Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2024.055180

ARTICLE

KubeFuzzer: Automating RESTful API Vulnerability Detection in Kubernetes

Tao Zheng[1], Rui Tang[1,2,3], Xingshu Chen[1,2,3,*] and Changxiang Shen[1]

[1]School of Cyber Science and Engineering, Sichuan University, Chengdu, 610065, China

[2]Cyber Science Research Institute, Sichuan University, Chengdu, 610065, China

[3]Key Laboratory of Data Protection and Intelligent Management (Sichuan University), Ministry of Education, Chengdu, 610065, China

*Corresponding Author: Xingshu Chen. Email: chenxsh@scu.edu.cn

Received: 19 June 2024 Accepted: 14 September 2024 Published: 15 October 2024

ABSTRACT

RESTful API fuzzing is a promising method for automated vulnerability detection in Kubernetes platforms. Existing tools struggle with generating lengthy, high-semantic request sequences that can pass Kubernetes API gateway checks. To address this, we propose KubeFuzzer, a black-box fuzzing tool designed for Kubernetes RESTful APIs. KubeFuzzer utilizes Natural Language Processing (NLP) to extract and integrate semantic information from API specifications and response messages, guiding the generation of more effective request sequences. Our evaluation of KubeFuzzer on various Kubernetes clusters shows that it improves code coverage by 7.86% to 36.34%, increases the successful response rate by 6.7% to 83.33%, and detects 16.7% to 133.3% more bugs compared to three leading techniques. KubeFuzzer identified over 1000 service crashes, which were narrowed down to 7 unique bugs. We tested these bugs on 10 real-world Kubernetes projects, including major providers like AWS (EKS), Microsoft Azure (AKS), and Alibaba Cloud (ACK), and confirmed that these issues could trigger service crashes. We have reported and confirmed these bugs with the Kubernetes community, and they have been addressed.

KEYWORDS

Kubernetes; RESTful APIs; API fuzzing; black-box fuzzing

1 Introduction

Kubernetes, also known as K8s, is an open-source software system hosted by the Cloud Native Computing Foundation (CNCF) for managing containerized applications across multiple hosts [1]. It provides the primary mechanism for application deployment, maintenance, and scaling. Kubernetes is frequently used in real-world container cloud platforms, as it can dramatically accelerate deployments by removing the burden of manual, repetitive operations during container deployment. According to the survey [2], organizations such as the US Department of Defense use Kubernetes to manage their application deployments, reducing their release time from 3–8 months to one week for 37 projects, saving more than 100 years.

Due to the massive adoption of Kubernetes, it has become a significant target for hackers carrying out cloud attacks in recent years [3]. From the official CVE website, it can be found that of the 179 existing Kubernetes-related CVEs, 40 are caused by API design vulnerabilities. Among the 24,995 issues related to bugs in the official Kubernetes community, 4666 bugs were caused by the API [4]. Consequently, the security issues stemming from the Kubernetes API design are growing more critical, necessitating a prompt solution in the form of an automated API fuzzing tool specifically tailored for Kubernetes. This tool would identify inherent design flaws within the Kubernetes API. Current API testing tools face efficiency challenges when applied to REST API vulnerability mining for Kubernetes. More specifically, these tools first take the OpenAPI specification or API blueprint as test input, use various strategies to generate random test cases to execute the API endpoints defined in the document, and then determine whether unexpected errors are triggered to uncover problems in the API design with the returned response information. However, these tools randomly select the value of each parameter when generating a request, making it almost impossible to generate valid test cases that pass the syntax and semantic checks of the Kubernetes API gateway. For example, RESTler [5] first parses the input OpenAPI document and infers producer-consumer dependencies between operations. It then uses a search-based algorithm to random generate a request sequence that matches the test dependencies for testing. However, because RESTler randomly picks values from the dictionary to render the request sequence with parameters, the request sequence it constructs is not semantic and cannot pass the Kubernetes API server. Due to insufficient semantic information, current RESTful API fuzzing tools are unable to generate high-quality requests for Kubernetes.

During the testing process, we observed that the return information from failed REST API request sequences often contains rich semantic details that could significantly enhance our testing tools. For instance, a failed request with a 400 status code and a return message including terms such as *bad request* or *content* reveals the underlying cause of the failure. Specific phrases like *content-length* or *invalid-param* in the *content-type* header can directly pinpoint the nature of the error. By leveraging this information, our testing tools could be guided to generate more semantically informed request sequences. However, relying on either manual or syntax rule parsing tools makes it difficult to deal with the huge number of API return data values. In addition, the problem of constructing valid request sequences in an oriented manner while extracting API data also needs to be addressed.

In contrast to other studies, our work focuses on generating REST API requests specifically designed to bypass the Kubernetes API gateway and efficiently detect API vulnerabilities. How to extract useful semantic information values from the return values of failed request sequences so as to pass the inspection of the API gateway is the critical issue to be addressed in this research. To fill this gap, we propose KubeFuzzer, a comprehensive RESTful API fuzzing tool based on CoreNLP (an open-source NLP framework designed by Stanford) for Kubernetes API testing scenarios. The workflow of KubeFuzzer comprises three key procedures. Firstly, it acquires API specification documents and gathers pertinent information about API test responses. Secondly, leveraging CoreNLP, KubeFuzzer extracts valid values from the specification documents and response information that conform to the syntax and semantics. Lastly, it dynamically incorporates these valid values into test cases to generate well-crafted requests. To evaluate the vulnerability detection performance of KubeFuzzer, we selected three state-of-the-art API testing tools (RESTler [5], Resttestgen [6], RESTest [7]) to compare with it. Meanwhile, we locally built two different versions of Kubernetes clusters (stable Kubernetes cluster V1.18.5 and the popular version of Kubernetes cluster V1.25.3) and executed API fuzzing on these clusters using KubeFuzzer. The results show that KubeFuzzer found 901 and 1252 service crashes, respectively. All bugs were categorized, de-duplicated and responsibly submitted to the Kubernetes community, where they were confirmed and fixed by community staff. In summary, our contributions are as follows:

1. An automated vulnerability fuzzing tool for Kubernetes REST APIs is proposed that utilizes NLP techniques to solve the parameter semantics problem, which can improve the efficiency of vulnerability detection.
2. The performance of KubeFuzzer is fully evaluated. Experiments prove that it outperforms the state-of-the-art in terms of the number of valid requests generated, code coverage, and the number of bug found.
3. A large-scale empirical evaluation experiment was conducted. Multiple types of REST API vulnerabilities were discovered in a large number of real Kubernetes services using KubeFuzzer, all of which have been identified and fixed by the vendor.

2 Related Work

2.1 REST API Fuzzing

Black-box testing methods do not require access to source code and are therefore more useful in practical testing scenarios. Several black-box RESTful API tools have been proposed to generate the meaningful RESTful API requests sequence. RESTler [5] builds the API call sequences with a bottom-up approach which starts with single operation call sequences and gradually extend the call sequences by appending more operations after trial and error. RESTTESTGEN [6] generates the API call sequences with Operation Dependency Graphs (ODGs) to model RESTful services and build the call sequence with graph traversal. Compared to the above approaches, RESTest [7] supports automated management of inter-parameter dependencies and supports deeper and faster evaluation of APIs by systematically generating combinations of valid and invalid inputs. White-box testing approaches for RESTful APIs are not common in the literature [8]. Morest [9] builds the call sequences with RESTful-service Property Graph (RPG) and can enjoy both high-level guidance and the flexibility of dynamic adjustments to achieve better performance. Pythia [10] for RESTler uses grammar-based syntax fuzzing, where each property is filled with fixed value types, such as strings or numbers. These values are sourced from user dictionaries or API specifications and remain unchanged over time, lacking any prioritization. This approach does not address which requests or request parts should mutate, nor does it determine the extent of mutation required. Some black-box tools [11] proposed to analyze the output returned by the service after similar requests. When inconsistencies between these outputs were detected, they managed to find errors, e.g., the API returned more data when filters were used than when filters were not applied. Other tools [12–14] have also been subsequently improved for invalid input, extracting attribute values from historically valid responses to populate subsequent attribute inputs, and adding additional security checkers to check for fixed errors.

2.2 Kubernetes Security

The security of Kubernetes clusters is increasingly gaining attention from researchers [15]. For example, researcher studied cluster resource management and scheduling [16]. Shamim [17] studied the current status and future perspectives of machine learning applications to clusters, and also studied the recent practice of knowledge-based security for Kubernetes clusters. Zhang et al. [18] presented a method for generating recommended configuration policies for clusters, and Viktorsson et al. [19] deploied three runtimes in the same Kubernetes cluster, the security focused Kata and gVisor, as well as the default Kubernetes runtime *runC* to understand the current state of the technology in order to make the right decision in the selection, operation and/or design of platforms—and to scholars to illustrate how these technologies evolved over time. In addition, there are researchers who utilize graph mining [20] and deep learning methods to study the detection methods of cluster insecure policies [21]. In terms of dynamic risk detection, Yang et al. [22] proposed for the first time a cluster takeover method based on excessive privileges, Zeng et al. [23] studied on full-stack vulnerability of the cloud-native platform, and He et al. [24] proposed a cluster attack method using ebpf, which is capable of realizing cross-cloud attacks on clusters. Li et al. [25] systematically studied the container security defense mechanism based on Kubernetes. Lyu et al. [26] used a deep learning algorithm to collect response data from historical valid request sequences, while Pan et al. [27] developed an efficient API vulnerability detection tool for data exposure vulnerabilities. Previous methods reveal two critical aspects for effective API vulnerability detection in Kubernetes: (i) bypassing API gateway authentication, and (ii) constructing highly semantic requests.

3 Design and Implementation of KubeFuzzer

3.1 Challenges and Overview

To achieve efficient automated vulnerability detection for Kubernetes' REST APIs, we have the following key challenges (C1–C3):

C1: **Enhancing parameter semantics.** How to generate parameters with semantics that populate the request sequence, which in turn is semantically checked by Kubernetes' API gateway is the primary challenge for KubeFuzzer.

C2: **Generating valid request sequences.** Another key challenge for KubeFuzzer is to render effective request sequence templates in order to detect as many deep vulnerabilities as possible under complex API operations.

C3: **Optimizing response analysis.** How to automate the analysis of valid information in request responses to dynamically feedback to parameter semantic enhancement tasks is a key challenge for KubeFuzzer.

The proposed KubeFuzzer is illustrated in Fig. 1. During the semantic extraction phase, KubeFuzzer uses NLP technology to extract semantic information from the OpenAPI specification document. This information is then utilized in the test case generation phase to create a new OpenAPI specification and a fuzz dictionary, which are used to generate high-quality request sequence test cases. These test cases are sent to the API server for testing, and KubeFuzzer collects and analyzes the response messages. Detected errors are reported for bug analysis. Additionally, ***4XX*** responses are filtered and fed back into the semantic extraction phase to extract enhancements from these responses, which are then used to refine and generate higher-quality test cases.

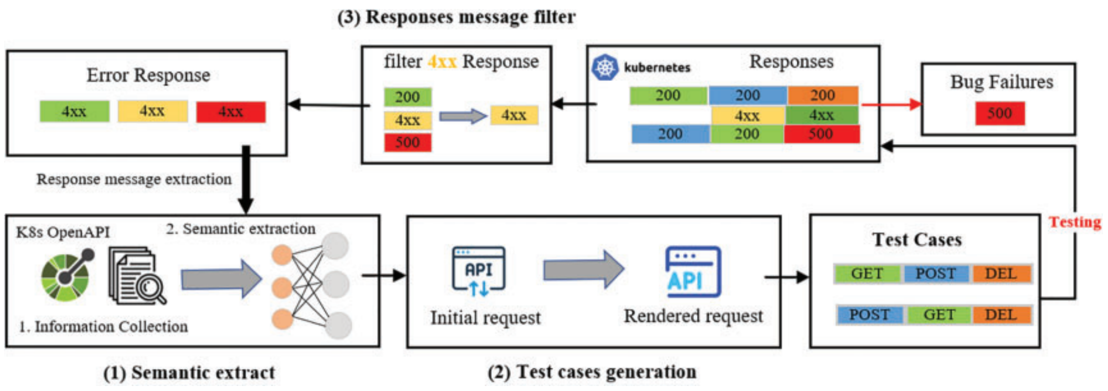

Figure 1: The workflow of KubeFuzzer

3.2 Semantic Extract with NLP

To address Challenge 1, this module initially gathers description information from the OpenAPI specification documents and message values from the request responses. It subsequently extracts semantic information associated with the API request parameters from the collected sequence of information. Therefore, this module encompasses two primary sub-modules: Information Collection and Semantic Extraction.

Specifically, 1) Information collection module collects the description fields corresponding to all parameters in the specification document and the message fields in the returned test case responses. 2) Semantic extraction module extracts the semantic information from the original information collected in sub-module Information collection. More specifically, inspired by Wong et al. [16], we introduced the NLP function uses the word tokenization model, part-of-speech model and dependency parsing model to extract semantic information. The description and message fields are processed by the NLP dependency parsing module to produce a dependency syntax tree consisting of dependency relationships, as shown in Fig. 2. A dependency relationship connects two words: **head word**, **dependent word**, and from **head word** to **dependent word**.

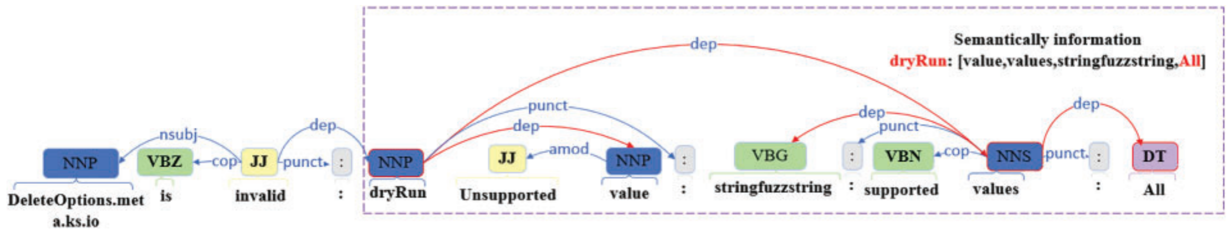

Figure 2: Parameter and semantic information dependence analysis

When we extract the semantic information of a parameter, we take the parameter as the head word and the semantic information as the set of dependent words that depend on the head word. But owing to format inconsistencies, the semantic information of certain parameters may not exhibit a direct correspondence with their respective description or message fields. To elaborate, in Fig. 2, the message field manifests the presence of the "*dryRun*" parameter and the recommended parameter "All". We use "*dryRun*" as the head word and the "*All*" should be his dependent word. Nonetheless, it has been noted that the value of the "*All*" parameter does not overtly rely on the "*dryRun*" parameter, but rather interfaces with it via the "*values*" field. Given this premise, we define the set of semantic information M as follows:

$$\mathrm{M} = \{x | \varepsilon \leq 2\} \tag{1}$$

where x represents the semantic information that satisfies the condition and ε represents the dependency distance between the parameter and the semantic information. When extracting, it is determined that the dependent words with a distance of less than or equal to 2 from the head word are the semantic information of the head word. Once ε more than 2, more words can be included but there is a loss of accuracy. By the definition, we can extract the set of semantic information about the parameter "*dryRun*":["value","values","stringfuzzstring","All"].

Our selection of Stanford CoreNLP was driven by key considerations. Firstly, CoreNLP excels in syntactic parsing and semantic role labeling, crucial for accurately interpreting complex natural language instructions in our study. Its holistic approach to semantic understanding aligns well with our requirements, surpassing other frameworks that may offer specialized functionalities. Secondly, CoreNLP's extensive library of pre-trained models, refined over years, ensures stability and efficiency, especially with large datasets. While alternative frameworks may have similar features, their performance in our specific context may not be as robust. Lastly, CoreNLP's clean and intuitive API facilitated seamless integration with our Kubernetes API testing environment, reducing development costs and complexity during implementation.

As shown in Algorithm 1, KubeFuzzer extract first defines *mess_list* and *para_list*. *mess_list* is used to store the description field collected by the collection function and the contents of the message field in the response message. *para_list* stores the values of the parameters involved in the API documentation, then defines the list of dependency relations to filter, *noMeaningRels*, including *det*, *cc*, *punct*. Next, word separation, lexical inference and dependency inference are performed on each message. The *id*, *value*, *pos*, head of each word in the message is obtained according to the inferred result. Iterate through each word in the message, and if the current word matches the element in the *para_list*, the current word is taken as the root node, and the id of that node is taken as the *root_id*. Again, iterate through each word in the message, determine if the root word id with distance less than or equal to 2 is the same as *root_id* and the *deprel* does not exist in the *noMeaningRels*, if the condition is satisfied, the word is considered as possibly containing semantic information and extracted.

Algorithm 1: Extract the semantic message with NLP

```
Input: Responses file with 4XX, Description value in API Specification
Output: Semantic enhanced information
1:  $para\_list \leftarrow \varnothing$
2:  $noMeaningRels \leftarrow$ [“cc”, “det”, “punct”]
3:  $mess\_list \leftarrow \varnothing$
4:  for message in mess_list do
5:      $text\_result \leftarrow$ nlp.word_tokenize (message)
6:      $pos\_result \leftarrow$ nlp.pos_tag (message)
7:      $deprel\_result \leftarrow$ nlp.dependency_parse (message)
8:      $word\_dir \leftarrow \{\}$
9:      $word\_list \leftarrow \varnothing$
10:     for id in range (len(text_result)) do
11:         $word\_dir \leftarrow$ {“id”, “value”,“pos”,“head”}
12:         word_list.append (word_dir)
13:     for para in para_list do
14:         for word in word_list do
15:             if word equal para then $root\_id \leftarrow word$ [‘id’]
16:         for word in word_list do
17:             if word [‘deprel’] in noMeaningRels then continue
18:             else if word [‘head’] equal root_id then return word [‘value’]
19:             else if result_list [word [‘head’] $-$ 1] [‘head’] equal root_id then
20:                 return word [‘value’]
```

3.3 Test Cases Generation

To address Challenge 2, similarly, as is shown in Algorithm 2, this module contains two main submodules: **Request Sequence Generation** and **Request Sequence Tendering**. KubeFuzzer first generates the request sequence template based on the dependencies defined in the RESTler [5] by parsing the OpenAPI specification. After that, it renders each request in the template respectively.

Request Sequences Generation. This sub-module first iterates through *req_seqs* and *req_list* to get the current request sequence template (*seq*) and the Kubernetes API request (*req*), and then determines if the *req* matches the dependency with *seq*. If it does, *req* is added to *seq*, then a new request sequence template is assembled. The specific parameter value of each request in the request sequence template is replaced with a placeholder and waits for the *request_sequences_render* function to render it. Upon successful rendering, the request sequence collection adds the newly constructed request sequence to complete this process.

Request Sequences Render. This sub-module renders each request in the API request sequence, replacing the attribute value placeholders of each request in the request sequence with specific parameter values. Inspired by [3], we introduce a dynamic feedback dictionary and a document-based approach to populating parameter values. The dynamic feedback dictionary stores semantic information about the parameters extracted from the extraction module of the parameter composition mapping dictionary. During request rendering, the specific values of the parameters are populated by matching the same parameters in the request against the dynamic feedback dictionary. Since the Swagger documentation supports the ability to specify default values for input parameters and examples of values to be used, and if these values are specified, test operations can be attempted. Therefore, the document-based approach also makes use of the semantic information extracted by the Extract module and, based on the semantic information, modifies the default values of the response parameters and provides example values for the parameters as a way of constructing a new OpenAPI specification document to be parsed again and thus populated with the default values during the rendering process. Iterate through each parameter in the request, and if the parameter has a sample value in the specification, KubeFuzzer will use the sample value to render it. If there is a mapped value in the dynamic feedback dictionary, KubeFuzzer uses the mapped value to fill, if there is none, it uses the random value to fill.

Algorithm 2: Test cases generation

```
Input: $K8s\ OpenAPI\ Specification$
Output: $Request\ Sequence$
1:   $req\_list \leftarrow$ Analysis ($K8s\_Spec$)
2:   $req\_seqs \leftarrow \{\}$
3:   def request_seq_generate($req\_seqs$, $req\_list$):
4:       for $seq$ in $req\_seqs$ do
5:           for $seq$ in $req\_seqs$ do
6:               if $seq$ is $empty$ then
7:                   $new\_seq \leftarrow$ concat ($seq$, $req$)
8:                   $request\_seq\_render$ ($new\_seq$)
9:                   $req\_seqs.append$ ($new\_seq$)
10:              else if consums ($req$) $\in$ produces ($req$) then
11:                  $new\_seq \leftarrow$ concat ($seq$, $req$)
12:                  $request\_seq\_render$ ($new\_seq$)
13:                  $req\_seqs.append$ ($new\_seq$)
14:      def request_seq_render($seq$):
15:          for $req$ in $seq$ do
```

(Continued)

Algorithm 2 (continued)

```
16:       for para in spec do
17:         if para in spec then
18:           put example value in req
19:         else if para in dict then
20:           put dict value in req
21:         else
22:           put fuzzing value in req
```

3.4 Response Message Filter

To address Challenge 3, this module first reads all the responses generated by the test cases from the log file. It then matches all responses starting with the message about "*HTTP/1.1 4XX*" by constructing regular matching rules. Finally, the successful matches are written to a file and returned to the Extract module. More specifically, for all the responses returned by the test cases with status code ***2XX***, we consider that all the parameters in the request meet the semantic requirements of Kubernetes. For the response with a status code of ***500***, we consider that the parameters in the request may be the specific cause of the error, and even if they do not meet the semantic requirements of Kubernetes, we need to retain their specified values in order to analyse the cause of the error. Therefore, responses with status codes ***2XX*** and ***500*** are not targeted for collection. The ***4XX*** responses often contain information about Kubernetes' semantic requirements for the parameters.

4 Experiment

We have implemented KubeFuzzer based on Python 3.9.0 and conducted experiments to evaluate the bug detection performance of KubeFuzzer. We select Restler [5], RESTTESTGEN [6], and RESTest [7] as baseline tools. We design several experiments and try to answer the following questions to validate our contributions in different aspects:

RQ1 (Coverage): How is the code coverage and successful response rate capability of KubeFuzzer compared with baselines?

RQ2 (Bug Detection): How is the bug detection capability of KubeFuzzer compared to the baselines?

RQ3 (Ablation Study): How do input guidance and response feedback affect the performance of KubeFuzzer separately?

4.1 Evaluation Settings

Evaluation Criteria. We use the following three criteria to evaluate KubeFuzzer and the baselines: (1) Code Coverage. Since line coverage is the most refined performance metric for black-box testing tools, we use line coverage to reflect the state space exploration capability of KubeFuzzer. We combine *SonarQube* (A Code coverage tool for GO) with *integration test method* (Test Coverage of Go Services during Integration Tests) to test code coverage. (2) Successful Response Rate (SRR). We applied SRR to reflect whether KubeFuzzer can generate valid requests to test the deeper code logic of the RESTful service, and valid requests are partly the result of correct call sequences. When a request returns a status code of ***2XX***, we define it as a successful request. As the number of request cases generated by each tool varies, we use the ratio of the number of successful requests to the number of request cases as an

SRR indicator. (3) Bug Number. The number of unique bugs found is the most important measure of a black-box testing tool. If an API request is detected and a ***5XX*** status code is returned, we assume that the API request is causing a service failure. In our experiments, we manually classify errors as unique bugs based on the response body, server logs, etc.

Kubernetes Cluster. We found that the Kubernetes versions involved in the services that are orchestrated using the Kubernetes native API, provided by the well-known cloud service providers are in the range of 1.18–1.25. Therefore, we set up the evaluation benchmark with the stable and the popular Kubernetes cluster (V1.18.5 and V1.25.3). We hosted these two clusters on a local machine and ran each technique with three time budgets—8, 16 and 24 h. Each evaluation lasts for 48 h on a docker container configured with 8 CPU cores, 20 GB RAM, Python3.8.2, and the OS of Ubuntu16.04 LTS.

4.2 Coverage (RQ1)

4.2.1 Code Coverage

To comprehensively compare the performance of KuberFuzzer with the baselines in terms of code coverage, we ran this experiment 10 times of 8, 16, and 24 h on each tool to measure the code coverage of two different versions of kubernetes, and took the average value to represent the code coverage of each tool. Fig. 3, Tables 1 and 2 depict the code coverage of the four tools, due to space limitations, Table 2 only shows the results of 10 run times of the four tools, each time for 24 h. We define α as the average code coverage of KubeFuzzer, and β as the average code coverage of baseline technology. We calculate the average increased code coverage number (ρ) by:

$$\rho = \frac{\alpha - \beta}{\beta} \times 100\% \quad (2)$$

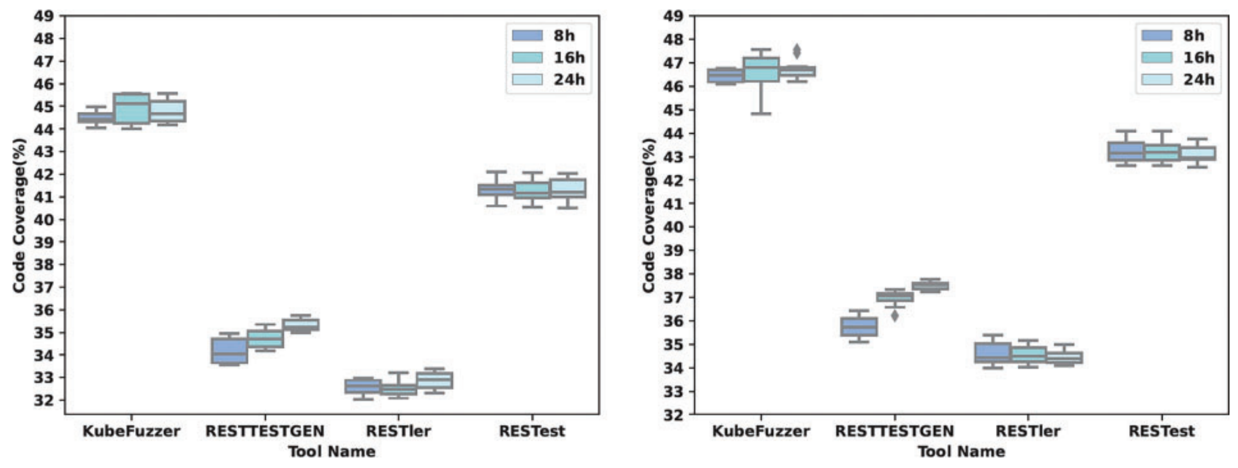

Figure 3: Code coverage under stable (Left) and popular Kubernetes versions (Right)

Table 1: Compare KubeFuzzer and baseline code coverage over 8, 16, and 24 h in V1.18.5. KF: KubeFuzzer, RL: RESTler, RT: RESTest, RG: RESTTESTGEN

Loc-8 h				Loc-16 h				Loc-24 h			
KF	RL	RT	RG	KF	RL	RT	RG	KF	RL	RT	RG
44	35	33	41	45	36	34	43	45	35	33	44
45	35	32	41	46	35	33	43	46	33	33	44
45	34	33	41	45	37	34	42	46	36	34	43
44	35	32	42	46	37	36	43	46	35	34	43
44	34	33	42	47	36	34	42	45	36	35	42
45	35	32	42	47	37	35	44	46	34	34	41
44	35	33	40	47	38	35	44	45	35	35	42
45	34	32	41	47	36	34	43	46	36	34	42
44	35	32	41	46	36	35	43	45	35	33	41
44	35	32	41	47	36	34	44	46	36	34	42

Table 2: Compare KubeFuzzer and baseline code coverage over 8, 16, and 24 h in V1.25.3. KF: KubeFuzzer, RL: RESTler, RT: RESTest, RG: RESTTESTGEN

Loc-8 h				Loc-16 h				Loc-24 h			
KF	RL	RT	RG	KF	RL	RT	RG	KF	RL	RT	RG
46	35	34	43	46	36	33	42	46	37	32	42
46	35	34	44	46	36	34	42	46	37	32	43
46	36	35	42	46	35	35	43	47	37	32	43
46	35	34	42	47	36	34	42	47	37	33	42
46	36	35	43	47	37	35	43	46	37	33	43
46	35	35	43	46	36	36	42	46	37	32	43
46	36	34	43	46	36	35	42	46	37	33	43
46	35	34	43	47	35	36	42	46	36	33	44
46	35	35	43	46	37	34	43	46	37	32	43
46	35	34	42	44	37	35	42	46	37	33	42

From Tables 1 and 2, we observed that KubeFuzzer outperformed the baseline tool in terms of code coverage, but all tools did not achieve high coverage in general. By observing test cases, we identified the key factors responsible for these low coverage rates. In addition, because the baseline tools use dictionary and random generation methods to generate parameter values, in the absence of semantic information to guide them, they generate many invalid requests that are rejected by the Kubernetes API Server. In most cases, this is the case when parameters have specified restrictions on value and format. For example, for the "*dryRun*" parameter, Kubernetes only allows *null* and *All* as parameter values. The baselines waste a lot of time trying to pass this check and still fail.

4.2.2 SRR

In Tables 3 and 4, we have tested the Request Cases (rc), Successful Requests (sr) and their ratio SRR (srr) of the four tools at 1, 4, 8, 16 and 24 h, respectively. Firstly, analysing the three baseline tools, we find that the Request Cases generated by RESTest are much smaller than RESTler and RESTTESTGEN, but the SRR is higher than RESTler and RESTTESTGEN. By analysing the test cases, we found that the reason for this result is that RESTler and RESTTESTGEN generate request sequences based on dependencies between dependent operations, and construct request sequences by adding a new request template at the end of the current request template to construct a new sequence.

Table 3: Successful request operation of different Tool in V1.18.5. KF: KubeFuzzer, RL: RESTler, RT: RESTest, RG: RESTTESTGEN

Tools		1 h	4 h	8 h	16 h	24 h
KF	rc	32,864	87,892	157,912	196,489	211,942
	sr	821	1560	2002	2002	2002
	srr	2.50	1.77	1.27	1.02	0.94
RL	rc	8896	16,888	29,894	38,878	40,839
	sr	32	40	55	55	55
	srr	0.36	0.24	0.18	0.14	0.13
RT	rc	2687	6879	8488	9968	10,564
	sr	26	46	55	61	61
	srr	0.97	0.67	0.65	0.64	0.61
RG	rc	695	1302	1521	1564	1684
	sr	56	102	117	117	117
	srr	8.06	7.83	7.69	7.48	6.95

On the one hand, if the last request in the sequence template fails Kubernetes' syntactic semantic check, it will infer that the sequence template currently in use is invalid, interrupt the sequence expansion process, and discard this sequence. Thus, in the absence of semantic information to guide them, RESTler and RESTTESTGEN will generate a large number of invalid test cases, causing the request at the beginning of the sequence to be an invalid request. Other requests that depend on this request cannot be constructed, resulting in some operations in some API documents that are not covered. On the other hand, RESTest generates test cases based on individual API operations, enabling it to cover a broader range of operations and increasing the likelihood of successful requests. Similarly, RESTTESTGEN produces a smaller total number of requests, with a higher proportion of legitimate ones, contributing to its higher SRR compared to RESTler.

Table 4: Successful request operation of different tool in V1.25.3. KF: KubeFuzzer, RL: RESTler, RT: RESTest, RG: RESTTESTGEN

Tools		1 h	4 h	8 h	16 h	24 h
KF	rc	33,453	88,962	166,527	198,452	333,014
	sr	876	1596	1952	2014	4996
	srr	2.62	1.79	1.17	1.01	1.15
RL	rc	8642	17,866	24,561	33,561	36,100
	sr	30	58	67	67	67
	srr	0.35	0.32	0.27	0.20	0.18
RT	rc	5689	15,879	27,891	30,564	32,487
	sr	68	147	208	221	221
	srr	1.19	0.92	0.75	0.72	0.68
RG	rc	615	921	1279	1360	1483
	sr	47	69	95	95	95
	srr	7.64	7.56	7.43	6.99	6.41

As is shown in the Formula (3), we define SR as the successful requests, and SR_I as the increased degree of successful requests:

$$SR\_I = \frac{SR(24\,\text{h}) - SR(1\,\text{h})}{SR(1\,\text{h})} \times 100\% \tag{3}$$

Therefore, we can obtain the SRR by:

$$SRR = \frac{SR\_I(\text{KubeFuzzer}) - SR\_I(\text{baseline})}{SR\_I(\text{baseline})} \times 100\% \tag{4}$$

We collected code coverage and SRR data for KubeFuzzer and the benchmark tools over 10 time periods. To evaluate the superiority and stability of KubeFuzzer, we conducted independent samples *t*-tests comparing its code coverage and SRR against RESTler, RESTest, and RESTTESTGEN. Using Python's *scipy.stats* package, we calculated the *p*-values for each comparison, resulting in (0.03, 0.04), (0.02, 0.03), and (0.04, 0.04), respectively. These results indicate that KubeFuzzer significantly outperforms the benchmark tools in both code coverage and SRR.

Based on these results, we can answer **RQ1** as: Compared to the baselines, the code coverage of KubeFuzzer improved by 7.86%–36.34%, 7.78%–43.13%, and the SRR improved by 6.7%–83.33% and 108.89%–360.78% on the two clustered versions, respectively.

4.3 Bug Detection (RQ2)

4.3.1 Bug Detection in Local K8s Cluster

We analyse the bug detection capabilities of KubeFuzzer in this section. Table 5 shows the number and type of bugs detected by KubeFuzzer and baseline tools in the two versions of Kubernetes cluster.

Table 5: Number and type of bugs detected by KubeFuzzer and baselines

K8s version	Tool	Number	Type
V1.18.5	KubeFuzzer	901	7
	RESTler	18	5
	RESTest	189	6
	RESTTESTGEN	27	3
V1.25.3	KubeFuzzer	1113	7
	RESTler	1	1
	RESTest	192	6
	RESTTESTGEN	168	2

We define $N(x)$ as the bug number found by KubeFuzzer and the baseline tools, as is shown in Formula (5), our calculations indicate that KubeFuzzer can detect 16.7% to 133.3% more bugs on the stable version of Kubernetes clusters and 16.7% to 600% more on the popular version.

$$\omega = \frac{N(KubeFuzzer) - N(baseline)}{N(baseline)} \times 100\% \tag{5}$$

We can conclude that all three baseline tools were able to find bugs on both two Kubernetes clusters, but the number and type of bugs found differed. The bugs found by RESTest are better than RESTler and RESTTESTGEN in terms of number and type. More specifically, as analysed in **RQ1**, when extending a request sequence, it can only be added to the RESTler and RESTTESTGEN request sequences if the current request is a valid request, but due to the lack of semantic information in the generated requests, most requests fail Kubernetes' semantic checks and are simply discarded. RESTest is based on a single API request rather than building sequences for testing, which is usually not a disadvantage, but when testing Kubernetes, RESTestest can trigger more bugs because a single API request can cover most of the operations in the API documentation. KubeFuzzer uses the Extract module to extract a large amount of semantic information from the request built sequences that can effectively pass the Kubernetes API server's semantic checks, so it detects a much larger number and type of bugs than baselines.

4.3.2 Bug Detection in Real-World K8s Service

The experimental validation is conducted on a set of RESTful APIs. However, considering that test cases are supposed to exploit a security vulnerability, it would not be ethical to test publicly cloud services that are orchestrated using the Kubernetes native API, since their integrity could be compromised by a successful attack. Hence, we opted for case studies that we can download or run in a controlled environment. With these considerations in mind, to explore the bug detection capabilities of KubeFuzzer in real-world project, we constructed 7 API requests that contained unique bugs found by KubeFuzzer, first, we constructed a controlled experimental environment in 6 real-world cloud service projects (as shown in Table 6), and then used the 7 API requests unobtrusively to continuously access the Kubernetes cluster in a controlled container environment, and then we looked for open-source projects hosted on GitHub, so that we can compile and install them locally. We found that these API requests also triggered bugs in these 4 real-world projects.

Table 6: Bug detection in real-world projects that are orchestrated using the K8s native API

Case study	Version	Type	Bug status
Amazon EKS	1.18.5 1.25.3	1,2,3,6,7	Fixed
Azure AKS	1.18.5 1.25.3	1,2,4,6,7	Fixed
Alibaba ACK	1.25.3	1,2,5,6,7	Fixed
Google GKE	1.18.5	1,2,4,6,7	Fixed
Tencent TKE	1.18.5 1.25.3	1,2,3,6,7	Fixed
Oracle OKE	1.18.5 1.25.3	1,2,3,6,7	Fixed
Kubeflow	1.18.5	1,2,6,7	Confirmed
KubeVirt	1.18.5 1.25.3	1,2,7	Fixed
KubeMQ	1.25.3	1,2	Fixed
KubeArmor	1.18.5 1.25.3	1,2,5,6,7	Fixed

Thus, we can conclude **RQ2** by stating that KubeFuzzer detects 16.7% to 133.3% (and up to 600%) more unique bugs compared to the three baseline tools. Moreover, these bugs can be validated on real-world projects orchestrated using the Kubernetes native API.

4.4 *Ablation Study (RQ3)*

We implemented three variants of KuberFuzzer to evaluate the contribution of semantic information in the API specification document and the response message:

(1) KubeFuzzer-1: No-Input version by removing the extraction of semantic information from the API specification document in the Extract module.

(2) KubeFuzzer-2: No-Output version by removing the extraction of semantic information from the response message in the Extract module.

(3) KubeFuzzer-3: No-All version by directly removing the Extract module.

The results are averaged using five runs (with time budget 24 h) to avoid the statistics bias. For better visualisation and understanding, we divided the SRR of the three variant tools by the SRR of KubeFuzzer as the *normalised SRR* value (π).

$$\pi = \frac{\pi(\text{variant tools})}{\pi(\text{KubeFuzzer})} \times 100\% \tag{6}$$

Table 7 and Fig. 4 display the normalized SRR of KubeFuzzer and its variants across different Kubernetes versions. KubeFuzzer-1 and KubeFuzzer-2 outperformed KubeFuzzer-3 but lagged behind KubeFuzzer. Integrating Kubernetes semantic information into API specs and responses improves detection efficiency, highlighting the need for comprehensive inclusion. KubeFuzzer-2's weaker performance compared to KubeFuzzer-1 is attributed to less semantic information in input documents *vs*. output responses.

Table 7: Number of bugs in the three variants of KubeFzzer

K8s version	KubeFuzzer	KubeFuzzer-1	KubeFuzzer-2	KubeFuzzer-3
V1.18.5	901	368	850	18
V1.25.3	1252	496	1113	11

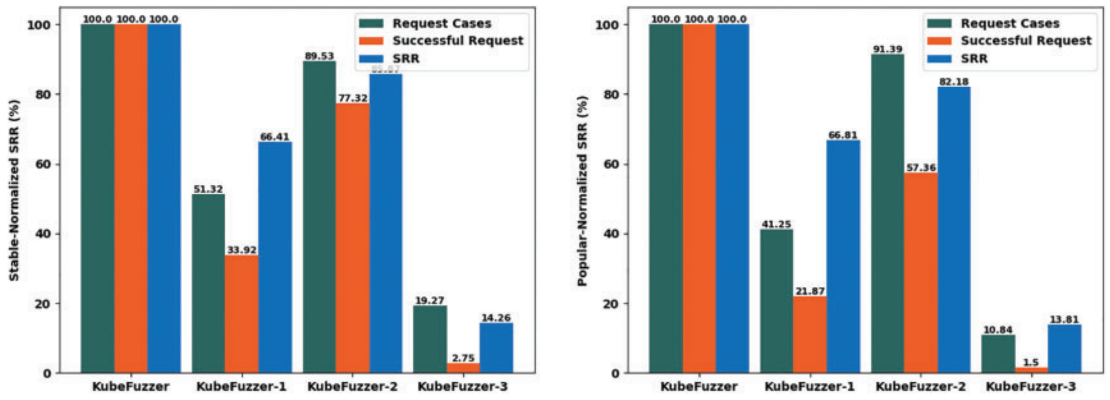

Figure 4: Normalized SRR for three variants of KubeFzzer under different K8s version

From these findings, we can address **RQ3** as follows: Overall, semantic details in both API specification documents and response messages impact KuberFuzzer's performance. While both play a role, the richness of semantic content in response messages notably influences KuberFuzzer's effectiveness. Optimal performance is attained when all semantic information is fully integrated.

5 Discussion

Unlike traditional RESTful API testing solutions that typically identify vulnerabilities ***5XX*** error codes in Table 8 [4,14,26,27], KubeFuzzer does not focus on this approach. In practice, RESTful API vulnerabilities often lead to exploit attacks, particularly DoS attacks, due to data mishandling on the server side. Our goal is to automate the discovery of RESTful vulnerabilities on Kubernetes without manual analysis or exploit development. In future work, we will focus on the challenges we may face when identifying and responding to RESTful API vulnerabilities during Kubernetes migrations, as well as possible resolution strategies. This will include insights into potential security threats, vulnerability detection methods in complex environments, and technical means to improve system robustness in real-world applications. In addition, we will also investigate vulnerability-oriented REST API vulnerability mining methods in K8s in conjunction with LLM.

Table 8: Comparison of API testing studies in the last two years

Fuzzer	Conference	Testing approach	Para Generation
NAUTILUS [4]	USENIX SEC 23	Black-box	Dictionary-based
NLPtoREST [14]	ISSTA 2023	Black-box	Dictionary-based

(Continued)

Table 8 (continued)

Fuzzer	Conference	Testing approach	Para Generation
Miner [26]	USENIX SEC 23	Black-box	DNN
EDEFuzz [27]	ICSE 2024	Black-box	Dictionary-based

6 Conclusion

In this paper, we propose KubeFuzzer to automate detect the REST API vulnerabilities in Kubernetes. Experiments conducted on both local and real-world Kubernetes clusters demonstrate that KubeFuzzer leverages NLP to effectively enhance the semantics of request sequences, allowing them to pass Kubernetes API gateway inspections and significantly improving vulnerability detection performance. Future work will focus on developing effective Kubernetes testing approaches based on RESTful API vulnerabilities.

Acknowledgement: We thank the anonymous reviewers for their useful comments.

Funding Statement: This work was supported by the National Natural Science Foundation of China (No. 62202320), the Fundamental Research Funds for the Central Universities (Nos. SCU2023D008, 2023SCU12129), the Natural Science Foundation of Sichuan Province (No. 2024NSFSC1449), the Science and Engineering Connotation Development Project of Sichuan University (No. 2020SCUNG129), and the Key Laboratory of Data Protection and Intelligent Management (Sichuan University), Ministry of Education.

Author Contributions: The authors confirm contribution to the paper as follows: study conception and design: Tao Zheng; data collection: Rui Tang; analysis and interpretation of results: Xingshu Chen; draft manuscript preparation: Changxiang Shen. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The data that support the findings of this study are available from the corresponding author, Xingshu Chen, upon reasonable request.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security misconfigurations in open source kubernetes manifests: An empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, pp. 1–36, 2023. doi: 10.1145/3607179.

[2] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Comput. Surv.*, vol. 55, no. 7, pp. 1–37, 2022. doi: 10.1145/3539606.

[3] P. Godefroid, B. Y. Huang, and M. Polishchuk, "Intelligent REST API data fuzzing," in *Proc. 28th ACM ESEC/FSE*, USA, 2020, pp. 725–736. doi: 10.1145/3368089.3409719.

[4] G. L. Deng *et al.*, “NAUTILUS: Automated RESTful API vulnerability detection,” in *Proc. USENIX Secur. 23*, Anaheim, CA, USA, 2023, pp. 5593–5609. doi: 10.5555/3620237.3620550.

[5] V. Atlidakis, P. Godefroid, and M. Polishchuk, “RESTler: Stateful REST API fuzzing,” in *Proc. 41th IEEE/ACM ICSE*, Montreal, QC, Canada, 2019, pp. 748–758. doi: 10.1109/ICSE.2019.00083.

[6] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, “RestTestGen: An extensible framework for automated black-box testing of restful apis,” in *Proc. 38th IEEE ICSME*, Limassol, Cyprus, 2022, pp. 504–508. doi: 10.1109/ICSME55016.2022.00068.

[7] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, “RESTest: Automated black-box testing of RESTful web APIs,” in *Proc. 30th ACM ISSTA*, Denmark, 2021, pp. 682–685. doi: 10.1145/3460319.3469082.

[8] A. Arcuri, “RESTful API automated test case generation with EvoMaster,” *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 1, pp. 1–37, 2019. doi: 10.1145/3293455.

[9] Y. Liu *et al.*, “Morest: Model-based RESTful API testing with execution feedback,” in *Proc. 44th IEEE/ACM ICSE*, Pittsburgh, PA, USA, 2022, pp. 1406–1417. doi: 10.1145/3510003.3510133.

[10] V. Atlidakis, R. Geambasu, P. Godefroid, M. Polishchuk, and B. Ray, “Pythia: Grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations,” 2020, *arXiv:2005.11498*.

[11] V. Atlidakis, P. Godefroid, and M. Polishchuk, “Checking security properties of cloud service REST APIs,” in *Proc. 13th IEEE ICST*, Porto, Portugal, 2020, pp. 387–397. doi: 10.1109/ICST46399.2020.00046.

[12] H. Ed-Douibi, J. L. Cánovas Izquierdo, and J. Cabot, “Automatic generation of test cases for REST APIs: A specification-based approach,” in *22nd EDOC*, Stockholm, Sweden, 2018, pp. 181–190. doi: 10.1109/EDOC.2018.00031.

[13] S. Karlsson, A. Čaušević, and D. Sundmark, “QuickREST: Property-based test generation of OpenAPI-described RESTful APIs,” in *Proc. 13th ICST*, Porto, Portugal, 2020, pp. 131–141. doi: 10.1109/ICST46399.2020.00023.

[14] M. Kim *et al.*, “Enhancing REST API testing with NLP techniques,” in *Proc. 32nd ACM ISSTA*, Seattle, WA, USA, 2023, pp. 1232–1243. doi: 10.1145/3597926.3598131.

[15] C. C. Chang, S. R. Yang, E. H. Yeh, P. Lin, and J. Y. Jeng, “A kubernetes-based monitoring platform for dynamic cloud resource provisioning,” in *GLOBECOM 2017—2017 IEEE Global Commun. Conf.*, Singapore, 2017, pp. 1–6. doi: 10.1109/GLOCOM.2017.8254046.

[16] A. Y. Wong, E. G. Chekole, M. Ochoa, and J. Y. Zhou, “On the security of containers: Threat modeling, attack analysis, and mitigation strategies,” *Comput. Secur.*, vol. 128, 2023, Art. no. 103140. doi: 10.1016/j.cose.2023.103140.

[17] S. I. Shamim, “Mitigating security attacks in kubernetes manifests for security best practices violation,” in *Proc. 28th ACM ESEC/FSE*, 2021, pp. 1689–1690. doi: 10.1145/3468264.3473495.

[18] Y. Zhang, R. Meredith, W. Reeves, J. Coriolano, M. A. Babar and A. Rahman, “Does generative AI generate smells related to container orchestration?” in *Proc. IEEE/ACM 21st MSR*, Lisbon, Portugal, 2024, pp. 192–196. doi: 10.1145/3643991.3645079.

[19] W. Viktorsson, C. Klein, and J. Tordsson, “Security-performance trade-offs of kubernetes container runtimes,” in *Proc. 28th IEEE MASCOTS*, 2020, pp. 1–4. doi: 10.1109/MASCOTS50786.2020.9285946.

[20] Ł. Wojciechowski *et al.*, “NetMARKS: Network metrics-aware kubernetes scheduler powered by service mesh,” in *Proc. IEEE INFOCOM*, 2021, pp. 1–9. doi: 10.1109/INFOCOM42981.2021.9488670.

[21] T. Goethals, F. De Turck, and B. Volckaert, “Extending kubernetes clusters to low-resource edge devices using virtual kubelets,” *IEEE Trans. Cloud Comput.*, vol. 10, no. 4, pp. 2623–2636, 2020. doi: 10.1109/TCC.2020.3033807.

[22] N. Z. Yang, W. B. Shen, J. K. Li, X. Q. Liu, X. Guo and J. F. Ma, “Take over the whole cluster: Attacking kubernetes via excessive permissions of third-party applications,” in *Proc. ACM SIGSAC Conf. Comput. Communicati. Security*, Copenhagen, Denmark, 2023, pp. 3048–3062. doi: 10.1145/3576915.3623121.

[23] Q. Zeng, M. Kavousi, Y. Luo, L. Jin, and Y. Chen, “Full-stack vulnerability analysis of the cloud-native platform,” *Comput. Secur.*, vol. 129, no. 5, 2023, Art. no. 103173. doi: 10.1016/j.cose.2023.103173.

[24] Y. He *et al.*, “Cross container attacks: The bewildered eBPF on clouds,” in *Proc. USENIX Secur. 23*, Anaheim, CA, USA, 2023, pp. 5971–5988. doi: 10.5555/3620237.3620571.

[25] Z. Li *et al.*, "Lost along the way: Understanding and mitigating path-misresolution threats to container isolation," in *Proc. ACM CCS*, Copenhagen, Denmark, 2023, pp. 3063–3077. doi: 10.1145/3576915.3623154.

[26] C. Lyu *et al.*, "MINER: A hybrid data-driven approach for REST API fuzzing," in *Proc. USENIX Secur. 23*, Anaheim, CA, USA, 2023, pp. 4517–4534.

[27] L. Pan, S. Cohney, T. Murray, and V. T. Pham, "EDEFuzz: A web API fuzzer for excessive data exposures," in *Proc. IEEE/ACM ICSE*, Lisbon, Portugal, 2024, pp. 1–12. doi: 10.1145/3597503.3608133.