF program accelerate Cilium's packet processing performance. On the other hand, the custom eBPF program of Cilium enables querying containers not only by their IP but also by various properties. These advantages of Cilium facilitate innovative management of containers, as (i) potential container clouds are large-scale, and (ii) volatility of containers frequently changes their IP addresses.

Cilium policies are either JSON or YAML, as shown in Fig. 1 (a). Cilium policies share a **Global Selector** across the policy. At the same level of policy hierarchy, **Direction and Action** such as "ingress" and "egressDeny" follow. Beneath, **Rule** then followed by **Direction and Action**. At the top level of each Rule, **Per-rule Selector** composed of multiple **Expressions** comes. Global Selector and Per-rule Selector become the source and destination selectors based on the Direction of the corresponding policy Rule. Finally, **Packet Specification**, which defines finer-granularity rules

Manuscript received May 23, 2022.
Manuscript revised August 9, 2022.
Manuscript publicized August 26, 2022.

[†]The authors are with Electrical Engineering, KAIST, Korea.
a) E-mail: haney1357@kaist.ac.kr
b) E-mail: claude@kaist.ac.kr

DOI: 10.1587/transinf.2022EDL8046


Global Selector
Direction and Action
Rule
Per-rule Selector
Packet Specification

```
apiVersion: "cilium.io/v2"
kind: "CiliumNetworkPolicy"
metadata:
  name: "nginx-policy"
spec:
  endpointSelector:
  - matchLabels:
      id: "nginx"
  ingressDeny:
  - fromEndpoints:
    - matchLabels:
        id: "db"
    toPorts:
    - ports:
      - port: "80"
        protocol: "TCP"
  - …
```

```
apiVersion: "cilium.io/v2"
kind: "CiliumNetworkPolicy"
metadata:
  name: "nginx-policy"
spec:
  endpointSelector:
  - matchLabels:
      group: "A"
  ingressDeny:
  - fromEndpoints:
    - matchLabels:
        group: "B"
  ingress:
  - fromEndpoints:
    - matchLabels:
        group: "B"
```

(a) (b)

Fig. 1 (a) Example of Cilium policy which defines connectivity between Nginx and Database Containers. (b) Example of policy conflict

such as L4 and L7, comes at the lowest level. We use the term **Treatment** to refer to a pair of Action and Packet Specifications.

2.2 Motivating Example

Although Cilium network policy provides an intuitive interface to define rules, human error hinders the availability of cloud networks. Kubernetes [13], one of the primary container orchestration tools provided by Google, supports containers up to 300K; the number of policies also scales similar containers, resulting in an increased possibility of the human factor. For example, Fig. 1 (b) shows the simplest form of policy error. The "ingress" rule of `policy1` allows container group A to communicate with container group B, but the "ingressDeny" rule precludes communication between the same pair of container groups. The main reason for conflicts is the design of policy language, in which identical policies potentially intersects the semantics of selectors.

3. Design

3.1 Requirements and Overview

Throughout the design, we consider several requirements and approaches to mediate each.
***Verikube* uses memory efficiently.** We noticed that containers filtered by the same selectors always have the same treatments and verification results. Denoting group of containers filtered by the same policies as *container class*, we design *Verikube* to perform verification for container class rather than verifying each container. With this approach, unlike existing approaches managing entire data of cloud, *Verikube* no longer needs to manage a list of containers and reduces the number of verification required. *Verikube* may complain about policy conflict that currently does not target any container. Yet, it is worth to be identified as it is a potential error for future containers.
***Verikube* efficiently queries treatments to guarantee runtime verification.** Querying treatment is a bottleneck since treatments responsible for a container class are scattered

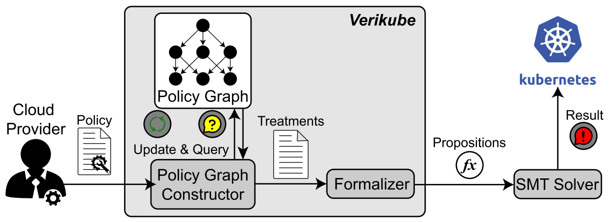


Fig. 2 The overall architecture of *Verikube*

around many policies in the cloud. Therefore, we introduce Policy Graph that immediately reflects changes in cloud and query treatments from selectors. As Policy Graph manages links between policy rules in which selectors are logically implying, *Verikube* traverse the graph to find treatments without exploring entire policies in the cloud.
***Verikube* verifies fine-grained L4 and L7 policies.** Packet specification of Cilium Policy is composed of simple action and various L4 and L7 protocols, yet existing solutions only verify L3 point-to-point connectivity. Thus, *Verikube* adopts SMT-solver Yices2 [11] by converting L4 and L7 policies into formal propositions. *Verikube* efficiently detects the conflict and opens flexibility for Cilium's features that will be extended in the future by adopting SMT-solver.

As shown in Fig. 2, *Verikube* is mainly composed of two key modules. Once policies change in the cloud, **Policy Graph Constructor**, an interface module to manage Policy Graph, updates policies and reflects the changes into Policy Graph. It then queries treatments corresponding to the container class of the new policy. Treatments returned by Policy Graph are then passed to **Formalizer** and transformed into propositions to be applied by SMT solver. The SMT solver identifies the verification result, and policy changes are finally delivered to the cloud if verification succeeds.

3.2 Policy Graph Constructor

Policy Graph $G(V, E)$ is a directed acyclic graph representing policy rules and their logical relation. We designed each node to represent a unique pair of container classes identified by a selector. Thus, we denote a node $v(S, D) \in V$ defined by selectors S and D, which denote source and destination container class, respectively. Besides, nodes are associated with treatments $T(S, D)$. Edge, meanwhile, is denoted by $e(v_i, v_j)$, meaning directed link from v_i to v_j. Each edge satisfies following proposition, $(S_i \implies S_j) \land (D_i \implies D_j) \implies e(v(S_i, D_i), v(S_j, D_j)) \in E$.

Policy Graph Constructor, an interface module of Policy Graph, performs three operations: (i) creating, (ii) removing, and (iii) querying policies. When cloud providers modify policies, we regard they remove and newly create policies.

Algorithm 1 describes a simple graph algorithm to reflect the cloud's policy when cloud providers introduce the new policy. The algorithm takes Policy Graph $G(V, E)$, Treatments T, and the new policy *Policy* consists of multiple entries of policy rules as an input. Each policy rule is composed of a pair of selectors S, D, and treatments. If a node identified by S and D exists, it extends treatments of the new policy to $T(S, D)$ (Line 13). Otherwise, Policy Graph Constructor creates the new node for an S and D pair (Line 3), then figures out any node that has logical implications with the new node (Line 4-10). Finally, it initializes treatments of a selector pair S, D (Line 11-13). In the case of dropping policies from the cloud, Policy Graph Constructor reversely performs the above process to remove nodes and edges from Policy Graph.

Once policies in the cloud change, verification is needed. *Verikube* queries a set of treatments by a pair of selectors that specifies the container class to be verified to perform verification. Policy Graph Constructor first access the node $v(S, D)$, then traverse edges to figure out nodes connected by $v(S, D)$ to collect all treatments affecting the particular container class.

3.3 Formalizer

From treatments queried from Policy Graph, Formalizer converts treatments to formalized format. As a prototype, *Verikube* only considers the L4 layer and action. For verification, we first introduce an uninterpreted function, $f(Proto, Port) \rightarrow Action$, to initially allow every type of packet [14]. The function's first argument is protocol and takes one of `TCP`, `UDP`, and `ANY`. Note that `ANY` semantically indicates "don't-care". However, a second argument, Port, is an integer value from 0 to 65535. We use a bit vector of size 16 to describe the data type of Port. At last, f returns `ALLOW` or `DENY`, an alias of True and False.

Formalizer converts each treatment to an assertion that limits the result of f for given inputs. For example, the treatment allowing Port 80 and TCP packets becomes formalized as $\forall p \in Port, \forall pr \in Proto, p = 80 \wedge pr = TCP \implies f(p, pr) = True$.

Algorithm 1: Policy Graph Construction when the new policy introduced

```
input  : $G(V, E)$, $T$, $Policy$
output: None
1  for $(S, D, Treatments) \in Policy$ do
2      if $v(S, D) \notin V$ then
3          $V.append(v(S, D))$
4          for $v(S', D') \in V$ do
5              if $(S' \implies S) \wedge (D' \implies D)$ then
6                  $E.append(v(S', D'), v(S, D))$
7              else if $(S \implies S') \wedge (D \implies D')$ then
8                  $E.append(v(S, D), v(S', D'))$
9              end
10         end
11         $T(S, D) \leftarrow \emptyset$
12     end
13     $T(S, D).extend(Treatments)$
14 end
```

4. Evaluation

4.1 Implementation

We only use Kano [10] for our evaluation since it is the only related work verifying container networks and is compatible with Cilium Network Policy for evaluation comparison. However, as Kano is not publicly available, we implemented its logic by referring to its algorithm described in the paper. To diversify *Verikube*'s ability, we experimented with *Verikube* under two different conditions. On the one hand, we set experiment settings *verikube-min*, which creates random policies to have a unique single expression to avoid having an edge between any nodes. On the other hand, we performed *verikube-randedge*, which randomly selects expressions and lets nodes having edges connected to other nodes.

We implement *Verikube* with 674 LoCs C code for high memory efficiency and performance. Kano, however, is not publicly opened; thus, we manually implemented its verification algorithm described in the paper [10]. We implemented Kano also with C for a fair comparison, having 515 LoCs. We evaluated both Kano and *Verikube* in the same environment. Our machine is composed of Intel i5-6600K 3.50 GHz quad-core CPU and two 8 Gb of M378A1G43EB1-CPB 2133 MHz DDR4 memory manufactured by Samsung. The host operating system is Ubuntu 20.04.3 LTS focal, and the corresponding kernel version is 5.4.0-89-generic.

4.2 Results

To empirically show advantage of *Verikube*, we measure memory usage, CPU cycle for data structure maintenance, and CPU cycle for verification with different number of containers and policies. We fixed number of policies to 10K for measuring resource consumption with different number of containers, whereas we fixed number of containers to 100K for measuring affects on number of policies. Our evaluations are done for five times, and average the resulting values to avoid dependency on the configuration of randomly selected node and expressions.

Memory efficiency: As described in Fig. 3 (a), heap memory usage of *Verikube* is independent to number of containers. This result come from our design choice that consumes

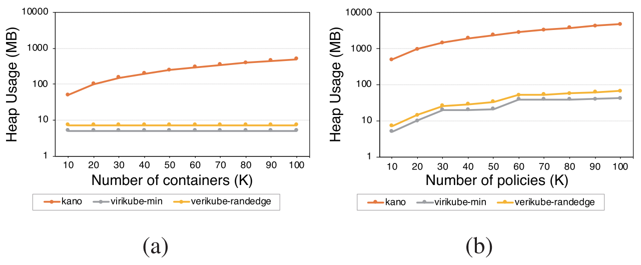


Fig. 3 Heap usage comparison of *Verikube* and Kano with different number of containers (a) and policies (b).

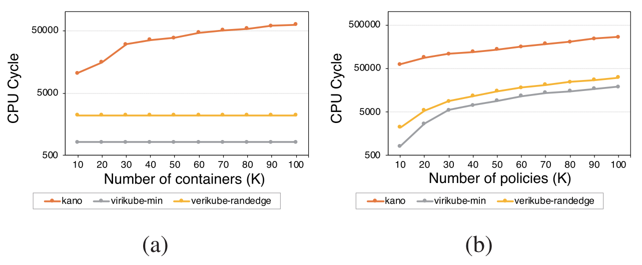


Fig. 4 CPU cycle used by *Verikube* and Kano to add new policy with different number of containers (a) and policies (b).

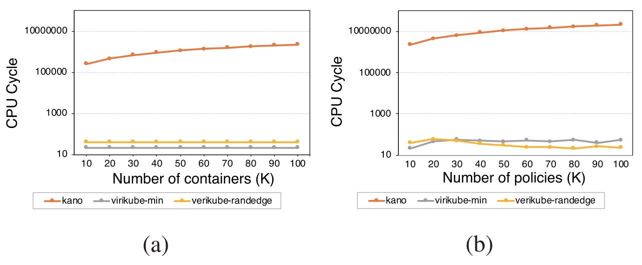


Fig. 5 CPU cycle used by *Verikube* and Kano for verification with different number of containers (a) and policies (b).

memory space only for storing policies unlike Kano which requires memory space for every container's policies regardless of policy duplication. By comparing *verikube-min* and *verikube-randedge*, *Verikube* requires a small amount of memory for edges; still, it is small enough and does not disrupt scalability. In the case of increasing number of policies, shown in Fig. 3 (b), both *Verikube* and Kano require additional memory. Nevertheless, as the growing rate of memory used by *Verikube* is much smaller than Kano, it does not exceed 100 MB even if the number of policies reaches 100K.

Data Structure Management: Figs. 4 (a) and 4 (b) describe CPU cycles used when cloud providers introduce the new policy into the cloud. CPU usage of *Verikube* when inserting new policy does not varies with number of containers, due to same reason of the result on memory usage. Thus, *Verikube* performs identically under different container numbers but is only affected by policy numbers. However, the performance of Kano gets worse when the number of containers and policies increase. Meanwhile, under the different number of containers and policies, *Verikube* performs more than 1.5 faster in any case. By comparing the case of none-edge and random-edge, there is a significant performance gap between the two evaluating scenarios caused by the iterating edges of all related node in Policy Graph.

Verification: As described in Figs. 5 (a) and 5 (b), the verification performance of *Verikube* outperforms Kano. As shown in the figures, the verification performance of *Verikube* does not vary by container and policy numbers. The SMT solver takes in charge of policy verification in *Verikube*, and its performance only depends on number of expression which is commonly low in much cases regardless of number of both containers and policies. Meanwhile, Kano iterates policies of every container, which fails to scale. In conclusion, the scalability of *Verikube* does not depend on the verification process itself.

5. Conclusion

We propose *Verikube* that efficiently manages data structure representing cloud environment in terms of memory and performance. With *Verikube*, cloud providers are available to figure out network-wide policy conflict in run-time under much more strict resource conditions. Our evaluation shows that *Verikube* only affected by policy numbers. *Verikube* performs better under the same number of policies, implying that *Verikube* is much more scalable and efficient.

Acknowledgements

This work has been supported by the Future Combat System Network Technology Research Center program of Defense Acquisition Program Administration and Agency for Defense Development. (UD190033ED)

References

[1] "Linux containers - LXC," https://linuxcontainers.org/lxc/introduction/, accessed May 15 2022.
[2] "Cloud functions," https://cloud.google.com/functions, accessed May 15 2022.
[3] N.P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), pp.499–512, 2015.
[4] G.D. Plotkin, N. Bjørner, N.P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," ACM SIGPLAN Notices, vol.51, no.1, pp.69–83, 2016.
[5] S. Bleikertz, C. Vogel, and T. Groß, "Cloud radar: near realtime detection of security failures in dynamic virtualized infrastructures," Proc. 30th annual computer security applications conference, pp.26–35, 2014.
[6] T. Probst, E. Alata, M. Kaâniche, and V. Nicomette, "An automated approach for the analysis of network access controls in cloud computing infrastructures," Network and System Security, Lecture Notes in Computer Science, vol.8792, pp.1–14, Springer International Publishing, Cham, 2014.
[7] Y. Wang, T. Madi, S. Majumdar, Y. Jarraya, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi, "Tenantguard: Scalable runtime verification of cloud-wide vm-level network isolation," Proc. 2017 Network and Distributed System Security Symposium, 2017.
[8] "Cilium - Linux native, API-aware networking and security for containers," https://cilium.io, accessed May 15 2022.
[9] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin, "{BASTION}: A security enforcement network stack for container networks," 2020 USENIX Annual Technical Conference (USENIX ATC 20), pp.81–95, 2020.
[10] Y. Li, C. Jia, X. Hu, and J. Li, "Kano: Efficient container network policy verification," 2020 IEEE Symposium on High-Performance Interconnects (HOTI), pp.63–70, IEEE, 2020.
[11] "The Yices SMT Solver," https://yices.csl.sri.com, accessed May 15 2022.
[12] "eBPF - Introduction, tutorials & community resources," https://ebpf.io, accessed May 15 2022.
[13] "Kubernetes," https://kubernetes.io, accessed May 15 2022.
[14] "Yices Manual Version 2.6.4," https://yices.csl.sri.com/papers/manual.pdf, accessed May 23 2022.