In-depth analysis of Kubernetes manifest verification tools for robust CNF deployment

Boubacar Diarra, Karine Guillouard, Meryem Ouzzif, Philippe Merle, Jean-Bernard Stefani

▶ **To cite this version:**

Boubacar Diarra, Karine Guillouard, Meryem Ouzzif, Philippe Merle, Jean-Bernard Stefani. In-depth analysis of Kubernetes manifest verification tools for robust CNF deployment. ICIN 2024 - Conference on Innovation in Clouds, Internet and Networks, DNAC, Mar 2024, Paris, France. pp.1-8. hal-04421758

HAL Id: hal-04421758
https://inria.hal.science/hal-04421758

Submitted on 28 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

In-depth analysis of Kubernetes manifest verification tools for robust CNF deployment

Boubacar Diarra*†, Karine Guillouard*, Meryem Ouzzif*, Philippe Merle†, Jean-Bernard Stefani‡
*Orange Innovation Rennes, †Inria Lille, ‡Inria Grenoble-Alpes, France
{boubacar1.diarra, karine.guillouard, meryem.ouzzif}@orange.com, {philippe.merle, jean-bernard.stefani}@inria.fr

***Abstract*—Kubernetes has emerged as the leading platform for orchestrating cloud-native network functions (CNF). The description of the hardware and software architecture of systems to be deployed with Kubernetes is defined in configuration files called manifests. In order to address the errors that these manifests may contain, manifest verification tools have been developed. However, evaluating and selecting the most appropriate verification tool for a given project is a difficult task. To facilitate this evaluation process, we identified and characterized the most relevant verification capabilities, useful also in the CNF context. The selected capabilities serve as criteria for the evaluation of manifest verification tools. In total, we have identified 13 criteria, like the variety of verification types or the ability to carry out JSON Schema-based verification. The identified criteria were used to evaluate 16 of the most commonly used verification tools in the Kubernetes community. This evaluation revealed several shortcomings in the existing tools.**

***Index Terms*—Kubernetes, manifests verification, verification tools, evaluation criteria, CNF, robust CNF deployment.**

I. Introduction

The growing need for flexibility, efficiency and cost reduction in the hosting of their network infrastructure components is pushing telecoms operators to adopt new approaches. One of these approaches is Network Functions Virtualization (NFV). This approach involves separating network functions (router, firewall, etc.) from the underlying hardware infrastructure [1]. In the early stages of their adoption, virtualized network functions (VNFs) were deployed on virtual machine (VM)-based infrastructures. However, with the advent of cloud computing, these functions are now containerized and referred to as Cloud-native Network Functions (CNF). A concrete example of this approach is the free5GC solution [2]. Due to the complexity of managing containerized systems, it is imperative to use orchestrators. A container orchestrator automates the deployment, management, scaling and networking of containers. Among the orchestrators available in the market, Kubernetes [3] has emerged as the de facto standard. Kubernetes uses declarative configuration files in YAML or JSON format, called manifests, to define the configuration (i.e., the desired state) of the system to be deployed.

Like any other configuration file, Kubernetes manifests can contain errors. These errors can have various consequences, including failure of the deployment process, security issues, inaccessibility or failure of a CNF instance [4], [5]. To prevent errors in manifests, different tools like Datree [6], Checkov [7], Gatekeeper [8] or Kyverno [9] have been developed. However, although these tools are available as open source, evaluating their capabilities, behavior and determining the level of confidence to place in them is difficult. This difficulty stems not only from the lack of clarity and precision in the documentation for these tools, but also from the fact that many of them have been developed without any real effort to standardize either the nature of the errors they target to address, or their verification capacity. As a result, finding the right tool for a given verification context can be challenging.

Our goal in this paper is to provide an in-depth analysis of Kubernetes manifest verification tools and, as a result, to facilitate the selection of the most appropriate Kubernetes manifest verification tool for a given context.

The paper is structured as follows: Section II presents an overview of a Kubernetes manifest, including some definitions of terms. Section III shows some examples to highlight the importance of using verification tools. Section IV presents the verification capabilities we have selected as evaluation criteria. Section V presents the results of applying our criteria to 16 manifest verification tools. Furthermore, we will discuss the shortcomings of the analysis tools. Section VI will cover our work to address the identified shortcomings. Finally, related work and conclusions are presented in Sections VII and VIII, respectively.

II. Manifest Overview and definitions

A Kubernetes manifest takes the following form:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: upf-example
spec:
  containers:
  - name: upf
    image: upfImage
...
```

Fig. 1. Pod manifest for instantiating a UPF (User Plane Function)

This manifest allows the creation of an instance of a 5G UPF (User Plane Function) [10] container. Furthermore, the term `field` is used to describe a particular property of a manifest. `metadata.name` and `spec.containers[_].name` are examples of fields in this manifest.

In addition, we consider the following definitions.

`Cluster`: Refers to a server, or group of servers, where the Kubernetes control plane and system instances run.

`Resource`: Is a description of a system component. Figure 1 illustrates a Pod resource. This kind of resource is used to manage the instantiation of one or more containers.
`Verification rules`: Refer to the constraints with which a manifest must comply.
`Built-in verification rules`: Refer to verifications performed natively by a tool.

III. Motivating Examples

As mentioned earlier, manifests can contain errors. To illustrate this, let's take the following examples.

```yaml
#Ingress resource definition
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: wara-ingress
spec:
  rules:
    - host: /wara-front
      http:
        paths:
          - path: /app
            pathType: Prefix
            backend:
              service:
                name: warra-service
                port:
                  number: 80
---
#Service resource definition
apiVersion: v1
kind: Service
metadata:
  name: wara-service
...
```

Fig. 2. Example of binding error

1) Binding error: `Ingress` and `Service` are two kinds of Kubernetes resource. `Service` resources are used to manage traffic within the cluster. By default, a `Service` is restricted to the cluster, meaning that only cluster-internal applications can access it. There are several ways of avoiding this limitation, including the use of `Ingress`. In Kubernetes, an `Ingress` resource enables traffic to be routed from outside the cluster to one or more `Services` inside the cluster. To establish a binding between an `Ingress` and a `Service`, the value of the `spec.rules[_].http.paths[_].backend.service.name` field of the `Ingress` must be equal to the value of the `metadata.name` field of the `Service`. As a result, any inaccuracy in the specification of these two fields will lead to an interruption in the traffic flow. Figure 2 demonstrates a situation where a binding error occurs due to a typography issue in the `spec.rules[_].http.paths[_].backend.service.name` value, more precisely, the word `'warra'` contains an extra `'r'`. The consequences of this kind of error can be varied, and include the unnecessary and costly use of resources like CPU and RAM, since a number of containers will continue to run without being reachable nor usable.

2) Security error: In the manifest of Figure 3, the field `securityContext.capabilities.drop` is used to specify a list of Linux capabilities to be removed from the set of capabilities available to a process inside the container. In this case, we are excluding the ability to run the Linux `KILL` signal. Regarding the `securityContext.privileged` field, when set to true, this indicates that the container has the same privileges as the host system, enabling it to have full access to the host functionalities. This manifest can lead to a security error for the following reason: the `securityContext.privileged` field takes priority over the `securityContext.capabilities.drop` field. So, whatever capability we try to remove using the `securityContext.capabilities.drop` field, it will be systematically reassigned as soon as `securityContext.privileged` is set to true. Thus, there is a cause-and-effect relationship between these two fields, such that when the `securityContext.capabilities.drop` field is set, the `securityContext.privileged` field must be set to false for the instructions in the `securityContext.capabilities.drop` field to be taken into account. This error has the consequence of increasing the attack surface of the cluster, which compromises its security, due to the fact that the user is granted privileges that they should not have had, and which were supposed to have been revoked. Since Kubernetes does not detect this error during deployment, it will only be detected in the event of an attack, when the system is already in production and it is too late to act.

```yaml
containers:
- name: my-container
  image: my-image:latest
  securityContext:
    privileged: true
    capabilities:
      drop: ["KILL"]
  livenessProbe:
    httpGet:
      path: /
      port: 80
    initialDelaySeconds: 3
    periodSeconds: 3
```

Fig. 3. Pod manifest with defined `capabilities.drop`

In general, it is essential to perform verifications against the contents of the `securityContext` section in a Kubernetes manifest. To illustrate, in one of the Proof of Concepts (PoCs) of the NEPHIO project [11], which is based on free5GC, the UPF CNF encountered an issue when initializing a filter on the IP table [5]. This issue arose because the `NET_ADMIN` capability had not been included in the list of capabilities. Like binding errors between `Ingress` and `Service`, finding this kind of error in production leads to inefficient and costly use of resources, as well as potential degradation of quality of service. [4] gives a good overview of security errors related to the Kubernetes manifests.

3) Availability error: To illustrate this kind of error, let's take the example of the liveness probe. As the name suggests, the liveliness probe is used to check whether a container is

still working properly, or whether it has reached some kind of deadlock and needs to be restarted. Although a liveness probe is important, it is possible to deploy containers without specifying it, which can lead to a decrease in availability. Therefore, it is recommended as best practice to ensure that practitioners have defined it in their manifests, more specifically, ensure that `livenessProbe` field is defined as illustrated in Figure 3.

The examples above show how error-prone manifests can be, and how these errors can impact the robustness, security and availability of a CNF-based system. As a result, Kubernetes telco practitioners need to use verification tools.
Unfortunately, in terms of verification capabilities, the tools are not equivalent. For example, the above binding error cannot be detected by Kubeval, Datree, and Config-lint. Even if a tool can detect a specific error, there is no guarantee of the accuracy of the performed verification. Let's consider the case of Datree's verification rule, which aims to ensure that containers run as non-root user. Upon analyzing the implementation of this rule, it is found that Datree only verifies if the fields `spec.securityContext.runAsNonRoot` and `spec.containers[_].securityContext.runAsNonRoot` are set to true. However, even if these two fields are set to true, the container can still run as root if the `spec.securityContext.runAsUser` field is set to 0. Due to this missing check, Datree's verification rule is inaccurate.
For all these reasons, it is essential to establish a set of evaluation criteria for manifest verification tools, while carrying out an in-depth study of existing verification tools.

IV. Proposed Evaluation Criteria

Figure 4 shows our selected criteria. This figure was created using the feature modeling technique [12], and the term `OR group` in the figure legend means that at least one of the features in this group must be selected if the parent feature is selected. So, Figure 4 can be read as follows: a tool is characterized by at least the type of its verification, its ability to access external data, its ability to access internal data, etc.

A. Variety of verification types

The first evaluation criterion that we considered concerns the variety of verification types that a tool can perform. Indeed, each type of verification is related to a specific error kind or situation, and therefore, the wider the variety of verification types offered by a tool, the greater its ability to detect different error natures. In this paper, we distinguish three types of verification: field-based verification, structure-based verification and architecture-based verification. These three types are defined as follows.

1) Field-based verification: This type of verification aims to validate the conformity of a field of a resource with respect to a set of constraints.
Example: for availability needs, a practitioner may decide that a minimum of 5 instances is required for a CNF. In a Kubernetes manifest, the `spec.replicas` field is used to indicate the desired number of instances. Therefore, the condition to check would be that `spec.replicas >= 5`.
The example mentioned in Section III-3 regarding the fact that `livenessProbe != null` illustrates another case of field-based verification.

2) Structure-based verification: Structural constraints refer to dependency and cause-and-effect relationships between the fields of a same resource. Thus, a structure-based verification aims to validate the conformity of two (or more) fields of a resource with respect to such relationships.
Example: two update strategies are available in Kubernetes: `RollingUpdate` and `Recreate`. While the `RollingUpdate` strategy gradually replaces old instances with new ones to minimize service interruptions, the `Recreate` strategy takes a different approach: it simultaneously replaces all existing instances with the new version. The field `spec.strategy.type` is used to define the strategy. In addition, if the `spec.strategy.type` field is set to `RollingUpdate`, additional parameters can be specified in the `spec.strategy.rollingUpdate` field. Since these parameters are only required when `spec.strategy.type` is equal to `RollingUpdate`, it is necessary to impose the constraint that `spec.strategy.rollingUpdate` must be null if `spec.strategy.type` is equal to `Recreate`. To resume:

```
if spec.strategy.type = Recreate then
↪  spec.strategy.rollingUpdate = null
```

The example mentioned in Section III-2 regarding the priority between the `securityContext.privileged` and `securityContext.capabilities` fields is another case illustrating structure-based verification.

3) Architecture-based verification: Architecture-based verification is a generalization of structure-based verification, by extending its scope to include multiple resources. In other words, its purpose is to analyze dependency or cause-and-effect relationships between fields from different resources.
The example mentioned in Section III-1 regarding binding errors between an `Ingress` and a `Service` is a good illustration of architecture-based verification.

B. External data access capabilities

This criterion refers to the ability of a tool to access data outside the cluster, for instance via HTTP requests, and use these data as an operand in the verification process. The use of external data sources, like databases, Web Service or APIs, enables manifests to be verified against data from maintained reference sources.

C. Internal data access capabilities

This criterion refers to the ability of a tool to use resource instances already present in the cluster as operands in the verification process. This capability is important, as it reflects the scope and accuracy of architecture-based verification rules of a tool. Indeed, it is impossible to write architecture-based verification rules without having access to resources (manifests) other than the resource under test. These additional resources can be obtained in two distinct ways: either in the form of resources not yet instantiated (in the form of YAML files, but not yet deployed in the cluster), or in the form of resources already instantiated and currently running in the

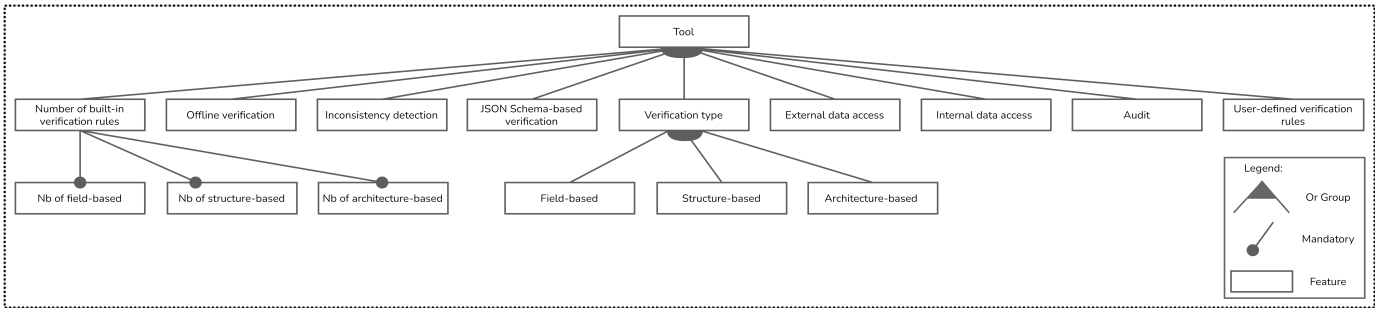

Fig. 4. Feature model of Tool Verification Capabilities

cluster.
Each of these two methods of accessing resources is required. To illustrate, let's go back to the example, we mentioned in Section III, of binding errors between an `Ingress` and a `Service`. This error can be detected by performing the following verifications:

1) When creating the `Ingress`, check that there is a resource definition of type `Service`, not yet instantiated, but which will be instantiated later and whose `metadata.name` field corresponds to the `spec.rules[_].http.paths[_].backend.service.name` field of the `Ingress`.
2) When creating the `Ingress`, check that there is a `Service` instance whose `metadata.name` field matches the `spec.rules[_].http.paths[_].backend.service.name` field of the `Ingress`, by accessing the cluster internal data.

Without one of these two verification steps, the rule would lack precision, which could lead to a number of false negatives.
We have only considered the second step as a criterion, as the first step is quite simple to implement and is already implemented in the majority of tools offering architecture-based verification.

D. Audit capabilities

This criterion refers to the ability of a tool to re-evaluate instances already present in the cluster, either periodically or as soon as a change occurs in the verification rules. This capacity is important, because it ensures that the content of the cluster always remains in accordance with the verification rules.

E. JSON Schema-based verification capabilities

A JSON Schema [13] is a specification that describes the structure and constraints of a data model. For each field in the model, it defines the expected data type, value constraints and relationships with other fields. To illustrate, the schema in Figure 5 specifies that the type of the `containerPort` field is an integer.
Thus, this criterion refers to the ability of a tool to check if a manifest conforms to the schema of the Kubernetes Resource Model (KRM), whether this schema comes from the official Kubernetes source or from a third-party source. This capability is important because it ensures that the arrangement

```
{
  "containerPort": {
    "description": "Number of port to expose
        on the pod's IP address.",
    "format": "int32",
    "type": "integer"
  }
}
```

Fig. 5. Example of a JSON Schema

of manifest fields and their types are consistent.
As an example, a verification against the official Kubernetes schema [14] will reveal two errors in the manifest of Figure 6. The first concerns the omission of the `metadata` field, while the second relates to the incorrect type of the `containerPort` field, which should be an integer and not a string.

```
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: phd-pod
    image: nginx:latest
    ports:
    - containerPort: "nginx-port"
```

Fig. 6. Example of erroneous Pod manifest

F. User-defined verification rules support

This criterion refers to the ability of a tool to enable users to define their own verification rules. This capability is important because it illustrates the flexibility and adaptability of the tool to the user needs. Indeed, verification needs can vary considerably from one company or user to another. By allowing users to define their own rules, the tool becomes more flexible and can adapt to a wide range of use cases.

G. Inconsistency detection capabilities

This criterion concerns the ability of a tool to detect inconsistencies between verification rules, whether they are specified by the user or integrated into the tool.
The following two rules provide a simple illustration of inconsistencies that can occur between two verification rules:

- Rule 1: for security reasons, each CNF deployed on the cluster must be configured so that all its capabilities are disabled. In other words: `capabilities.drop = ["ALL"]`.
- Rule 2: For the UPF's CNF to work properly, it must be able to modify the IP table. This means that the following conditions must be met: `capabilities.add` must contain `NET_ADMIN` and `capabilities.drop` must not contain `NET_ADMIN`.

With the two previous rules, a manifest which respects Rule 2 will be in contradiction with Rule 1 and vice versa. In other words, no manifest can satisfy a set containing the two previous rules.

The tools we considered in our study are not capable of identifying such inconsistencies, as they evaluate each rule individually. Indeed, to detect inconsistencies in a set of rules, it is necessary to verify the following statement: there is a manifest that can satisfy every rule in the set of verification rules. Regardless of the method used to verify this consistency, an individual evaluation of the rules does not allow us to conclude that such a manifest exists.

Inconsistencies in a set of rules, combined with the way the tools operate, generally lead to deployment failures. Indeed, since there is no manifest that can satisfy all the rules, whatever manifest is used, there will always be at least one rule to which the manifest does not conform. As a result, the manifest verification phase remains an unachievable step. Since the number of verification rules can be significant, detecting these inconsistencies quickly becomes a laborious and time-consuming task. Therefore, it is crucial that a verification tool can guarantee the consistency of its built-in rules, as well as user-defined rules.

H. Offline verification capabilities

This criterion refers to a tool ability to carry out its verifications without needing to access an operational Kubernetes cluster. There are two main advantages to offline verification:

a) Verify manifests locally to prevent any possible alteration of the cluster execution context.

b) Allow the verification process to be integrated into a CI/CD pipeline.

I. Number of built-in verification rules

In general, the built-in verification rules are composed of best practices for specifying manifests as well as security vulnerabilities identified within the Kubernetes community.

In our paper, we did not only count the tools built-in verification rules, we also classified them according to their type, i.e. field-based, structure-based and architecture-based.

The number of verification rules and the classification we made of them are an indicator of the coverage of a verification tool. To illustrate, consider the Datree tool, which has about a hundred built-in verification rules. However, none of these rules are able to detect architectural errors. Therefore, despite the amount of rules built into Datree, its ability to identify one of the most critical error categories remains insufficient.

V. Tools evaluation

A. Evaluation methodology

We followed a three-stage evaluation process: (1) analysis of the source code, (2) analysis of the documentation, and (3) experimentation.

Table I shows the results of applying this process to the most commonly used manifest verification tools in the Kubernetes community.

B. Precisions in regard to Table I

Before starting a qualitative analysis of Table I, the following clarifications should be provided.

1) Precisions regarding field-based, structure-based and architecture-based verification: In our study, we considered that a tool is able to perform field-based, structure-based and architecture-based verification only when its specification language allows the specification of such verification rules. So, even if a tool has built-in architecture-based verification rules, if it does not allow the specification of architecture-based verification rules via its specification language, we will not grant this capability to that tool. Kube-Score is a good example.

2) Precisions regarding Datree, Checkov and Config-lint architecture-based verification capabilities: These three tools are unable to perform architecture-based verification as they are limited to processing a single resource at a time. As explained in Section IV-C, it is impossible to write architecture-based verification rules without having access to (manifest) resources other than the resource under test.

3) Precisions regarding KubeLinter architecture-based verification capabilities: The expressiveness of the tool specification language is very limited. Indeed, the tool provides the user with a set of templates, and new rules can only be added by modifying these templates. As a result, its field-based, structure-based and architecture-based verification capabilities are limited.

4) Precisions regarding Gatekeeper, Kyverno architecture-based verification capabilities: Like Datree, Gatekeeper and Kyverno can only take a single resource as input. However, it retains its architecture-based verification capability, for the following reason: the ability to access resource instances present in the cluster. As a result, it is possible to verify causal or dependency relationships between the fields of the instances and those of the resources provided as input.

5) Precisions regarding Kubeconform and Kubeval: We assign the capability discussed in Section IV-F to these two tools, as they allow verification of user manifests using custom JSON Schemas, i.e. schemas where the user has defined their own verification rules. It is also important to note that JSON Schemas are limited to field-based and structure-based verification.

6) Precisions regarding Terrascan: Terrascan does not perform schema-based verification, but during the deserialization of YAML in Go, it detects some errors related to the type and arrangement of fields.

TABLE I
TOOL EVALUATION RESULTS

Tool	Variety of verification types [based on the rules specification language capabilities]			External data access capabilities	Internal data access capabilities	Audit capabilities	JSON Schema-based verification capabilities	User-defined verification rules support		Offline verification capabilities	Inconsistency detection capabilities	Number of built-in rules		
	Field-based	Structure-based	Architecture-based					Is Supported	Rule specification language			Nb of field-based	Nb of structure-based	Nb of architecture-based
Copper [15]	○	○	○	×	×	×	×	○	JavaScript	○	×	0	0	0
Datree [6]	○	○	×	○	×	○	○	○	Rego	○	×	100	11	0
KubeLinter [16]	○	○	○	×	×	×	×	○	YAML / Go	○	×	33	9	9
Kubevious [17]	○	○	○	×	○	○	○	○	Kubik	×	×	1	0	35
Kube-score [18]	×	×	×	×	×	×	×	×	N/A	○	×	11	13	11
Polaris [19]	○	○	○	×	×	×	○	○	YAML / JSON Schema / Go Template	○	×	25	4	7
Checkov [7]	○	○	×	○	×	×	×	○	Python / YAML	○	×	50	3	0
Config-lint [20]	○	○	×	○	×	×	×	○	YAML / JMESPath	○	×	0	0	0
Conftest [21]	○	○	○	○	×	×	×	○	Rego	○	×	0	0	0
Terrascan [22]	○	○	○	○	×	×	×	○	Rego	○	×	40	3	0
Trivy [23]	○	○	○	○	×	×	○	○	Rego	○	×	46	18	0
Gatekeeper [8]	○	○	○	○	○	○	×	○	Rego	×	×	37	0	5
Kyverno [9]	○	○	○	○	○	○	×	○	YAML / JMESPath	○	×	128	26	22
Kubeconform [24]	○	○	×	×	×	×	○	○	JSON Schema	○	×	0	0	0
Kubernetes-Validate [25]	×	×	×	×	×	×	○	×	N/A	○	×	3	0	0
Kubeval [Deprecated] [26]	○	○	×	×	×	×	○	○	JSON Schema	○	×	0	0	0

C. Limitations of existing tools

The in-depth study we carried out in order to propose Table I enabled us to reach the following conclusions.

1) Lack of schema-based verification: Only a third of tools offer the ability to validate manifests against the Kubernetes Resource Model (KRM).

2) No inconsistency detection capabilities: None of the verification tools include a mechanism to detect inconsistencies between verification rules.

3) Lack of internal data access capabilities: A limited number of tools offer the ability to access internal data. As explained in Section IV-C, not offering this capability reduces the expressiveness of architecture-based verification rules.

4) Lack of audit capabilities: A limited number of tools offer audit capability.

5) Difficult behavior analysis: Analyzing the behavior (operation) of built-in verification rules is difficult, due to the use of languages and specification methods specific to each tool. For instance, to analyze a particular rule in the KubeLinter tool, it is necessary to analyze the Go specification of that rule, while for Datree, it is necessary to examine the YAML specification of the rule. This lack of standardization in rule

specification makes it difficult and time-consuming to analyze the behavior of a specific rule in a given tool. As a result, it is also difficult to compare the same verification rule from different tools.

In conclusion, the analysis of Table I reveals that none of the tools studied in this paper exhaustively meets all our criteria.

VI. Proposition and future work

To address the shortcomings identified in Section V-C, we have undertaken the following work.

A. Rewriting of built-in rules in a standard formalism

The aim of this rewriting work is to address shortcoming discussed in Section V-C5, i.e. difficult behavior analysis. We chose the CEL (Common Expression Language) as the standard formalism [27]. This choice stems from the recent adoption of CEL as the standard for specifying verification rules for VAP (Validating Admission Policy), the official Kubernetes rule validation tool, currently under implementation. By using CEL as a rewriting language, we ensure that the majority of rewritten rules can be executed in VAP. However some rules, like those requiring external data access, are currently not supported in VAP.

Rewriting the rules in CEL also allowed us to set up a methodology to automatically test the capabilities of a manifest verification tool. Indeed, rewriting a rule requires checking the conformity of the rewritten rule with the original one. In our study, this verification is carried out by creating a set of incorrect manifests to test the detection ability of the rewritten rule. Later, these manifests can be used to automatically evaluate the number of built-in verification rules of a verification tool. To do this, it will be sufficient to provide the incorrect manifests as input and to count those that the tool has succeeded in detecting.

B. Design of a manifests verification tool

In order to address the shortcomings discussed in Section V-C, we have started to develop a new tool. It aims to combine the functionality of existing tools, while providing an additional feature: the ability to detect inconsistencies between rules. For this purpose, the following methodology is applied: (1) translation of the CEL files into Alloy language [28], (2) model checking on the results of step (1) using Alloy Analyzer. Model checking is a formal verification method that checks if a model of a system satisfies a set of properties. In our case, the results of step (1) and the formal model of the Kubernetes Resource Model (KRM) make up the model, and the property being checked is inconsistency. This property can be reformulated as follows: for a set `S` of CEL rules written for a resource kind `R`, there must exist at least one instance of `R` that satisfies `S`.

VII. Related work

Several papers in the literature have been conducted to evaluate verification tools for configuration files. These papers often take the form of surveys that create a classification of verification tools [4], [29], [30]. This classification provides an in-depth understanding of the available tools and, as result, help to select the most appropriate ones.

One notable instance is the study conducted by T. Xu and Y. Zhou in [30]. This survey paper examines different approaches to resolving configuration errors: building configuration-free systems, making systems easy to configure, hardening systems against configuration errors, checking the correctness of configurations, automating deployment and monitoring procedures, troubleshooting configuration errors. Of all the approaches considered, the most relevant to our paper is the verification of configuration correctness. T. Xu and Y. Zhou categorize configuration correctness verification tools based on whether verification rules are manually defined (defining-rules) or automatically generated using machine learning techniques (learning-rules). In our paper, we refer to these two categories as user-defined rules and built-in rules. While the association between user-defined rules and defining-rules is straightforward, the association between built-in rules and learning-rules may not be as obvious. Built-in rules refer to rules that are embedded to a verification tool, regardless of the method used to obtain or specify them (including machine learning). As long as these learning-rules are inherently part of the tool, they are considered as built-in rules. While the work done in [30] is great, it still does not cover all the dimensions of analysis considered in Table I.

In contrast to the general scope of configuration files addressed in [30], the work conducted in [4] specifically concentrates on Kubernetes manifests. In this paper, the authors categorize security misconfiguration in Kubernetes manifests and identify a total of 11 categories, namely: absent resource limit, absent `securityContext`, activation of `hostIPC`, activation of `hostNetwork`, activation of `hostPID`, capability misuse, docker socket mounting, escalated privileges for child container processes, hard-coded Secret, insecure HTTP, and finally privileged `securityContext`. The consequences of these error categories range from susceptibility to denial-of-service attacks, to granting full access to host functionality.

Once the categories were defined, they were used to evaluate five verification tools: SLI-KUBE (authors own tool), Checkov, KubeLinter, Datree and Synk.

Although the work done in this study is great, unfortunately all the error categories it proposes to verify are exclusively field-based verification, and in the context of CNF-based systems, as demonstrated in Section III and IV-A, this is insufficient. In general, it is important to mention that while structure-based and architecture-based verifications have been extensively studied in other contexts, such as Hadoop [31] and Software Architectures [32], the same level of research and application has not yet been extended to Kubernetes.

In addition to academic work, there has been work done to evaluate manifest verification tools, and most of these works take the form of blog posts, as illustrated by references [33]–[35]. The criteria outlined in these works cover some of the criteria discussed in our study, such as JSON Schema-based verification or number of built-in verifications. However, they

do not seek the same degree of granularity as our study. Take for example the criterion of number of built-in verification rules. Our study stands out by creating a classification of built-in rules according to their verification type, a classification crucial for evaluating the scope of each tool, as explained in more detail in Section IV-I.

Finally, our verification tool under development will be in line with the implementation methodology carried out in [36], [37]. These studies used Alloy [28] to formally verify different types of manifests, such as Docker Compose and OpenStack Heat languages.

The aforementioned discussions reveal a lack of research on the definition of revelant verification capabilities in the context of Kubernetes and, by extension, in the context of CNF-based systems, which we address in our paper.

VIII. Conclusion

Kubernetes manifests may be subject to various kinds of errors that can have an impact on the robustness, security and availability. To enable practitioners to select manifest verification tools, we have introduced a set of 13 evaluation criteria relevant in the CNF context. In addition, we used our criteria to evaluate 16 verification tools. This evaluation revealed 5 key shortcomings: (1) lack of schema-based verification, (2) no rules inconsistency detection capability, (3) lack of internal data access capability, (4) lack of audit capability, (5) difficult behavior analysis of built-in verification rules.
Furthermore, the analysis of the built-in verification rules allowed us to rewrite them in the standard CEL language. The resulting rules can be directly used with VAP, the official Kubernetes rule validation tool. However, it is worth mentioning that VAP, like other analyzed tools, does not detect inconsistencies between rules. In addition, the in-depth analysis also enables the creation of a set of erroneous manifests that can be used to benchmark manifest verification tools. Developers of the evaluated tools could benefit from these results, i.e. a set of CEL rules and erroneous manifests, to improve their tools. Finally, this analysis conducted to the design of a new customizable and formal-based verification tool. A complementary study might consider additional evaluation criteria such as performance and resource consumption (CPU, RAM, etc.), which are important aspects in the field of model checking.

References

[1] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network Function Virtualization: State-of-the-Art and Research Challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2015.

[2] R. Botez, A.-G. Pasca, and V. Dobrota, "Kubernetes-Based Network Functions Orchestration for 5g Core Networks with Open Source MANO," in *2022 International Symposium on Electronics and Telecommunications (ISETC)*, 2022, pp. 1–4.

[3] "Production-Grade Container Orchestration," 2023. [Online]. Available: https://kubernetes.io/

[4] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security Misconfigurations in Open Source Kubernetes Manifests: An Empirical Study," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–36, 2023.

[5] "Resolve issue: iptables v1.6.1: can't initialize iptables table 'filter': Permission denied (you must be root)," 2022. [Online]. Available: https://github.com/matysiaq/nephio-pocs/commit/7860e8e2ac8cce6adbde4f7de3128fc9d0ec0957

[6] "Datree." [Online]. Available: https://github.com/datreeio/datree

[7] "Checkov." [Online]. Available: https://github.com/bridgecrewio/checkov

[8] "Gatekeeper." [Online]. Available: https://github.com/open-policy-agent/gatekeeper

[9] "Kyverno." [Online]. Available: https://github.com/kyverno/kyverno/

[10] "System architecture for the 5G system (5GS)." [Online]. Available: https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3144

[11] "Nephio: Proofs of Concepts." [Online]. Available: https://github.com/matysiaq/nephio-pocs

[12] D. Nešić, J. Krüger, u. Stănciulescu, and T. Berger, "Principles of feature modeling," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 62–73.

[13] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of JSON Schema," in *Proceedings of the 25th International Conference on World Wide Web*, 2016, pp. 263–273.

[14] "Kubernetes API Schema," 2023. [Online]. Available: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.27/

[15] "Copper." [Online]. Available: https://github.com/cloud66-oss/copper

[16] "Kubelinter." [Online]. Available: https://github.com/stackrox/kube-linter

[17] "Kubevious." [Online]. Available: https://github.com/kubevious/kubevious

[18] "Kube-score." [Online]. Available: https://github.com/zegl/kube-score

[19] "Polaris." [Online]. Available: https://github.com/FairwindsOps/polaris

[20] "Config-Lint." [Online]. Available: https://github.com/stelligent/config-lint

[21] "Conftest." [Online]. Available: https://github.com/open-policy-agent/conftest

[22] "Terrascan." [Online]. Available: https://github.com/tenable/terrascan

[23] "Trivy." [Online]. Available: https://github.com/aquasecurity/trivy

[24] "Kyverno." [Online]. Available: https://github.com/yannh/kubeconform

[25] "Kubernetes-Validate." [Online]. Available: https://github.com/willthames/kubernetes-validate

[26] "Kubeval." [Online]. Available: https://github.com/instrumenta/kubeval/

[27] "Common Expression Language," 2023. [Online]. Available: https://github.com/google/cel-spec/blob/master/doc/langdef.md

[28] D. Jackson, "Alloy: a Language and Tool for Exploring Software Designs," *Commun. ACM*, vol. 62, no. 9, pp. 66–76, 2019.

[29] A. A. Jabal, M. Davari, E. Bertino, C. Makaya, S. Calo, D. Verma, A. Russo, and C. Williams, "Methods and Tools for Policy Analysis," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–35, 2019.

[30] T. Xu and Y. Zhou, "Systems Approaches to Tackling Configuration Errors: A Survey," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 70:1–70:41, 2015.

[31] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and discovering software configuration dependencies in cloud and datacenter systems," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 362–374.

[32] C. Araújo, E. Cavalcante, T. Batista, M. Oliveira, and F. Oquendo, "A Research Landscape on Formal Verification of Software Architecture Descriptions," *IEEE Access*, vol. 7, pp. 171 752–171 764, 2019.

[33] "Top Kubernetes YAML Validation Tools." [Online]. Available: https://kubevious.io/blog/post/top-kubernetes-yaml-validation-tools

[34] "Kubernetes Validation Tools." [Online]. Available: https://github.com/HighwayofLife/kubernetes-validation-tools

[35] "Validating Kubernetes YAML for best practice and policies." [Online]. Available: https://learnk8s.io/validating-kubernetes-yaml

[36] A. N. Sylla, K. Guillouard, F. Klamm, M. Ouzzif, P. Merle, S. B. Rayana, and J.-B. Stefani, "Formal Verification of Orchestration Templates for Reliable Deployment with OpenStack Heat," in *2019 15th International Conference on Network and Service Management (CNSM)*, 2019, pp. 1–5.

[37] P. Merle, S. Ben Rayana, L. Seinturier, R. Pissard-Gibollet, J.-B. Stefani, and A. Ndeye Sylla, "Towards a formal reference computational model for cloud configuration management," INRIA, Research Report RR-9317, 2020. [Online]. Available: https://inria.hal.science/hal-02940938