Vrije Universiteit Amsterdam


Honours Programme, Project Report

# Analysing Helm charts: the security of K8 configurations

**Author:** Julia Teerhuis (2691867)

*supervisor:* Dr. Katja Tuma

*A report submitted in fulfillment of the requirements for the Honours Programme, which is an excellence annotation to the VU Bachelor of Science degree in Computer Science/Artificial Intelligence/Information Sciences*

May 31, 2022

## The Project Card

(Fill in this part as you make progress with your research. When done, send this and the entire project report to the Honours Programme (HP) coordinators.)

This is a one-page (recto and verso) summary of the research:

Research question(s):

**RQ1** What are the characteristics of security misconfigurations and how to automate microservice configuration security assessment to identify security problematic configurations?

**RQ2** How to evaluate the automated approach on community provided configurations?

Research method(s):

**M1** Descriptive statistical analysis on (mis)configurations.

**M2** Flow diagram.

**M3** Code

Research approach:

**A1** Since automated support is not extensive, to adress **RQ1** we want to focus on the misconfigurations that occur most often. Because these will have a larger impact on the overly security posture of the deployment. On the other hand, we also want to focus on the misconfigurations that existing tools do not support (eg, Checkov only has 1 check for ServiceAccount while it is still important from security perspective, as best practices say [K8 Benchmarks].)

To answer the first part of the question, what are the characteristics of security misconfigurations, we perform a quantitative analysis of 4306 yaml files.

Since the main cause of security problems are caused by misconfigurations, therefore it is important to investigate the characteristics of misconfigurations [16]. This has not been done before on such a large scale. We investigate the common characteristics of security misconfigurations by doing a descriptive statistical analysis of a large body of K8 configurations in 4306 yaml files. The repository we will analyze for the descriptive statistical analysis will be

the Helm chart. This Helm chart will give us a good overview of the resource files.

We have designed an automated process to collect and prepare the data set of the Helm charts. To this aim, we developed a simple crawler [5]. Roughly, we follow the steps: (i) Download all Helm charts from Artifacthub using a crawler, (ii) The Helm charts will all be rendered and saved as YAML files, (iii) will convert these YAML files into JSON object files.

The descriptive statistical analysis will help us in answering the full research question. Based on the descriptive analysis we can start programming the rules to capture the identified misconfiguration characteristics. The program will be our answer to the research question.

Furthermore, we will investigate microservice secure configuration from the perspective of evaluating the automated way of capturing the misconfigurations.

**A2** To address **RQ2**, Some of the implemented checks could be compared against a baseline tool: Checkov [6]. Therefore, we designed specific measures to capture our tool's performance. We compared the amount of passes and fails per check.

For the newly proposed check we adopted a manual evaluation with studying a few examples in detail and manually checking 20% of the hits in our data set of 4306 files.

Main results and contribution:

**C1** A descriptive statistical analysis of a large body of K8 configurations.

**C2** A tool that can capture the security of commonly reused K8 configurations.

**C3** A presentation.

**R1** A GitHub repository containing:

- The crawler that downloads all helm templates from Artifacthub
- The code to render helm templates easily and convert them to JSON files
- The passed and failed checks of all files per check
- The automated tool

## Abstract

Misconfigurations are one of the leading causes of many vulnerabilities within the microservice applications and orchestration framework [16]. In spite of the fact that the security of microservice-based systems has been researched [15], these results have not been used much in practice. According to security-by-design principles, we developed an automated way to assess the configurations before the actual deployment. While most studies focus on gathering the issues within the architecture, we created a tool that can capture the security of commonly reused K8 configuration.

We evaluated the tool with 4306 files from all public Helm Charts on Artifacthub, and we obtained on average a precision of 0.9885 and for recall on average 0.9979.

## 1 Introduction

Most companies have built their systems initially with a monolithic architecture. However, over time the system will grow, and the code and the architecture will become too complex to maintain. With a monolithic structure, the more the system grows the more it will become slower and less flexible [17]. Therefore, companies are changing their systems architecture and using microservices to improve their systems. Also, younger software companies and start-ups often opt for the microservice architectural style [14].

Nonetheless, the microservice architecture also has its additional challenges. For example, the effort required to deploy, scale, and operate the microservices in cloud infrastructures [17]. Furthermore, there currently is a lack of automated tool support for developers to analyse security risks of microservices configuration alternatives. Given the fast-paced development in the industry, security is not always a top priority for companies [16, 18].

Despite this need for rapid application development, there are not many automated tools that check the implemented configurations. One of the tools that can be used to check for misconfigurations is the CheckOV tool. Checkov scans cloud infrastructure configurations to find misconfigurations before they are deployed [6].

According to the research of [15], there are many works on how to secure specific parts of a system. Of these many works, there is only little work on systematic approaches. Secure network configurations are a critical part of microservices security. Namely, compared to a monolithic application, microservices applications are easier to infiltrate since they have a larger attack surface [15]. The security of microservice configurations is important since the data gets transferred between the microservices regularly. When having a microservice architecture, communications between the microservices are exposed through the network. This creates a potential attack surface. Secure network configurations are therefore a critical part of microservices security.
With more and more systems growing and more companies adopting microservices to keep up with the fast-changing environment this problem cannot be ignored.

According to the [16], the leading cause of security incidents in microservices is misconfigurations. However, to this day, there is little automated support for analyzing the security of cloud configurations (with the exception of e.g., CheckOV). Therefore, the ad-

ministrators have to manually analyze the complex cloud infrastructure to understand the security state of their applications and deployment. In this research we aim to automate microservice configuration security assessment to help the administrators with finding misconfigurations more easily. Furthermore, the expected contribution will be an analysis of security problematic configurations and an analysis of the automated tool.
With these contributions, misconfigurations will get noticed more easily and configuring microservices can be done more securely.

Firstly, we downloaded 4731 helm templates from Artifacthub. Then, we rendered them and converted them into JSON files. After converting, there were 4306 files remaining. We could easily loop through all of the files. Then, we conducted a descriptive analysis of the errors that could occur by using the CheckOV tool [8]. We used the CheckOV tool as the ground truth. Furthermore, we have implemented some of the checks ourselves and also added one ourselves.

From the descriptive analysis we concluded that the most common type of configuration to occur was "Service". Nonetheless, this was one of the least common entity to have a CheckOV check. Out of the 154 checks, there were only 2 concerning "Service". The second most common entity to occur was the Deployment. This entity had 98 checks.

The error that would occur the most within our pool of yaml files was the CKV_K8S_21: "The default namespace should not be used". In total we had 122423 errors and 4380 YAML files.

The results of our tool were close to the results of the CheckOV tool. Both the precision and recall were for all checks above 0.9.

## 2 Background Concepts and Models

### 2.1 Overview of K8 architecture

Kubernetes is an open source software that is mainly used for the deployment of software. Kubernetes automates the usage of container objects. A working Kubernetes deployment is called a cluster. The design of these clusters is based on 3 principles. They should be secure, easy to use and extendable.

A Kubernetes cluster can be divided in two parts: the control plane and the compute machines (nodes). The nodes can either be a physical or a virtual machine where each node is its own Linux environment. Within each node, there are pods running which are made up of containers.

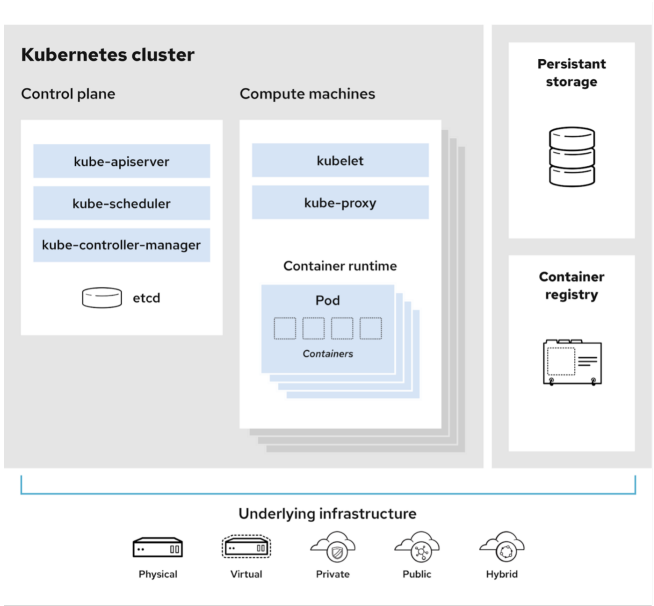


Figure 1. Overview of the Kubernetes architecture [3].

### 2.2 Orchestration steps

Within the orchestration we have different steps for creating it. The lifecycle of cloud-native deployment consists of different layers. It is composed of layers of the foundation, lifecycle, and environment. The lifecycle process consists of the management of the supply chain and the curation of applicable security benchmarks.

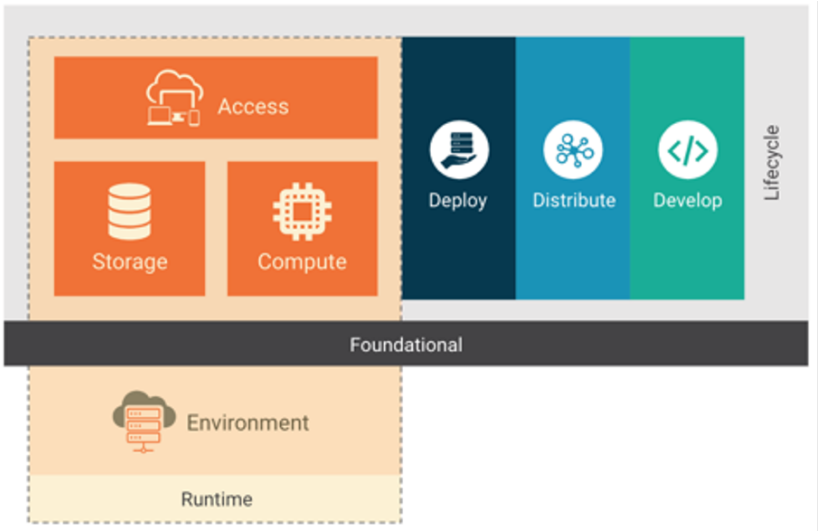


Figure 2. Cloud Native Layers [2].

## 3 Research Design

***RQ1:*** What are the characteristics of security misconfigurations and how to automate microservice configuration security assessment to identify security problematic configurations?

Since automated support is not extensive, we want to focus on the misconfigurations that occur most often. Because these will have a larger impact on the overly security posture of the deployment. On the other hand, we also want to focus on the misconfigurations that existing tools do not support (eg, Checkov only has 1 check for ServiceAccount while it is still important from security perspective, as best practices say [K8 Benchmarks].)

To answer the first part of the question, what are the characteristics of security misconfigurations, we perform a quantitative analysis of 4306 yaml files.

Since the main cause of security problems are caused by misconfigurations, therefore it is important to investigate the characteristics of misconfigurations [16]. This has not been done before on such a large scale. We investigate the common characteristics of security misconfigurations by doing a descriptive statistical analysis of a large body of K8 configurations in 4306 yaml files. The repository we will analyze for the descriptive statistical analysis will be the Helm chart. This Helm chart will give us a good overview of the resource files.

We have designed an automated process to collect and prepare the data set of the Helm charts. To this aim, we developed a simple crawler [5]. Roughly, we follow the steps: (i) Download all Helm charts from Artifacthub using a crawler, (ii) The Helm charts will all be rendered and saved as YAML files, (iii) will convert these YAML files into JSON object files.

The descriptive statistical analysis will help us in answering the full research question. Based on the descriptive analysis we can start programming the rules to capture the identified misconfiguration characteristics. The program will be our answer to the research question.

Furthermore, we will investigate microservice secure configuration from the perspective of evaluating the automated way of capturing the misconfigurations.

***RQ2:*** How to evaluate the automated approach on community provided configurations?

Some of the implemented checks could be compared against a baseline tool: Checkov [6]. Therefore, we designed specific measures to capture our tool's performance. We compared the amount of passes and fails per check.

For the newly proposed check we adopted a manual evaluation with studying a few examples in detail and manually checking 20% of the hits in our dataset of 4306 files.

# 4 Results and Analysis

This section describes the descriptive analysis of the Helm charts and the evaluation of the implemented checks.

### 4.1 The Data Set

Helm is a package manager running on top of Kubernetes that deploys charts. These charts are Kubernetes YAML manifests combined into a single package. The charts are used to deploy an application, or one component of a larger application. You can easily manage the application structure with simple commands. [1, 4]

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
```

Figure 1. Example of a rendered template of Kind "ConfigMap" [7].

We followed the approach of the first research question. We started with 4731 files which we all rendered with the command "helm template".

| | Discarded files | Files remaining |
|---|---|---|
| Rendering all helm charts | 149 | 96.8% |
| Converting all templates to YAML files | 202 | 92.5% |
| Converting all teplates to JSON files | 74 | 91.0% |

In total we excluded 8.96% of the files. When rendering the helm charts, some charts required manually added values to be added by the programmer (e.g., password in case of limesurvey-0.5.0.tgz). Furthermore, when converting all templates to YAML files, we also filtered out the empty helm charts. The empty helm charts were helm charts that resulted in an empty YAML file with no lines. Therefore, this number of lost files is higher than the others. Furthermore, we looked at the number of lines of each file.

| Files | total lines | Average lines | Largest | Smallest |
|---|---|---|---|---|
| 4306 | 3142807 | 718,4 | 54835 | 1 |

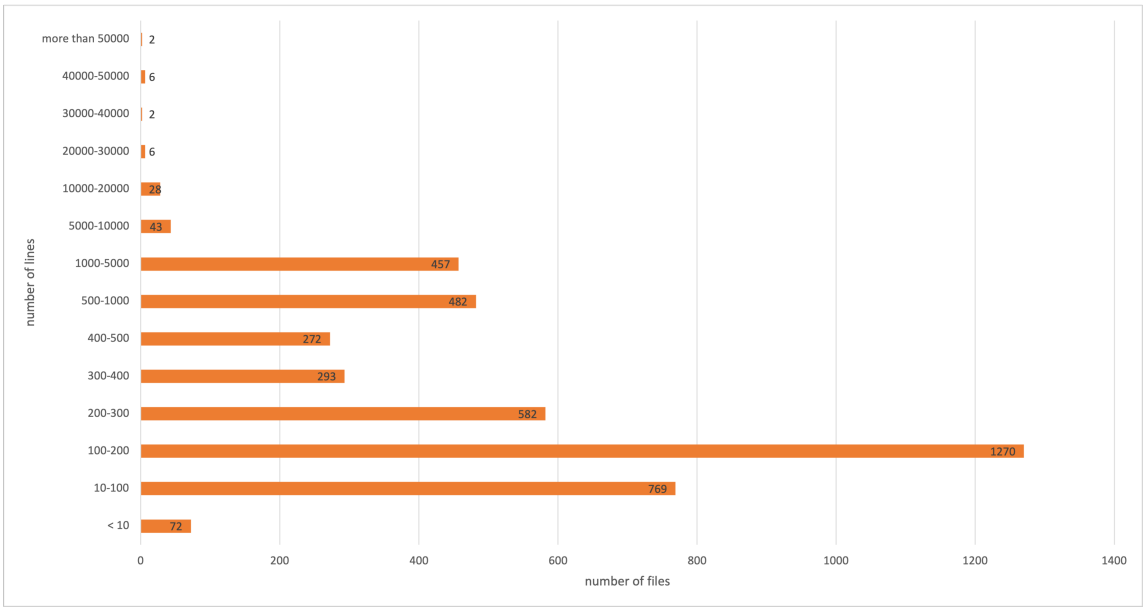


Figure 2. Overview of the amount of the lines per file.

We see that most of the files have between 100 and 200 lines of code, this was 30% of all files. There were only two files that had more than 50000 lines of code. Furthermore, rendering the helm charts for the files with more lines took longer than files with less lines.

**The Configuration types**

After this, we researched the number of CheckOV checks developed per configuration type.

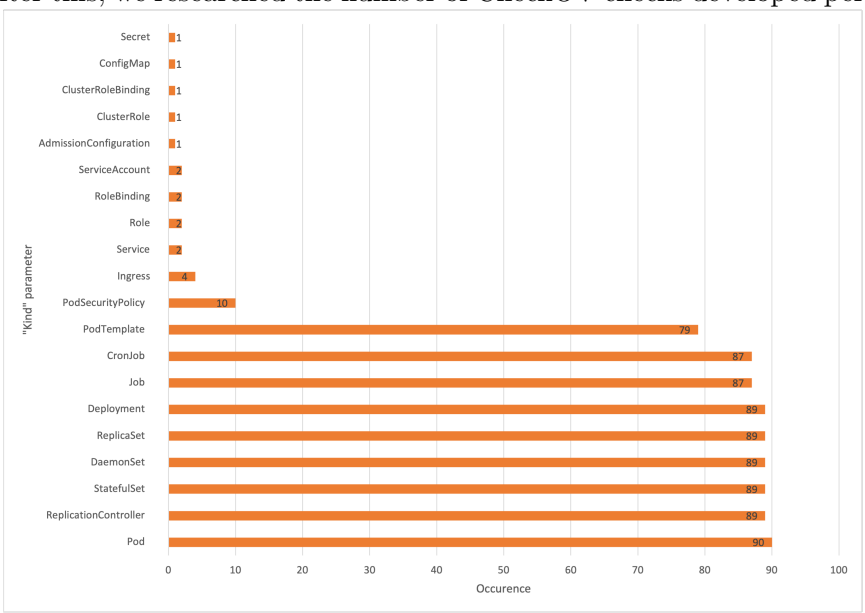


Figure 3. The number of CheckOV checks developed per configuration type.

If we look at the Checkov tool checks we can see that there is only 1 check for the configuration type "Secret". This is also the case for "AdmissionConfiguration", "Clus-

terRole", "ClusterRoleBinding", and "ConfigMap". As seen in the figure 4, "Service" is in fact the most frequent configuration type. Nonetheless, CheckOV has only 2 checks for the parameter "Service".

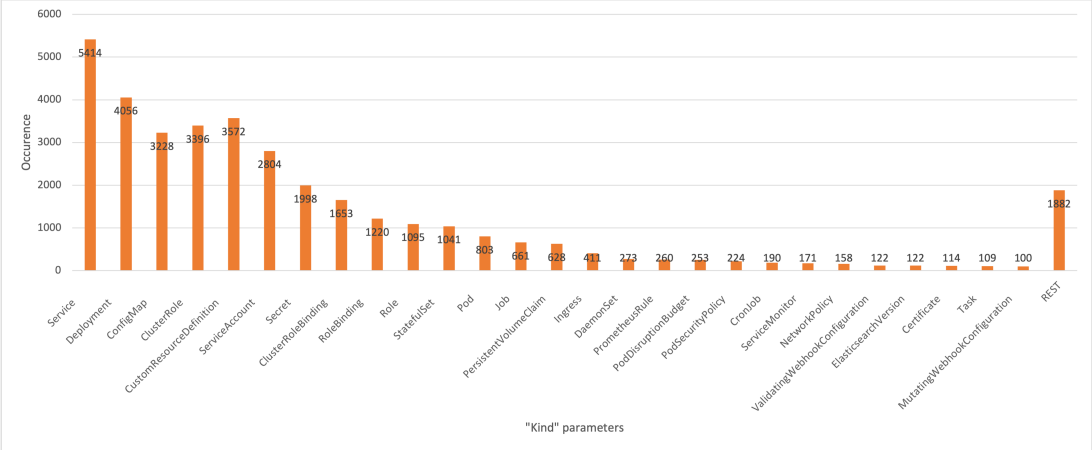


Figure 4. Occurence of configurations per type.

Since there are such few checks for Role and ClusterRoleBinding, we have added one check for these. The "Added Check" checks if the amount of verbs in the "rules" of the object is higher than a certain threshold. The user can choose this threshold themselves.

**The result of CheckOV on our Data set**

We ran the Checkov tool on the files to see which Errors are the most frequent.


Figure 5. Overview of the amount of the errors per file by the CheckOV tool.

As seen in the graph, the most frequent error was the CKV_K8S_21. The message this error gave was "The default namespace should not be used". This rule was checked by CheckOV for almost all of the configuration types. Furthermore, we had one check CKV_K8S_49 which checked the configurations with the configuration type, which can be found under the "Kind" parameter, "Role" and "ClusterRole". These occurred many

times in the files of the data set.

### 4.2 Implementation

We want to package the solution as part of the security tests in the applicable security benchmarks. These are running just before the deployment stage (See figure 6).

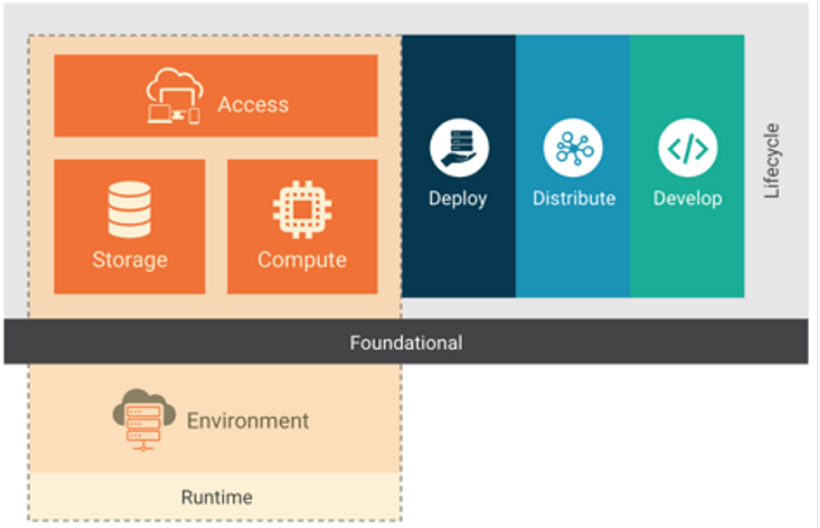


Figure 6. Cloud Native Layers [2].

As explained in the results, we could only check 91% of the files. This dataset still gives us a reliable overview. The average time to check one file with the Checkov tool was 40,58 seconds, even if we would do only one type of check. On the other hand, the tool we created only takes 3 milliseconds on average to check one file. The reason why the difference is so big is probably because the CheckOV tool is actually building a graph from all sorts of files and configurations first, so they can do all the other checks also, while we only looked at a few simple ones and implemented them directly using JSON objects.

Based on the outcome of figure 5 we chose to re-implement 5 checks. For each check, there are certain "Kind"-parameters for which it must check for the misconfiguration. For example, CKV_K8S_49 only myst be checked for the "Kind"-parameters "Role" and "ClusterRole". Therefore, we will have a list that contains all the "Kind"-parameters for which the check has to be executed. All yaml files have been converted into JSON objects. The input is therefore always a JSON object.

- **CKV_K8S_21: The default namespace should not be used**

```
if CKV_K8S_21 contains kind then
    if object metadata namespace != 'default' then
        "Passed CKV_K8S_21"
    else
        "Failed CKV_K8S_21"
    end if
end if
```

Within the object metadata we will search for the 'namespace'. If there is a namespace in metadata and the namespace value is set to 'default'. The configuration has passed the check. In all other cases the configuration has failed the check and the error message "CKV K8S 21: The default namespace should not be used" will be shown.

- **CKV_K8S_38: Ensure that Service Account Tokens are only mounted where necessary**

```
if CKV_K8S_38 contains kind then
    if kind == 'Pod' then
        route = object/spec
    else if kind == 'CronJob' then
        route = object/spec/jobTemplate/spec/template/spec
    else
        route = object/spec/template/spec
    end if
    if 'automountServiceAccountToken' in route == false then
        "Passed CKV_K8S_38"
    else
        "Failed CKV_K8S_38"
    end if
end if
```

There are different places to check for different configuration types. For example, If the parameter is "Pod" we will have another route than if the configuration type is "CronJob".
After following the correct route to the key value pair that supposedly contains the "automountServiceAccountToken" key, we will check if this key value is set to "false". If this is the case the configuration has passed the check. Otherwise, the configuration has failed the check.

- **CKV_K8S_49: Minimize wildcard use in Roles and ClusterRoles**

```
if CKV_K8S_49 contains kind AND object has property 'rules' then
    wildcards = []
    for i to object/rules-size do route = object/rules[i]
        if verbs contains "*" then
            add "verbs" to wildcards
        else if resources contains "*" then
            add "resources" to wildcards
        else if apiGroups contains "*" then
            add "apiGroups" to wildcards
        end if
    end for
    if wildcards size  0 then
        "Failed CKV_K8S_49"
    else
        "Passed CKV_K8S_49"
    end if
```

For the object rules it will check if none of the key value pairs, in the apiGroups, verbs and resources are a wildcard. The values of these keys are called verbs. It is important to limit these verbs to keep the configuration secure. If you would put a wildcard as value, any verb could be put in that place.

Therefore, there can not be any wildcards as values. If an star (*) is found for one of the values of one of the keys (apiGroups, verbs, resources), the configuration has failed the check. In all other cases it passes the check.

- **CKV_K8S_43: Image should use digest**

```
if CKV_K8S_43 contains kind then
    Recurse through all key values
    if key == image then
        if key-value contains "@sha" then
            "Passed CKV_K8S_43"
        else
            "Failed CKV_K8S_43"
        end if
    end if
end if=0
```

For all images in the whole configuration, we will check of they all are using a digest. A container image digest uniquely and immutably identifies a container image. A docker pull will download the same image every time. By using a digest, you avoid the downsides of deploying by image tags, which are mutable.
If the value of the key "image" contains "@sha" it uses a digest and it passes the check. However, if it does not contain "@sha" it does not use a digest and therefore it will fail the check.

- **CKV_K8S_31: Ensure that the seccomp profile is set to docker/default**

**or runtime/default**

```
if CKV_K8S_31 contains kind then
    if kind == 'Pod' then
        route = object/spec
        if route/securityContext/SeccompProfile/type == 'RuntimeDefault' then
            "Passed CKV_K8S_38"
        else
            metadata = object/metadata
        end if
    else if kind == 'CronJob' then
        metadata = object/spec/jobTemplate/spec/template/metadata
    else if kind == 'StatefulSet' OR kind == 'Deployment' then
        route = object/spec/template/spec
        if route/securityContext/SeccompProfile/type == 'RuntimeDefault' then
            "Passed CKV_K8S_38"
        else
            metadata = object/metadata
        end if
    else
        route = object/metadata
    end if
    if all metadata/annotation values are set to default values then
        "Passed CKV_K8S_38"
    else
        "Failed CKV_K8S_38"
    end if
end if
```

For the configuration types "Pod", "CronJob", "StatefulSet", and "Deployment" there are different routes to find the object that contains the settings for the seccomp profile. If the key value pair is not found in the route, the metadata will be saved in the variable "metadata".
Also, in case of another "Kind" parameter than the ones listed above we will save the object metadata in the variable "metadata". If the variable "metadata" is undefined, we will check if the annotations contain the correct key pair values.

These checks have been combined together in one file. In figure 7, the combined checks are shown in a flow chart. The user will choose a file to check, which is a Helm template or already a YAML file. After converting the file to a YAML file and creating a JSON object all checks will be executed until the end of the file. For the added check, the user will choose the maximum number of verbs that can be used. In figure 7 the JavaScript code is shown of the added check.

```
if(ADDED_CHECK.includes(key) && object.hasOwnProperty('rules')){
    var nums = [];
    if(object.rules){
        for(num in object.rules){
            var route = object.rules[num];
            if(route.apiGroups && route.verbs && route.resources){
                if(route['verbs'].length > verbs_max){
                    nums.push(num);
                }
            }
        }
        if(nums.length > 0){
            console.log("   FAILED ADDED CHECK: Amount of verbs is more than " + verbs_max + " for verbs number(s)" + nums.join(", "));
            ADDED_CHECKP +=1;
        }
        else{
            console.log("   PASSED ADDED CHECK");
            ADDED_CHECKF +=1;
        }

    }
}
```

Figure 7. JavaScript code of the added check. This checks if the amount of verbs exceeds the amount the user has entered.

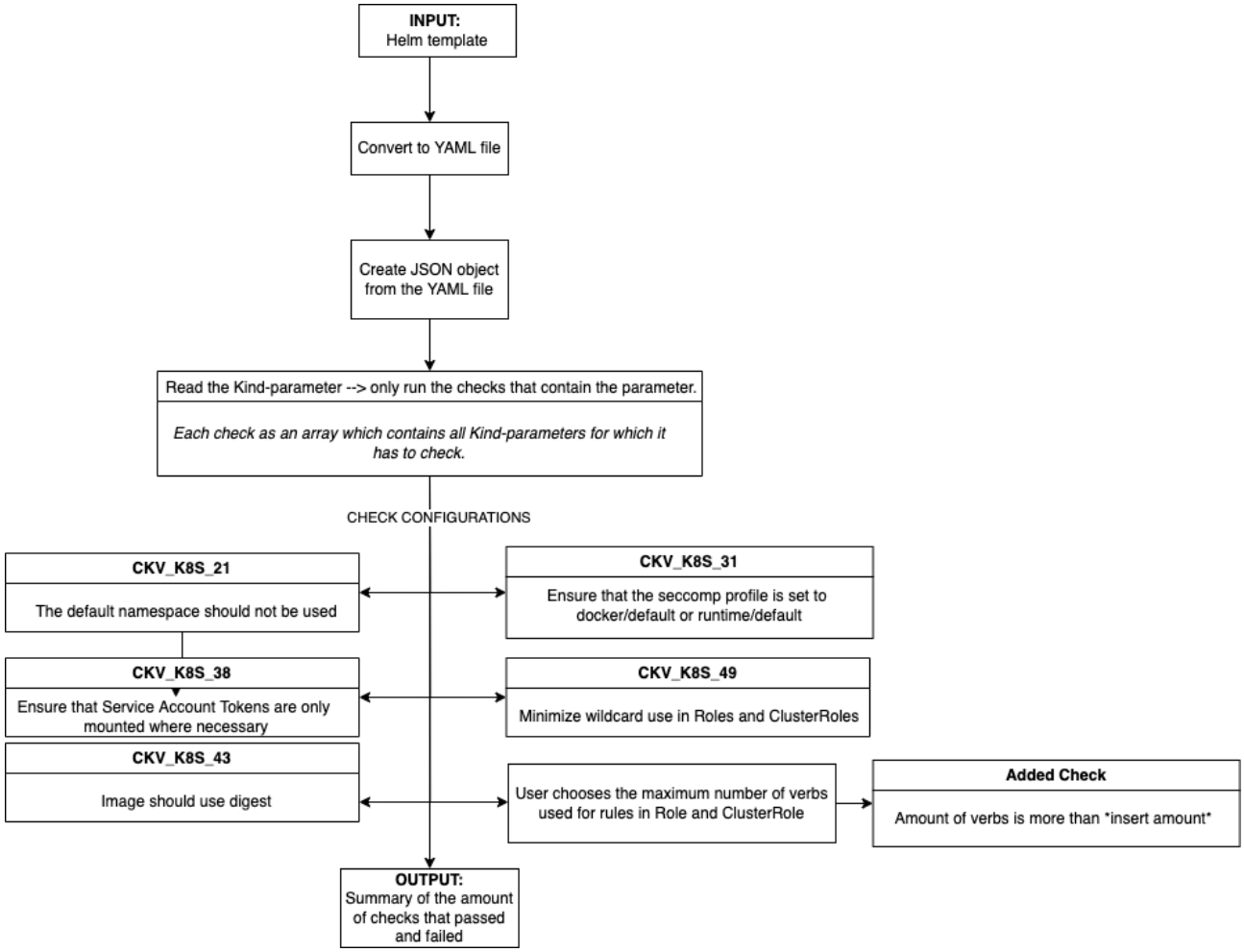


Figure 8. Flow chart of the automated tool.

The input was a Helm template. This Helm template was then converted to a YAML file which was converted into a JSON object.

### 4.3 Evaluation

The results of our tool were close to the results of the CheckOV tool. Both the precision and recall were for all checks almost 1.

| Check type | TP | FP | FN | Precision | Recall |
|---|---|---|---|---|---|
| CKV_K8S_21 | 22245 | 0 | 23 | 1 | 0.9990 |
| CKV_K8S_31 | 6966 | 3 | 6 | 0.9996 | 0.9991 |
| CKV_K8S_38 | 6849 | 0 | 6 | 1 | 0.9991 |
| CKV_K8S_43 | 6694 | 0 | 52 | 1 | 0.9923 |
| CKV_K8S_49 | 2039 | 124 | 0 | 0.9427 | 1 |

To obtain this data, we first ran the CheckOV tool and saved the failed checks in a text file. Then, we did the same for our own created tool. We compared these two text files with each other to get the true positives, false positives, and false negatives.
For the JSON object We could not keep the exact lines saved anywhere and could therefore not get a precise location. This was the same for the lines that CheckOV returns, they also refer to the lines of the full configuration and not the specific line of the error. We verified that the errors were the same by looking at the name combined with the namespace and the type of the YAML configuration. For example, this is the information that CheckOV returns.

```
Check: CKV_K8S_49: "Minimize wildcard use in Roles and ClusterRoles"
FAILED for resource: ClusterRole.default.aad-pod-identity-mic
File: /aad-pod-identity-4.1.7.yaml:29-64
```

We took the part after "FAILED for resource": ClusterRole.default.aad-pod-identity-mic. The first part is the type, the second is the namespace and the last is the name of the configuration. In my own tool, I also formatted the information of the configuration in the same way:

```
var namespace = object.metadata["namespace"];
if (!("namespace" in object.metadata)) namespace = "default";
else if(namespace == undefined) namespace = "None"
list.push(object.kind + "." + namespace + "." + object.metadata['name']);
```

In the end, I added the list to a file that got compared with a file that contained all the lists of errors of the CheckOV tool.

The true positives (TP) are the fails that have been detected correctly. We add one true positive for each name that is included in both the list of our error tool as the list of the CheckOV tool.

The false positives (FP) are the fails that have been detected incorrectly by our tool. This is when the number of fails the CheckOV tool detected were lower than the number of fails our tool detected. We add one false positive for each time that a name is included in the list of our tool, but not in the list of the CheckOV tool. The false positives are caused by the fact that my tool will fail something automatically if it can not find the key value pair. Take for example CKV_K8S_31, it has a lot of different places where the key value pair we want to check is located. CKV_K8S_49 has the highest amount of false positives, this is caused by the fact that the description was not clear on what "minimize" meant. Our tool immediately counted a fail if there was a wildcard for one of the keys in rules. However, CheckOV does not work as strict as our tool since it only will fail the check after a certain amount of wildcards.

The false negatives (FN) are the fails that have not been detected by our tool, but should have been. In this case, the number of fails the CheckOV tool detected were higher than the number of fails our tool detected. We add one false negative for each time that a name is not included in the list of our tool, but should be according to the list of the CheckOV tool. The main cause of the false negatives was that it did not take into account the type "List". This type was not taken into consideration for the checks since it was not added to the array of types that should be checked for each error. Therefore, it automatically skipped all checks. Nonetheless, within the list, there are multiple other types that had to be checked. In this case all misconfigurations for all checks that were not picked up by my tool, but should have been according to CheckOV were part of a "List".

For the added check, I manually checked 20% of the files. There were no false positives, therefore the precision was 1. The results above already showed me that the main cause of false negatives would be the Lists. In total there are 28 files that contain a List. However, there was only two Lists in the file spot-config-webhook-0.1.6.yaml and tsorage-0.4.11.yaml that contained rules. In the file spot-config-webhook-0.1.6.yaml, there were in total 6 wildcards used. In the file tsorage-0.4.11.yaml the largest amount of verbs used was two. The tool we created would not have found these and would have resulted in two false negatives in the case that the chosen amount of verbs was lower than two.

# 5 Related Work

The literature we have on secure software architecture design already includes systematic approaches such as threat modelling, applying security patterns, detecting design flaws and vulnerabilities. However, this knowledge is not fully utilised in the later stages, i.e., when the implemented application is configured for deployment [11].

A. Pereira-Vale et Al. Conducted a multivocal literature review on the security in microservice-based systems. The literature review researched grey literature and academic literature. Furthermore, they created a set of quality criteria for publications and calculated quality scores as percent of articles that meet each one of them [15]. There was a big difference in quality scores between the academic literature and the grey literature. All researches explicitly addressed security problems. Moreover, 92% described clearly their research objectives and explicitly stated the security solution that could be provided. On the other hand, the grey literature studies were often not clearly dated (21%) and 55% of the studies did not provide background. Furthermore, The problem addressed by their security solution was described by 66% of the studies and only 5% were deemed as having conclusions supported by data [15].

Nonetheless, adding grey literature to the systematic literature review can provide benefits and certain challenges. In the study by Garousi et al. they concluded that grey literature should be added, since evidence is based on experience and opinion [12].

To get an understanding of what existing challenges in designing, developing, and maintaining microservice systems. J. Ghofrani and D. Lübke [13] did an empirical survey to assess the current state of practice in microservices. The outcome of this survey also stated that there is a lack of notations, methods, and frameworks to architect microservice architecture. Furthermore, they stated that the top main gaps are the lack of tool or framework support for selecting third-party artefacts according to their features, as well as the shortage of knowledge of the practitioners about systematic methods [13].

The Checkov tool is a static code analysis tool for infrastructure-as-code. Checkov is a static code analysis tool for scanning infrastructure as code (IaC) files for misconfigurations that may lead to security or compliance problems. Checkov includes more than 750 predefined policies to check for common misconfiguration issues. Checkov also supports the creation and contribution of custom policies. [9].

As stated in the introduction, one of the most important policies in Kubernetes are the network policies [16, 10]. In a research conducted by G. Budigiri et Al. they evaluated the performance and analysed the security of the network policies in kubernetes. In the paper, it is considered that an attacker has compromised a microservice in one of the edge application namespaces. To hinder these attacks, we can use networks policies effectively. When configuring network policies, there are plenty of opportunities for weak or faulty configurations that as a result threaten the container network isolation. In the research they name wrong pod label specifications, manual errors like typos, or even completely forgetting to enforce network policies after writing them [10].

# 6 Conclusion and Future Work

In conclusion, the automation of microservice configuration security assessment is really important. We currently have the CheckOV tool that can help us with a static code analysis. The tool contains a total of 154 different checks for Kubernetes misconfigurations [6].

However, as mentioned in the results, there are multiple configuration types that are being used often in charts, but have only a few checks. For example the parameter "Service" was the most common configuration type in our dataset. Nonetheless, this configuration type is only checked by two CheckOV checks; CKV_K8S_21, which is also the most common error in our dataset, and CKV_K8S_44, this error does not occur many times in our dataset.

There is no related work on developing an automated tool for microservice configuration security assessment. Nonetheless, there have been many researches on the problems that can be caused by incorrect microservice configurations.

We created our own tool in JavaScript to try and automate the microservice configuration security assessment to identify security problematic configurations [5]. The tool had high precision and recall for each check. The main reason why there were problems was because of the Kind parameter "List". Too find the characteristics of misconfigurations we analysed the documentation of the checks of the CheckOV tool.
The CheckOV tool can help leverage this work and is already developing a lot around the microservice configuration security assessment. They have a live terminal execution and an extensible integration interface. This way misconfigurations can be prevented from being deployed by embedding it into existing developer workflows.

Furthermore, CheckOV has the possibility of adding your own checks with the help of their developer guide. Anyone can contribute to the tool, so with enough interest this tool can help prevent more and more misconfigurations in microservices.

# References


[1] The chart repository guide. `https://helm.sh/docs/topics/chart_repository/`, 2021. [Online; accessed Dec 5, 2021].

[2] Cloud native security whitepaper. `https://tinyurl.com/CNCF-CNSWP`, 2021. [Online; accessed nov 28, 2021].

[3] Introduction to kubernetes architecture. `https://www.redhat.com/en/topics/containers/kubernetes-architecture`, 2021. [Online; accessed nov 28, 2021].

[4] What is helm? `https://helm.sh/`, 2021. [Online; accessed Dec 5, 2021].

[5] `https://github.com/julia-grdt/researchproject`, 2022. [Online; created apr 2022].

[6] Checkov tool. `https://www.checkov.io/`, 2022. [Online; accessed jan 21, 2022].

[7] helm template example. https://helm.sh/docs/chart_template_guide/getting_started/, 2022. [Online; accessed feb 21, 2022].

[8] kubernetes resource scans. https://github.com/bridgecrewio/checkov/blob/master/docs/5.Policy{\%}20Index/kubernetes.md, 2022. [Online; accessed jan 21, 2022].

[9] What is checkov. https://www.checkov.io/1.Welcome/What%20is%20Checkov.html, 2022. [Online; accessed feb 26, 2022].

[10] G. Budigiri, C. Baumann, J. T. Muhlberg, E. Truyen, and W. Joosen. Network policies in kubernetes: Performance evaluation and security analysis. In *2021 Joint European Conference on Networks and Communications &amp 6G Summit (EuCNC/6G Summit)*. IEEE, June 2021.

[11] T. Colanzi, A. Amaral, W. Assunção, A. Zavadski, D. Tanno, A. Garcia, and C. Lucena. Are we speaking the industry language? the practice and literature of modernizing legacy systems with microservices. In *15th Brazilian Symposium on Software Components, Architectures, and Reuse*. ACM, Sept. 2021.

[12] V. Garousi and M. V. Mäntylä. When and what to automate in software testing? a multi-vocal literature review. *Information and Software Technology*, 76:92–117, Aug. 2016.

[13] J. Ghofrani and D. Lübke. Challenges of microservices architecture: A survey on the state of the practice. 05 2018.

[14] Osterman Research and Radware. Radware research: The state of web application and api protection. https://blog.radware.com/wp-content/uploads/2021/01/Radware-AppSec-Report-FINAL-1.pdf, 2021. [Online; accessed Dec 5, 2021].

[15] A. Pereira-Vale, E. B. Fernandez, R. Monge, H. Astudillo, and G. Márquez. Security in microservice-based systems: A multivocal literature review. *Computers & Security*, 103:102200, Apr. 2021.

[16] Red Hat Inc. State of kubernetes security report. https://www.redhat.com/rhdc/managed-files/cl-state-kubernetes-security-report-ebook-f29117-202106-en.pdf, 2021. [Online; accessed Dec 6, 2021].

[17] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang. Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS lambda architectures. *Service Oriented Computing and Applications*, 11(2):233–247, Apr. 2017.

[18] Y. Yu, H. Silveira, and M. Sundaram. A microservice based reference architecture model in the context of enterprise architecture. In *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*. IEEE, Oct. 2016.

# A Problem Statement

Include here the Problem Statement guiding the HP project.
See other file.

# B Self-Reflection

I have learned a lot from the research. In my opinion, this project has helped develop me in my writing en how to work more precisely. Moreover, I learned how to think more forward with programming. I have gotten a lot more experience with JavaScript and have learned how to evaluate data with Python. In the end I spend more time on writing the report as expected, this could have been avoided if I planned more precisely. However, for the rest I have followed the project plan quite well.

**Projectplan:**

1. Formulate the research topic and write the proposal - finished on December 6th
2. Setting up the environment and render the Helm charts - finished December 13th
3. Execute the approach of research question 1, sub question A – finished February 10th
   (a) 3 weeks for study of characteristics
   (b) 3 weeks for descriptive analysis
4. Execute approach of research question 2 – finished February 28st
5. Write the Project Report during the process of the research. - finished March 21st