Bugs in Pods: Understanding Bugs in Container Runtime Systems

Jiongchi Yu
Singapore Management University
Singapore
jcyu.2022@phdcs.smu.edu.sg

Xiaofei Xie
Singapore Management University
Singapore
xfxie@smu.edu.sg

Cen Zhang*
Nanyang Technological University
Singapore
cen001@e.ntu.edu.sg

Sen Chen
Tianjin University
Tianjin, China
senchen@tju.edu.cn

Yuekang Li
University of New South Wales
New South Wales, Australia
yuekang.li@unsw.edu.au

Wenbo Shen
Zhejiang University
Hangzhou, China
shenwenbo@zju.edu.cn

Abstract

Container Runtime Systems (CRSs), which form the foundational infrastructure of container clouds, are critically important due to their impact on the quality of container cloud implementations. However, a comprehensive understanding of the quality issues present in CRS implementations remains lacking. To bridge this gap, we conduct the first comprehensive empirical study of CRS bugs. Specifically, we gather 429 bugs from 8,271 commits across dominant CRS projects, including `runc`, `gvisor`, `containerd`, and `cri-o`. Through manual analysis, we develop taxonomies of CRS bug symptoms and root causes, comprising 16 and 13 categories, respectively. Furthermore, we evaluate the capability of popular testing approaches, including unit testing, integration testing, and fuzz testing in detecting these bugs. The results show that 78.79% of the bugs cannot be detected due to the lack of test drivers, oracles, and effective test cases. Based on the findings of our study, we present implications and future research directions for various stakeholders in the domain of CRSs. We hope that our work can lay the groundwork for future research on CRS bug detection.

CCS Concepts

• **Software and its engineering** → **Software verification and validation**; **Software testing and debugging**; • **Computer systems organization** → **Cloud computing**.

Keywords

Container Runtime, Software Testing, Empirical Study

ACM Reference Format:
Jiongchi Yu, Xiaofei Xie, Cen Zhang, Sen Chen, Yuekang Li, and Wenbo Shen. 2024. Bugs in Pods: Understanding Bugs in Container Runtime Systems. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680366

*Corresponding author

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680366

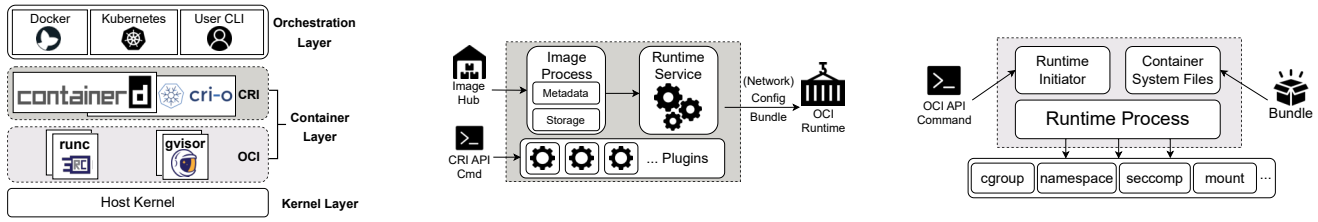
1 Introduction

Containers provide a highly flexible and portable solution for managing and deploying application workflows in the cloud. According to the Cloud Native Computing Foundation (CNCF) Annual Survey 2023 [5], containerization has become the predominant standard, with over 66% of organizations adopt cloud native technologies in production. It highlights the benefit of widespread containerization and the ongoing shift towards a container-centric approach in cloud computing. As a result, ensuring the quality, reliability and security of container infrastructure is paramount.

As depicted in Fig.1a, the container cloud architecture comprises three layers: *orchestration*, *container*, and *kernel* layers. The orchestration layer includes container cluster management platforms like Kubernetes [30], which automates and scales containers. The kernel layer provides basic isolation features for containers. The container layer, positioned as the middle layer, is composed of Container Runtime Systems (CRSs) that serve two primary functions: interpreting container orchestration commands complying with Container Runtime Interface (CRI) protocol [31] (e.g., `containerd` and `cri-o`) and operating containers within the Open Container Initiative (OCI) [47] (e.g., `runc`, `gvisor`). CRSs work as the foundational infrastructure of container clouds by enabling the deployment and management of containers. Therefore, it is important to ensure the reliability and security of runtime systems.

Motivation. Given the unique features of CRSs, such as multi-tenant application scenarios, complex configurations and elevated privileges for system-level communication, the bug patterns in CRSs can manifest in different ways, leading to characterized issues with reliability and security. As the foundational infrastructure in container services, faults in CRSs often necessitate immediate remedial actions for dependent cloud services, such as Docker [22] and Kubernetes [30]. For instance, the container vulnerabilities CVE-2019-5736 [17] and CVE-2024-21626 [21] can result in privilege escalation and container escape due to mishandling of process and file descriptor isolation in CRS. The root causes of these vulnerabilities are related to the unique lifecycle management of CRS. Moreover, considering the frequent updates and high security demands of CRSs, the rapid code changes can also introduce regression vulnerabilities, e.g., CVE-2023-27561 [20]. It is necessary to understand the characteristics of CRS and its corresponding issues.

While many studies have explored bugs in various software systems [4, 50, 51, 58], including container security [1, 35, 63] and performance issues [23], there is currently no comprehensive study



(a) Overview of Container Cloud (b) Architecture of CRI Runtime (c) Architecture of OCI Runtime

Figure 1: Overview of Container Cloud and CRSs

that delves into the characteristics of various bugs in CRSs. Specifically, there is a lack of understanding regarding the types of bugs in CRSs, as well as their symptoms, root causes, and the challenges involved in detecting them.

To fill this gap, we present the first study targeting understanding the bugs present in CRSs, aiming to answer three key research questions: **RQ1:** What are the common symptoms of bugs occurred in CRSs? **RQ2:** What are the prevalent root causes of these bugs? **RQ3:** How effective are existing methods in detecting different types of bugs in CRSs, and what are the main challenges?

Contributions. To answer these research questions, we select two representative projects from each of the two container layers, specifically `runc` and `gvisor` from the OCI layer, and `containerd` and `cri-o` from the CRI layer. The CRSs we selected are widely used in the production environment, serving billions of users. Among them, `runc` and `containerd` are the default runtimes for Docker and Kubernetes. `gvisor` is sandbox [65] runtime implementation developed by Google, which is used as the default runtime for the first generation execution environment of Google Cloud [25]. `cri-o` is specifically designed as the high-level runtime for Kubernetes. It is highly compatible with the CRI protocol and supports hybrid security-level OCI runtime workloads.

We collect commits from the four projects and extract bug-related commits. After filtering out irrelevant commits, we obtain a dataset of 429 commits containing bugs and corresponding fixes in total from the above four projects. We manually analyze each bug and its repair to gain a quantitative understanding of these bugs and summarize the symptoms (RQ1) and root causes (RQ2). Specifically, we summarize a total of 16 types of symptoms and 13 types of root causes. We further perform in-depth analysis to understand the unique features of CRS bugs, the differences compared to other software and the relationship between symptoms and root causes.

We further investigate the challenges involved in detecting these bugs by studying the three most widely used testing methods: unit testing, integration testing, and fuzz testing. We collect existing tests, analyze whether the collected bugs can be covered and identify the reasons behind these outcomes to address RQ3. Our results indicate that only approximately 20% of the gathered bugs can be automatically detected with existing testing approaches. This is largely attributable to the absence of test oracles 9.32%, test drivers 41.96%, or test cases 27.97%. By addressing these research questions, this study provides a comprehensive understanding of container bugs, which serve as a foundation for future work in enhancing the quality of CRSs.

In summary, this paper makes the following contributions:

- We conduct the first empirical study to systematically explore the characteristics of bugs in Container Runtime Systems (CRSs). We provide taxonomies for bug symptoms and root causes, offering insights into understanding CRS bug characteristics.
- We conduct the study on the effectiveness of existing testing methods in detecting different types of bugs in CRSs.
- We offer findings based on the developed taxonomies and provide recommendations for different stakeholders.
- We collect a dataset of bugs from CRSs, which can serve as a valuable benchmark for further research and testing of CRSs.

2 Background

As illustrated in Fig. 1, container cloud systems can be categorized into three layers: orchestration layer, container layer, and kernel layer. The orchestration layer handles the management of containers and resources, while the container layer is responsible for container images and container isolation. The container layer contains two sub-layers: the high-level container layer which complies with Container Runtime Interface (CRI) protocol, and the Open Container Initiative (OCI) regulated lower-level container layer. The kernel layer receives commands from the container layer and provides container isolation and resource limitations.

Container Runtime Initiative (CRI). `containerd` [7] and `cri-o` [15] are the top two recommended CRI runtime implementations by Kubernetes, which function as a tool for managing the lifecycle of lower-level container runtimes. The main tasks of the CRI layer include providing the OCI runtime with a prepared and configured image packages, managing image versions and snapshots, and handling networking configuration for the container, which is done through the gRPC service to receive commands. The CRI runtime is designed to be loosely coupled, depending on a variety of plugins for its functionality. These include built-in plugins such as snapshots and contents, as well as external plugins like `hcsshim` [39] or user-customized plugins. The unified container initiative protocol allows a CRI runtime to manage multiple life cycles of OCI runtimes with different implementations.

Open Container Initiative (OCI). OCI runtimes act as vital components for container services. The primary function of OCI runtimes is to parse the input configuration and communicate with the kernel to customize isolation and resource usage. The OCI runtime initiator will prepare all the processes required to start a container, after which the functions of the OCI runtime will be controlled by the runtime engine. For instance, `runc` [48], the *de facto* default OCI runtime for CRI runtimes, which transfer the received commands into the API of `libcontainer` for managing `cgroup`, `namespace`,

seccomp and etc. gvisor is a popular sandbox [65] OCI runtime, which interacts with the host system based on runsc and sentry.

3 Methodology

To comprehend the CRS bugs, we first gather the commits of runc, gvisor, cri-o and containerd, and then filter the results with keywords to identify bug-repairing commits, which are kept as candidate bugs. The collected commits are then used to study the first two research questions. Specifically, we manually read, triage, and label the bug-repairing commits, and then develop the taxonomy of both symptoms and root causes for each bug. To answer RQ3, we collect existing tests and subsequently execute them on corresponding software versions. We manually analyze the testing results and identify the specific reasons.

3.1 Data Collection

3.1.1 Bug Collection. Following previous works [51, 53, 54], we collect and analyze bug-repairing commits from four of the most widely-used CRS projects for bug analysis. Specifically, we select runc and gvisor for the respective OCI runtime layer. For the CRI runtime layer, we choose containerd and cri-o. These systems are all implemented in the Go programming language and are pre-configured as the default runtime combination in mainstream container orchestration systems such as Kubernetes and Docker/-Moby. We also attempt to include GitHub issues in our dataset. We find that all the bug-related issues are already linked to our selected bug commits, since the resolved issues are typically accompanied by fix commits. However, bug related commits may not always involve an issue (e.g., bugs fixed by developers independently).

We collect and analyze the commits over a two-year period, starting from June 1st, 2021. In total, we collect 1,081, 2,305, 2,981, and 1,904 commits from runc, gvisor, containerd and cri-o, respectively. Among all the commits we collected, bug-repairing commits are what we need for further analysis. Therefore, we follow the previous works [33, 51] by selecting suitable keywords for filtering the bug related commits. Specifically, we filter the commits messages using keywords including fix, error, bug, mistake, incorrect, flaw, fault, issue, performance, security, cve and vulnerability. Then we remove the duplicated commits, such as those with keywords like merge or pr. We are left with a total of 253, 458, 356 and 215 commits for the four projects, respectively.

For each commit, we conduct a thorough manual review of the developer's comments within the source code, the commit messages, and the discussions associated with attached issues and pull requests. After our manual confirmation and filtering process, we retain commits explicitly related to bug-repairing activities, totaling 99, 115, 152, and 63 bug-repairing commits, respectively. We then summarize the symptoms and root causes of CRS bugs from these 429 commits. During the analysis, we also check whether the commits are for patching security vulnerabilities by matching the commit with records from MITRE CVE Database [14] and Github Security Advisory of each project. In total, we assemble 19 unique security vulnerabilities, which are listed on our website [28].

3.1.2 Test Collection. To understand the challenges associated with detecting the collected bugs in CRSs, we examine the existing testing suites within CRS projects. We focus on three popular testing methods: unit testing, integration testing, and fuzz testing, which are commonly integrated in the selected projects (e.g., within the *tests* directory). Our main objective is to evaluate the effectiveness of these tests in detecting the collected bugs and gain insights into why certain bugs fail to be detected.

- *Unit test*. Since all four selected projects are primarily implemented in the Go programming language, the unit test adheres to the native Go test paradigm. Test files end with "test", function names begin with "Test", and the testing log is prefixed with "RUN". Based on these patterns, we track each unit test drivers following testing flags from the Makefile and source code. Eventually, we collect 314, 449, 734 and 62 unit tests for runc (release 1.1.4), gvisor (release 20230710.0), containerd (release 1.6.15) and cri-o (release 1.27.0).
- *Integration test*. Unlike unit tests, which focus on testing individual component or function, integration tests in CRS usually verify the assembled functionalities such as container create or delete. We examine the test build script of CRS projects and filter related flags such as integration. We also gather all the Go tests that are called during the integration testing. In total, we collect 170, 22, 190, and 309 integration tests for runc, gvisor, containerd and cri-o, respectively.
- *Fuzz testing*. The fuzz testing includes test cases written by OSS-Fuzz [3] and CNCF Fuzz [6], which are used to test key functions of CRS project. We follow the build scripts of CRS projects for these two fuzzing platforms and collect 11, 1, 28, 17 fuzz tests for runc, gvisor, containerd and cri-o, respectively.

After collecting the tests, we execute them to determine whether the corresponding bugs could be detected. Subsequently, we conduct a manual analysis of the dynamic execution information and summarize the challenges associated with detecting bugs from different categories.

3.2 Manual Analysis

To create a taxonomy for symptoms and root causes in CRS bugs, we follow the prior work with an open coding procedure [51–53] and split the commits into two halves for analysis. In the first round of analysis, two authors independently analyze the informative messages, bug behavior, modified files and testing results of each commit, and group them according to their symptoms or root causes. If a commit had unique symptoms or root causes, a new category will be created. As the taxonomies are refined during the manual analysis, the two authors discuss and clarify any differences in their categories. For any dispute, an arbitrator will be introduced to jointly discuss the resolution of the taxonomy result until they can reach the consensus. If they cannot reach consensus, the bug is classified as "Others". The second round repeats the procedure with the remaining 50% of the commits, and the rate of arbitration falls from 50% to 12%. In the third round, the authors sample 20% of the commits in each project five times and examine the proposed taxonomy. The controversial commits are discussed by all authors and lead to a reevaluation of the related categories.

The final version of the taxonomies are reviewed and confirmed by all the authors. For the security vulnerabilities of CRSs, the corresponding fix commits are picked with the keywords (e.g., CVE) or the assigned GitHub security advisory ID (e.g., GHSA). We jointly analyze and confirm the commits that are indeed the fix commits for

the security vulnerabilities. We evaluate the inter-rater reliability of labeling in each round with Cohen's Kappa (k) coefficient. In the initial round, the inter-rater reliability k is 0.55, which improves to 0.67 in the second round. After detailed analysis in the third round, the k value increases to 0.83, signifying good agreement [32].

4 RQ1: Symptom Taxonomy

Fig. 2 presents the hierarchical taxonomy of symptoms of container bugs in CRSs, organized into 4 major categories: *Build Failure*, *Unexpected Termination*, *Unexpected Functionality* and *Poor Performance*. Each high-level category is further subdivided into subcategories based on the bug characteristics.

To highlight the security vulnerabilities in CRSs, we label part of subcategories, i.e., leaf nodes in the taxonomy tree using red color. To distinguish the severity of each category, we adapt four color intensities based on the average CVSS score of the bugs. Our analysis reveals that over half of the existing CRS vulnerabilities (11 out of 19) exhibit the symptom of *Escalated Privilege*, underscoring the severity of this bug category.

Finding 1: We identify a total of 16 distinct leaf categories of bug symptoms. Among these categories, 5 (31.25%) exhibit the association with the security vulnerabilities of CRSs.

4.1 Build Failure (A)

The CRSs require compilation before providing the service, with various configurations on different supporting architectures. We find that 7.69% of the bugs occur during the building phase, falling under the category of *Build Failure*. The build failure mainly manifests as the dependencies errors, which occur when there are issues with the dependencies required to build CRSs.

Dependency errors can occur at *package level* and *API level*. Dependency errors at the package level can arise when there are frequent updates to upstream dependencies, leading to incompatibility issues during the building process. Dependency errors at the API level occur due to the need for distinct implementations and configurations to support various computing architectures and OS distributions, given that OCI and CRI software is designed to be platform-agnostic. As a result, the building process may encounter platform-specific unsupported API errors, leading to build failures.

Finding 2: A total of 7.69% of the bugs fall under the category of *Build Failure*, which is mainly attributed to the inherent requirement for CRSs to be platform-agnostic, leading to cross-platform compatibility issues such as package or API dependency errors.

4.2 Unexpected Termination (B)

A significant portion of bugs (20.98%) are related to unexpected terminations during the running of CRSs, including two main subcategories: *Unexpected Crash* and *Incorrect Exit Code*.

4.2.1 Unexpected Crash (B.1). The majority of Unexpected Termination bugs (88.89%) fall into the *Unexpected Crash* category. Although crashes are a common symptom, they occur in a variety of container-specific situations such as *Plugin Management Error (B.1.1)*, *Runtime Daemon Crash (B.1.2)*, and *System Communication Error (B.1.3)*. Notably, 25.00% of crashes occur during the management of plugins. CRSs, especially CRI runtimes, use many plugins to facilitate functionality and reduce coupling. However, the improper usage of plugins may lead to unexpected crashes. For instance, `containerd` crashes during the management of snapshot plugin, which is caused by the incorrect calling for *Commit* method [10].

There are 53.75% of crashes that directly break down the runtime daemon. For example, if `runc` starts the *systemd* in a container without setting specific system configuration (e.g., deviceAllowList), it will return a fatal error in accessing *devnull* location and lead to the crash of runtime daemon [45]. As CRSs usually manage container services with a resident daemon, the crash of the daemon could have severe consequences to the service of CRSs.

The remaining 21.25% of the crashes happen when CRSs communicate with the system-related API calls (e.g., using system API to request *cgroups*, *namespaces*, and *networking* resources from the host machine). It is a unique feature of CRSs, as common application software rarely handles these system-related resources.

4.2.2 Preset Exit Code (B.2). 11.11% of the bugs that caused unexpected termination are classified under the *Preset Exit Code* category. These bugs do not cause a CRS crash but instead terminate them with a predetermined exit code due to some errors. These exit codes represent abnormal termination status. An example is when using the `shim.Delete` command in `containerd`'s shim, which always returns a 137 exit code when cleaning up temporary resources (i.e., the task has been killed) [11].

Finding 3: 20.98% of the collected bugs manifest as *Unexpected Termination*. These symptoms are highly related to the features of CRSs workflow from managing runtime daemon, managing different plugins to communicating with the host using system calls.

4.3 Unexpected Functionality (C)

The prevalent error *Unexpected Functionality* constitutes 59.44% of the analyzed bugs. These bugs can be further categorized into: *Incorrect Execution Output*, *Authorization Error*, and *Logging Error*. Those bugs are closely related to the core functionalities of CRSs.

4.3.1 Logging Error (C.1). *Logging Error* accounts for 10.98% of the unexpected functionality bugs. As a core function in cloud systems, logging is crucial for cloud service maintenance and error diagnosis. Logging errors represent situations where logging is not implemented correctly, such as providing inaccurate logging information or creating overfilled log files. For example, overfilled log files can pose challenges in analysis and consume significant system storage space.

4.3.2 Incorrect Execution Output (C.2). The most prevalent category of unexpected functionality bugs in CRSs is *Incorrect Execution Output*, accounting for 70.59% of all bugs. This category refers to situations where the function output assigned to the runtime's internal variables leads to incorrect computational results, either due to incorrect or improperly formatted output. In addition to the common symptom of *Incorrect Return Value (C.2.1)*, there are two symptoms that are more related to CRSs: *Wrong Container Behavior* and *Incorrect Configuration Effect (C.2.3)*.

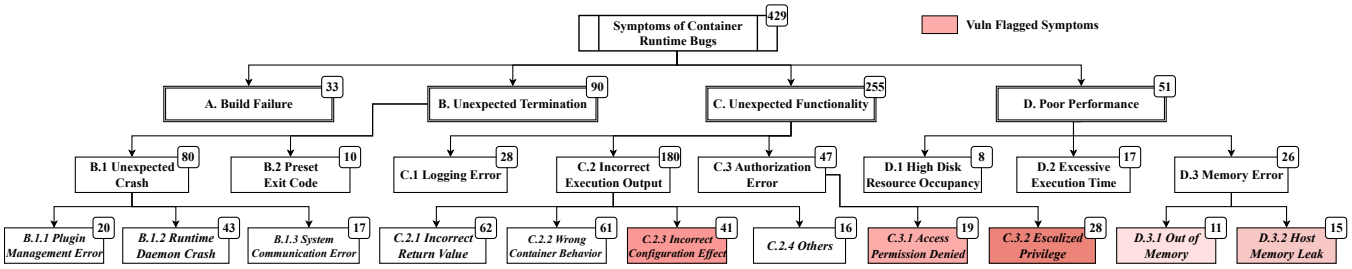


Figure 2: Symptoms of CRS Bugs

* **The numbers on the rectangles are the number of bugs, while the red colored categories are security vulnerability related symptoms.**

Wrong Container Behavior (C.2.2), which accounts for 34.44% of the incorrect execution output bugs, refers to the container operating with incorrect or unexpected behaviors from user commands.

Incorrect Configuration Effect, accounting for 22.78% of the incorrect output issues, occurs when the preset configuration does not work as expected, leading to incorrect computational results. For example, the configuration (*ENOSYS*) for `runc` does not work correctly on the *s390x* platform due to different syscall support policies for the kernel. There is also a category of *Other (C.2.4)* bugs, which account for 8.89% of the total, that rarely occur and have no identifiable characteristics, such as wrong output for exception warnings and misleading command line hints.

4.3.3 Authorization Error (C.3). Authorization is another critical feature in CRSs, granting them specific operating system privileges for specific usage within container instances. *Authorization Error* refers to bugs that cause improper authorization [56] and accounts for 18.43% of unexpected functionality bugs.

We identify two main symptoms for authorization errors: *Access Permission Denied (C.3.1)* and *Escalated Privilege (C.3.2)*, accounting for approximately half of the authorization errors, respectively. Access permission denied bugs typically occur when users are blocked from accessing specific container resources, while they should have permission to access the location or file. For instance, `runc` may falsely block the mounting point of *procsyskernelns_last_pid*, resulting in no write permission for a process. Conversely, *Escalated Privilege* bugs allow users to have access to container resources they are not supposed to have access to. These bugs can have severe consequences, ranging from container permission escalation to sandbox breakout of the preset isolation environment, reflecting the majority amount (59.57%) of security vulnerabilities.

Finding 4: The most common category of bugs in CRSs (59.44%) is *Unexpected Functionality*, with *Incorrect Execution Output* being the most prevalent subcategory (70.59%). From a security perspective, the most critical symptom is *Escalated Privilege*, often resulting from authorization errors. This symptom is closely related to the inherent characteristics of CRSs.

4.4 Poor Performance (D)

Poor Performance, which represents 11.89% of the total bugs, encompasses performance issues related to storage, memory, and execution time. This category includes *High Disk Resource Occupancy*, *Excessive Execution Time*, and *Memory Error*, which account for 15.69%, 33.33%, and 50.98% of *Poor Performance* bugs, respectively.

4.4.1 High Disk Resource Occupancy (D.1). *High Disk Resource Occupancy* often manifests as excessive disk usage and slow input/output (I/O) pipeline processing. Improper allocation of storage resources not only leads to I/O pipeline resource bottlenecks, but can also cause the over-consumption of the host storage that impacts on other containers. A typical example is the incorrect implementation of slice copy with append in `containerd` [12], which results in the storage space being used up more quickly than anticipated.

4.4.2 Excessive Execution Time (D.2). *Excessive Execution Time* can slow down the runtime significantly, potentially leading to non-termination. For instance, a bug was discovered in `runc` [49], where the incorrect execution results in a much longer execution time.

4.4.3 Memory Error (D.3). *Memory Error* bugs are mainly caused by *Out of Memory (D.3.1)* and *Host Memory Leak (D.3.2)*. For *Out of Memory*, the amount of memory allocated for a container is determined by the `memcfg` configuration. However, the bugs may cause the container to consume more memory than intended, even exceeding memory usage limits. This could lead to severe issues, including malicious memory consumption, which creates vulnerabilities for denial of service attacks and container escapes [64].

Host Memory Leak refers to the memory boundary leaks of CRSs, which can have serious consequences. For example, the byte order representation used in `runc` [46] differs from that expected by `systemd`, causing reversed `cpuset` ranges to be set in `systemd` transient unit [36].

Finding 5: Bugs with *Poor Performance* symptom account for 11.89% of all bugs and can manifest as the abnormal behaviors. These bugs are primarily attributed to the complexities involved in managing memory and storage within containerized environments.

5 RQ2: Root Cause Taxonomy

We further conduct an analysis of the root causes of the bugs collected in CRSs. The taxonomy of root causes is presented in Fig. 3, which includes categories of *Coding Error*, *Configuration Error*, and *Others*. There are a total of 13 leaf categories, including *others* that do not fit logically into the other categories.

Similarly, the categories in red represent the security vulnerabilities flagged bug root causes, *i.e.*, they have caused real world security consequences. These categories can be ranked into four groups based on their prevalence and colored with different intensities to represent their severity.

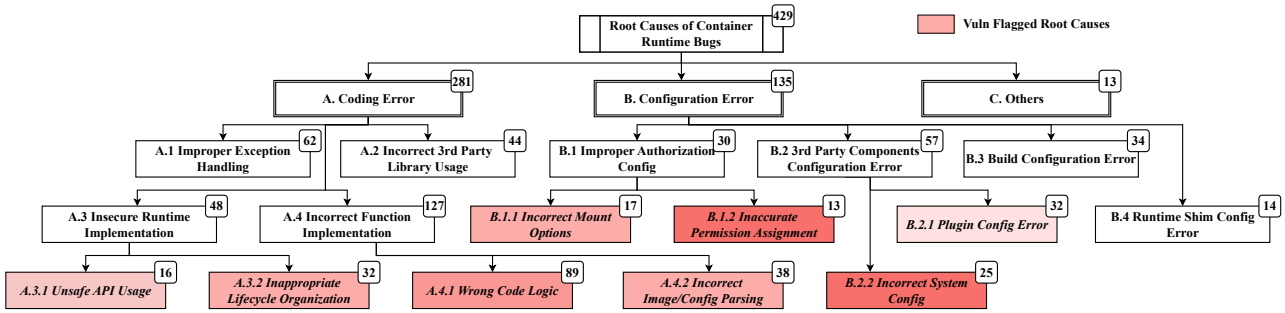


Figure 3: Root Causes of CRS Bugs.

* **The numbers on the rectangles are the number of bugs, while the red colored categories are security vulnerability related root causes.**

5.1 Coding Error (A)

The most prevalent root cause for CRS bugs is coding errors, accounting for 65.50% of all bugs, including *Improper Exception Handling*, *Incorrect 3rd Party Library Usage* as well as security vulnerability flagged root causes like *Insecure Runtime Implementation* and *Incorrect Function Implementation* (e.g., image parsing).

5.1.1 Improper Exception Handling (A.1). The *Improper Exception Handling* category is a root cause of 22.06% of the *Coding Error* bugs. When exceptions are not handled properly, the runtime may terminate unexpectedly and error prompts may be missing, resulting in dysfunctional runtime services and inaccurate error message. For instance, in `containerd`, the errors in function `gorestrl.rdt.ContainerClassFromAnnotations` were not properly handled, leading to ineffective configuration parsing and incorrect runtime functionality.

5.1.2 Incorrect 3rd Party Library Usage (A.2). 15.66% of the *Coding Error* bugs are caused by the incorrect usage of third-party libraries. Since CRSs rely on numerous third-party libraries during the development and runtime phases, it is important to use them correctly. Improper use includes issues such as unnecessary variable usage, inconsistencies in function implementation with the initially settled API, and compatibility problems with the runtime environment. For example, the third-party library `libseccomp` [44] changed its method from `ActKill` to `ActKillThread`. Calling the outdated version of the API may lead to undefined issues.

5.1.3 Insecure Runtime Implementation (A.3). 17.08% of the *Coding Error* bugs are caused by insecure runtime implementation, which can result in security issues such as incorrect authorization illustrated in Section 5. This category can be further divided into *Unsafe API Usage (A.3.1)* and *Inappropriate Lifecycle Organization (A.3.2)*. The improper use of internal or external *unsafe* APIs can introduce potential exploitable vulnerabilities into CRSs. For instance, CVE-2021-43816 [19] shows an example where the CRI plugin of `containerd` misused the API of SELinux security module, resulting in improper file relabeling of arbitrary files and directories [9]. It allows the bind mounts in `hostPath` volumes, thereby elevating container permissions.

Inappropriate Lifecycle Organization (A.3.2) refers to container lifecycle management issues that may cause the miss of permission checks, leading to issues such as memory leaks, wrong isolation allowance, and action handling race conditions. For example, a bug occurs when `containerd` crashes during the creation of a container, and the shim process is not properly cleaned up, which may bring leak of shim process and potential security issues.

5.1.4 Incorrect Function Implementation (A.4). *Incorrect Function Implementation* is a general category of errors that greatly affects the functionality of CRSs, and accounts for 45.20% of coding errors. The majority of the incorrect implementations (70.08%) are caused by *Wrong Code Logic (A.4.1)*. Another category is *Incorrect Image and Config Parsing (A.4.2)*, which accounts for the remaining 29.92% of incorrect implementation bugs. Due to the nature of cross-platform, CRSs need to parse different images and have many configurations that vary from memory, storage, to host mounting and networking. Therefore, incorrectly implemented parsing functions may result in wrong container behaviors or incorrect CRSs functionality.

Finding 6: *Coding Error* is the most common root cause of bugs in CRSs, accounting for 65.50% of all bugs. In addition to common coding errors such as incorrect API usage and flawed logic, coding errors in CRSs are often related to the importing of images and the management of container lifecycles.

5.2 Configuration Error (B)

In addition to coding errors, configuration errors are another significant root cause of bugs in CRSs, accounting for 31.47% of all bugs. Configuration errors can impact the performance of the runtime lifecycle, resulting in errors and affecting its overall functionality. Four sub-categories of configuration errors include: *Improper Authorization Configuration*, *3rd Party Component Configuration Error*, *Build Configuration Error*, and *Runtime Shim Configuration Error*.

5.2.1 Improper Authorization Config (B.1). CRSs, especially CRI runtimes, encounter numerous scenarios in processing authorization configurations to provide containerization services. Incorrect setting of such permission related configurations for containers can lead to functional issues and even security vulnerabilities. Specifically, these bugs can be classified into two categories: *Incorrect Mount Options (B.1.1)* and *Inaccurate Permission Assignment (B.1.2)*.

CRSs require multiple mount settings for CRSs to normally start a container. Mount options are critical in starting containers, as they include mount points, parameters (e.g., auto mount option, root option, synchronize option) and accessibility. For instance, the dysfunctional behavior of the *pause* container [8] is due to the absence of three specific flags (`nosuid`, `nodev` and `noexec`) at the `/dev/shm` mount point. Inaccurate permission assignments can also lead to significant risks in CRSs, as many of these configurations are pre-loaded and not altered by the users. Therefore, these bugs can

have a significant impact on the performance and security of CRSs. For example, the default seccomp configuration in `containerd` did not block socket calls to `AF_VSOCK`, posing security concerns for the runtime.

5.2.2 3rd Party Components Configuration Error (B.2). The incorrect configurations of 3rd Party components are a major root cause of configuration errors, accounting for 42.22% of bugs. This category mainly concerns errors caused by improper plugin handling configurations and system-related configurations. CRI runtimes, in particular, rely on a large number of plugins to support their functionality, making them more susceptible to configuration errors in plugins compared to OCI runtimes (64.29% vs. 48.28%). For example, a strict *AppArmor* configuration can block reads of containerd traces, making it difficult for diagnostic facilities to collect crash or hang dumps [13]. In contrast, OCI runtimes, which have more functions for interacting with system-related services, are more vulnerable to errors in setting system-related configurations than CRI runtimes (51.72% vs. 35.71%). For instance, a misconfigured *libseccomp* configuration in `runc` could lead to a process with a broken *argc* check [43].

5.2.3 Build Configuration Error (B.3). During the building process of CRSs, incorrect configurations can affect the building result and further container functionality. This root cause accounts for 25.19% of all configuration errors. For example, misconfiguration of the Go modules in `containerd` can cause a crash when using the jq command [13] during container creation.

5.2.4 Runtime Shim Config Error (B.4). In containerized runtime systems, there is a runtime shim that acts as an interface between container managers (e.g., `containerd`) and runtimes (e.g., `runc`). Incorrect configuration of the shim can lead to discrepancy issues between CRI and OCI runtimes. This type of configuration error is responsible for 10.37% of all configuration errors. An example is that the incorrect configuration of the `runhcs` shim caused an error in `containerd` for supporting the Windows hypervisor isolation.

6 In-Depth Analysis on CRS Bugs

Building on the taxonomy of bug symptoms and root causes in CRSs, this section delves deeper into the common types of bugs (Section 6.1), compares bug characteristics between CRS and other software (Section 6.2), and explores the relationship between symptoms and root causes (Section 6.3).

6.1 CRS-Specific Bug Characteristics

We conduct a detailed analysis of bug characteristics within the specific context of CRSs, focusing on four main types: performance bugs, functional bugs, configuration bugs, and security issues.

Performance Bugs are characterized by the program operating correctly but exhibiting sub-optimal performance. Our findings indicate that a majority of performance bugs (88.24%) in CRS are due to coding errors, while the remaining are mainly caused by the improper configuration and dependencies on software in other layers. This leads to issues such as high disk resource occupancy, slow execution, and excessive memory usage, which are closely related to the unique features of CRS, such as the parallel use of multiple containers and complex resource management. The performance

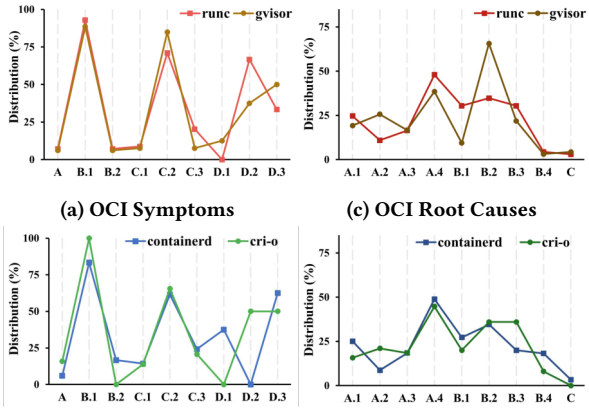


Figure 4: CRS Bug Taxonomy Distribution

issues can also cause serious consequences, high computational resource demands are often exploited to create Denial of Service (DoS) threats. For example, Microsoft Azure cloud platform experienced intermittent outages of more than 11 hours, disrupting an untold number of customer websites along with Microsoft Office 365, Xbox Live and other services across many countries [67]. More critically, memory-related performance issues can lead to security concerns, including privilege escalation and container escape scenarios.

Given the diverse root causes and the various consequences, there is an urgent need to design effective methods for detecting CRS performance errors, considering both performance and security impacts in cloud systems.

Functional Bugs identified in CRSs include three main types: *Logging Errors*, *Incorrect Execution Output*, and *Authorization Error*. Logging bugs and authorization bugs are particularly relevant to CRS features, as logging and authorization are core functionalities of cloud systems. Incorrect execution output, while a general issue, manifests uniquely in the context of CRS, such as incorrect container behaviors, improper execution of container isolation-related components, and flawed container inheritance.

Given the widespread use of CRS in cloud environments, it is noteworthy that functional bugs in CRS have a higher relevance to security issues. These bugs can be easily exploited by attackers, leading to concerns such as privacy issues, unauthorized privilege escalation, and isolation bypass. The primary challenge in testing the functionality of CRS is its complexity, which involves various functions and interactions with multiple software layers.

Configuration Bugs are a distinct category of CRS bugs due to the intricacies of cloud configuration. As shown in Fig. 4, there is a significant portion of bugs (31.47%) that are caused by configuration errors. This prevalence is attributed to the CRS architecture, which heavily depends on configurations to manage the operation of diverse components, including system and third-party components. CRI plugins require specific configurations for operation, the OCI runtime relies on `config.json` to initiate containers, and different images need configurations to specify dependencies. Incorrect configurations in CRSs can also lead to severe security issues.

The key challenges in detecting CRS configuration bugs lie in the myriad configurations across different components and their inter-dependencies. For example, CRS configurations span both

management-level settings (e.g., plugin configurations) and system-level settings (e.g., system mounts). While some configurations adhere to explicit specifications (e.g., the OCI runtime specification [47]), many others are customized (e.g., relying on add-on plugins). Therefore, developing methods to better extract specifications and model their impact and dependencies should be considered.

Security Issues in CRSs primarily encompass DoS attacks, privilege escalation or escape, memory leaks, and excessive resource consumption, as highlighted in Fig. 2 and 3. As discussed earlier, CRS bugs (such as performance, functional, or configuration bugs) are more likely to lead to security issues due to the multiple attack surfaces inherent in cloud architecture. For instance, our analysis reveals that certain bug types (Root Cause A.4.1, B.1.2, and B.2.2) are classified as vulnerabilities with relatively higher threat levels (CVSS > 7.0), designated as High Severity.

Finding 7: Complex configurations during development and usage phases of CRSs make *configuration errors* a significant cause of bugs (31.47%). These errors often involve mount options, permission configurations, plugin configurations, system call configurations, and shim configurations.

6.2 Comparison with Other Software

We compare the bug types of CRSs with other software, including the common software, the relevant distributed software and the similar CRS software.

Common Software. Compared to common software, such as autonomous driving systems, deep learning libraries, protocols and compilers, CRS exhibits specific bug symptoms and root causes attributable to its *distributed nature* and *containerization features*. Specifically, we can identify characterized CRS bug categories, including the symptoms of *B.1.2*, *B.1.3*, *C.2.2*, *C.2.3*, *C.3.1*, *C.3.2* and root causes of *A.3.2*, *A.4.2*, *B.1.1*, *B.1.2*, *B.2.1*, *B.2.2*, *B.4.* bugs.

Relevant Software. Except for general software, CRSs share similarities in application scenarios and architecture with Distributed Systems (DS) and Operating Systems (OS). We observe that they share many commonalities in bug types [4, 33, 40, 51, 53, 66], such as performance bugs, configuration bugs and functional bugs due to their distributed nature. However, it is important to note that while these bug types may appear similar at a high level, their context specifications differ significantly. As the middle layer to handle containers, CRSs introduce unique bugs related to container isolation and orchestration. While these bugs can be broadly classified as functional bugs, their specific nature and implications are distinct to the CRSs. For instance, *Runtime Shim Config Error* arises from the distinct lifecycle and architectural design of CRSs.

In contrast, Operating Systems, serving as the interface between hardware and software, encounter bugs related to hardware resource management, interaction, and system security. Distributed Systems, on the other hand, face challenges with maintaining consistency, availability, and partition tolerance. As a result, traditional static analysis or testing methods used in OS and DS may not be effective in detecting CRS-specific bugs, highlighting the need for the bug understanding and tailored approaches in this domain.

Other CRSs. Additionally, we compare the distribution of symptoms and root causes between two sets of software within the same layer: `containerd` and `cri-o` for the CRI layer, and `runc` and `gvisor` for the OCI layer. We calculate the average distribution differences for each pair of software regarding symptoms and root causes. As illustrated in Fig. 4, the average difference in symptom distribution is 12.58% for the OCI layer and 12.91% for the CRI layer. In terms of root causes, the average difference is 13.35% for the OCI layer and 13.80% for the CRI layer.

While these results indicate consistent similarities, we also observe notable differences in the distribution of symptoms and root causes between the two software implementations within each layer. For example, we observe that `runc` does not exhibit High Disk Resource Occupancy (D.1). This absence can be attributed to `runc`'s direct interaction with the host kernel when handling disk call functions, theoretically reducing the likelihood of encountering bugs related to unreasonable disk resource usage. Similarly, there is a significant difference between `gvisor` and `runc` regarding the root cause 3rd Party Component Configuration Error (B.2). This discrepancy is due to their varying dependencies on third-party libraries. Specifically, `runc` typically utilizes the host kernel's functionalities directly, resulting in fewer dependencies on external libraries. These differences suggest that, even for software within the same layer, there can be distinct characteristics and bug distributions due to different implementations.

6.3 Symptoms and Root Causes

During the analysis, we also observe clear relationships between symptoms and root causes. Specifically, we found that the majority (84.85%) of Building Failures (A) are caused by Configuration Errors (B.3), highlighting the significant impact of configurations on successful building processes. For common bugs like Crash (B.1), Logging Error (C.1), and Incorrect Execution Output (C.2), we found that they can be attributed to various root causes, including Coding Errors (A.1, A.2, A.3, and A.4) and Configuration Errors (B.1, B.2, B.3, and B.4). Incorrect Function Implementation (A.4) accounts for the main reasons for crashes and incorrect execution output, constituting 30.0% and 33.89%, respectively.

Authorization Error (C.3) is primarily caused by Incorrect Function Implementation (A.4), Improper Authorization Configuration (B.1), and 3rd Party Component Configuration Error (B.2), with respective percentages of 30.04%, 38.30%, and 12.77%. Excessive Execution Time (D.2) is mainly attributed to Coding Errors (A), accounting for 95.24%. Memory Error (D.3) is primarily caused by Insecure Runtime Implementation (A.3), Incorrect Function Implementation (A.4), and 3rd Party Component Configuration Error (B.2), with respective percentages of 53.84%, 23.08%, and 15.38%. This indicates that runtime implementation is likely to contribute to memory errors. Our further analysis reveals that Unsafe API Usage (A.3.1) and Inappropriate Lifecycle Organization (A.3.2) account for 64.29% and 35.71% of memory errors, respectively.

7 RQ3: Study of CRS Tests

In general, the testing process involves running the *target program* with given *test cases* and using *test oracles* to determine if any bugs occur. Therefore, our study focuses on three main questions: ❶ whether the target functions with bugs can be tested, ❷ whether the test cases can trigger the buggy code when the functions are

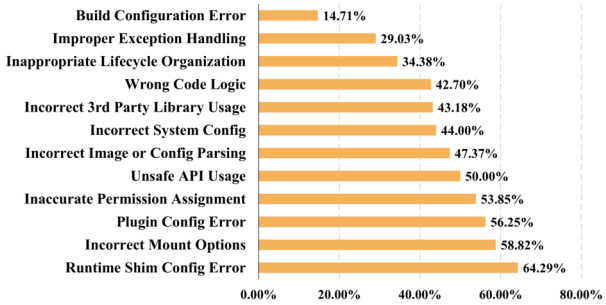


Figure 5: Proportion of Bugs Lacking Test Drivers in Different Root Causes

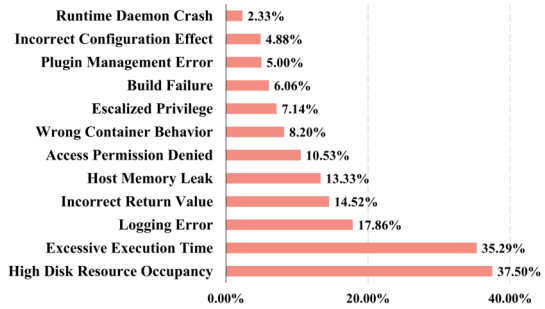


Figure 6: Proportion of Bugs without Oracles in Different Symptoms

executed, and ❸ whether the test oracles can identify the bugs when the buggy code is covered by the test cases.

Lack of Test Drivers. To evaluate whether the existing test drivers can execute the functions with bugs, we collect all functions involved in buggy-related commits (called buggy functions) as well as the functions that have been executed by the collected tests (called executed functions). Then we compare the buggy functions and executed functions, and check whether the buggy functions can be covered. We find that 41.96% of the bugs are not detected since the corresponding buggy functions are not covered.

To detect the presence of a bug, one necessary condition is that the code related to the root cause should be executed. Therefore, we further study the relationship between the leaf categories of root causes and the target function coverage to check the buggy functions for which type of root causes are not covered.

From Fig. 5, we find that buggy functions with *Runtime Shim Config Error* errors are less likely to be covered (64.29%), indicating the lack of consideration of testing these container `shim` related modules. In addition, most uncovered buggy functions are related to configuration errors, such as *Runtime Shim Config Error, Plugin Config Error, Incorrect Image and Config Parsing*, and *Incorrect System Config*, where more than 40% are not covered.

Lack of Test Cases. We further analyze the bugs which lack proper test cases. For these bugs, the tests will execute the buggy function with proper oracle, while lacking proper test inputs for triggering the bug. We find that 27.97% of the bugs failed to be detected by existing tests due to the lack of proper inputs. Particularly, we discover that unit tests of CRSs often have simple inputs that may not be enough to trigger the buggy code, indicating the need for automated testing algorithms to generate effective test cases.

Lack of Oracles. Our analysis reveals that 9.32% of the bugs are undetected due to the lack of appropriate oracles even though they are executed during tests. This indicates that existing tests could reach the faulty code, but lack the oracles to identify the bugs. Since oracles are directly related to symptoms, we study the relationship between the leaf categories of symptoms and undetected bugs that are caused by the lack of oracles. This can help understand which symptoms are (or not) considered by the existing tests.

In Fig. 6, we find that 37.50% of high disk occupancy bugs and 35.29% of the excessive execution time bugs lack proper oracles. This suggests a deficiency in storage performance and timeout checks within testing designs, likely due to the challenge of determining the ground truth regarding storage usage and time constraints. It is surprising that the percentage of other bugs lacking oracles is low, particularly for logic-related bugs. Our in-depth analysis reveals two main reasons: ❶ many buggy functions are not covered by existing tests and are therefore not included in the statistics; ❷ most of the tests are designed by developers, and the oracles (e.g., for logic errors) have been manually assigned.

> **Finding 8:** Common testing methods, such as unit tests, integration tests, and fuzz tests, are insufficient for detecting the collected bugs, with only a small portion (21.21%) being detectable by existing tests. The reasons for the missing bug detection include the lack of test drivers (41.96%), test oracles (9.32%), and test inputs (27.97%).

Example. The code above illustrates CVE-2023-27561 [20], which is a significant regression vulnerability introduced during an attempt to fix CVE-2021-30465 [18] in `runc`. This fix inadvertently nullified a previous patch for CVE-2019-19921 [16]. Specifically, `runc` before version 1.0.0 had container escape bug by allowing mounting directories through symbolic links under `procfs` race conditions. This was patched with a check for the mount of sensitive locations such as the root directory (lines 12-15). The subsequent patch for CVE-2021-30465 [18] employs the `securejoin` library to verify mount targets to mitigate `symlink`-based race vulnerabilities (lines 5-7). However, this patch inadvertently reintroduced CVE-2019-19921 [16] because the mount point `dest` would be set before the checking of sensitive locations. While these bugs can be categorized as functional or security bugs, they highlight the relevance to the unique features of CRS, particularly regarding the *isolation* properties of containerized environments and the potential for race conditions in *high-load, multi-tenant* cloud environments.

```
1  func mountToRootfs(m *configs.Mount, c *mountConfig)
       error {
2    rootfs := c.root
3    dest := m.Destination
4    if !strings.HasPrefix(dest, rootfs) {
5      dest = filepath.Join(rootfs, dest)
6    dest, err := securejoin.SecureJoin(rootfs, m.
         Destination) // Patch for CVE-2021-30465
7    if err != nil {
8      return err
9    }
10   switch m.Device {
11       case "proc", "sysfs":  // Patch for CVE-2019-19921
12       ...
```

Regarding bug detection, we find that this regression issue (CVE-2023-27561) was not discovered until 2023. Despite the existence of

many testing methods for `runc`, such as unit testing and fuzzing suites, they are ineffective in detecting such bugs due to: ❶ the complexity of designing a test driver. In this bug, it requires the simulation of high concurrency tasks and specific race conditions (e.g., via a mount/unmount script), and ❷ the lack of oracles to accurately capture the bugs, as they do not cause crashes or other explicit symptoms. This example demonstrates the challenges of detecting CRS-specific bugs and the need for the development of new methods in the context of cloud environments.

Discussions on Bug-Specific Detection Methods. Although we find that the existing built-in test suites face challenges in detecting bugs in CRSs, there can be other testing methods devised for specific bug types that might aid in CRS bug detection. We discuss the relevant tailored approaches as follows:

- *Performance Bug Detection.* Torpedo [38], based on framework syzkaller [26], employs resource usage-based oracles for detecting out-of-band workload issues associated with high disk or CPU occupancy. MANTA [64] targets the detection and validation of missing-account bugs within systems, which could be adapted to detect memory exhaustion issues in CRSs.
- *Functional and Security Issue Detection.* Xiao et.al [59] designed layer-specific strategies to exploit operation forwarding attacks in sandbox containers. Yang et.al [61] proposed attacking paths-specific methods for detecting excessive permissions from third parties in cloud orchestration software. However, these approaches are limited in scope and may not generalize for other CRS bugs.
- *Configuration Bug Detection.* Some research works have explored techniques in detecting configuration bugs. For instance, PCheck [60] generates configuration checking code to detect latent configuration errors. Ctest [55] integrates software testing with production configurations and code to identify failure-inducing configuration changes. However, these methods may face challenges when testing CRSs due to the complex and multi-resource-based configurations (including containers, runtimes, and plugins).

8 Discussion

8.1 Implications and Lessons Learned

In this section, we summarize the implications of this work. Our analysis aims to provide guidance tailored to the different features of CRS bugs discussed in Section 6.1, for stakeholders of the container cloud ecosystem, including users, developers, and researchers.

For Users. We provide the following suggestions:

- *Monitoring performance and functionality*. Users should periodically check whether the performance and functionality of CRS are as expected (see Section 4.3) to prevent more severe consequences like security-related abuse of privileges. For examples, users should meticulously monitor for performance issues, including excessive disk or memory usage, which can stem from hardware, operating systems, configurations, or general implementation (see D.1, D.3 in Fig. 2). Our study indicates that such monitoring should be performed especially carefully on certain platforms such as s390x or Windows.
- *Careful configuration and confirmation.* In CRSs, numerous configurations require input from users, and improper configuration can easily lead to functional or security issues. For example, specifying improper system capabilities when launching containers can be used by attackers for privilege escalation, even leading to container escape. To mitigate such risks, users should adhere to CRS configuration specifications and conduct thorough checks before deployment, particularly in security-critical scenarios and on platforms that may lack comprehensive support. Minimizing configurations related to authorization to reduce potential abuse should also be seriously considered by users. Additionally, many comments in the manual or README may become outdated, which users should carefully confirm.

For Developers. We offer the following suggestions for developers:

- *Confirming functional correctness.* CRSs involve numerous plugins and API calls (e.g., system calls). In addition to common coding best practices, developers must ensure correct configurations, API compatibility, and proper API usage (see A.2, B.2 in Fig. 3). Particular attention should be given to configurations with default parameters to check for incorrect specification constraints or improper authorization allocation.
- *Focusing on cloud security.* Developers should focus more on improving CRS security. Our study indicates that developers should rigorously assess unsafe API usage, container lifecycle management consistency, correct permission assignments, and authorization configuration bugs in CRSs (refer to A.3.1, A.3.2, B.1.1, B.1.2, B.4 in Fig. 3). Developers might also consider tracking platform variances, abnormal inputs for CRSs, and verifying the consistency of CRS functionalities across different versions.
- *Adding high-quality tests and oracles.* Our study shows that the existing built-in test suites are limited. It is suggested for existing projects to incorporate more tests and benchmarks, particularly focusing on the different categories of bugs summarized in Fig.2 and Fig.3. Furthermore, CRSs should extend their support to the greatest extent possible, copping with rigorous compatibility tests, including correct and consistent container behaviors resulting from different commands and configurations (see C.2.2, C.2.3 in Fig. 2). Developers should enhance unit tests by augmenting the number of test harnesses and corresponding oracles.

For Researchers. Based on our analysis of the unique features of CRSs, we have identified potential future research directions:

- *Automated Test Driver and Test Oracle Generation.* Existing built-in tests are inefficient and insufficient to address the complexity and unique features of CRSs. Our findings highlight the need for methods capable of generating diverse test drivers, effective test cases, and various types of oracles. Test driver generation techniques should cover different aspects, particularly configurations (e.g., plugin configuration and mounting configuration), image parsing, and incorrect code logic. The symptoms and root causes summarized in this paper (see Figures 2, 3, 5, and 6) can benefit researchers in designing test oracles and test drivers. A potential direction is the LLM-based methods due to the capability of LLMs in understanding code semantics and business logic, which can help generate corresponding drivers [68] and oracles.
- *Designing Bug-Specific Methods.* Our study shows that designing a general end-to-end testing approach (e.g., integration testing and fuzzing) for detecting all kinds of bugs in CRSs is challenging. We suggest researchers divide the tasks and design detection methods specific to specific types of bugs. For example, for performance bugs, researchers can focus on collecting appropriate

resource information (e.g., CPU and memory) and designing metrics for systematic monitoring. For configuration bugs, building a causal model between configurations and their impact on CRSs could be more effective than simple rule-based matching.

- *Detecting Regression Bugs.* Our analysis find that frequent updates in CRSs can introduce regression bugs, which can cause security issues. For example, CVE-2023-27561 [20] (see Section 7) is caused by the break of the previously vulnerability patch during software updates. The key challenge is evaluating the impact of code changes, especially for those related to security patches. Researchers can develop effective regression tests by considering different file or method granulates. Patch fuzzing is a potential direction that leverages previous PoCs (e.g., from CVEs) as initial seeds to test patches effectively.
- *LLM-based Scanning, Diagnosing and Bug Detection.* We found that Large Language Models have been applied to the cloud maintenance. For example, k8sgpt [29] is developed to assist scanning, diagnosing, and triaging issues for Kubernetes configurations. We foresee leveraging LLMs to detect bugs and analyze vulnerabilities in CRSs as a promising direction. LLM agents can be developed to test CRSs, debug failed tests, triage failures, and even repair identified bugs. Therefore, how to build CRS-specific intelligent LLM (e.g., CRS GPT) is an interesting direction.
- *Log-based Monitoring.* Considering the complexity of CRSs, testing-based methods may not detect all bugs due to the lack of test drivers, oracles and tests. As a complement, online error detection (e.g., log analysis and anomaly detection) could be used to detect some errors. For example, based on the log information, we can detect incorrect runtime behaviors (see A.3.1, A.3.2 in Fig. 3) and some configuration errors (see B in Fig. 3). The main challenges include the large volume of CRS logs and specific log representations. Designing effective log collection and embedding construction methods for attack detection could be important directions for future research.

8.2 Threats to Validity

Our dataset selection (including the projects, the bugs and tests) is a potential threat. We mainly selected `runc`, `gvisor`, `containerd` and `cri-o` since they are the most commonly used CRSs in industry. The selection of tests could pose another validity threat. There might exist test cases that are used internally by development teams or generated by bug-specific tools. Since these tests are either not publicly available or challenging to adapt, we have focused on various tests from the official codebases of CRS projects.

Time interval is a threat to the bug distribution. We limit our commit dataset to last two years for covering the latest bugs. We will keep updating the results by adding more analysis of new bugs in the future. Another potential threat to our study is the subjectivity involved in manual analysis. To address this, we take measures such as discussing and cross-checking the labeling results among the authors involved in the study. Additionally, we follow a systematic and rigorous process for bug classification, and any discrepancies are resolved through discussion and consensus. To validate the representatives of our findings, we sample commits from the entire dataset and compare the taxonomy results with our researched commit data.

9 Related Work

Although there are some works studying errors in operating systems [4, 50, 58], distributed systems [24, 66], and DevOps systems [27, 41], these studies primarily focus on domain specific characteristics of the study target instead of container cloud systems. Moreover, these studies miss many container-characterized bug categories e.g., *Escalized Privilege* etc.

Container Cloud Security. Given the high security impact of container cloud systems, much research has been devoted to studying their security challenges. For example, Yang et al.[63] discussed the current security challenges in container cloud systems, and suggested multiple solutions for future development. Nordell [42] proposed a systematic evaluation of CVEs and mitigation strategies for the Kubernetes stack. Abbas et al.[1] designed PACED, a real-time uses privileged monitor system to detects container escape attacks. Li et al.[34] examined path misresolution vulnerabilities in container systems and proposed kernel-based filesystem isolation method to enhance container access control. Yang et al.[62] proposed a novel abstract resource attack technique that can exhaust host memory without breaking the container limit. McDonough et al. [38] developed Torpedo to fuzz container cloud services using the popular Linux kernel fuzzing framework syzkaller[26]. Torpedo is specifically designed to detect out-of-band workloads in multi-tenant container cloud services. While these studies focus on specific domains of container cloud security, none of them provides a comprehensive understanding of various bugs in CRSs.

Container Runtime Performance. The performance of CRSs has also garnered much attention. Specifically, Avino et al.[2] tested the performance of `Docker` under heavy computing workloads. Mavridis et al.[37] evaluated the performance of `Docker` on virtual machines such as `KVM` and `HyperV`. Espe et al.[23] conducted a performance evaluation of CRSs using the self-developed tool TouchStone to test the CPU and memory performance. Wang et al.[57] researched the performance and isolation functionality of the `runc`, `gVisor`, and `Kata` CRSs. Their assessment of performance was based on several metrics, including the number of supported system calls, startup time, and isolation functionality.

10 Conclusion

In this work, we conduct the first comprehensive study on bugs of Container Runtime Systems by manually inspecting 429 related bugs. We develop taxonomies for the symptoms and root causes of these bugs, analyze their distributions, and evaluate the effectiveness of existing test codes. The study reveals that CRSs have characteristic bugs and only around 20% bugs are detectable through built-in tests. These findings offer practical insights on improving the quality of CRSs. Based on our research, we also provide recommendations for the use, development, and testing of CRSs.

Acknowledgment

This research is partially supported by the Lee Kong Chian Fellowship, the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] Mashal Abbas, Shahpar Khan, Abdul Monum, Fareed Zaffar, Rashid Tahir, David Eyers, Hassaan Irshad, Ashish Gehani, Vinod Yegneswaran, and Thomas Pasquier. 2022. PACED: Provenance-based Automated Container Escape Detection. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 261–272.

[2] Giuseppe Avino, Marco Malinverno, Francesco Malandrino, Claudio Casetti, and Carla-Fabiana Chiasserini. 2017. Characterizing docker overhead in mobile edge computing scenarios. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*. 30–35.

[3] Oliver Chang, Jonathan Metzman, Max Moroz, Martin Barbella, and Abhishek Arya. 2016. OSS-Fuzz: Continuous Fuzzing for Open Source Software. *URL: https://github.com/google/ossfuzz* (2016).

[4] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An empirical study of operating systems errors. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*. 73–88.

[5] CNCF. 2023. 2023 CNCF Cloud Native Survey. https://www.cncf.io/reports/cncf-annual-survey-2023/.

[6] CNCF. 2023. CNCF Fuzzing. https://github.com/cncf/cncf-fuzzing.

[7] Containerd. 2023. Containerd. https://github.com/containerd/containerd".

[8] Containerd. 2023. incofunc. https://github.com/containerd/containerd/commit/42a386c8164bef16d59590c61ab00806f854d8fd.

[9] Containerd. 2023. misapi. https://github.com/containerd/containerd/security/advisories/GHSA-mvff-h3cj-wj9c.

[10] Containerd. 2023. plugincrash. https://github.com/containerd/containerd/commit/9f9ebbd99103bb60b0b0b45be8d9520b8c047b44.

[11] Containerd. 2023. preexit. https://github.com/containerd/containerd/commit/31a710c492ecb225ecbc4d43fddc2c36ec962f16.

[12] Containerd. 2023. storage. https://github.com/containerd/containerd/commit/0c63c42f8183d13c2c106c01f5bb3560d39b3295.

[13] Containerd. 2023. thirdconfig. https://github.com/containerd/containerd/commit/8d868dadb746aabfd3583d834510109afe9b1919.

[14] The MITRE Corporation. 2023. The CVE List. https://cve.mitre.org/.

[15] cri o. 2023. cri-o. https://cri-o.io/.

[16] National Vulnerability Database. 2019. CVE-2019-19921. https://nvd.nist.gov/vuln/detail/CVE-2019-19921.

[17] National Vulnerability Database. 2019. CVE-2019-5736. https://nvd.nist.gov/vuln/detail/CVE-2019-5736.

[18] National Vulnerability Database. 2021. CVE-2021-30465. https://nvd.nist.gov/vuln/detail/CVE-2021-30465.

[19] National Vulnerability Database. 2021. CVE-2021-43816. https://nvd.nist.gov/vuln/detail/cve-2021-43816.

[20] National Vulnerability Database. 2023. CVE-2023-27561. https://nvd.nist.gov/vuln/detail/CVE-2023-27561.

[21] National Vulnerability Database. 2024. CVE-2024-21626. https://nvd.nist.gov/vuln/detail/CVE-2024-21626.

[22] Docker. 2024. Docker, Inc. https://www.docker.com/.

[23] Lennart Espe, Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. 2020. Performance Evaluation of Container Runtimes. In *CLOSER*. 273–281.

[24] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An empirical study on crash recovery bugs in large-scale distributed systems. In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 539–550.

[25] Google. 2023. Google Cloud Documentation. https://cloud.google.com/run/docs/container-contract.

[26] Google. 2023. Syzkaller. https://github.com/google/syzkaller.

[27] Jordan Henkel, Christian Bird, Shuvendu K Lahiri, and Thomas Reps. 2020. Learning from, understanding, and supporting devops artifacts for docker. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 38–49.

[28] Bugs in Pods. 2024. Understanding Bugs in Container Runtime Systems. https://sites.google.com/view/understand-bugs-in-crs.

[29] k8sgpt ai. 2023. k8sgpt. https://github.com/k8sgpt-ai/k8sgpt.

[30] Kubernetes. 2022. Kubernetes. https://kubernetes.io/.

[31] Kubernetes. 2023. Container Runtime Interface. https://kubernetes.io/docs/concepts/architecture/cri/#api.

[32] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.

[33] Jiaxin Li, Yiming Zhang, Shan Lu, Haryadi S Gunawi, Xiaohui Gu, Feng Huang, and Dongsheng Li. 2023. Performance Bug Analysis and Detection for Distributed Storage and Computing Systems. *ACM Transactions on Storage* 19, 3 (2023), 1–33.

[34] Zhi Li, Weijie Liu, XiaoFeng Wang, Bin Yuan, Hongliang Tian, Hai Jin, and Shoumeng Yan. 2023. Lost along the Way: Understanding and Mitigating Path-Misresolution Threats to Container Isolation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 3063–3077.

[35] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. 2018. A measurement study on linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 418–429.

[36] Linux. 2023. Systemd-run. https://www.linux.org/docs/man1/systemd-run.html.

[37] Ilias Mavridis and Helen Karatza. 2019. Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing. *Future Generation Computer Systems* 94 (2019), 674–696.

[38] Kenton McDonough, Xing Gao, Shuai Wang, and Haining Wang. 2022. Torpedo: A fuzzing framework for discovering adversarial container workloads. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 402–414.

[39] Microsoft. 2023. Windows Host Compute Service Shim. https://github.com/microsoft/hcsshim.

[40] Dongliang Mu, Yuhang Wu, Yueqi Chen, Zhenpeng Lin, Chensheng Yu, Xinyu Xing, and Gang Wang. 2022. An In-depth Analysis of Duplicated Linux Kernel Bug Reports.. In *NDSS*.

[41] Ali Rezaei Nasab, Mojtaba Shahin, Seyed Ali Hoseyni Raviz, Peng Liang, Amir Mashmool, and Valentina Lenarduzzi. 2023. An empirical study of security practices for microservices systems. *Journal of Systems and Software* 198 (2023), 111563.

[42] Fred Nordell. 2022. A Systematic evaluation of CVEs and mitigation strategies for a Kubernetes stack. *Master's Thesis* (2022).

[43] Opencontainers. 2023. 3rdpartyconfigrunc. https://github.com/opencontainers/runc/commit/6d2067a4bf1ae0a9ae1c570a0d1f2004e80eb229.

[44] Opencontainers. 2023. apiincomp. https://github.com/opencontainers/runc/commit/29a56b5206a3fc7b2f5fe4b43c09c8ba09b25495.

[45] Opencontainers. 2023. crash. https://github.com/opencontainers/runc/commit/462e719cae227a990ed793241062a8d2d6145332.

[46] Opencontainers. 2023. memory. https://github.com/opencontainers/runc/commit/77cae9addc0c7c9ef52513b4e46b2e6485e4e469.

[47] Opencontainers. 2023. Open Container Initiative Runtime Specification. https://github.com/opencontainers/runtime-spec.

[48] Opencontainers. 2023. Runc. https://github.com/opencontainers/runc".

[49] Opencontainers. 2023. time. https://github.com/opencontainers/runc/commit/18e286261ec6e94bb9ccaee2bbc91fc63bf08f89.

[50] Fangyun Qin, Zheng Zheng, Xiaodan Li, Yu Qiao, and Kishor S Trivedi. 2017. An empirical investigation of fault triggers in android operating system. In *2017 IEEE 22Nd pacific rim international symposium on dependable computing (PRDC)*. IEEE, 135–144.

[51] Lili Quan, Qianyu Guo, Xiaofei Xie, Sen Chen, Xiaohong Li, and Yang Liu. 2022. Towards Understanding the Faults of JavaScript-Based Deep Learning Systems. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[52] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering* 25, 4 (1999), 557–572.

[53] Xiuhan Shi, Xiaofei Xie, Yi Li, Yao Zhang, Sen Chen, and Xiaohong Li. 2022. Large-Scale Analysis of Non-Termination Bugs in Real-World OSS Projects. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*. ACM, 256–268.

[54] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An empirical study of deep learning models for vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2237–2248.

[55] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing configuration changes in context to prevent production failures. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 735–751.

[56] Kevin Wang and Yiqi Sun. 2022. CVE-2022-0492. https://access.redhat.com/security/cve/cve-2022-0492.

[57] Xingyu Wang, Junzhao Du, and Hui Liu. 2022. Performance and isolation analysis of RunC, gVisor and Kata Containers runtimes. *Cluster Computing* 25, 2 (2022), 1497–1513.

[58] Guanping Xiao, Zheng Zheng, Beibei Yin, Kishor S Trivedi, Xiaoting Du, and Kai-Yuan Cai. 2019. An empirical study of fault triggers in the Linux operating system: An evolutionary perspective. *IEEE Transactions on Reliability* 68, 4 (2019), 1356–1383.

[59] Jietao Xiao, Nanzi Yang, Wenbo Shen, Jinku Li, Xin Guo, Zhiqiang Dong, Fei Xie, and Jianfeng Ma. 2023. Attacks are Forwarded: Breaking the Isolation of {MicroVM-based} Containers Through Operation Forwarding. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7517–7534.

[60] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 619–634.

[61] Nanzi Yang, Wenbo Shen, Jinku Li, Xunqi Liu, Xin Guo, and Jianfeng Ma. 2023. Take over the whole cluster: Attacking kubernetes via excessive permissions of third-party applications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 3048–3062.

[62] Nanzi Yang, Wenbo Shen, Jinku Li, Yutian Yang, Kangjie Lu, Jietao Xiao, Tianyu Zhou, Chenggang Qin, Wang Yu, Jianfeng Ma, et al. 2021. Demons in the shared

kernel: Abstract resource attacks against os-level virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 764–778.

[63] Yutian Yang, Wenbo Shen, Bonan Ruan, Wenmao Liu, and Kui Ren. 2021. Security challenges in the container cloud. In *2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 137–145.

[64] Yutian Yang, Wenbo Shen, Xun Xie, Kangjie Lu, Mingsen Wang, Tianyu Zhou, Chenggang Qin, Wang Yu, and Kui Ren. 2022. Making Memory Account Accountable: Analyzing and Detecting Memory Missing-account bugs for Container Platforms. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 869–880.

[65] Ethan G Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2019. The true cost of containing: A {gVisor} case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.

[66] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed {Data-Intensive} Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 249–265.

[67] Jason Zander. 2014. Update on Azure Storage Service Interruption. https://azure.microsoft.com/zh-cn/blog/update-on-azure-storage-service-interruption/.

[68] Cen Zhang, Yaowen Zheng, Mingqiang Bai, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. 2024. How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation. *arXiv* (2024). arXiv:2307.12469

Received 2024-04-12; accepted 2024-07-03