Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Erroneous Kubernetes Object Generation using Structure-aware Fuzzing

Author: Prashanth Varma Dommaraju(2688076)

1st supervisor:	Prof. dr. ir. Alexandru Iosup
daily supervisor:	Ir. Sacheendra Talluri
2nd reader:	Dr. Tiziano De Matteis

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

October 1, 2024

"Problems that remain persistently insoluble should always be suspected as questions asked in the wrong way." — ,

from The Book: On the Taboo Against Knowing Who You Are*, by Alan Wilson Watts*

Abstract

Kubernetes is evolving rapidly. Private and public vendors increasingly rely on custom Kubernetes configurations to support the demand for containerized applications and global distributed systems. Such custom configurations need resilience and reliability. Any misconfiguration can disrupt crucial production workloads. Kubernetes has lower test coverage, thus making it difficult for developers to debug issues or errors. Kubernetes needs faster error localization in the vast code base. Kubernetes' complexity challenges users to debug erroneous configurations and identify issues stemming from software bugs.

We propose an erroneous object generator using the structure-aware fuzzer. We design and implement the structure-aware fuzzer to generate these real-world error configurations and evaluate if Kubernetes can handle such erroneous objects and create the errors. Our experiments demonstrate that the tool can create erroneous objects, and the structure-aware fuzzer can explore more code paths than the String-based input fuzzer. Creating such error objects will help vendors or developers to localize and fix the errors created by these error objects.

Acknowledgements

I want to thank Sacheendra Talluri for guiding and supervising me through such a fantastic project and introducing me to fuzzing and Kubernetes. I also extend my gratitude to my First reader, Prof. Dr. Ir. Alexandru Iosup, and second reader, Prof. Dr. Tiziano De Matteis. I wouldn't have been able to complete this project without the support of family and friends throughout the project, during stressful situations, and during illness. This project pushed me hard to make me understand new perspectives and throw puzzling, challenging scenarios. Grateful for everyone who supported me through this project.

Contents

List of Figures

List of Tables

1

Introduction

Datacenter resource managers such as the Kubernetes (1) are complex software systems. Kubernetes has many functionalities, exceeding 256 features and encompassing thousands of configurations. Comprised of tens of components and millions of lines of code, Kubernetes' complexity challenges users in debugging erroneous configurations and identifying issues stemming from software bugs. This work addresses the technical and scientific problem of optimizing Kubernetes error localization by integrating structured data generation using fuzzing to generate errors in Go (2) components. Current fuzzing methodologies in Kubernetes predominantly rely on Go-fuzz (3) (4) and OSS-fuzzers (5) with unstructured random data, potentially missing errors and bugs due to the lack of structured input data scenarios. Fuzzing is a cornerstone in identifying errors and security issues (6), and incorporating its methodology directly contributes to Kubernetes core APIs' overall efficiency and reliability. The problem is timely, given the dynamic nature of Kubernetes development and its continuous evolution (7). As Kubernetes is widely adopted across diverse cloud environments (8), a structured fuzz testing approach becomes increasingly important to address the error localization issue. Solving this problem will significantly improve error detection accuracy within Kubernetes core APIs. A more precise fuzzing methodology can prevent potential crashes, errors, and bugs, ensuring the integrity and reliability of applications orchestrated by Kubernetes. Failure to address this problem may result in hardness finding errors and localizing within Kubernetes core APIs. This poses a severe risk of crashing containerized applications and service disruptions and the inability to localize the error to mitigate the problem in reasonable times. Solving this problem will enable a new class of Kubernetes systems characterized by enhanced efficiency and reliability.

1.1 Context

We analyze Kubernetes test coverage using existing literature and our own static analysis. Existing work shows that end-to-end (e2e) test feature coverage is only 41% across 256 features (9). The feature coverage distribution follows: Quota management and Container security yield a feature coverage of 100% and 90%, respectively. Application configuration and deployment exhibit a feature coverage of 83%, while Quality of Service(QoS) management aspects average 44% coverage. The study also notes that four other elements have coverage ranging from 33%

A detailed analysis in the study (9) also reveals significant differences among vendors, particularly in terms of customization interfaces and error occurrences. Despite offering extensive customization interfaces, Amazon Elastic Kubernetes Service (EKS) (10) is found to have the most errors in e2e tests, potentially impacting its usability for highly customized Kubernetes clusters. Conversely, Google Kubernetes Engine GKE (11) stands out for its out-of-the-box feature support, with solid feature lock-in tendencies. These observations underscore the importance of Kubernetes' reliability and integrity through improved error detection methods, such as structured fuzzing. By addressing the identified challenges, Kubernetes users and vendors can benefit from enhanced error detection accuracy and improved cluster migration experiences, thereby ensuring the resilience and dependability of containerized applications orchestrated by the platform.

To investigate Kubernetes errors further, I have retrieved data from ServerFault (12), a platform known for its technical discussions. A data mining approach was used to gather relevant data and preprocess it to analyze it meaningfully. Analyzing the distribution of these terms can provide insights into common issues, troubleshooting methods, and areas requiring further investigation within Kubernetes environments. This approach offers valuable insights into the frequency and contextual relevance of errors encountered within Kubernetes deployments by specific terms within relevant discussions. The most repeated term is kubelet with the error-related strings in the question and answer section. as shown in Table 1.1, and this also adds relevance to the study (9) as there were issues with Volume plugins and Node authorizations, which are part of kubelet. This can help vendors, customers, and providers enhance the orchestration process and quickly resolve conflicts.

Software development demands different testing methodologies to ensure the integrity of a system. Kubernetes is a complex and large distributed system. Testing methodologies don't cover everything that is required for a code base. One of the reasons or

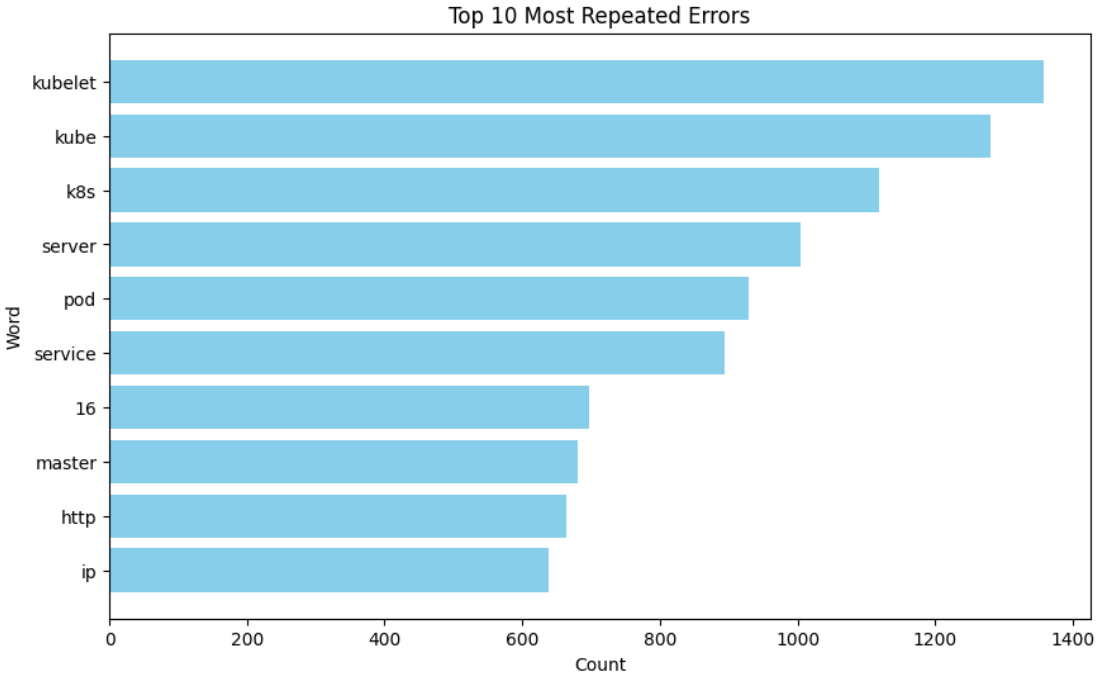

Figure 1.1: Frequency distribution of words after an error string (max. 100 characters).

bottlenecks is the lack of or insufficient data needed and input structures. This is evident in the case of Kubernetes, where the code coverage analysis shows significant gaps in critical components, but not limited to modules such as `kubelet_network_linux.go` and `kubelet_server_journal_linux.go` from Table 1.1, which have zero percent test coverage.

The error-handling paths remain unexplored because of insufficient test coverage or data generation. This could lead to reliability when the system is subjected to unexpected conditions, as mentioned with volume plugins in the case of this study (9). Given Kubernete's role in managing the containerized application, including the generated input structures and developing test cases that can handle errors and edge cases is essential.

1.2 Problem Statement

Given the nature of Kubernetes and its multitude of features and configurations. The challenge lies in enhancing error detection and localization within its core APIs. The project aims to increase the reliability of the Kubernetes by using the structure-aware fuzzing to generate erroneous objects. Mapping errors to specific configuration parameters is critical for troubleshooting and debugging in complex systems like Kubernetes. With

File Path	Coverage Percentage (%)
k8s.io/kubernetes/pkg/kubelet/active_deadline.go	77.3
k8s.io/kubernetes/pkg/kubelet/kubelet.go	40.8
k8s.io/kubernetes/pkg/kubelet/kubelet_getters.go	32.2
k8s.io/kubernetes/pkg/kubelet/kubelet_network.go	40.0
k8s.io/kubernetes/pkg/kubelet/kubelet_network_linux.go	0.0
k8s.io/kubernetes/pkg/kubelet/kubelet_node_status.go	19.8
k8s.io/kubernetes/pkg/kubelet/kubelet_node_status_others.go	0.0
k8s.io/kubernetes/pkg/kubelet/kubelet_pods.go	50.7
k8s.io/kubernetes/pkg/kubelet/kubelet_resources.go	0.0
k8s.io/kubernetes/pkg/kubelet/kubelet_server_journal.go	0.0
k8s.io/kubernetes/pkg/kubelet/kubelet_server_journal_linux.go	0.0
k8s.io/kubernetes/pkg/kubelet/kubelet_volumes.go	44.4
k8s.io/kubernetes/pkg/kubelet/pod_container_deletor.go	16.2
k8s.io/kubernetes/pkg/kubelet/pod_workers.go	1.1
k8s.io/kubernetes/pkg/kubelet/reason_cache.go	84.0
k8s.io/kubernetes/pkg/kubelet/runonce.go	0.0
k8s.io/kubernetes/pkg/kubelet/runtime.go	53.2
k8s.io/kubernetes/pkg/kubelet/volume_host.go	14.8

Table 1.1: Test coverage report of Kubelet.

new test cases that cover previously untested parts of the code, the likelihood of behavioral regression and unintended changes in functionality due to updates or modifications can be reduced. This makes the system more stable and ensures that new changes do not inadvertently disrupt existing functionalities. By reaching the uncovered code and creating errors, there will be a local store of multiple input configurations (erroneous configurations) that can be used for regression testing and a local database of coverage, which can be used to localize the errors with precision and reduce the time for developers or vendors to debug the issues in lesser time.

1.3 Research Questions

- **RQ1**: *What fuzzing design choices enable generating erroneous objects that target error modes in Kubernetes?*

  The design needs to generate error objects capable of creating the error states in Kubernetes. Error data generation needs to understand the Kubernetes configuration inputs, which are structures. The design choices made here need to pass structures

to Kubernetes APIs and must be evaluated to see if they will be accepted.

- **RQ2**: *How to implement the fuzzing framework using Go structures to explore Kubernetes errors?*

  The focus would be on developing a tool that leverages Go's capabilities to produce data that can systematically trigger a variety of failure scenarios within Kubernetes' system, integrating fuzzing libraries, which are predominantly unsupported or developed for a wide range of cases. These libraries are highly customizable, and complexities arise while combining them. We must ensure the integration does the erroneous objects we desire.

- **RQ3**: *How to evaluate the efficacy of the fuzzing approach for Kubernetes error object generation?*

  Does the initial hypothesis of generating the erroneous objects work with the design choices made? What's the code coverage and effectiveness compared to the standard unstructured fuzzing inputs?

1.4 Challenges

The critical challenge is generating these erroneous structured objects by getting dependencies to work together without disruption. One crucial challenge is implementing custom mutators that understand Kubernetes' specific structures and can develop meaningful mutations that are likely to uncover errors in component interaction and create a fuzzing framework tailored to Kubernetes that is capable of generating structured aware mutations. Designing a fuzzing framework using a custom mutator tailored to Kubernetes is complex. The framework must generate syntactically correct and semantically meaningful inputs to Kubernetes. It must also be seamlessly integrated with Kubernetes testing environments to automate the fuzzing process effectively. This framework should integrate with Kubernetes environments to create errors within Kubernetes components. Moreover, it is important to employ static analysis as well, for instance, using the AST to inspect the kubelet for issues and to find out if any fuzzing errors have affected the coverage that has been inspected with the AST. Coverage stands as a critical evaluation metric for this project.

1.5 Societal Relevance

As highlighted by Iosup et al., our society and economy's dependence on computer systems has been a substantial requirement for jobs and a large share of the GDP in the Netherlands (13). As stated in the manifesto, there are four grand societal challenges. Research questions are developed in relevance to building better distributed systems, which have a better understanding of reliable computing mechanisms, which are highly relevant in this era of massive containerized cloud computing. The findings of this work help to address two out of four challenges.

(**Challenge 1: Manageability**) The focus on optimizing error detection, localization, and dealing with extensive configurations and features directly relates to managing the growing complexity of large-scale distributed systems such as Kubernetes'. (**Challenge 2: Responsibility**) Ensuring Kubernetes' core APIs are more reliable by enhancing error detection aligns with responsible infrastructure development. The framework increases security, performance, and availability.

1.6 Plagiarism Declaration

I confirm that this thesis work is my own independent work, is not copied from any other source (Person, Internet, or Machine), and has not been submitted elsewhere for assessment.

1.7 Thesis Outline

The following is the organization of the remainder of this thesis. In Chapter 2, I will discuss the necessary background knowledge required to understand the fuzzing framework that will be built. In Chapter 3, I will explain the high-level and low-level design in detail. In Chapter 4, we discuss the evaluation of the results and their implications.

2

Background

Kubernetes has critical components that communicate with each other through the Kubernetes API server. Kubernetes manages its containers using the master-worker architecture. Containers run on each worker node and report to the master node using the Kubernetes API server. (14)

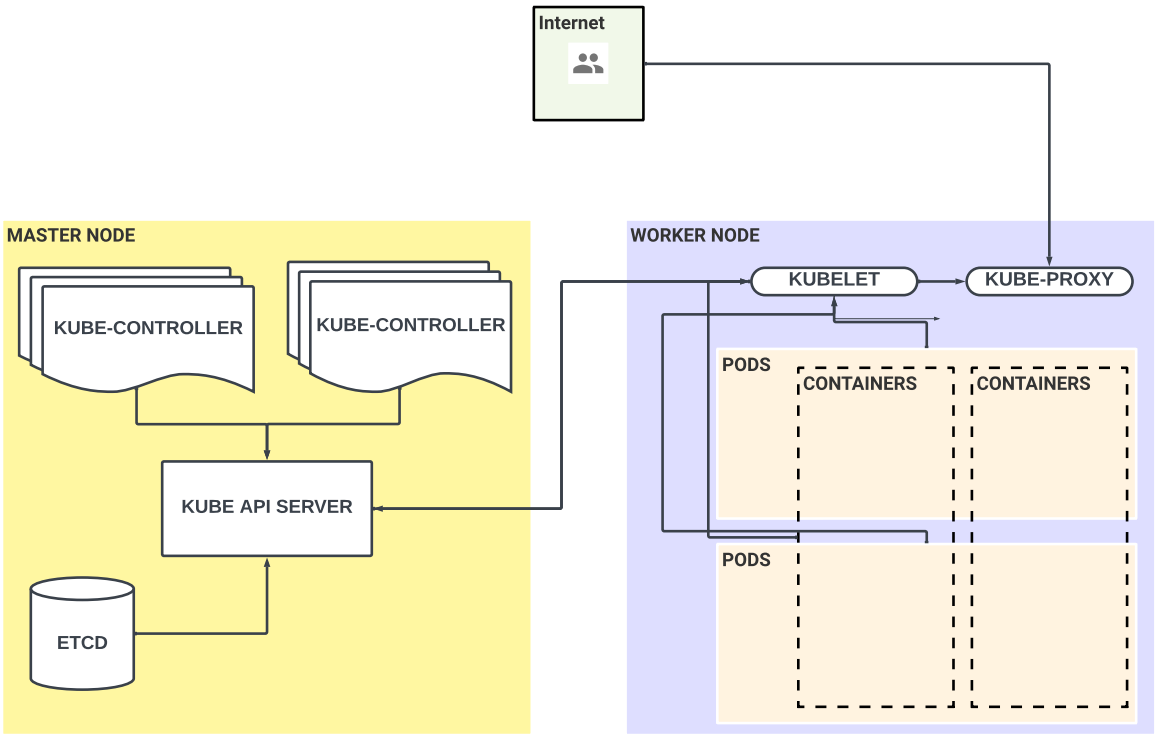

Figure 2.1: Kubernetes architecture.

2.1 Kubelet, Kubernetes-API-server, and ETCD

Kubelet is Kubernetes's (K8s) most essential and primary controller in Kubernetes (15). The Kubelet implements the primary Pod and Node APIs that drive container execution in Kubernetes. Without these APIs, Kubernetes (K8s) would be a basic CRUD-oriented REST application framework (15). By default, Kubernetes (K8s) runs isolated application containers decoupled from each other and from the hosts on which they execute. This allows for independently managing individual applications and the underlying cluster infrastructure (15). Kubernetes (K8s) runs application containers as its default mode of execution. Containers are isolated from each other and from the hosts they run on. Kubernetes (K8s) provides pods that host multiple containers and storage volumes, making packages and deploying applications easier. Pods help separate deployment and build-time concerns, facilitating migration from physical or virtual machines.

Kubernetes-API-server: Kubernetes-API-server is the core part of K8s because it is in charge of offering K8s API, which is used to control and manage K8s' components. Since the Kube-API-server needs to talk with each component, each component must connect to the Kube-API-server.

ETCD server ETCD is a distributed key-value database, which makes retrieval and management very flexible. It stores the cluster's confirmation data and the status of resources. When the Kube-API-server receives new commands, it will first update the ETCD data and inform other control components.

2.2 Protobuf in Kubernetes

The primary data format Kubernetes utilizes to exchange objects between these components is protobuf, which is then translated into go structs using a serializer hosted within the Kubernetes API machinery. Most traffic within Kubernetes is directed toward intra-cluster components, with up to 90% of all requests served by the APIs being for internal cluster components, such as nodes, controllers, and proxies. During initial versions of kubernetes, JSON serialization and deserialization were causing network size issues and significant memory and CPU usage due to the amount of garbage collection required during operation. The Kubernetes team has conducted experiments using protobuf as a medium to exchange objects (16) and has shown impressive results regarding CPU usage reduction during serialization and deserialization. Specifically, the Kubernetes developers have achieved a 10x reduction in CPU use, a 2x reduction in the size of data being transmitted

over the network, and a 6-9x reduction in the number of objects created on the heap during serialization.

Protocol Buffers (protobuf) is a language-neutral, platform-neutral, extensible mechanism for serializing the structured data (17). Protobuf provides a serialization format for packets of typed, structured data up to a few megabytes. Protocol buffer messages and services are described by .proto files, and the format of the protobuf is shown below with an example of a schema that's required for personal details in Listing 1.

```
syntax = "proto2";
package tutorial;

message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    PHONE_TYPE_UNSPECIFIED = 0;
    PHONE_TYPE_MOBILE = 1;
    PHONE_TYPE_HOME = 2;
    PHONE_TYPE_WORK = 3;
  }

  message PhoneNumber {
    optional string number = 1;
    optional PhoneType type = 2 [default = PHONE_TYPE_HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

Listing 1: Protobuf file for AddressBook and Person messages.

Proto compiler is invoked at build time on .proto files to generate code for different languages. Each generated class contains accessors for each field and methods to serialize and parse the whole structure to and from raw bytes. As in the above example, we can see that nested messages can be derived. For example, the PhoneNumber type is defined inside Person. Kubernetes uses this leverage to convert go structs and package the proto files accordingly. Each component has proto files generated using the predefined structures

in Go files. To generate these proto files, Kubernetes uses a third-party library called gogo-protobuf (16). In this project, we use the same protobuf files, convert them to C++ headers, and include files to integrate them with libprotobuf-mutator. Listing 2 is the example of converting the protobuf file into a C++ format with protobuf C compiler.

```
// name
 inline bool has_name() const;
 inline void clear_name();
 inline const ::std::string& name() const;
 inline void set_name(const ::std::string& value);
 inline void set_name(const char* value);
 inline ::std::string* mutable_name();

 // id
 inline bool has_id() const;
 inline void clear_id();
 inline int32_t id() const;
 inline void set_id(int32_t value);

 // email
 inline bool has_email() const;
 inline void clear_email();
 inline const ::std::string& email() const;
 inline void set_email(const ::std::string& value);
 inline void set_email(const char* value);
 inline ::std::string* mutable_email();

 // phones
 inline int phones_size() const;
 inline void clear_phones();
 inline const
 ::google::protobuf::RepeatedPtrField< ::tutorial::Person_PhoneNumber >&
 phones() const;
 inline
 ::google::protobuf::RepeatedPtrField< ::tutorial::Person_PhoneNumber >*
 mutable_phones();
 inline const ::tutorial::Person_PhoneNumber& phones(int index) const;
 inline ::tutorial::Person_PhoneNumber* mutable_phones(int index);
 inline ::tutorial::Person_PhoneNumber* add_phones();
```

Listing 2: C++ methods of the Person message from compiled protobuf file.

2.3 Structure Aware Fuzzing

The term Fuzzing has become a synonym for penetration testing. Bart Miller first used the term fuzzing in the paper: *An empirical study of the reliability of UNIX utilities* (18). Bart Miller could crash or hang many utility programs on each system (from 24%-33%) like vi, cshell, and emacs. There were also new fuzzing cases on GUI apps done by the same group (19). The latest recent research (20) allowed smart fuzzers to use differential testing as the test oracle instead of random bits used in fuzzing, allowing them to find 80 gcc (21) bugs and 200 clang (22) bugs. Microsoft research group took a step further to introduce fuzzing by using symbolic execution and dynamic analysis (23). In general, the fuzzing taxonomy is divided into three parts:

White box fuzzing: It is a testing technique that uses internal program structure and code to execute the program with random inputs. It combines static and dynamic analysis to ensure maximum code coverage and test all possible execution paths. This method typically includes a "symbolic execution" component to explore various branches and gather them into constraints. Let's assume the code Listing 3.

```
int foo(int x) {
    int y = x + 2;
    if(y == 5)
        abort();
}
```

Listing 3: C function example with potential abort condition.

In this case, black box fuzzing has a low probability of getting the check conditional statement above, depending on the input x. However, in the case of white box fuzzing: SAGE (23), the input variable x takes the branch of the conditional statement and generates the path constraint x+2 != 5. Once this constraint is negated and solved, it yields x = 3, providing a new input that causes the program to follow the then branch of the conditional statement. This will generate a crash because of abort().

Grey box fuzzing: This fuzzing technique uses coverage feedback to learn how to reach deeper into the program. One example is AFL(American fuzzy loop) (24), which leverages some program analysis but not heavyweight analysis or constraint solving. Instead, it uses lightweight program instrumentation to get information about the input coverage. If the input increases coverage, it is added to the seed corpus for further fuzzing. Below are two categories in grey box fuzzing. Our current project falls into the grey box fuzzing as

we know the structure of input corpus, giving feedback to the fuzzer and allowing further mutations.

- **Generative-based fuzzers**: These fuzzers are complex and synthesize test cases from scratch according to a predefined grammar. One of the examples is CSmith (20)

- **Mutation-based fuzzers**: These kinds of fuzzers modify(non-)random test cases and treat inputs as a bag of bits. Examples of this are AFL(American fuzzy loop) (24) and libFuzzer (25). Mutation-based fuzzers can be simple or intelligent. A simple fuzzer randomly generates the input without knowing the data structure. These simple fuzzers only check the parsing of highly data-structured inputs, leading to an invalid rejection in initial parsing (26).

Black box fuzzing: In this type of fuzzing, there is no coverage-feedback. The input type is randomly mutated, as discussed before.

2.3.1 Structure Aware Mutations

Generation-based fuzzers create inputs based on a pre-defined grammar for a single input type. In contrast, mutation-based fuzzers like libFuzzer and AFL are not limited to one input type and require no grammar definitions. However, lacking an input grammar can lead to inefficient fuzzing for complicated input types like protobuf (4). Coverage Guided Fuzzers (CGF) work by changing inputs at the level of their bit and byte representations. CGF considers mutated inputs attractive when they cause the program to explore new sections or paths of the code. While this method is effective for small and unorganized inputs (27), it may not work well for highly structured inputs that need to follow specific rules or formats. Essentially, uncontrolled mutations can cause the fuzzer to waste time generating inputs that the initial stages of the program reject, resulting in little to no improvement in code coverage (28). These mutations generate inputs that spend more time rejecting inputs in the initial parsing stages, resulting in very low code coverage. Structure-aware mutations solve the exact problem by generating highly structured inputs. This minimizes the time and allows adherence to specific rules and formats set by input requirements in a program. The importance of structured aware mutations is being actively explored. Google Chromium team used libprotobuf-mutator(29) to find more than 50 bugs or vulnerabilities in libraries like SQLite (30) and Linux kernels (31)

Integrating libprotobuf-mutator (LPM) with libFuzzer can address some of the inefficiencies faced by traditional mutation-based fuzzers, as mentioned above when dealing with complex input types such as protobufs.

- **Structured Input Generation**: This Structure-aware input generation ensures that the inputs are syntactically correct and semantically meaningful within the context of the target application, thereby bypassing the initial rejection faced by improperly structured inputs. Libraries such as libprotobuf-mutator (29) and AFLsmart (32) are designed to create and mutate structures while respecting their schema.

- **Coverage-Guided Mutation with Structure Awareness**: By integrating LPM with libFuzzer, fuzzers benefit from both coverage-guided mutation and Structure-aware input generation. libFuzzer adapts its mutations based on the code paths activated by the inputs. When these inputs are structured correctly (LPM), libFuzzer can more effectively direct its efforts toward unexplored and potentially vulnerable code areas.

- **Feedback Loop Enhancement**: The feedback from libFuzzer about which code paths have been triggered by the inputs can be used to refine the mutations applied by LPM further. This continuous loop of generation, mutation, and feedback allows for a more targeted approach to discovering vulnerabilities, particularly in complex software systems that use structured formats like protobufs.

- **Reduced Wastage**: With LPM, the chances of generating irrelevant or syntactically incorrect inputs are significantly reduced, decreasing the computational waste commonly seen in fuzzers when dealing with complicated input types. The inputs are more likely to reach deeper into the application logic before being rejected, if at all, enhancing the overall efficiency of the fuzzing process.

libFuzzer mainly has two functions leveraged by the libprotobuf-mutator to generate custom mutation functions as shown in the Listing 4 (33).

This is the following execution flow of the Custom mutators in LLVM using libfuzzers 2.2:

- **Initialization**: When libFuzzer initializes, it starts with an empty corpus or the set of initial inputs provided.

```
// Mutates raw data in [Data, Data+Size) inplace.
// Returns the new size, which is not greater than MaxSize.
// Given the same Seed produces the same mutation.
size_t LLVMFuzzerCustomMutator(uint8_t *Data, size_t Size,
size_t MaxSize, unsigned int Seed);
// libFuzzer-provided function to be used inside LLVMFuzzerCustomMutator.
// Mutates raw data in [Data, Data+Size) inplace.
// Returns the new size, which is not greater than MaxSize.
size_t LLVMFuzzerMutate(uint8_t *Data, size_t Size, size_t MaxSize);
```

Listing 4: LLVM custom mutator in libFuzzer.

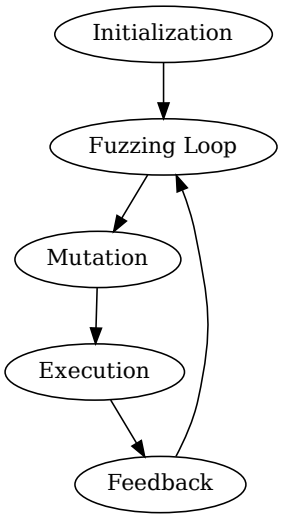

Figure 2.2: Fuzzing flow with feedback loop.

- **Fuzzing Loop**: LibFuzzer then enters a loop where it repeatedly calls the fuzz target function (LLVMFuzzerTestOneInput) with mutated inputs generated by the mutator (LLVMFuzzerCustomMutator).

- **Mutation**: Inside the LLVMFuzzerCustomMutator function, the mutator mutates the input data and returns it to libFuzzer.

- **Execution**: LibFuzzer then passes this mutated input to LLVMFuzzerTestOneInput for execution. If the input triggers a crash or some other error within LLVMFuzzerTestOneInput, libFuzzer detects it.

- **Feedback**: Upon detecting a crash or error, libFuzzer reports it, and the fuzzing process continues.

This allows the libprotobuf-mutator to mutate the protobuffer's tree structures and individual fields. The Structure-aware fuzzer uses this flow to generate erroneous configurations that create errors. libprotobuf-mutator manipulates protobuf data structures to produce new, potentially exciting inputs fed into the software being tested. This manipulation is based on the protobuf's schema, allowing the mutator to create logically consistent but unexpected message configurations.

3

Design of Kubernetes Erroneous Object Generation

Main Contributions

- **Main Contribution 3.1 (MC3.1)**: We analyze the requirements to generate error objects that adhere to Kubernetes input structures and trigger meaningful errors.
- **Main Contribution 3.2 (MC3.2)**: We design Structure-aware fuzzers to inject data into Kubernetes APIs and handle nested structures such as Pods, Nodes, and Containers.
- **Main Contribution 3.3 (MC3.3)**: We implement coverage-directed fuzzing to enhance input exploration by increasing the code paths executed in the kubelet using a feedback-driven system and a global input corpus.

This chapter presents a design for erroneous object generation using struct-aware fuzzing. The chapter is organized as follows: Section 3.1 analyzes the requirements for the design choices to generate the error objects. Section 3.2 provides detailed requirements for the design of a struct-aware fuzzing system and passes them to Kubernetes components. Section 3.3 presents a high-level design of the system. Section 3.4 presents a low-level design of the system, a more thorough system view, and specifies each step required to obtain the error objects.

3.1 Requirement Analysis

This project's primary goal is to generate the erroneous configuration for Kubernetes, mainly the kubelet component. Kubelet is a vital part of the whole Kubernetes ecosystem.

The primary node agent runs on each node in the cluster. Its primary responsibility is maintaining the set of pods, which are essential for encapsulating containerized applications.

Kubelet takes instructions from the Kubernetes API server to manage the lifecycle of the Pods, like starting, stopping, and maintaining application containers based on the system and Kubernetes instructions (34). As such, it directly impacts the deployment speed and stability of applications running in the Kubernetes environment. Kubelet checks the health of the pods and reports back to the control plane, as shown in Figure 3.1. This allows the control plane to maintain the cluster's desired state by restarting failed containers. (35). The ability of the kubelet to accurately monitor and report container health is important for auto-healing functionalities.

Kubelet also plays an important role in resource allocation. It ensures that each container has enough resources to run as specified but does not exceed the allocated resources. This will prevent resource issues and maintain the quality of service across all nodes. Kubelet interacts with the container runtime through the Container Runtime Interface (CRI) to manage the lifecycle of containers within pods. This includes pulling images from container registries, starting and stopping containers, and collecting container logs. This interface allows Kubernetes to support multiple container runtimes, promoting flexibility and choice in deployment architectures (15)

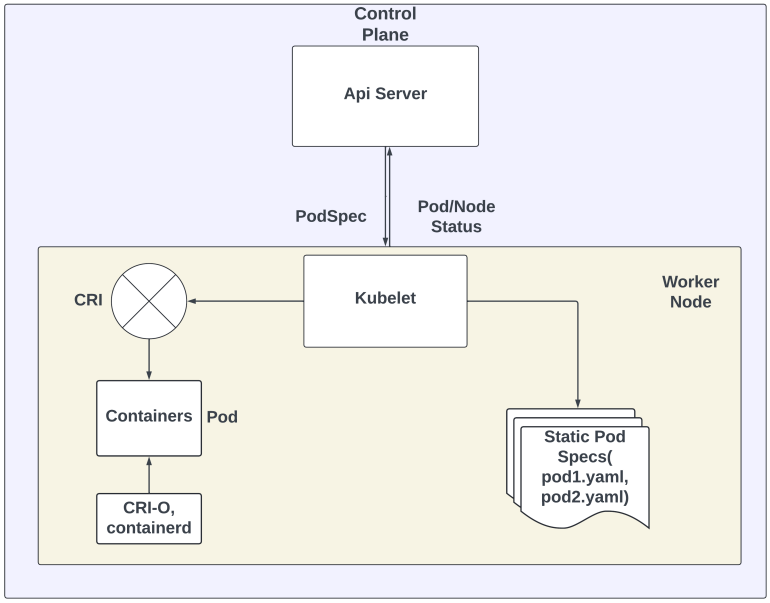

Figure 3.1: Control-plane to kubelet flow.

Since kubelet uses protobuf for a significant portion of its configuration and communication (especially in settings like network configurations and API interactions), protobufs can generate structured inputs that respect the required format but vary significantly in content. This is required for triggering edge cases and allows us to cover the code that regular unstructured inputs can't reach. protobufs allows for the customization of mutations based on the specific use cases of Kubernetes. By understanding the fields and values that are more likely to lead to states of interest within kubelet, fuzzers can be designed to explore particular areas of the code more broadly or stress test specific functionalities.

3.1.1 Requirement Analysis 1: Structured Input Generation

To invoke a series of erroneous configurations for a particular kubelet component, the structure of injection objects must be in the nested form as multi-level sub-objects. Let's take a simple Pod structure with ObjectMeta and Status.

```
pods := []*v1.Pod{
        {ObjectMeta: metav1.ObjectMeta{
                                UID:         "1",
                                Name:        "completed-pod1",
                                Namespace:   "ns",
                                Annotations: make(map[string]string),
        },
        Status: v1.PodStatus{
                Phase: v1.PodFailed,
                ContainerStatuses: []v1.ContainerStatus{
        {
                State: v1.ContainerState{
                Terminated: &v1.ContainerStateTerminated{},
        },}}}}
```

Listing 5: Go code defining a list of Pods with status information.

These rules need to be considered for erroneous configuration generation for the structure in Listing 5.

- **Invalid Pod Phases**: Set the Phase to a non-existent or deprecated value to test how kubelet handles unknown states.
- **Container State Corruption**: Alter the ContainerState to simultaneously include conflicting states, such as Running and Terminated.

If any considerations are not taken, it will not cover the path required by a particular function or component in kubelet. Such considerations are necessary before we generate structure inputs to create error objects. The design choice should be capable of understanding such complex input structures.

3.1.2 Requirement Analysis 2: Key Considerations for Generating Erroneous Objects

- **Understanding the Structure**: Kubernetes objects are complex and nested. Kubernetes objects can be described as quite complicated and hierarchical. To produce error objects for the kubelet, the fuzzer needs to understand the configurations of the kubelet. This means understanding protobuf definitions and the relationships that are employed by kubelet. Also, a structured input generator should understand the hierarchy and relationships.
- **Field Interdependencies**: There are many configuration fields whose usage is conditional on other fields. For instance, a particular field may be valid only when another is set to a specific value. It is required to recognize these interdependencies to create a valid configuration that contains errors.
- **Boundary Values**: Using input values that are either the maximum or the minimum value for a numerical field or string's maximum/minimum lengths can help bring out off-by-one errors or buffer overflows or help find new coverage paths. Such inputs should be a part of the erroneous object generation strategy.
- **Unexpected Combinations**: Although each field may contain some valid values, some combinations of these fields can cause problems. It is important to generate such configurations that check all these combinations, as this will help detect bugs.

3.1.3 Requirement Analysis 3: Coverage Directed Fuzzing

Coverage-directed fuzzing is essential to explore the vast input space of kubelet configurations effectively. This approach maximizes the code paths exercised during the error object generation.

- **Code Coverage** looks into the amount of the kubelet code executed during fuzzing. Various methods for evaluating coverage exist, such as line, branch, path coverage, and others. Within kubelet, it is required to maximize branch and path coverage to avoid testing unnecessary execution paths.

- **Feedback Mechanism** is established where the fuzzer receives information about which a given input executed parts of the code. This is done with the help of a global input corpus. This feedback is used to guide the generation of new inputs, aiming to explore previously unexecuted paths.

- **Guided Exploration** Employ a coverage-guided exploration strategy where the fuzzer prioritizes inputs that trigger new coverage. This requires dynamically updating the global input corpus and using default configurations on pod and node structs.

- **Long-Running Fuzzer** Since kubelet configurations can involve complex interactions and states, support long-running fuzzers to capture state-dependent issues. This ensures that the kubelet runs in a stable test environment and can recover from any erroneous states introduced by the fuzzing process.

3.2 Requirements

These requirements specify the functional and non-functional requirements and technical constraints. They clearly understand what the system should do and its expected behavior.

3.2.1 Functional Requirements

Functional requirements specify what the system should do. For this project, these include, as shown in Figure 3.2:

- **FR1 - Structured Data Generation**: The system must generate structured input data that adheres to the Kubernetes API specifications. The system should support the creation of erroneous objects to target known function calls in kubelet

- **FR2 - Input Injection and Error Detection**: The tool must be able to inject generated erroneous objects into Kubernetes mock functions, which call the kubelet APIs. The system should detect and log the errors triggered by these injections.

- **FR3 - Data analysis and feedback using corpus**: Provide detailed reports on the errors detected, including traces, status, and the conditions that triggered them. Maintain a database of generated configurations and associated coverage metrics.

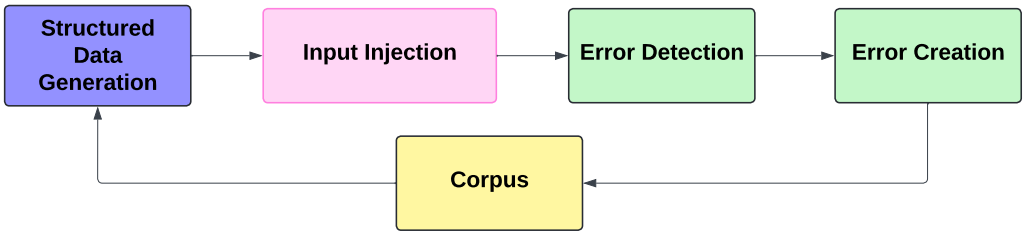

Figure 3.2: Design functional requirments.

3.2.2 Non-Functional Requirements

Non-functional requirements describe how the system performs its functions. These include performance, usability, reliability, etc.

- **NFR1 - Performance**: The tool must efficiently generate and process error configurations to avoid excessive computational overhead.
- **NFR2 - Scalability**: The system should handle a variety of Kubernetes components and complex configurations. It should also ensure scalability to generate continuous error configurations as Kubernetes evolves.
- **NFR3 - Reliability**: Ensuring the accuracy and consistency of generated error objects. The system should handle and recover gracefully from failures during execution.
- **NFR-4 - Usability**: Providing a user-friendly configuring and running fuzzer interface. Include documentation and examples for users to understand and utilize the tool effectively.
- **NFR-5 - Distributed Corpus**: The system should be able to share the corpus between different instances to maintain coverage uniformity across the fuzzing process.

3.2.3 Technical Constraints

These are the limitations or constraints imposed on the project.

- **Compatibility**: Ensure compatibility with the latest stable versions of Kubernetes and Go. The tool should support major operating systems used in Kubernetes deployments (e.g., Linux, Windows).

- **Resource Usage**: Optimize the tool to run efficiently within the available hardware and network constraints.

3.3 High-Level Design

This section presents the design of error object generation, which is applied specifically to kubelet. The Figure 3.3 below gives a high-level picture of how the data is passed until it reaches the kubelet functions. The requirements from the previous section should be reflected in the design system.

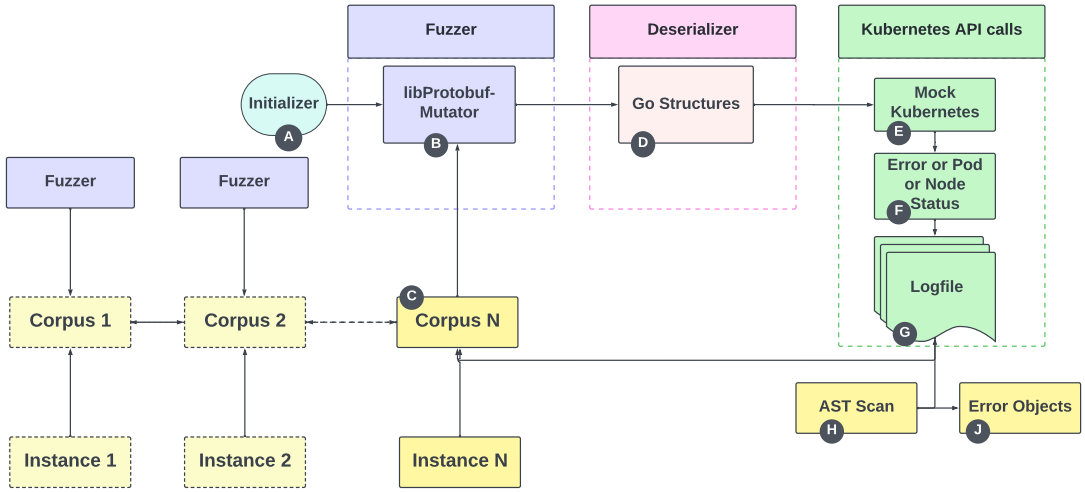

Figure 3.3: High-Level design of erroneous object generation flow.

The design system consists of three steps. The first step is the **Fuzzer**; the libprotobuf mutator(B) is initiated in this step (A) using different initialize calls provided, and as for the input, we need to pass Kubernetes structs as protobuf objects to the fuzzer input. Hence, it generates the structured data, which satisfies the { FR1 }. To fulfill the { FR2 }, All the individual mock kubelets (E) must be rewrapped with the unmarshaler. The protobuf data needs to be converted into a format that kubelet APIs accept. Before passing the go structs to kubelet in the second step, they must be converted using **Unmarshaler**. This function takes the protobuf data and changes them to the structures accepted by the kubelet APIs. The Go structs (D) are passed on to all mock **Kubelet APIs** in the third step. In case of errors or crashes (F), they are sent to the log file (G) for further evaluation later, then further sent back to the input corpus. If Errors or pod or node status (F) are reported during the fuzzing, the field inputs are sent back to the

input corpus. The input corpus is also used as a feedback system to the fuzzer (C), as we use the fuzzer for the individual field mutation, which satisfies the { FR3 }. The feedback system is distributed, and the input corpus should be synced temporally.

3.4 Components of the Erroneous Object Generation Flow

The error object generation flow has three main components. Examining each component is essential, and this section explains its specific functionalities. These components depend on each other as the generation flow is a continuous loop, and the dependencies are interlinked with Kubernetes' tight specifications and requirements.

3.4.1 Design of the Structure-Aware Fuzzer

This Structure-Aware Fuzzer follows the { FR1 }, which is structure data generation (3.1). The project uses the libprotobuf-mutator and libfuzzer (A) as discussed. But these are the primary rationale behind choosing the fuzzer design, as shown in Figure 3.3. Kubernetes uses advanced data structures that are encoded with the aid of protocols such as Protocol Buffers (protobuf). Most fuzzers with simple text formats like string, bytes, integers, and bits cannot create even slightly valid and relevant test cases for advanced data structures such as Go Structures in kubelet.

- **Complexity of Kubernetes Structs** :
  - *Hierarchical and Nested Structures*: Kubernetes defines various resources such as Pods, Services, Deployments, StatefulSets, ConfigMaps, Secrets, etc. Deployments, StatefulSets, ConfigMaps, Secrets, etc. Each resource type has its schema, which has a schema with child elements.
  - *API Versions and Compatibility*: Kubernetes has several API versions where the fuzzer must accommodate different resources, such as v1 and v1beta1. This, however, ensures that older web application versions do not stop working yet increases challenges such as knowing which version of API to work with and why.
  - *Deprecations and Migrations*: Users will eventually need to adopt new versions and structures as some API versions and fields go out of use and are replaced with new ones.

  - *Declarative Configuration*: Kubernetes works using a declarative model where all the issues related to the state are dealt with during cluster-level declaration. The actual state is persistent at the Kubernetes control plane, which constantly works to resolve issues between the two state models.
  - *Persistent Volumes (PVs) and Persistent Volume Claims (PVCs)*: This includes creating PVs, PVCs, and Storage Classes, as well as knowing how binding works.

- **Ability to Fuzz Complex Structs**:

  Let's consider a simple pod definition in YAML. By converting this to protobuf, the nestfields are represented in the tree from, shown in Figure 3.4. A normal fuzzer randomly mutates the string or inputs, which will not adhere to the expected structure or sometimes doesn't provide exciting results. This will lead to many invalid test cases that do not provide helpful feedback for the fuzz. This will be explored more in evaluation with examples.

  A protobuf-mutator generates inputs that conform to the schema defined by the protobuf specifications. This ensures the generated data is syntactically and semantically valid, increasing the likelihood of discovering errors that only manifest with valid inputs. Protobuf-mutator employs intelligent mutation strategies that are aware of the data structure (4). This means they can perform more meaningful mutations, such as changing values within a range, swapping nested elements, or adjusting the length of repeated fields, which can lead to the discovery of more errors. Using a protobuf-mutator for fuzzing stateful APIs might be slower than String-based input fuzzing or more complicated than fuzzing action traces encoded as a sequence of bytes { NFR2 }. However, this approach is more flexible and maintainable since the protobuf type is easier to understand and extend than a custom byte encoding.

3.4.2 Design of the Unmarshaler: Converting Protobufs to Go Structs supporting Kubernetes

This section delves into how serialized data, particularly in Protocol Buffers format, is converted into Go structs usable within Kubernetes systems. The unmarshaler ensures data integrity and operational correctness in dynamic, distributed environments like Kubernetes and needs to satisfy the { FR2 }.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  labels:
    app: example
spec:
  containers:
  - name: example-container
    image: nginx:latest
    ports:
    - containerPort: 80
    env:
    - name: EXAMPLE_ENV_VAR
      value: "example-value"
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
    volumeMounts:
    - name: example-volume
      mountPath: /example/path
  volumes:
  - name: example-volume
    configMap:
      name: example-config
```

Listing 6: Kubernetes Pod definition for example-container.

3.4.2.1 Struct Creation and Encoding

The initial stage in the lifecycle of Kubernetes objects within the system involves creating and serializing Go structs. The focus is on struct integrity and alignment with Kubernetes API specifications.

- **Structure Initialization** Structures coming from the fuzzer should reflect the schema Kubernetes objects expect (D). This includes fields representing metadata, specs, and statuses essential for Kubernetes operations. The incoming Protocol Buffers data is in hex format and needs to be de-serialized into Go structures. In any fuzzing

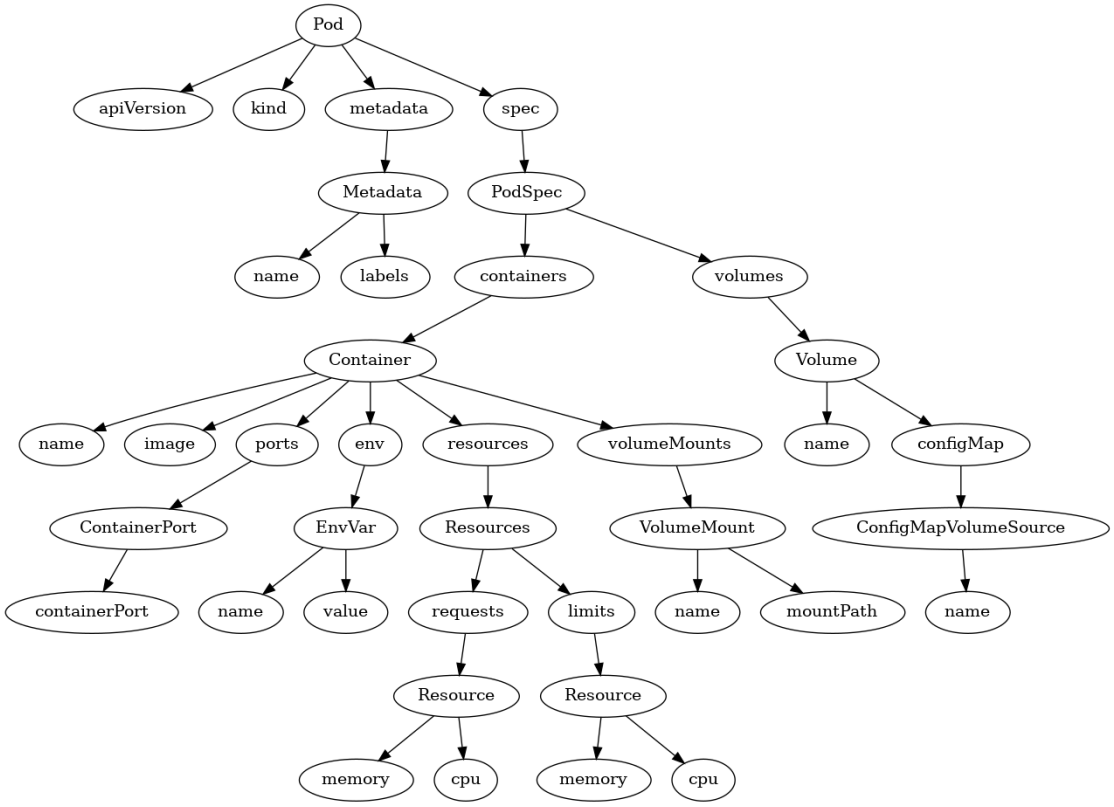

Figure 3.4: Protobuf tree structure of converted Pod yaml.

input, the incoming fuzzing data should be de-serialized or converted to the Go structures, which Kubernetes can take as input, as shown in Figure 3.5.

3.4.2.2 Error Handling and Logging

Effective error handling and logging are required in the fuzzer, where the complexity and scale can lead to numerous failure points. The unmarshaler is designed to handle and log errors. Which are logged for further use in coverage and error analysis (G).

- **Error Detection and Recovery** Errors during unmarshaling should be detected promptly. The system should be designed to recover gracefully from such errors, ensuring continuous service availability and reliability { NRF3 } .

- **Logging Mechanisms** All errors should be logged with detailed contextual information, including timestamps, the function names where errors occurred, and the nature of the error.

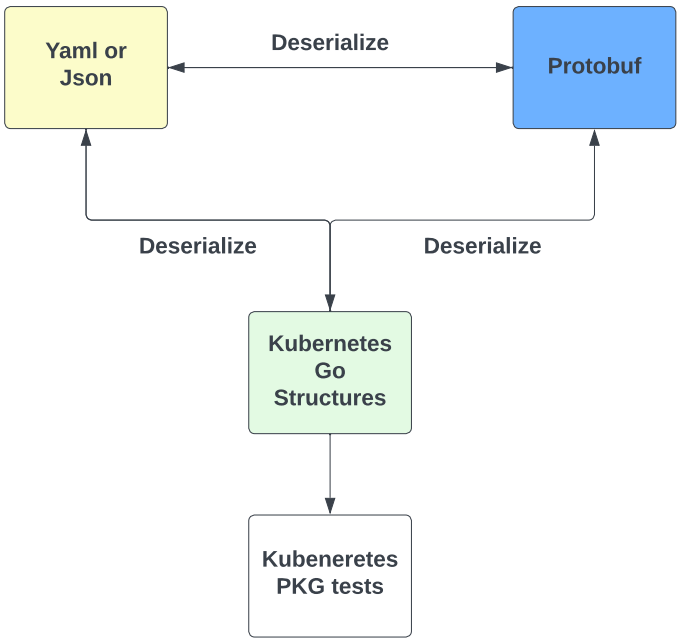

Figure 3.5: Protobuf to Structs, Structs to Protobuf, Yaml to Protobuf, Deserializer flow.

3.4.3 Design of the Kubelet API Calls

This section explores the final step in the flow where Kubernetes structures, now correctly initialized and transformed into Go structs (D), interact directly with the kubelet's APIs (MC3.2). This interaction is important as it directly impacts the kubelet's ability to manage node resources and handle pods effectively.

The Main rationale for choosing these design choices is to evaluate if the error state creation can be reached faster using specific input handling techniques that satisfy the mock kubelet API's requirements. In Structure-Aware fuzzing, reaching specific paths might take longer, which means more computing time. The design hypothesis here is to verify and evaluate if that is true.

3.4.3.1 Integration with Kubelet Functions

The interaction between the Go structs (D) and kubelet APIs (E) involves invoking kubelet functions using the generated and unmarshaled Go structs. These calls simulate real-world operational conditions to validate the kubelet's responses to varied and potentially erroneous inputs.

- **Mock Integration**: Utilizing mock kubelet APIs allows for controlled testing without needing an active Kubernetes cluster. This integration is important for testing the kubelet's behavior under simulated conditions that will not be feasible in a live environment.

- **API Function Calls**: Specific kubelet functions are targeted based on the testing requirements. These include managing pod lifecycle events, handling node resource allocations, and responding to cluster state changes.

3.4.3.2 Handling of Erroneous Inputs

The design must ensure that the kubelet can gracefully handle erroneous inputs. This involves validating the kubelet's error-handling capabilities to maintain stability and reliability.

- **Error Simulation**: By passing erroneous data structures to the kubelet APIs, the system can validate the error handling protocols of the kubelet, ensuring it responds appropriately without leading to system crashes or unexpected behavior.

- **Response Validation**: Responses from kubelet APIs are captured and analyzed to ensure they meet the expected outcomes, whether handling errors correctly or managing resources under fault conditions.

3.4.3.3 Feedback Loop for Continuous Improvement

The integration with kubelet API calls is not just about testing but also about improving. The corpus from these fuzzers feeds back into the fuzzing and mutation process, enhancing the error cases' effectiveness (MC3.3).

- **Coverage and Error Analysis**: Information from the input executions, such as coverage data and error logs, is used to refine the fuzzing process, aiming to cover more code paths and discover new potential errors.

- **Distributed Feedback Corupus**: The fuzzer runs in a distributed pattern, and the input corpus might differ from each instance. To maintain a consistent corpus across the instances, we sync the input corpus between the instances { NFR5 }.

3.5 Summary

In this chapter, we address RQ1 by proposing a design for a Structure-aware fuzzer system for generating erroneous Kubernetes objects, by targeting the Kubelet component. We identify the key considerations for generating error objects (MC3.1) and propose a fuzzer design to create structured inputs for Kubernetes APIs that satisfy the { FR1 }. A structured input generator is built using libprotobuf-mutator to handle Kubernetes' nested and complex objects (MC3.2). The coverage-directed fuzzing system is used in code exploration using feedback loops and a global input corpus (MC3.3) which satisfies the { FR3 }. libprotobuf-mutator is configurable for custom resources and uses the llvm custom mutator, which efficiently fuzzes the inputs. Information during a panic or error is stored and exits safely. There is validation for the inputs to avoid the nil pointer or panics by which only valid structure inputs are passed to the Kubernetes APIs, satisfying the { FR2 }.

4

Implementation of Structure Aware Fuzzer with Kubernetes

Main Contributions

- **Main Contribution 4.1 (MC4.1)**: We integrate effective fuzzing of structured data (protobuf messages) for Kubernetes components.
- **Main Contribution 4.2 (MC4.2)**: We handle incoming mutated protobuf data, encapsulating and prefixing it with encoding headers and transforming it into Go Structs to simulate real Kubernetes scenarios.
- **Main Contribution 4.3 (MC4.3)**: We integrate error handling and logging to capture execution errors (e.g., panics) with metadata.

This section outlines the development of a struct-aware fuzzer that integrates Google's libFuzzer with the libprotobuf-mutator to efficiently test programs that process structured data, specifically protobuf messages (MC4.1). The fuzzer aims to uncover errors or defects by subjecting the target software to varied and valid erroneous configurations. Figure 4.1 represents the overall flow of the fuzzer implementation.

4.1 Implementation of Structure-Aware Fuzzer

Initialization and Configuration The implementation commences with including the necessary libraries:

- Fuzzer: libprotobuf-mutator mutates protobuf messages, and libfuzzer_macro provides the macros required to define the fuzzing behavior.
- Protobuf definitions: Protobuf headers **.pb.h* contains the generated class definitions

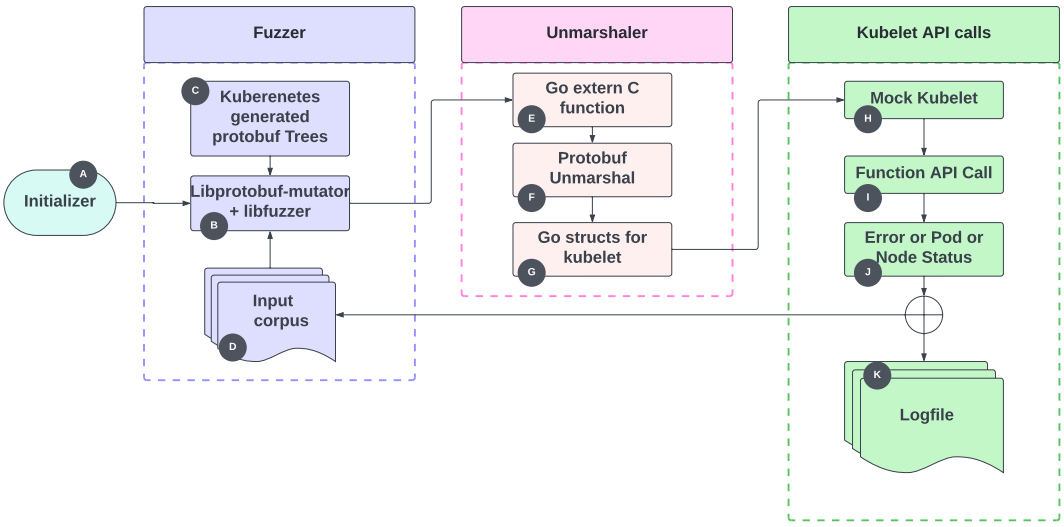

Figure 4.1: Implementation of erroneous object generation flow.

from the protobuf specifications. which includes structured data to be fuzzed. These protobuf files are compiled from already existing Kubernetes components. This fulfills the need to define the structures for the protobuf mutator to understand what to fuzz.

- There are two header files: The protobuf header files and Go Exported C headers. These contain definitions and implementations explicitly used for the fuzzing process, like hooks for data manipulation and debugging tools.

DEFINE_PROTO_FUZZER : The fuzzer's entry point (A) is defined using a macro from libprotobuf-mutator (B), which integrates seamlessly with libFuzzer. This macro from libprotobuf-mutator defines the main function that libFuzzer will invoke. This function:

- Registers a custom post-processor only once (controlled by hasRegister), which could modify the protobuf data after each mutation, depending on certain conditions or targeted fuzzing strategies, as shown in Figure 4.2.

Input Corpus : Corpus provides a set of well-formed inputs that mutator can use as a base to generate new input cases (D). The input cases are more meaningful because they are based on valide inputs rather than random data. Having realistic inputs, the mutator

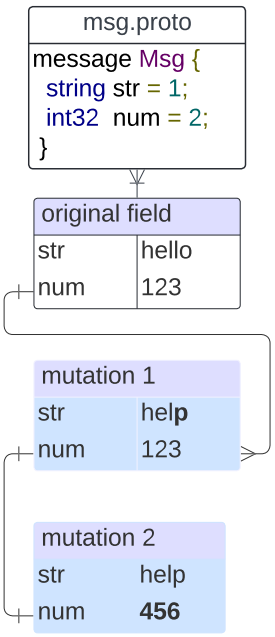

Figure 4.2: Protobuf struct mutation.

can modify specific part of structured inputs. libprotobuf-mutator can generate mutations that explore the different paths in code more effectively. A good input corpus will generate fewer invalid or malformed inputs that Kubernetes components reject immediately.

4.2 Implementation of Unmarshaler

In the context of our struct-aware fuzzer, the decoding and creation of structured data are vital for generating error objects that are correct and meaningful, thus reflecting realistic scenarios encountered in Kubernetes environments.

Kubelet Binaries : The extern "C" (E) keyword in C/C++ is typically used to indicate the function that should have C linkage. If we want to use libraries such as libProtobuf-mutator on the Kubelet, there should be a link between the go and C/C++. We use Go's c-shared build mode to compile the Go code as an archive file (Groups of objects or static libraries that are also input into the linker). The exported Go functions are callable from C++ because Go creates a compatible interface with C through the export directive. C++ uses the archive object file and the generated header to call Go functions, linking the Go-generated library at compile time. This would allow C++ to call

Go functions to perform specific tasks while still utilizing the core fuzzing capabilities of libprotobuf-mutator.

4.2.1 Struct Creation and Decoding

As the fuzzer generates the data, it is presented in hex format, which is the protobuf format. However, this cannot be directly translated into Go structs, where Kubernetes components would understand the data (MC4.2). A few steps are required to convert the protobuf hex data into Go structs. Here are the steps that need to be followed:

Encoding (Marshaling) : Initially, Incoming protobuf format `Pod` hex data is marshaled into its wire format by converting the `Pod` object into a byte slice, employing protobuf for efficient serialization. This byte slice represents the object's serialized form.

Preparing Mutated Data : The raw data from the fuzzer(encoded byte slice) is encapsulated in a `runtime.Unknown` object, simulating the reception and handling of unknown objects within Kubernetes. In this context, the raw byte data is encapsulated within a `runtime.Unknown` object, accompanied by appropriate type metadata. The encapsulated object is then re-marshaled into a byte slice (`wire`), simulating the receiving of unknown or malformed objects. Header bytes must be used to entirely unmarshal the protobuf data (Ⓕ) according to the Kubernetes requirements. The incoming buffer has no headers and raw mutated protobuf data from libprotobuf-mutator. The header bytes are added to the incoming wire as below:

```
wire = append([]byte{0x6b, 0x38, 0x73, 0x00}, wire...)
```

protoEncodingPrefix is a magic number for an encoded protobuf message on this serializer. All proto messages serialized by this schema will be preceded by the bytes 0x6b 0x38 0x73, with the fourth byte reserved for the encoding style. The only encoding style defined is 0x00, meaning the rest of the byte stream is a message of type k8s.io.kubernetes.pkg.runtime.Unknown (proto2).

Struct-Aware Fuzzing and Test Cases : The mutated data is incorporated into test cases to assess the unmarshaling logic. Each test case specifies the expected object type and the mutated data (`wire`). The unmarshaling process is then evaluated against these cases to ensure validation.

Decoding and Validation : The protobuf-encoded data undergoes decoding, and any resultant errors are logged. The protobuf-encoded data is transformed back into a `Pod` object during the decoding process. Successful decoding operations validate the unmarshaling logic while errors are highlighted.

4.3 Error Handling and Logging

Error handling and logging are necessary to identify and rectify issues during fuzzing. Our implementation includes mechanisms to capture and log errors, including panics, ensuring thorough post-analysis MC4.3.

Capturing Panics : A deferred function captures any panics during execution, ensuring these events are logged appropriately. This deferred function leverages the `recover` function to intercept panics. The extracted error message is used to exclude nil pointer dereference errors, which are non-actionable, thereby maintaining the relevance and utility of the logged data.

Logging Errors : Errors captured during execution are logged with detailed information, including timestamps, error types, error messages, and function names. The log capture function constructs a log entry encapsulating the error type, message, function name, and timestamp. This logging facilitates analysis and identification.

```
logEntry := ErrorLog{
    Timestamp:    time.Now().Format(time.RFC3339),
    ErrorType:    "Unmarshal Error",
    ErrorMessage: err.Error(),
    FunctionName: "UnmarshalPod",
}
```

4.4 Summary

This chapter discusses the implementation of a Structure-aware fuzzer. The fuzzer is designed to uncover errors in Kubernetes components by generating and mutating structured protobuf data. (MC4.1) involves integrating the fuzzing of structured data (protobuf messages) into Kubernetes, enabling the fuzzer to handle complex object hierarchies and links Kubernetes components to the fuzzing process through protobuf definitions and Go-exported C headers. (MC4.2) introduces the handling of mutated protobuf data, which is encapsulated and transformed into byte slices to simulate real Kubernetes scenarios.

(MC4.3) implements error handling and logging mechanisms to capture execution errors such as panics and to document relevant metadata for post-mortem analysis.

5

Evaluation of the Erroneous Objects Generated

Main Findings

- **Main Finding 5.1 (MF5.1)**: Structure-aware fuzzing significantly improves code coverage and explores more execution paths than String-based input fuzzing.
- **Main Finding 5.2 (MF5.2)**: The Structure-aware fuzzer processes a larger and more complex input corpus, leading to broader input exploration and slower execution speeds.
- **Main Finding 5.3 (MF5.3)**: Specific errors are highly reproducible, indicating consistent issues in particular functions, while others require refinement due to inconsistent reproducibility.
- **Main Finding 5.4 (MF5.4)**: Feedback-driven corpus generation leads to faster and more consistent coverage growth, discovering additional code paths in less time.
- **Main Finding 5.5 (MF5.5)**: Long runtimes (3-5 days) are necessary to uncover deeper bugs, with functions like HandleMemExceeded running for almost 40 hours.

In this part of the thesis, we have discussed analyzing discovered errors using Structure-aware fuzzing on Kubernetes. In this chapter, we will evaluate the effectiveness and efficiency of our design and implementation by answering the relevant research questions.

Due to no public availability of the Structure-aware fuzzing benchmark or coverage approaches, particularly for this problem statement, as well as due to difficulty in comparison of the fuzzing approaches as mentioned at (36) by Klees et al., We focus evaluation mainly on error objects and newly discovered code coverage paths created during the Structure-

aware fuzzing approach. We will evaluate if this methodology is capable of producing the error objects. We will also measure the Structure-aware fuzzer with the String-based fuzzer, which only mutates the input values instead of structures.

The fuzzer efficiency is measured based on the number of distinct bugs it can find (36). If a Fuzzer finds more bugs than the baseline, then we can say it is more effective. In our case, the fuzzer efficiency can be measured on several error objects created during the run. The baseline is the code coverage of the String-based input fuzzer.

5.1 Abstract Syntax Tree for Static Analysis

We use Abstract Syntax Tree(AST) to extract the relevant information on errors and logging functions. The hierarchical nature of the Abstract Syntax Tree (AST) allows us to inspect nested function calls, arguments, and expressions, which is essential for identifying logging patterns such as *klog.V(...)*, *InfoS(...)*, *fmt.Errorf*, *log.ErrorS* and others. The Go Parser generates an AST representation of the Go source code. This AST is a hierarchical tree structure where each node represents a syntactic element such as function calls, variable declarations, etc. The main parameters we gather are function name, position, and error strings, and we store them in a JSON file.

- **Function Name**: The name of the logging function (e.g., InfoS, ErrorS, Errorf).
- **Position**: The location in the source file where the logging function call occurs. This is important for tracking log messages exactly back to the lines of code generated, providing context for the log entry.
- **Error Strings**: The message or format string with the logging call. This is particularly important for structured logging systems, such as klog, where the log messages are often key-value pairs.

Each of these patterns requires a different approach to extraction. By inspecting the AST, we can identify not only the top-level function call (e.g., InfoS, ErrorS) but also its nested components (e.g., V(...) for log levels or Errorf format strings). This allows us to understand how logging is handled throughout the codebase. It identifies the specific logging functions by traversing the AST and checking the structure of each function call. This allows us to track the position of tokens in the file and allows the static analysis on fuzzer logs to find precisely the location of function calls within the source file.

5.2 Expremiental Setup

The Experimental system includes three compute nodes that run the fuzzing workload based on the available hardware resources. This architecture in Figure 5.1 is ideal for LPM fuzzers such as LLVM's LibFuzzer, which benefits from parallel processing and high I/O throughput.

The corpus is central to this Experimental setup, and it creates a collection of error input cases that the fuzzer mutates and executes to discover new code paths or vulnerabilities. As new inputs are found, the corpus is dynamically updated during the fuzzing process.

Each machine processes a portion of the corpus based on the workload distribution. The setup periodically syncs results with the central repository. This syncing period is done manually via the rsync tool (37) in Linux. This ensures that new, exciting error cases are shared across all compute nodes, promoting faster discovery of edge cases.

Table 5.1: Technical specifications of the experiment infrastructure.

Host	CPU	Cores	RAM	Storage	Instances
1	Intel Xeon vCPU	16	32GB	320GB SATA SSD	1
2	Intel Xeon vCPU	8	16GB	120GB SATA SSD	3
3	AMD Ryzen 9 7800x3d	12	64GB	2TB NVMe SSD	1

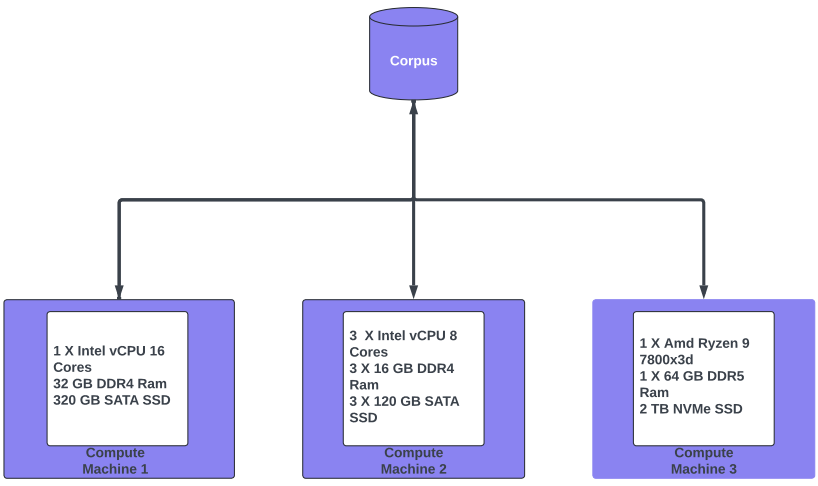

Figure 5.1: Expremiental setup to evaluate the fuzzer with corpus sharing.

5.2.1 Timeouts during runtime

One of the main factors evaluating the fuzzer on a particular target is the time. These generally range from hours to days and weeks. The recent papers that use LAVA as the benchmark choose 5 hours (38). In this paper (39) by Böhme et al., AFLFasts's evaluation with the AFL until 6 hours wasn't able to find bugs. But running it for longer, which is almost for the next 20 hours, 40 bugs were found. One of the main reasons the shorter runtimes are considered is because of computing power and hardware resources. Despite having fewer resources, we will have longer runtimes (3-5 days) only if the function does not quit because of Out-of-Memory and Slow Unit issues. If a function cannot run for more than 4-5 hours, we will stop the run on that function.

5.2.2 Key Elements Used in the Evaluation

- **cov**: This refers to the number of code coverage units (e.g., lines, blocks, or edges) that have been executed. A higher number indicates a broader exploration of the codebase.
- **ft**: refers to the number of features or function calls executed during the fuzzing process.
- **corp**: This shows the corpus size and the set of test cases libFuzzer uses. The corpus size is the number of files or inputs, followed by their total size in kilobytes.
- **lim**: This indicates the execution limit for each test case, typically measured in the number of instructions or some unit of time.
- **exec/s**: This is the number of executions per second, reflecting the speed of the fuzzing process.
- **rss**: This stands for Resident Set Size, which measures the memory used.
- **L**: Length of the inputs being tested.

5.3 Evaluation

In this section, We initially discuss comparing the String-based input fuzzer and the Structure-aware fuzzer. Then, we verify the status of error objects. Every error object found after fuzzing would be rerun and reported on the particular input function.

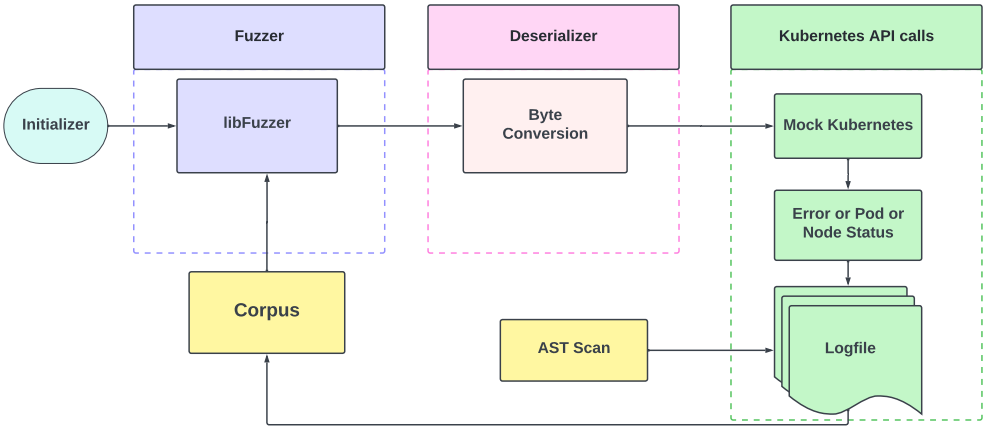

Figure 5.2: String-based input fuzzer for Kubernetes components.

5.3.1 Coverage Information of String-based Input Fuzzer and Structure-Aware Fuzzer

To compare the performance of the Structure-Aware Fuzzer, we need to establish a baseline. So, we choose the baseline as a String-based input fuzzer. String-based input fuzzers have a similar flow to Structure-aware fuzzers, but there are differences in input creation and deserialization. Input is created using the libFuzzer only, and the incoming bytes are converted as Strings and Integers depending on the kubelet component requirement. The String-based input fuzzer design is shown in Figure 5.2. Instead of passing structures, we pass only inputs mutated to the Kubernetes components.

When we compare the performance of the two fuzzers, the String-based input fuzzer on Table 5.2 and Structure-aware fuzzer on Table 5.3, significant differences are found. The Structure-aware fuzzer consistently achieves higher code coverage and processes more function calls (MF5.1). For instance, it achieves coverage of 13,008 for **HandleMemExceeded** compared to 86 in the String-based input fuzzer. This difference underscores the Structure-aware fuzzer's ability to explore various code paths.

It also tested a larger input corpus, contributing to its more extensive fuzzing process (MF5.2). Corpus sizes like 3,564Kb were compared to the smaller and simpler corpus of 28b, which the String-based input fuzzer tested. However, this increased thoroughness came at the cost of slower execution speeds. The functions like **SyncPodsDoesNotSetPodsThatDidNotRun -TooLongToFailed** where the execution rate dropped from 203

exec/s in the String-based input fuzzer to 87 exec/s in the Structure-aware fuzzer.

The Structure-aware fuzzer demonstrated efficient memory usage across most functions. These results suggest that the String-based input fuzzer operates at higher speeds. One notable difference across both fuzzers is the function **HandlePodRemovesWhenSourcesAreReady**, which exhibits high coverage but low execution speed and high memory consumption in the Structure-aware fuzzer but very high execution speed and lower memory usage in the String-based input fuzzer, indicating that this particular function is resource-intensive and complex. The Structure-aware fuzzer offers more extensive code coverage and input diversity, making it more suitable for in-depth error-generation scenarios.

5.3.2 Overview of Errors Objects Generated

The data analyzed includes various Error cases, each evaluated across multiple error metrics, including crashes (errors), slow units, false positives, reproducibility, coverage existence, and nil pointer exceptions. This analysis aims to determine the contribution of these factors to each Error Object Generated's overall performance and reliability.

The Table 5.3 represents the intensity of each error metric across the different error objects generated. The separation of crashes and slow units, along with the division by other factors like false positives, reproducibility, coverage existence, and nil pointer exceptions, yielded several key observations:

Certain Kubelet functions, such as **DoesNotDeletePodDirsIfContainerIsRunning**, experience many crashes and slow units. This highlights potential performance bottlenecks, where crashes and delays concurrently impact system reliability. In contrast, Kubelet functions such as **SyncPodsDoesNotSetPodsThatDidNotRunTooLong** have negligible crashes and slow units, indicating that these inputs are more efficient and stable.

Reproducibility : Reproducibility in fuzzing refers to the ability to consistently reproduce the conditions that lead to a specific crash or bug in a target application. In our case, the error input leads to a specific crash. Reproducibility varied significantly across errors and slow units generated. Specific errors demonstrated high reproducibility rates, while others have none (MF5.3). This indicates that some errors consistently produced the same results under repeated conditions. There are some error cases found where the functions exited without any errors.

Function Name	Cov	Ft	Corpus	Limit	Exec/s	Rss
SyncPodsDoesNotSet-PodsThatDidNotRunTooLong-ToFailed	86	109	28b	4,096	203	1260
DeleteOrphanedMirrorPods	86	109	24b	4,096	155	1931
DeletePodDirsForDeletedPods	86	109	24b	4,096	153	1904
DispatchWorkOfActivePod	86	109	24b	4,096	153	1798
DispatchWorkOfCompletedPod	86	109	24b	4,096	153	1831
GetPodsToSync	86	109	24b	4,096	153	1827
HandleHostNameConflicts	86	109	24b	4,096	153	1830
HandleMemExceeded	86	109	24b	4,096	154	1947
HandleNodeSelectorBasedOnOS	86	109	24b	4,096	154	1929
HandlePodAdditionsInvokes-PodAdmitHandlers	86	109	24b	4,096	191	1929
HandlePodRemovesWhen-SourcesAreReady	86	109	13b	4,096	34,065	548
SyncPodsSetStatusToFailed-ForPodsThatRunTooLong	86	109	24b	4,096	152	1918
CreateMirrorPod	86	109	24b	4,096	30,706	544
DoesNotDeletePodDirsFor-TerminatedPods	86	109	24b	4,096	143	1294
DoesNotDeletePodDirsIfCon-tainerIsRunning	86	109	24b	4,096	153	1904
FilterOutInactivePods	86	109	13b	4,096	129	1280
GenerateAPIPodStatusWith-DifferentRestartPolicies	86	109	24b	4,096	151	1921
GenerateAPIPodStatusWith-ReasonCache	86	109	24b	4,096	151	1983
GenerateAPIPodStatusWith-SortedContainers	86	109	24b	4,096	151	2001
HandlePluginResources	86	109	24b	4,096	153	1988
HandlePortConflicts	86	109	24b	4,096	132	1271
PurgingObsoleteStatusMap-Entries	86	109	24b	4,096	154	1936
ValidateContainerLogStatus	86	109	24b	4,096	195	134

Table 5.2: String-based input fuzzer max coverage and performance output on the Kubelet.

Function Name	Cov	Ft	Corpus	Limit	Exec/s	Rss
SyncPodsDoesNotSet-PodsThatDidNotRunTooLong-ToFailed	11,552	44,098	3,564	4,096	87	932
DeleteOrphanedMirrorPods	11,462	33,705	1,099	4,096	187	598
DeletePodDirsForDeletedPods	9,770	18,777	1,170	4,096	4,904	501
DispatchWorkOfActivePod	11,552	49,582	293	4,096	4,904	1,172
DispatchWorkOfCompletedPod	11,552	42,460	3,236	4,096	80	767
GetPodsToSync	11,552	48,524	4,485	4,096	27	1,891
HandleHostNameConflicts	7,226	13,041	628	4,096	961	369
HandleMemExceeded	13,008	47,900	3,884	4,096	65	961
HandleNodeSelectorBasedOnOS	13,008	47,900	5,924	4,096	454	1,028
HandlePodAdditionsInvokes-PodAdmitHandlers	13,008	49,437	4,449	4,096	122	1,782
HandlePodRemovesWhen-SourcesAreReady	11,521	33,942	1,896	4,096	108	545
SyncPodsSetStatusToFailed-ForPodsThatRunTooLong	11,351	27,011	1,506	4,096	157	479
CreateMirrorPod	4,927	6,612	318	4,096	7	485
DoesNotDeletePodDirsFor-TerminatedPods	8,103	12,350	700	4,096	4	461
DoesNotDeletePodDirsIfCon-tainerIsRunning	10,256	19,487	1,158	4,096	2	494
FilterOutInactivePods	11,552	46,912	4,045	4,096	108	1,430
GenerateAPIPodStatusWith-DifferentRestartPolicies	9,526	16,065	998	4,096	2	522
GenerateAPIPodStatusWith-ReasonCache	11,552	47,650	4,107	4,096	137	1,312
GenerateAPIPodStatusWith-SortedContainers	11,552	47,771	4,115	4,096	137	1,332
HandlePluginResources	7,397	10,985	418	4,096	341	182
HandlePortConflicts	10,231	16,946	735	4,096	195	390
PurgingObsoleteStatusMap-Entries	2,462	3,010	114	4,096	170	409
ValidateContainerLogStatus	11,552	48,201	4,174	4,096	136	1,307

Table 5.3: Structure-aware fuzzer max coverage and performance output on the Kubelet.

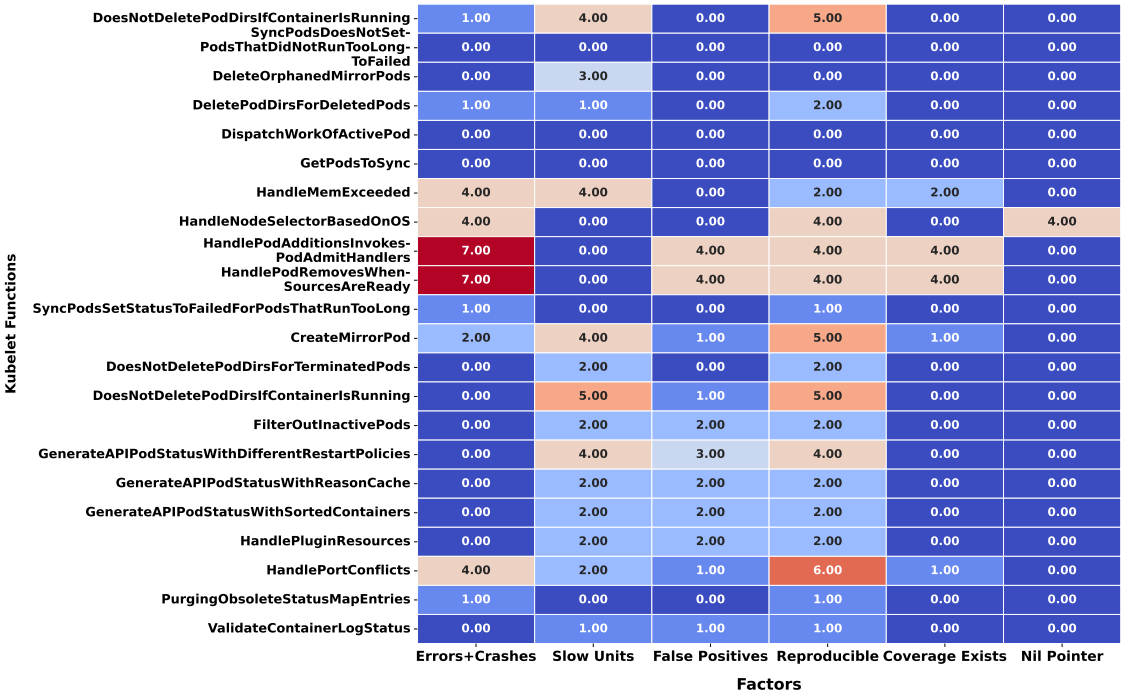

Figure 5.3: Metrics of the error objects generated.

False Positives : Across most Kubelet functions, false positives are minimal. However, the few instances of false positives may have led to misleading results in overall fuzzer performance, potentially requiring additional validation.

Coverage Exists and Nil Pointer : The factors of coverage existence and nil pointer exceptions are observed to be more uniform across, with relatively low values. This suggests that, while they are present, they do not contribute as significantly to failures compared to crashes, slow units, and reproducibility issues.

Timeouts During Runtime : One of the examples of early execution termination in our runtime is *TestHandleHostNameConflicts*. The function frequently terminates with slow units and OoM issues after 2.5 hours, as shown in Figure 5.4. *TestGetPodsToSync* could run for more than 30 hours, as shown in Figure 5.4. Still, a few instances(which is the parallel workloads) in the fuzzer workload quit early because of the slow unit and OoM cases (MF5.5).

- **Key Observations** :

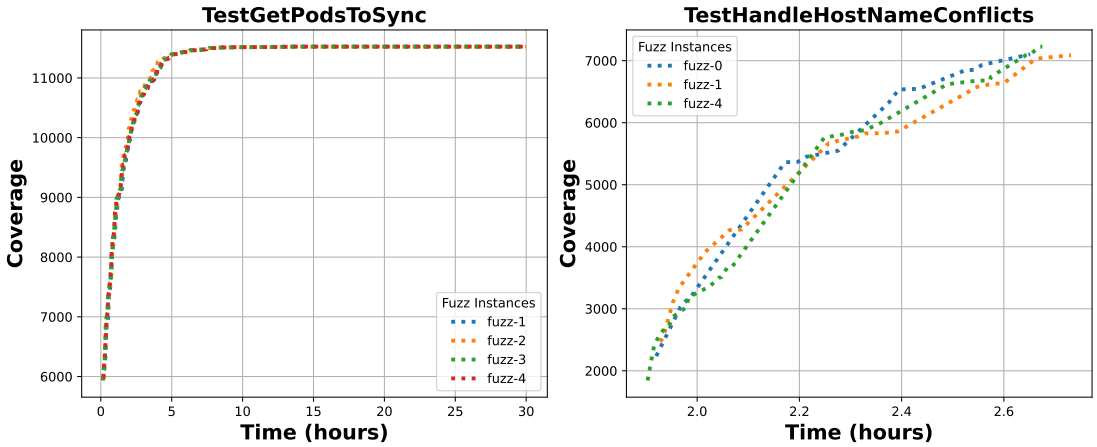

Figure 5.4: HandleHostNameConflicts and GetPodsToSync feature coverage on parallel workloads.

- **DoesNotDeletePodDirsIfContainerIsRunning**: This function case shows a high rate of both crashes and slow units, making it one of the most interesting in the fuzzing process. Despite having no false positives or nil pointer exceptions, the reproducibility factor is moderately high, suggesting that the issues identified are consistent and reproducible across different runs.
- **DeletePodDirsForDeletedPods**: This function case has fewer crashes but is marked by a moderate number of slow units. The lack of false positives and other errors indicated that while the function may be slow, it is relatively accurate and reliable.
- **DeleteOrphanedMirrorPods**: With low values across all metrics, this function case shows strong stability compared to other functions.

5.4 Impact of Feeback Driven Corpus Generation

We have taken a sample of three functions to measure the effectiveness of feedback-based corpus generation. The following functions coverage data is taken for 10 hours *TestHandlePluginResources*, *TestHandlePodAdditionsInvokesPodAdmitHandlers*, and *TestFilterOutInactivePods*. The first graph in Figure 5.5 represents the data for fuzzing data without the corpus, and the second graph in Figure 5.6 represents the data with a feedback-based corpus.

There is faster and more consistent growth in coverage over the iterations for feedback with corpus. This indicates that the fuzzer has taken advantage of the feedback to focus

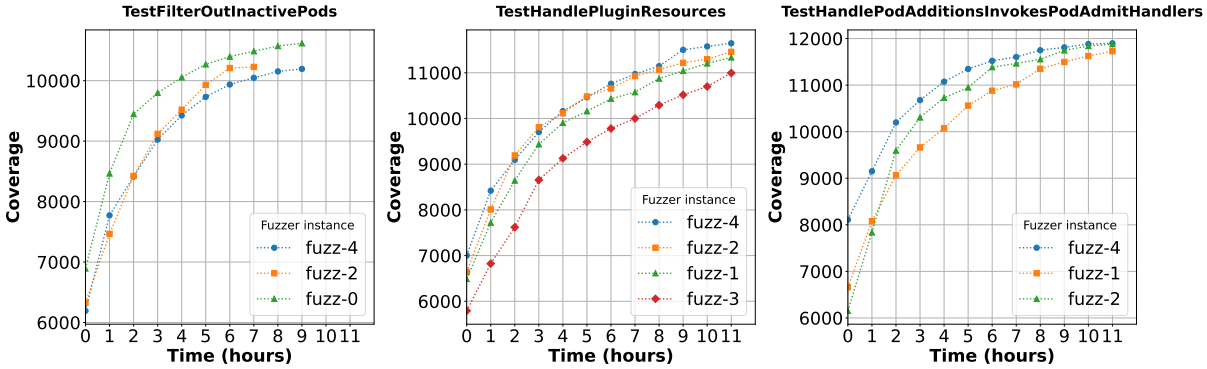

Figure 5.5: Feature coverage on parallel workloads without corpus feedback.

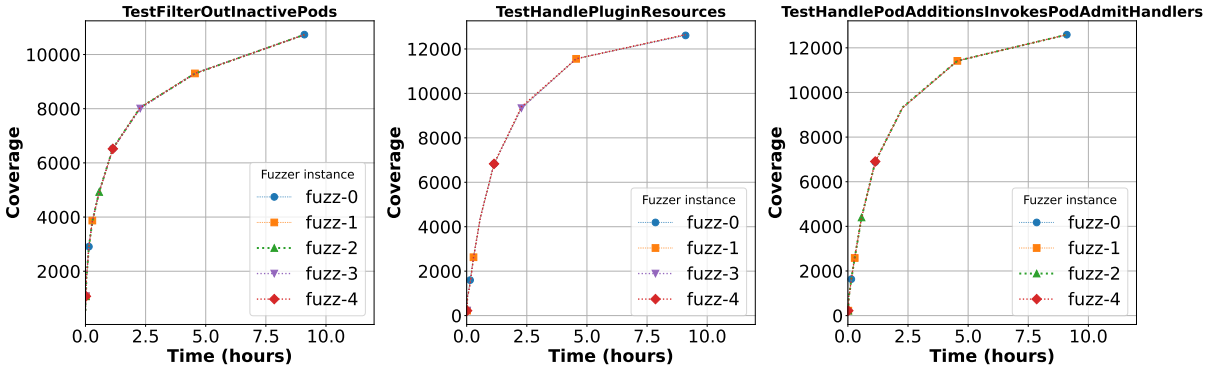

Figure 5.6: Feature coverage on parallel workloads with corpus feedback.

on and create inputs that would cover a greater number of code paths, thereby achieving more coverage within a shorter amount of time. Without Corpus, The fuzzer without a feedback-driven corpus shows slower coverage growth, indicating that the fuzzer is likely exploring many redundant paths or failing to focus on areas of the code that have yet to be explored (MF5.4). The mutation strategy is guided by feedback from prior iterations, which means the inputs evolve based on their success in triggering new code paths. This leads to an exponential increase in coverage, as indicated by the growth curve in Figure 5.7. The fuzzing process is more random without feedback corpus. The mutations might not be suitably directed to the areas of the code that should be investigated, leading to reduced efficiency in the search for new paths.

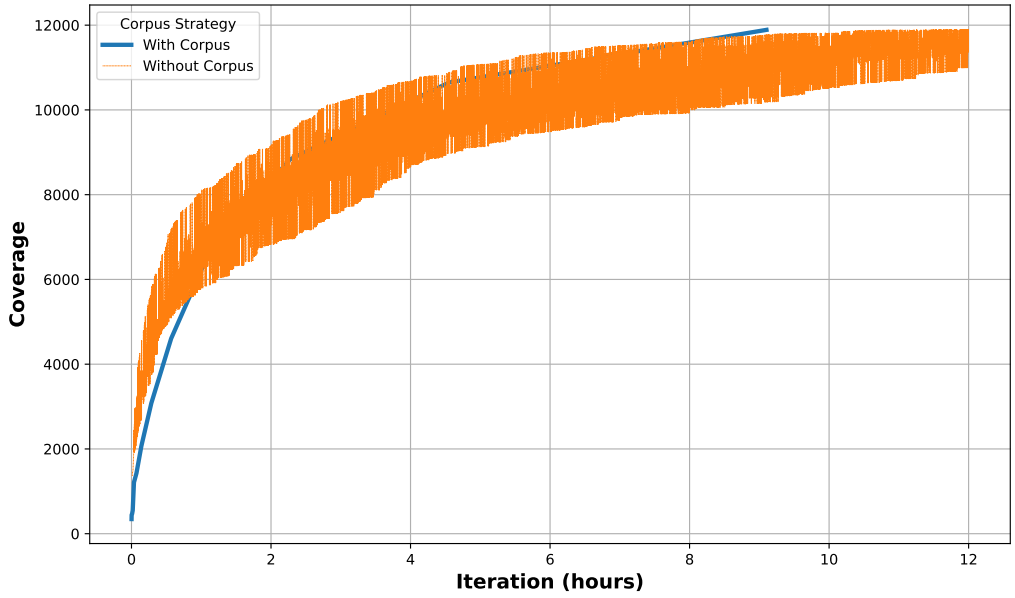

Figure 5.7: Coverage growth of 3 sample functions mentioned over time in hours (with vs without corpus).

5.5 Performance of AST Static Analyzer

The Static Analyzer starts by reading a JSON file containing log entries extracted from the AST, including function names, positions, and error strings. These AST entries form the foundation for error detection. The program then recursively scans directories to collect .log files, indexing each file using a Bleve search index (40). To improve efficiency, a worker pool is utilized where each worker processes and indexes a log file concurrently. The contents of each log file are indexed in batches, focusing on optimizing the indexing process. After indexing, the program searches the index for approximate matches (using fuzzy search) between the log lines and error strings from the AST. The results are written to an output file, identifying specific log lines that match or closely resemble the errors extracted from the AST. The Table 5.4 compares Time Taken to index against the log file size and many parallel workloads run on host machine three in Table 5.1.

If an error is detected, it logs the match along with details like the function name, position in the source file, and the log file path. A unique detection key is created for each error string and position to avoid duplicate logging, ensuring that the same error isn't logged multiple times for the same subfolder.

Table 5.4: Comparison of time taken to index against the LogFile size of each fuzzed component on 10 workers.

Function Name	Indexing Time in Min (Mean)	Log Size in MB (Mean)	Number of Fuzzers
DoesNotDeletePodDirsIf ContainerIsRunning	20.19	1,503	6
SyncPodsDoesNotSetPods ThatDidNotRunTooLongToFailed	6.11	81.63	6
SyncPodsSetStatusToFailed ForPodsThatRunTooLong	6.55	960	4
DeleteOrphanedMirrorPods	5.53	23.69	4
DeletePodDirsForDeletedPods	0.91	3.98	6
DispatchWorkOfCompletedPod	10.70	247	6
DoesNotDeletePodDirsIf ContainerIsRunning	2.22	62	4
DoesNotDeletePodDirs ForTerminatedPods	3.75	67	4
FilterOutInactivePods	23.68	2,920	4
GetPodsToSync	22.11	3,680	5
HandleHostNameConflicts	12.14	540	4
HandleMemExceeded	13.44	370	4
HandlePluginResources	13.12	360	4
HandlePodAdditions InvokesPodAdmitHandlers	12.96	360	4
PurgingObsoleteStatus MapEntries	13.82	1,500	4
ValidateContainerLogStatus	10.33	423	4
PodResourceAllocationReset	3.71	140	4
GenerateAPIPodStatusWith DifferentRestartPolicies	3.73	44	4
GenerateAPIPodStatus WithReasonCache	22.27	2,650	4
GenerateAPIPodStatus WithSortedContainers	3.06	139	4
HandleNodeSelectorBasedOnOS	18.86	957	4

6

Conclustion And Future Work

This chapter summarizes our main findings and contributions, associates them with the research questions they answer, and discusses future work.

6.1 Summary of Answers to main research questions

RQ1) **What fuzzing design choices enable generating erroneous objects that target error modes in Kubernetes**?

In Chapter 3, We proposed a fuzzer design for generating Error objects for Kubernetes objects. We go through different requirements that need to be satisfied to generate suitable error objects for Kubernetes to create errors. We use the existing Structure-aware fuzzers to inject the data into the Kubernetes APIs. Structure-aware fuzzing ensures that generated erroneous inputs conform to Kubernetes' schema while triggering meaningful errors. Nested structures like Pods, Nodes, and Containers require handling field interdependencies, boundary values, and invalid combinations during error generation. Coverage-directed fuzzing improves input exploration by maximizing code paths executed in the kubelet, using feedback loops and global input corpus.

- **Main Contribution 3.1 (MC3.1)**: We analyze the requirements to generate suitable error objects that conform to Kubernetes' structure and trigger meaningful errors.
- **Main Contribution 3.2 (MC3.2)**: We introduce the use of Structure-aware fuzzers to inject data into Kubernetes APIs, handling nested structures (Pods, Nodes, Containers)

- **Main Contribution 3.3 (MC3.3)**: We implement coverage-directed fuzzing to enhance input exploration by maximizing the code paths executed in the kubelet using feedback loops and a global input corpus.

RQ2) **How can the fuzzing framework with Go structures that explore Kubernetes errors be implemented**?

Integrating libFuzzer with libprotobuf-mutator enables effective fuzzing of structured data (protobuf messages) for Kubernetes components, uncovering errors through valid erroneous configurations. Protobuf definitions and Go Exported C headers are essential for providing fuzzing input structures and linking Kubernetes components with the fuzzing process. Handling of incoming protobuf data is encapsulated, prefixed with encoding headers, transformed into byte slices, and converted into Go Structs. Error handling and logging are important components of the system, capturing and logging execution errors (e.g., panics) with metadata and facilitating post-analysis. Panic recovery mechanisms ensure that panics are captured and logged during fuzzing, filtering non-actionable errors like nil pointer dereferences.

- **Main Contribution 4.1 (MC4.1)**: We integrate effective fuzzing of structured data (protobuf messages) for Kubernetes components.
- **Main Contribution 4.2 (MC4.2)**: We handle incoming mutated protobuf data, encapsulating and prefixing it with encoding headers and transforming it into Go Structs to simulate real Kubernetes scenarios.
- **Main Contribution 4.3 (MC4.3)**: We integrate error handling and logging to capture execution errors (e.g., panics) with metadata.

RQ3) **How to evaluate the efficacy of the fuzzing approach for Kubernetes Error object generation**?

Structure-aware fuzzing significantly improves code coverage and explores more execution paths than String-based input fuzzing. This highlights the benefit of understanding structures during fuzzing. Structure-aware fuzzer processes more input corpus and handles complex input cases but at the cost of slower execution speeds. The Structure-aware fuzzer dealt with a larger and more complex corpus, leading to a broader exploration of inputs. Despite higher code coverage, Structure-aware fuzzing showed more efficient memory usage in many cases, except for some resource-intensive functions. Some errors were highly reproducible, indicating consistent problems in specific functions, while others were

less consistent, requiring further refinement. Minimal false positives were observed across most functions, ensuring accuracy in error detection. Feedback-driven corpus generation resulted in faster and more consistent coverage growth, allowing the fuzzer to discover more code paths in less time. Long runtimes (3-5 days) were necessary to discover deeper bugs, with some functions like HandleMemExceeded running for almost 40 hours. Functions with slow units and Out-of-Memory (OoM) issues terminated prematurely, impacting fuzzing efficiency.

- **Main Finding 5.1 (MF5.1)**: Structure-aware fuzzing significantly improves code coverage and explores more execution paths than String-based input fuzzing.
- **Main Finding 5.2 (MF5.2)**: The Structure-aware fuzzer processes a larger and more complex input corpus, leading to broader input exploration and slower execution speeds.
- **Main Finding 5.3 (MF5.3)**: Specific errors are highly reproducible, indicating consistent issues in particular functions, while others require refinement due to inconsistent reproducibility.
- **Main Finding 5.4 (MF5.4)**: Feedback-driven corpus generation leads to faster and more consistent coverage growth, discovering additional code paths in less time.
- **Main Finding 5.5 (MF5.5)**: Long runtimes (3-5 days) are necessary to uncover deeper bugs, with functions like HandleMemExceeded running for almost 40 hours.

6.2 Future Work

Kubernetes is used in large-scale applications and distributed systems in sensitive production environments. Millions of Lines of Code are added every year. Future efforts should improve the fuzzing tool's operation to catch up with constant incoming changes from Kubernetes.

- **Optimize the Fuzzing Design and Tool**: The existing tool for fuzzing has good prospects but has yet to be improved further in effectiveness, particularly speed over the level of code coverage. There were some possibilities for improvement regarding input mutation processing and corpus handling efficiency. Optimizing the fuzzer's design will allow quicker test runs without sacrificing the coverage of Kubernetes components.

- **Faster Corpus Sharing**: The existing process of corpus sharing, which uses manual synchronization (e.g., via rsync), could be optimized for faster and more automated sharing of test cases across compute nodes. More efficient, automatic processes will make it possible to update the corpus in real-time, which will facilitate the creation of more test cases for edge cases.

- **Longer Runtimes**: As seen in the evaluation, extending the running time allows for more bugs to be captured and deeper code paths to be explored. Additional errors can be found by enabling the fuzzing to have a longer runtime of 3 to 5 days or weeks. Fuzzer testing strategy, from simple tests to more complex functions such as HandleMemExceeded, and continuous long-running fuzzing, will ensure that hard-to-reach errors are discovered and addressed.

- **More Kubernetes Components Fuzzed**: Currently, the target of the Structure-aware fuzzer is mainly kubelet. Future work should focus on fuzzing components outside the client, such as the API server, controllers, and networking layers. This will ensure that the system is resilient to erroneous inputs and errors, thus better evaluating the stability of Kubernetes systems.

- **More Compute Power Required**: The results show that some fuzzing inputs (e.g., fuzzing of complicated functions) are rather resource-demanding regarding memory and CPU. Increasing the computing capacity (such as issuing more computing machines or cloud resources) would lead to more parallel fuzzing jobs, faster test execution, and better corpus management. This would improve the performance of the fuzzer and the rate of finding bugs.

- **Enhanced Error Object Analysis**: Future iterations of the fuzzer should improve the handling of error objects, particularly in reproducibility and false positive reduction. A more sophisticated analysis framework for error objects could provide deeper insights into the causes of errors and facilitate faster debugging and patching of vulnerabilities.

- **Improved Feedback-driven Corpus Generation**: While feedback-driven corpus generation has demonstrated its effectiveness, there is room to refine this process further. Leveraging more advanced machine learning mechanisms could help the fuzzer prioritize inputs that are more likely to generate new coverage paths, resulting in even faster bug discovery.

- **Target Optimization for Performance Bottlenecks**: Specific Kubernetes functions, such as DoesNotDeletePodDirsIfContainerIsRunning, shows high crash and slow unit rates. Future work should optimize these performance bottlenecks by fine-tuning fuzzing parameters or investigating alternative fuzzing strategies designed explicitly for resource-intensive functions.

References

[1] The Kubernetes Authors. **Kubernetes: Production-Grade Container Orchestration**. `https://kubernetes.io/`, 2024. Accessed: 2024-09-12. 1

[2] Go Authors. **The Go Programming Language**, 2024. Accessed: 2024-05-19. 1

[3] Kubernetes Authors. **fuzzer.go**. `https://github.com/kubernetes/apimachinery/blob/master/pkg/apis/meta/fuzzer/fuzzer.go`, 2024. Accessed: 2024-05-19. 1

[4] Google. **Structure-Aware Fuzzing**. `https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md`. Accessed: March 18, 2024. 1, 12, 24

[5] CNCF Authors. **CNCF Fuzzers**. `https://github.com/cncf/cncf-fuzzing`, 2024. Accessed: 2024-05-19. 1

[6] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. **The Art, Science, and Engineering of Fuzzing: A Survey**. *IEEE Transactions on Software Engineering*, **47**(11):2312–2331, 2021. 1

[7] Eddy Truyen, Dimitri Van Landuyt, Davy Preuveneers, Bert Lagaisse, and Wouter Joosen. **A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks**. *Applied Sciences*, **9**(5), 2019. 1

[8] CNCF. **CNCF Annual Survey 2023**. `https://www.cncf.io/reports/cncf-annual-survey-2023/`, 2023. Accessed: 2024-09-22. 1

[9] Eddy Truyen, Nane Kratzke, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen. **Managing Feature Compatibility in Kubernetes: Vendor Comparison and Analysis**. *IEEE Access*, **8**:228420–228439, 2020. 2, 3

[10] **Amazon Elastic Kubernetes Service (EKS) User Guide**. https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html. Accessed: May 6, 2024. 2

[11] **Google Kubernetes Engine**. https://cloud.google.com/kubernetes-engine?hl=en. Accessed: May 6, 2024. 2

[12] **ServerFault: Q&A for System and Network Administrators**. https://serverfault.com/, 2024. Accessed: 2024-09-12. 2

[13] Alexandru Iosup, Fernando Kuipers, Ana Lucia Varbanescu, Paola Grosso, Animesh Trivedi, Jan Rellermeyer, Lin Wang, Alexandru Uta, and Francesco Regazzoni. **Future Computer Systems and Networking Research in the Netherlands: A Manifesto**, 2022. 6

[14] Nilton Bila, Paolo Dettori, Ali Kanso, Yuji Watanabe, and Alaa Youssef. **Leveraging the Serverless Architecture for Securing Linux Containers**. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 401–404, 2017. 7

[15] Kubernetes. **Architecture**. https://github.com/kubernetes/design-proposals-archive/blob/main/architecture/architecture.md, 2023. Accessed: 2024-04-15. 8, 17

[16] **Protocol Buffers in Kubernetes API Machinery**. https://github.com/kubernetes/design-proposals-archive/blob/acc25e14ca83dfda4f66d8cb1f1b491f26e78ffe/api-machinery/protobuf.md. Accessed: March 16, 2024. 8, 10

[17] Google. **Overview**. https://protobuf.dev/overview/, 2024. Accessed: 2023-04-23. 9

[18] Barton P. Miller, Lars Fredriksen, and Bryan So. **An empirical study of the reliability of UNIX utilities**. *Commun. ACM*, **33**(12):32–44, dec 1990. 11

[19] Barton Miller, David Koski, Cjin Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. **Fuzz Revisited: A Re-Examination of the Reliability of UNIX Utilities and Services**, 01 1998. 11

[20] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. **Finding and understanding bugs in C compilers**. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery. 11, 12

[21] GNU Project. **GCC, the GNU Compiler Collection**. `https://gcc.gnu.org/`, 2024. Accessed: 2024-09-22. 11

[22] LLVM Project. **Clang: a C Language Family Frontend for LLVM**. `https://clang.llvm.org/`, 2024. Accessed: 2024-09-22. 11

[23] P. Godefroid, M.Y. Levin, and D. Molnar. **SAGE: Whitebox fuzzing for security testing: SAGE has had a remarkable impact at Microsoft**. **10**:20–27, 01 2012. 11

[24] Google. **American Fuzzy Lop (AFL)**. `https://github.com/google/AFL`, 2024. Accessed: 2024-09-22. 11, 12

[25] LLVM Project. **LibFuzzer – A Library for Coverage-Guided Fuzz Testing**, 2024. Accessed: [Insert access date here]. 12

[26] George Torres, Davide Pesavento, Junxiao Shi, and Lotfi Benmohamed. **NFDFuzz: A Stateful Structure-Aware Fuzzer for Named Data Networking**. In *Proceedings of the 7th ACM Conference on Information-Centric Networking*, ICN '20, page 169–171, New York, NY, USA, 2020. Association for Computing Machinery. 12

[27] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. **Superion: Grammar-Aware Greybox Fuzzing**. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735, 2019. 12

[28] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. **WEIZZ: automatic grey-box fuzzing for structured binary formats**. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 1–13, New York, NY, USA, 2020. Association for Computing Machinery. 12

[29] Google. **libprotobuf-mutator: A library to randomly mutate protocol buffers**. `https://github.com/google/libprotobuf-mutator`, 2024. GitHub repository. 12, 13

[30] Google. **Chromium Issues: Protobuf Hotlist 2**. `https://issues.chromium.org/issues?q=hotlistid:5432475&lpm`, 2023. Accessed: 2024-05-11. 12

[31] **Chromium Issues: Protobuf Hotlist**. `https://issues.chromium.org/issues?q=hotlistid:5432475&protobuf&p=1`. Accessed: 2024-05-11. 12

[32] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. **Smart Greybox Fuzzing**. *IEEE Transactions on Software Engineering*, **47**(9):1980–1997, 2021. 13

[33] Kostya Serebryany. **Structure-Aware Fuzzing for Clang and LLVM with libprotobuf-mutator**. In *LLVM Developers' Meeting*, October 2017. Presentation at LLVM Developers' Meeting, San Jose, CA. 13

[34] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. **Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade**. *Queue*, **14**(1):70–93, jan 2016. 17

[35] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. **Large-scale cluster management at Google with Borg**. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. 17

[36] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. **Evaluating Fuzz Testing**. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery. 36, 37

[37] rsync. *rsync - Remote file and directory synchronization*, 2024. Accessed: 2024-09-18. 38

[38] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. **LAVA: Large-Scale Automated Vulnerability Addition**. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, 2016. 39

[39] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. **Coverage-based Greybox Fuzzing as Markov Chain**. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery. 39

[40] Bleve Search. **Bleve: A modern open-source search library for Go**, 2024. Accessed: 2024-09-18. 47

Appendix

This artifact description will explain how to setup the fuzzer and reproduce results as seen in this project. We explain how to obtain the software, set up the same environment, and run the fuzzer. We have two repositories for GitHub: the fuzzer environments and the Kubernetes forked environments, which are modified for fuzzer needs. There are also datanotebooks for plots and graphs

6.3 Artifact Check-List

- Program: Kubefuzzthesis (https://github.com/cymtrick/kubefuzzthesis/), kubernetes-error-fuzzer (https://github.com/cymtrick/kubernetes-error-fuzzer/),
- Compilation: C++11 above, Make, Go 1.22 and above, Python3.7 (Python3 is needed for the plotting tools only)
- Experiments: Experiments are maintained at

  https://github.com/cymtrick/kubefuzzthesis/tree/main/initial-tests.

6.4 Description

It's not required to fork the latest Kubernetes fuzzer as it's not supported. The software comes with correct compatible Kubernetes version :

```
$git clone https://github.com/cymtrick/kubefuzzthesis.git
$git submodule update --init
```

The implementation of the fuzzing on the kubelet can then be found in "src/kubelet" . An initial investigation for the fuzzing process on the go with C++ and a Make script to support it can be found in "initial-tests." Modified Kubernetes can be found under "libs/Kubernetes" . The datanotebooks for generating plots can be found "datanotebooks"

https://github.com/cymtrick/kubefuzzthesis/tree/main/initial-tests
https://github.com/cymtrick/kubefuzzthesis/tree/main/datanotebooks

6.5 Software Dependencies

The software will only run on GNU/Linux. It runs on only the latest versions of glibc and glibc++. Glibc 2.38 is recommended based on previous experience with failure on DAS-5, which has glibc-2.22 and can't be upgraded.

We recommend the libprotobuf mutator of version v1.2 as the next version runs into cmake build issues

https://github.com/google/libprotobuf-mutator/tree/v1.2

protoc –version should be less than v3.21.12 to support the current compilation. Otherwise, while running the libprotobuf mutator build, you would run into this issue.

```
CMake Warning at /usr/share/cmake-3.27/Modules/FindProtobuf.cmake:526 (message):
Protobuf compiler version 26.0 doesn't match library version 3.21.12
Call Stack (most recent call first):
CMakeLists.txt:129 (find_package
```

6.6 Experiment Workflow

There are a few steps to start the fuzzer successfully. The first would be compiling the mock Kubernetes layer along with the deserializer. These are present in **libs/kubernetes/pkg/fuzzer**.

```
$cd libs/kubernetes/pkg/fuzzer
$go build -o libpodfuzzermock.a -buildmode=c-archive fuzzpodmutator.go
$go build -o libyamlfuzzermock.a -buildmode=c-archive fuzzyamlmutator.go
```

Now, these static files need to be copied into the fuzzer location outside.

```
$cp libpodfuzzermock.* ../../../../src/kubelet
$cp libyamlfuzzermock.* ../../../../src/kubelet-yaml
```

The headers are already present, so the binaries are generated for each subfolder, which are named on function names. The make needs to be run first to develop the binaries. Before running make, we need to sure to remove the existing fuzzer files.

```
$cd src/kubelet
$find . -type d -exec sh -c '(cd "{}" &&
[ -f lpm_libfuzz_struct_aware_unguided ] &&
rm -rf lpm_libfuzz_struct_aware_unguided)' \;
$find . -type d -exec sh -c '(cd "{}" && [ -f Makefile ] && make)' \;
```

This will create fuzzer files inside the subfolders. Each subfolder has **lpm_libfuzz_struct_aware_guided** executable binary. This is the fuzzer entry point. To run this binary it can done in this way.

```
$mkdir corpus
$cd example-subfolder/
// 5 jobs in parallel
$./lpm_libfuzz_struct_aware_guided ../corpus -workers=5 -jobs=5
```

All of the different logs will be logged into **klog.log** and **fuzz-*.log** . You must pass the file to the fuzzer to reproduce a crash or slow unit.

```
$./lpm_libfuzz_struct_aware_guided ./slow-unit-*
$./lpm_libfuzz_struct_aware_guided ./crash-
```

For the AST Scan, there is a file called **src/check_errors.go** . This will run on the logs that are found in sub-directories and store the detected errors in the file **detected_errors.go**.

```
$go run check_errors.go
```

6.7 Self Reflection

This was challenging work. Initially, I had no idea about the internals of Kubernetes even though I had experience deploying on it as a dev-ops or any in-depth knowledge of fuzzing, even if I come from a security background. This topic has grounded me, taught me the fundamentals again, and made me more proficient in C and C ++ environments. I learned to code in Go lang from scratch and built a fuzzer on top of it. Additionally, I needed to understand the Kubernetes internals in-depth. Some Kubernetes internals were rewritten to support the fuzzing. This gave me in-depth confidence in understanding complex distributed systems such as Kubernetes. There were moments of doubt about whether the project was going in the right direction. I made the crucial mistake of fuzzing the inputs incorrectly in March, which could have generated invalid results. Thanks to my daily supervisor, Sacheen, for identifying the error and guiding me. Almost one week was spent correcting Kubernetes's protobuf data to support our run time. Resolving library dependency was challenging, mainly for the libprotobuf-mutator, as there's little to no documentation on resolving the compatibility issues. We had to do intensive research for a month in January to get the dependencies right. We also had trouble running the fuzzers on DAS-5 as the glibc version was not supported. Downgrading the fuzzer for almost 2

weeks didn't work, and there were many issues with the glibc compatibility . We went with the public cloud and local system. Happeniess was found when the AST scan revealed unique errors that were detected. Last month was purely data gathering, generating plots, and completing the writing part. Despite self-doubt and hardships, we could execute this project successfully.