Noname manuscript No.
(will be inserted by the editor)

An Empirical Study of Vulnerabilities in Edge Frameworks to Support Security Testing Improvement

Jahanzaib Malik · Fabrizio Pastore

Received: date / Accepted: date

Abstract Edge computing is a distributed computing paradigm aiming at ensuring low latency in modern data intensive applications (e.g., video streaming and IoT). It consists of deploying computation and storage nodes close to the end-users. Unfortunately, being distributed and close to end-users, Edge systems have a wider attack surface (e.g., they may be physically reachable) and are more complex to update than other types of systems (e.g., Cloud systems) thus requiring thorough security testing activities, possibly tailored to be cost-effective.

To support the development of effective and automated Edge security testing solutions, we conducted an empirical study of vulnerabilities affecting Edge frameworks. The study is driven by eight research questions that aim to determine what test triggers, test harnesses, test oracles, and input types should be considered when defining new security testing approaches dedicated to Edge systems.

We have inspected 147 vulnerabilities of four popular Edge frameworks. Our findings indicate that vulnerabilities slip through the testing process because of the complexity of the Edge features. Indeed, they can't be exhaustively tested in-house because of the large number of combinations of inputs, outputs, and interfaces to be tested. Since we observed that most of the vulnerabilities do not affect the system integrity and, further, only one action (e.g., requesting a URL) is sufficient to exploit a vulnerability we propose in-the-field testing as a possible solution to address the problem. Because of their prevalence, authentication and authorization issues are the ones requiring to be addressed with more urgency.

Keywords Security Testing · Edge Computing · Empirical Study

J. Malik, F. Pastore
SnT Centre, University of Luxembourg, Luxembourg
E-mail: jahanzaib.malik@uni.lu, fabrizio.pastore@uni.lu

1 Introduction

Business and private individuals are increasingly relying on data-intensive services provided by remote systems; examples include music streaming, video conferencing, E-gaming, cloud storage, and remote surveillance. Because of the real-time transmission of large amounts of data, latency is one of the main issues affecting the above-mentioned services. To minimize latency, the *Edge Computing* paradigm has been introduced [130]. It consists of distributed storage and computing resources close to the end-users with the objective of minimizing latency and ensuring real-time services.

When data is the main asset of a service, security is a major concern. Unfortunately, by moving data and computation closer to the end-user (e.g., TV boxes), service providers have less control on the infrastructure, which is often physically accessible, might be difficult to update (e.g., because updates take place overnight when the system is turned off), and might be installed on a large number of diverse hardware and OS layers whose configurations might be difficult to be tested extensively. Consequently, compared to services executed on traditional infrastructures (e.g., Cloud), services executed on Edge computing infrastructures may expose a wider set of attack surfaces (e.g., because physically accessible) and be more likely affected by vulnerabilities (e.g., because it is not possible to test all the configurations of the software, or because it is not possible to ensure that the underlying environment is up to date).

Because of the reasons above, infrastructure providers are looking for solutions to assess that Edge frameworks and applications are free from vulnerabilities. In this paper, we focus on software security testing, which, differently from other approaches (e.g., security analysis), provides evidence of the presence of vulnerabilities; for example, test failures show how an attacker can exploit a vulnerability.

As a starting point towards the definition of Edge security testing solutions we conduct an empirical study of the vulnerabilities affecting Edge frameworks. Our study partially relies on a recent study by Gazzola et al. [41]. The work of Gazzola et al., although focused on functional failures and not security aspects, has guided us towards the characterization of the vulnerable components (e.g., plugins), the type of failures being observed (e.g., signalled or silent), the complexity of the required testing procedures (i.e., how many actions should be performed to detect a vulnerability), and the reasons why vulnerabilities slip through the development process (e.g., because of the combinatorial explosion of the inputs to be tested). In addition, different from Gazzola et al., we characterized the preconditions (e.g., sub-nets should be set-up) and the inputs (e.g., sending crafted messages) required to exploit Edge vulnerabilities. Finally, similar to other vulnerability studies [82], we analyzed the distribution of CWE [88] identifiers (i.e., types of weaknesses leading to Edge vulnerabilities) reported in the Common Vulnerabilities and Exposures (CVE) database [87]. Also, we studied their severity, based on the CVSS entries of the National Vulnerability Database (NVD) [120].

In total, we defined eight research questions. We surveyed 263 bug reports concerning four Edge frameworks (Mainflux [79], K3OS [52], KubeEdge [59], and Zetta [142]). Among them, we identified 147 vulnerability reports. Our results show that the large number of combinations of configurations and inputs (i.e., combinatorial explosion)

is the main reasons for security vulnerabilities not being detected at testing time ($\mathbf{RQ}_1$). Vulnerabilities mostly affect the main Edge framework components (i.e., controllers), a minor presence is observed in network components and plugins, while other components (i.e., APIs, drivers, services, and resources) are less affected ($\mathbf{RQ}_2$). Generally, vulnerabilities can be observed when the software under test (SUT) is in a specific state or configuration ($\mathbf{RQ}_2$, $\mathbf{RQ}_4$), which clarifies why vulnerabilities are not detected at testing time because of combinatorial explosion. Security failures ($\mathbf{RQ}_3$) are silent (i.e., not detected by the SUT) and concern value failures (e.g., illegal data being returned), network (e.g., data erroneously routed), or actions (e.g., the software performs illegal operations on the environment). Once the SUT is in the vulnerable state, vulnerabilities can be exploited with a single action ($\mathbf{RQ}_{5A}$) that usually consists of providing specific data ($\mathbf{RQ}_{5B}$) to the SUT. The security property that is likely violated by Edge vulnerabilities is confidentiality ($\mathbf{RQ}_6$). Confidentiality issues are mainly due to developer mistakes concerning authentication mechanisms or information management errors ($\mathbf{RQ}_7$). Further, failures are observed because the SUT performs improper access control or improper control of resources over lifetime ($\mathbf{RQ}_7$). NVD data indicates that more than 50% of Edge vulnerabilities have a high severity and are easy to exploit, thus highlighting their criticality and the need for improved testing solutions ($\mathbf{RQ}_8$).

Based on the characteristics summarized above, to ensure timely discovery of vulnerabilities (e.g., before attackers), we suggest to automatically execute test cases directly in the field (e.g., on the deployed Edge system); such practice is known as field-based testing [15]. Indeed, automated testing might be executed, in the field, when configurations not tested in-house are observed; also, the detection of vulnerabilities might be simplified by the fact that only a single action is sufficient to exploit them. Further, testing might focus on confidentiality thus not requiring the identification of mechanisms to compensate for integrity problems caused by the testing process itself. All the data collected to perform our study are available online [80].

This manuscript proceeds as follows. Section 2 presents background information including a glossary. Section 3 describes the study design. Section 4 presents our results. Section 5 presents a discussion of threats to validity. Section 6 provides reflections on the research directions for Edge security testing, based on our results. Section 7 discusses related work. Section 8 concludes the manuscript.

2 Background

In this section we provide a brief overview of Edge technology, related studies, and a glossary.

2.1 Edge Computing

The Edge computing paradigm has been introduced to enable data transfer with extremely low latency for real-time services. Well known services relying on Edge computing include, for example, E-sports [1], live streams broadcasts [123, 134],

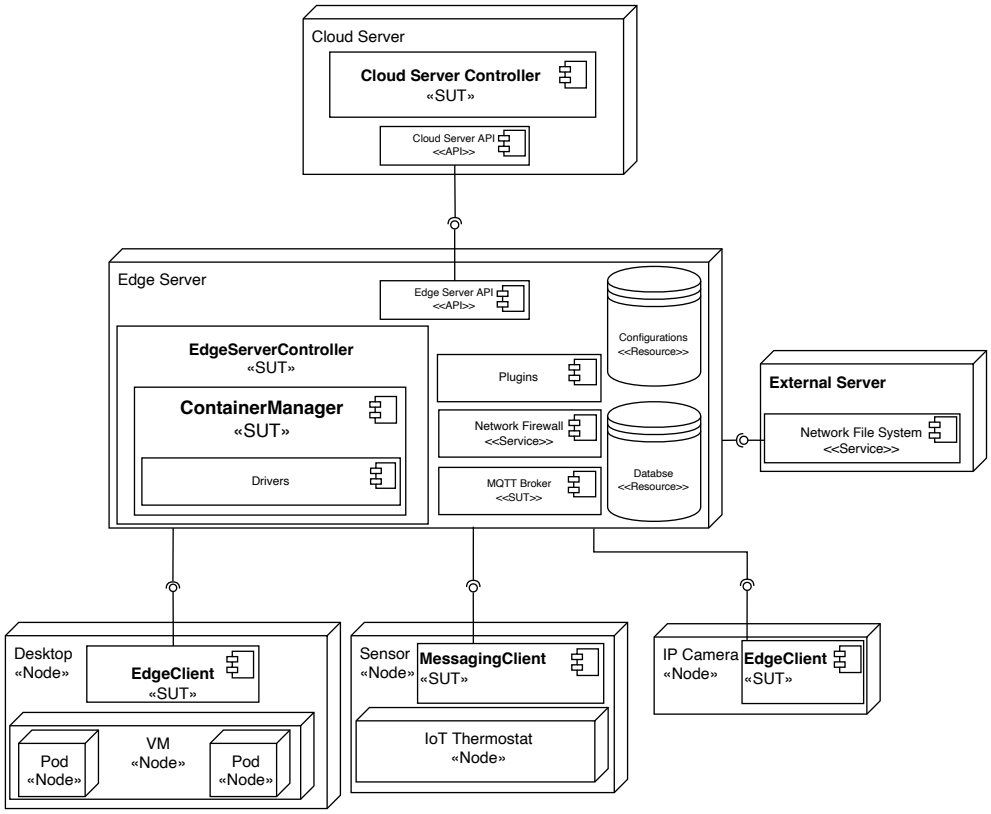


Fig. 1: UML deployment diagram capturing the architecture of Edge systems. Ball and socket notation is used to distinguish between the component providing a service (ball) and the component relying on the service (socket).

package tracking [40], and internet connectivity services for cruise lines [125] and aviation [124].

The development of services leveraging the Edge paradigm is supported by Edge frameworks; well known examples are KubeEdge [57], Yomo [137], K3os [52], and Mainflux [79]. In this paper, we rely on the term *Edge framework* to indicate a set of software components, including Web services and APIs, that are extended to provide a service relying on the Edge paradigm. Our definition is consistent with the definition of *framework* provided by IEEE: *partially completed software subsystem that can be extended by appropriately instantiating some specific plug-ins* [49]. Further, our definition of Edge framework recalls the definition provided by Fayad and Schmidt for *middleware integration frameworks*, which are used to *integrate distributed applications and components*; *middleware integration frameworks are designed to enhance the ability of software developers to modularize, reuse, and extend their software infrastructure to work seamlessly in a distributed environment* [36]. An

Edge framework integrates a broad range of technologies including Cloud services and virtualization environments; therefore, an Edge framework is often implemented as an integration of multiple frameworks developed by third parties. In this paper, we treat all the technologies cooperating with an Edge framework as one single framework. We call *Edge application* the software that implements the logic to provide a service to the end-user. We call *Edge system* what results from the integration of an *Edge framework*, one or more *Edge applications*, and external services that the Edge framework and applications may be configured to interact with.

Figure 1 provides a generic architecture of an Edge system. In Figure 1, the software components that constitute the Edge framework are annotated with the UML stereotype *SUT* (i.e., software under test). We use the term SUT to identify Edge frameworks' components because they are the target of our investigation.

The main architectural components in an Edge system are Cloud servers, Edge servers, and Nodes. *Cloud servers* provide centralized services (e.g., end-user authentication for a video streaming). *Edge servers* are deployed close to the end-user to minimize latency [11]; for example, they include caching mechanisms for the data provided by the Cloud server thus reducing latency. *Nodes*, instead, are deployed at the end-user's side; depending on the service provided through the Edge system, Nodes might be desktop computers, sensors, or IP camera. In Figure 1, Nodes are annotated with the stereotype *Node*. The Edge system may interact with external components providing specific services, for example a network file system. In Figure 1, we annotated external services with the stereotype *Service*.

The Cloud server executes a *Cloud server controller* component that interacts with the *Edge server controller* through the *Edge server API*. The Cloud server controller manages the Edge server instances (e.g., to provide monitoring and policy enforcement). Also, it provides and collect service data. Examples of provided data are on-demand video streaming and file streaming. Examples of collected data include information about devices (e.g., offline status of a surveillance camera) or end-user data (e.g., movies' rating or list of videos watched in a video streaming service).

The Edge server executes the *Edge server controller*, which has the responsibility of controlling access to resources, instantiate drivers, access plugins, manage resources, and control nodes. We use the term resource to indicate any medium used to store data, for example, configuration files or databases (see the *Resource* stereotype in Figure 1).

The *Edge server controller* includes a *container manager*, which is responsible for managing containers and Nodes. The *Edge server controller* usually integrates an MQTT broker [118] to communicate with devices.

Nodes execute the *Edge client*, which integrates the client of the MQTT component. The Edge client sends the data gathered from the *physical environment* (e.g., temperature) to the Edge server. Desktop Nodes usually execute Virtual Machine (VM) Nodes, which may execute multiple Pods. Pods are the smallest deployable units of computing that can be managed by Container Managers [66].

In the rest of the paper, we use the term *software environment* to refer to the operating system or any software component not belonging to the categories SUT and (SUT's) API.

2.2 Testing of Edge Systems

Edge frameworks are tested according to standard software engineering practices [44]. Information about the development process in place for proprietary frameworks is limited; however, we note that large companies embrace a testing culture and provide test automation support for the developers of Edge applications (e.g., for Azure [84]). The open-source frameworks considered in our study (i.e., KubeEdge, MainFlux [79], K3OS [52], and Zetta) are supported by private companies. KubeEdge is supported by the Cloud Native Computing Fundation and 27 additional private companies (e.g., ARM [7], Huawei [46], ci4rail [23]); Mainflux is developed and maintained by Mainflux Labs, which is a for-profit technology company; K3OS is part of Rancher, a framework developed by the open source software development company Suse [131]. However, since industry participation in open source projects does not provide any guarantee about software security [43], we investigated the testing procedures in place for the subjects of our study and describe them in the following paragraphs.

All the open-source frameworks considered in our study include automated test suites. KubeEdge includes automated unit [56], integration [60] and system test cases [58]. Also, KubeEdge's development process includes on code review activities (e.g., contributions are revised by senior members[1]) and two security teams [63, 64] that audit the system and respond to reports of security issues. Finally, KubeEdge is based on Kubernetes, whose development team includes a group of security experts [67]. Mainflux includes automated test cases and a dedicated benchmark [78]; further, MainFlux Labs perform security audits [76]. Finally, both K3OS [51] and Zetta [141] include automated test suites. To conclude, automated test execution is a state-of-the-practice approach for Edge frameworks; however, security seems to be better targeted by KubeEdge and, partly, by Mainflux.

The literature on Edge security highlights that security assurance of Edge systems should account for multiple attack surfaces (from physical layer to data security) and holistic, dedicated analyses are missing [50]. A recent survey of attack strategies and defense mechanisms for Edge systems points out that one of the causes of security vulnerabilities in Edge systems is the non-migratability of most security frameworks to the Edge context [136]; further, the provided attack descriptions show that, in general, the identification of security vulnerabilities is often delegated to manual activities (e.g., side-channel attacks or specification-based testing [21]) and automated tools concern vulnerabilities that might affect related systems (e.g., code injection or dictionary attacks for authentication). The lack of automated security testing solutions for Edge can be noticed from other surveys on the topic [2, 115], that suggest manual testing as a key solution to determine if the system appropriately respond to attack scenarios, thus further motivating our work. Finally, these surveys on attack methods do not provide details about the underlying vulnerabilities, thus, contrary to our work, not supporting the development of automated vulnerability testing solutions dedicated to the Edge.

[1] see https://kubeedge.io/en/docs/community/membership/

2.3 Field failures

Field failures are caused by faults that escape from the in-house testing process. For their characterization we refer to the work of Gazzola et al. [41], who performed a comprehensive study about causes and nature of field failures (i.e., failures affecting software deployed in the production environment or at end-user premises).

The study of Gazzola et al. is based on bug reports of open-source software (i.e., OpenOffice, Eclipse, and Nuxeo). The analysis in the study is based on four research questions:

- *Why are faults not detected at testing time?*
  Authors classified faults that are not detected at testing time into five categories (i.e., Irreproducible execution condition, Unknown application condition, Unknown environment condition, Combinatorial explosion, and Bad testing).
- *Which elements of the field are involved in field failures?*
  Authors identified five possible elements (i.e., Resources, Plugins, OS, Driver, Network) to be involved in field failures; sometimes *none* of them is involved.
- *What kinds of field failures can be observed?*
  Following the literature on the topic [9, 10, 18, 22, 24], authors classified failures according to failure types and detectability. They report three failure types: value, timing and system failures. As for detectability, they focus on three categories, which are signaled, unhandled, and silent.
- *How many steps are needed to reproduce a field failure?*
  Authors report on the number of user actions (called steps) required to reproduce a failure.

Different from Gazzola et al., we do not target faults affecting the functional properties of the software but faults affecting its security properties. Also, we have extended and refined the set of research questions considered in our study. Precisely, our refined research questions aim to facilitate the identification of security testing solutions to address the limitations of current security testing tools and practices. In our study, we address eight research questions instead of four.

2.4 Security Testing Glossary

Below, we provide definitions for security terminology appearing in the paper; we do not sort terms in alphabetical order but provide term definitions before their use in following descriptions.

Security failure. A *security failure* is a violation of the security requirements of the system.

Vulnerability. A *vulnerability* is a *"weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source" [33].* In our work we focus on vulnerabilities affecting

Edge frameworks, in other words, mistakes in the implementation, design, or configuration of the Edge framework that prevent either the framework or the software running on it from fulfilling its security requirements.

A vulnerability is said to be *exploited* by a malicious user U through an input sequence I, when (a) the malicious user provides the input sequence I to the software under test, (b) the input sequence exercises the vulnerability (i.e., the software executes the functionality affected by the weakness), and (c) a security failure is observed (i.e., there is a violation of security requirements). In a software testing context, it is the software tester who aims to identify input sequences that may reveal the presence of vulnerabilities.

Test oracle. A test oracle (or, simply, an oracle) is a procedure to determine if the software behaves according to its specifications [12], otherwise a test failure should be reported. In the context of security testing, test oracles should report security failures. Test oracles may either be automated or manual; in this paper, we focus on automated test oracles because we look for testing solutions that can be automatically executed.

CVE. Common Vulnerability Enumeration (CVE) [87] is a database managed by the Mitre corporation [89]. It lists publicly disclosed vulnerabilities. The CVE list is enumerated and managed by the CVE Numbering Authorities (CNA) [29]. All the registered vulnerabilities are characterized with a univocal identifier, a textual description, and additional details including severity, registration date, vulnerable product.

CWE. Common Weaknesses Enumeration (CWE) is a public database managed by the Mitre corporation [89]. It lists the weaknesses that may lead to a vulnerability; a weakness can be an invalid action taken by the software or a developer mistake performed when implementing or designing the software. For each weakness, the CWE database reports the CWE ID, its description, the creation date, a link to the NVD database, and references to external links (e.g., GitHub) to further explain the details about the vulnerability.

The CWE weaknesses constitute a catalog of vulnerability types organized according to different *views* (i.e, taxonomies) that group them in a hierarchical structure. The top level entries of such structures are called *pillars*. The views considered in our study are *research concepts* and *software development*. We excluded views that concern hardware design, are mappings to other taxonomies, or concern problems related to specific systems. The *research concepts* view focuses on the software behaviour and includes the following categories: Improper Access Control, Improper Interaction Between Multiple Entities, Improper Control of a Resource Through its Lifetime, Incorrect Calculation, Insufficient Control Flow Management, Protection Mechanism Failure, Incorrect Comparison, Improper Handling of Exceptional Conditions, Improper Neutralization, Improper Adherence to Coding Standards. The *software development concepts* view focuses on the development (e.g., design and programming) mistakes that lead to the vulnerability; it consists of 40 pillars including, among the others, API / Function Errors, File Handling Issues, Data Validation Issues, and Memory Errors.

Security properties. In our work we consider three security properties of software (i.e., Confidentiality, Integrity, Availability — CIA) that we define according to the NIST Information Security report NIST-800-137 [33]:

- *Confidentiality* concerns *"preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information"* [33].
- *Integrity* concerns *"guarding against improper information modification or destruction, and includes ensuring information non-repudiation and authenticity"* [33].
- *Availability* concerns *"ensuring timely and reliable access to and use of information"* [33].

NVD. The *National Vulnerability Database (NVD)* is the U.S. government repository of vulnerability data [120]. Vulnerabilities are reported using the Security Content Automation Protocol (SCAP), which consists of information including, among others, the CVE data and the Common Vulnerability Scoring System (CVSS). All the CVE vulnerabilities appears also on the NVD repository.

CVSS. The *Common Vulnerability Scoring System (CVSS)* is a framework for communicating the characteristics and severity of software vulnerabilities [27]. According to CVSS, each vulnerability is associated to a set of attributes: *Attack Vector*, which captures the context of the attack (Network, Adjacent, Local, Physical), *Attack Complexity* (Low, High), *Privileges Required* (None, Low, High), *User Interaction*, which indicates if the attacker needs to interact with another user (None, Required), *Scope*, which indicates whether a vulnerability in one vulnerable component impacts resources in components beyond its security scope (Unchanged, Changed), and *Impact Metrics*. Impact Metrics report how much the software security properties (i.e., Confidentiality, Integrity, and Availability) might be impacted (High, Low, None) by an exploit for the vulnerability. The CVSS attributes are represented through a string that reports the initials of each attribute along with its value. For example, for CVSS version 3.1, the string

AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H

indicates a Local (*L*) Attack Vector (*AV*), Low (*L*) Attack Complexity (*AC*), Low (*L*) Privileges Required (*PR*) to exploit the vulnerability, No interaction with an additional user being required (User Interaction, *UI*), Unchanged (*U*) Scope (*S*), and High impact (*H*) on Confidentiality (*C*), Integrity (*I*), and Availability (*A*).

The CVSS attribute values are used to derive a score between 0 and 10 that captures the severity of a vulnerability; score ranges are interpreted as follows: None (0.0), Low (0.1-3.9), Medium (4.0-6.9), High (7.0-8.9), Critical (9.0-10.0).

3 Study Design

The *goal* of our study is to investigate security vulnerabilities affecting Edge computing frameworks. The *purpose* is to identify the characteristics of Edge vulnerabilities

with the aim of driving improvements in the security testing process and supporting the identification of appropriate solutions for the development of security testing tools. The *context* consists of 147 vulnerabilities reported between January 2019 and December 2021. They concern four Edge frameworks, which are KubeEdge, Mainflux, K3os and Zetta. All the data used in our study are available online [80].

This study addresses *eleven research questions*, which we defined by focusing on those aspects that may drive the definition of an automated security testing technique. We focus on aspects that help identifying the testing opportunity (i.e, determine in which scenarios existing methods are insufficient), evaluating the feasibility of security testing automation (e.g., to avoid severe consequences on the integrity of the system), and defining the technical solution (i.e., design an input selection strategy, an automated test oracle, test harnesses and, in general, supporting procedures).

Figure 2 provides an overview of the relations between our research questions (RQs) and the final objective of this work (i.e., support the development of effective testing approaches for Edge systems); precisely, in Figure 2, we organize our RQs according to their objectives (i.e., identifying the testing opportunity, evaluating the feasibility of security testing automation, and defining the technical solution) and indicate which information is acquired by addressing each RQ. In this manuscript, our RQs are sorted according to the data used to address them: first we present the RQs addressed through the manual inspection of vulnerability reports ($\mathbf{RQ}_1$ to $\mathbf{RQ}_6$, with the four RQs inspired by Gazzola et al.'s work first), then we present research questions addressed using data available on the CVE and NVD databases ($\mathbf{RQ}_{7A}$ to $\mathbf{RQ}_8$). A detailed description of our research questions follows:

$\mathbf{RQ}_1$: *Why are Edge vulnerabilities not detected during testing?*
Like any other software system, an Edge system shall undergo a security testing phase in which engineers verify that it meets its security requirements [37,74]. The presence of vulnerabilities not being detected during testing but discovered later (e.g., once the system has been already released and deployed in the field), indicates pitfalls in the testing process. This research question aims to determine the reasons that prevented the detection of vulnerabilities during testing and whether further research is needed to prevent security failures in the field or, alternatively, if field failures can be avoided simply through the improvement of the testing process in place (i.e., testing was insufficiently conducted).

$\mathbf{RQ}_2$: *What are the types of components involved in a security failure?*
Similarly to the study of Gazzola et al., we aim to determine which components are involved in a security failure. However, to better support the definition of automated security testing techniques, we aim to distinguish between (A) the failing components, which indicate what should be the targets of test oracles, (B) the components that should be in a specific state to exploit the vulnerability, which may indicate the conditions under which the software should be tested (e.g., with an overloaded network), (C) the components receiving the input, which influence the type of input interfaces that should be managed by the testing technique (e.g., a Web interface or an input file), and (D) the vulnerable components, which indicate what to test. The analysis of the types of components involved in a security failure should support the identification of appropriate testing strategies. Therefore, we refine our research question into four:

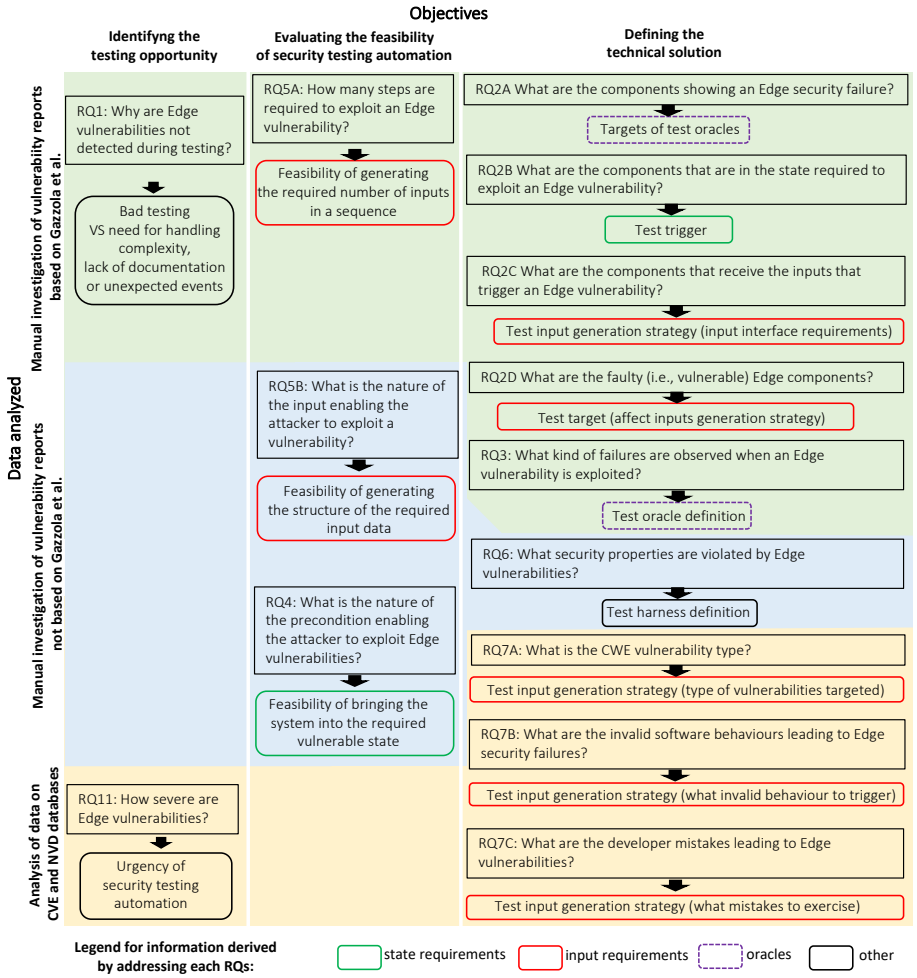


Fig. 2: Research Questions: Objectives, Data analyzed, and Information derived.

- **RQ**$_{2A}$ *What are the components manifesting an Edge security failure?*
- **RQ**$_{2B}$ *What are the components that are in the state required to exploit an Edge vulnerability?*
- **RQ**$_{2C}$ *What are the components that receive the inputs that trigger an Edge vulnerability?*
- **RQ**$_{2D}$ *What are the faulty (i.e., vulnerable) Edge components?*

RQ$_3$**:** *What kind of failures are observed when an Edge vulnerability is exploited?* To automatically test a software system, it is necessary to specify test oracles (see Section 2.4). The implementation of an automated test oracle depends on the nature of the failures to be detected; for instance, the program logic required to automatically detect a crash might be based on response timeout, which is likely different than the

logic required to detect unauthorized access to a resource, which might consist of verifying the data returned to the caller.

RQ$_4$: *What is the nature of the precondition enabling the attacker to exploit Edge vulnerabilities?*
A vulnerability may be exploited only if a certain precondition holds (e.g., a subnet has been set-up). Since it might be difficult for an automated approach to meet certain preconditions (e.g., automatically set-up a network), to evaluate the potential benefits of test automation (e.g., the proportion of vulnerabilities it might detect), we investigate the nature of such preconditions for different vulnerabilities.

RQ$_5$: *What inputs enable exploiting Edge vulnerabilities?*
The effectiveness of a test automation approach depends on the degree of complexity of the input to be generated, which we may characterize in terms of the number of required interactions with the SUT and the structure and type of input actions to perform (e.g., providing data, changing software configurations, or simulating network disruptions). For instance, a vulnerability that requires a long input sequence to be exploited may be more difficult to detect than one can be detected with single input. We therefore refine **RQ$_5$** into two separate questions:

- **RQ$_{5A}$:** *How many steps are required to exploit an Edge vulnerability?*

- **RQ$_{5B}$:** *What is the nature of the input action enabling the attacker to exploit a vulnerability?*

RQ$_6$: *What security properties are violated by Edge vulnerabilities?*
The type of security properties being violated by Edge security failures impact on the definition of automated oracles. Also, they may affect the test harness solutions[2] to put in place. For example, vulnerabilities that affect availability can be detected by oracles that look for the lack of responses from the system; instead, to detect authorization vulnerabilities it is necessary an oracle that is aware of the system's access policies. Concerning test harness, after discovering availability issues, it may be necessary to restart the system (e.g., to prevent blocking other testing processes), which is not required after discovering confidentiality problems (confidentiality issues do not alter the state of the system). Instead, the discovery of an integrity issue may imply restoring the configuration of the system after discovery.

RQ$_7$: *What faults cause Edge vulnerabilities?*
The input selection strategy implemented by a test automation approach depends on the types of faults being targeted. In the case of security testing, for example, the inputs to be selected to identify an SQL injection attack are different than the ones used to detect a path traversal vulnerability (e.g., they rely on different grammars). To categorize faults, we can rely on the CWE vulnerability types, which is well-known and largely adopted taxonomy. Additional aspects to take into account are the *erroneous software behaviors* caused by the vulnerability (e.g., improper access control) and by the *developer mistakes* leading to the vulnerability (e.g., memory buffer errors). Erroneous software behaviors are captured by the CWE pillars for the CWE view *Research concepts*; developer mistakes are captured by the CWE pillars

[2] We use *test harness* to indicate the technical solutions supporting test automation.

for the CWE view *Developer concepts*. We therefore refine $\mathbf{RQ}_7$ into three RQs that reflect the information collected in our process:

- $\mathbf{RQ}_{7A}$**:** *What is the CWE vulnerability type?*
- $\mathbf{RQ}_{7B}$**:***What are the erronous software behaviours leading to Edge security failures?*
- $\mathbf{RQ}_{7C}$**:** *What are the developer mistakes leading to Edge vulnerabilities?*

$\mathbf{RQ}_8$**:** *How severe are Edge vulnerabilities?*
To evaluate the importance of improving Edge security testing approaches, $\mathbf{RQ}_8$ discusses severity based on NVD CVSS scores (see Section 2.4); severity analysis provides an indication about the urgency for automated security testing approaches.

$\mathbf{RQ}_1$, $\mathbf{RQ}_2$, $\mathbf{RQ}_3$, and $\mathbf{RQ}_{5A}$ are inspired by the work of Gazzola et al.; however, we have extended the analysis method to better fit the context of this study. Precisely, the taxonomies used to address $\mathbf{RQ}_1$ and $\mathbf{RQ}_{5A}$ match the one used by Gazzola et al.; the taxonomies used for $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$ are an extension of the one proposed by Gazzola et al. Further, we address $\mathbf{RQ}_4$ and $\mathbf{RQ}_{5B}$ using a taxonomy that we introduce in this article. For $\mathbf{RQ}_6$ we rely on the CIA security properties (but we distinguish between data and system integrity). For $\mathbf{RQ}_{7A}$, $\mathbf{RQ}_{7B}$, $\mathbf{RQ}_{7C}$ we rely on CWE categories. Finally, for $\mathbf{RQ}_8$, we rely on NVD CVSS attributes.

3.1 Data collection

Figure 3 provides an overview of the process adopted to collect data and answer our research questions.

For our study, we selected Edge frameworks that fulfill the following criteria: (C1) being open-source and publicly available, which enables the investigation of software patches for a better understanding of the vulnerability, (C2) having active user base (i.e., users reporting bugs and vulnerabilities online) and support (i.e., responses are provided to 90% of the end-user issues), which ensures that the software provides features that are helpful for the development of Edge systems, (C3) having at least five vulnerabilities reported by end-users either on the CVE databases or GitHub (not all the vulnerabilities are necessarily reported on the CVE database).

We focus on Edge frameworks rather than services or applications developed to run on Edge frameworks since the latter delegate security management to the underlying frameworks [55].

First, we have identified 15 open-source Edge frameworks by executing a Web search with the Google search engine; we searched for the keywords '`edge framework`' and '`IoT framework`'. The identified frameworks are shown in Table 1, whereas columns C1, C2, and C3 indicate which of the above-mentioned criteria had been satisfied.

Based on our criteria, we selected as subjects of our study KubeEdge, Mainflux, Zetta, and K3os. *KubeEdge* [57] is the framework with the largest number of users providing comments in the issue tracker, probably because it is the most widely adopted one. It is developed as an open-source project by Cloud Native Computing

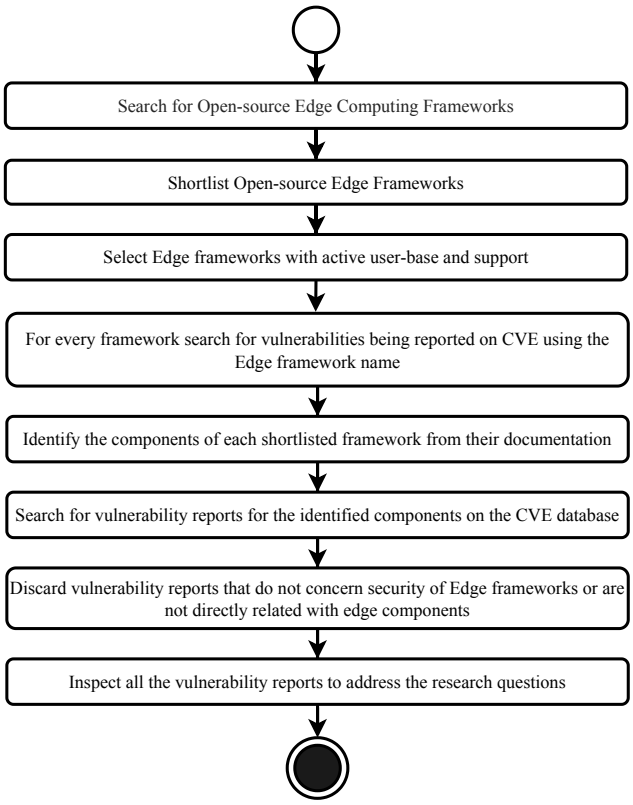


Fig. 3: Activity diagram for our approach in the manuscript

Foundation (CNCF) [26]. It is an open-source product built upon Kubernetes [69], which is a system for automating deployment, scaling, and management of containerized applications. KubeEdge extends containerization capabilities to Edge devices. KubeEdge's bug reports and vulnerabilities are available on its GitHub page [59] and CVE database [87], respectively.

Mainflux [79] is an open-source framework designed by *Mainflux Labs* to support smart devices in the Internet of Things (IoT) ecosystem. It has a simpler architecture than KubeEdge (i.e., less components) and serves as a middleware between Edge devices and cloud-based orchestration platforms; it targets systems that largely rely on the Edge paradigm (i.e., IoT). Its bug reports can be accessed on GitHub [77].

Zetta [142] is an open-source, Web-based Edge framework which provides connectivity to different types of smart devices. The Zetta's centralized device controller (Zetta hub) is designed to work on low-powered devices capable of running an OS such as BeagleBone Black, Intel Edison, or Raspberry Pi. Zetta's bug reports and vulnerabilities are available on the GitHub and CWE database [143].

K3os [52] is an open-source Edge framework designed to work in low resource environments with the capability of being managed through a light-weight Kuber-

Table 1: List of all the opensource Edge frameworks identified in our search (selected ones in bold). Labels C1, C2, and C3 refer to our selection criteria (see Section 3.1).

Framework	Selected	Organization	License	C1	C2	C3
KubeEdge	✓	Kubernetes	apache-2.0	✓	✓	✓
Wasm3	*x*	Volunteers	MIT	✓	✓	*x*
Baetyl	*x*	Linux Foundation Edge	apache-2.0	✓	*x*	*x*
Mainflux	✓	MAINFLUX LABS	apache-2.0	✓	✓	✓
Superedge	*x*	Volunteers	other	✓	*x*	*x*
Yomo	*x*	Volunteers	apache-2.0	✓	✓	*x*
Fog-flow	*x*	FIWARE	bsd-3-clause	✓	✓	*x*
Cloudsimsdn	*x*	Volunteers	gpl-2.0	✓	*x*	*x*
Deviceplane	*x*	Volunteers	apache-2.0	✓	*x*	*x*
Distributed Storm	*x*	Volunteers	apache-2.0	*x*	✓	*x*
ENORM	*x*	Volunteers	apache-2.0	✓	*x*	*x*
K3os	✓	Volunteers	apache-2.0	✓	✓	✓
Oci	*x*	Volunteers	BSD-2	*x*	✓	*x*
Zetta	✓	Volunteers	MIT	✓	✓	✓

netes dashboard called *k3s*. For example, it is used by Rancher, a multi-Cloud container management platform [122].

For each Edge framework, we analyzed the vulnerabilities reported in its bug repository (GitHub) and the ones appearing in the CVE database. To identify vulnerabilities in the GitHub repository, we used the GitHub built-in search functions to search for bug reports containing security-related keywords (i.e., security, vulnerability, crash, and privacy) either in their title or in the description of the vulnerability. To select vulnerabilities in the CVE database, we used the built-in search function to identify CVE records including the name of the framework. Also, we searched for vulnerabilities referring to components implementing the containerization and communication features used by our frameworks, which are MQTT brokers (e.g., Mosqitto [116] and VerneMQ [135]), Raspberry pi (configured as end-device or client manager for pods), and container managers (i.e., Kubernetes, Docker, and Cri-o). Precisely, KubeEdge components include Kubernetes, Cri-o, Raspberry Pi, Mosquitto or verneMQ, whereas Mainflux components include only Docker. K3os components include Kubernetes; Zetta's components include Raspberry Pi. However, to avoid duplicates in our study, the Edge vulnerabilities concerning Kubernetes (61, in total) and Raspberry Pi (two, in total) had been counted as part of KubeEdge only. Since we do not aim to compare frameworks but study the nature of Edge vulnerabilities, our choice should not bias our results.

In our study, we considered all the GitHub bug reports submitted till 31 November 2021, and all the CVE vulnerabilities dated between 1 January 2019 and 31 November 2021.

Table 2 provides the number of reports collected from GitHub and CVE, for each selected framework. The total number of reports ranges from 5 (Zetta) to 125 (KubeEdge); unsurprisingly, such number is related to the complexity of the framework (i.e., the largest frameworks, including their dependencies, are the ones with the largest number of vulnerabilities).

Table 2: Vulnerabilities selected for each case study subject

Framework	Reports				
	GitHub	CVE	Total	Vulnerabilities	Edge vulnerabilities
KubeEdge[1]	39	76	115	80	71
Mainflux[2]	7	118	125	119	74
K3os	18	-	18	1	1[3]
Zetta	5	-	5	1	1[4]
Total	**69**	**194**	**263**	**201**	**147**

Notes: [1] KubeEdge vulnerability count includes also vulnerabilities affecting Kubernetes, Cri-o, Raspberry-Pi, and MQTT brokers (Mosquitto or verneMQ). [2] Mainflux vulnerability count includes the vulnerabilities affecting Docker components used within Mainflux. [3] For K3OS, if we count also the vulnerabilities affecting Kubernetes and Raspbery Pi we end up with 64 *Edge vulnerabilities*. [4] For Zetta, if we count also the vulnerabilities affecting Raspbery Pi we end up with 3 *Edge vulnerabilities*.

Column *Vulnerabilities* in Table 2 provides the number of vulnerabilities reported in the collected reports, which are 201, in total. Vulnerability reports were identified by the first author of the paper who read all the report descriptions. Among all the vulnerability reports, we excluded the ones that concern Edge components (e.g., Docker) but affect features not used by Edge frameworks. An example is vulnerability CVE-2021-31938 [109] in Kubernetes [69], which concerns the Microsoft Visual Studio Code Kubernetes tool [85]. Such tool is not executed at runtime within the Edge system but is used at configuration time to implement scripts for the Kubernetes framework; therefore, the vulnerability is out of scope. After filtering, we count 147 vulnerabilities affecting the Edge frameworks considered in our study (see column *Edge vulnerabilities* in Table 2). Please note that the requirement of minimum five vulnerabilities to select an Edge framework for our study concern the total number of vulnerability-related reports in GitHub or CVE (i.e., column *Total* in Table 2), not the number of Edge vulnerabilities selected at the end of the process.

3.2 Analysis Method

This section explains the metrics and the procedures put in place to answer our research questions based on the collected vulnerability reports.

For our study, we proceeded as follows. The first author of the paper has carefully read the 147 vulnerability reports indicated above along with links to related electronic documents (e.g., detailed vulnerability descriptions provided on the frameworks' Web sites) and code commits registered on their versioning systems (e.g., git code commits selected by relying on either the vulnerability ID or a bug fix ID reported in related electronic documents). We resorted to the inspection of code commits when the description of the vulnerability was not clear (i.e., it did not enable us to answer some of our RQs). By reading the vulnerability descriptions and the related electronic resources, to address each RQ, the first author (1) classified each vulnerability according to the categories specified to address $\mathbf{RQ}_1$ to $\mathbf{RQ}_6$ and (2) collected the data required to address $\mathbf{RQ}_{7A}$ to $\mathbf{RQ}_8$. To minimize subjectivity in the manual classification, the authors of the paper have defined together the answers for each RQ

Table 3: Data collected for the vulnerabilities described in Section 3.2

Vulnerability ID	**RQ**$_1$	**RQ**$_{2A}$	**RQ**$_{2B}$	**RQ**$_{2C}$	**RQ**$_{2D}$	**RQ**$_3$-F	**RQ**$_3$-D	**RQ**$_4$	**RQ**$_{5A}$	**RQ**$_{5B}$	**RQ**$_6$	**RQ**$_{7A}$	**RQ**$_{7B}$	**RQ**$_{7C}$	**RQ**$_8$-A	**RQ**$_8$-P
2021-3499	IE	Nd	Se	Ne	Se	Ne	Sl	Co	1	D	SI	863	284	-	H	Non
2020-8565	UA	Su	Su	Non	Su	Ac	SL	Non	0	Non	C	532	664	-	Lo	Lo
2020-8559	UE	Su	Su	Ne	Su	V	SL	Non	No	D	SI	601	664	-	Lo	Lo
2021-25737	CE	Ne	Ne	Su	Su	Ne	SL	Co	No	Co	DI	601, 184	664	-	Lo	H
2020-28914	UA	R	Su	Su	Su	Ac	SL	Non	1	Co	DI	732	284	-	Lo	Lo
2021-39159	UA	Nd	Su	AP	P	U	SL	Non	1	D	SI	94	664	-	Lo	Lo
2021-34431	CE	Nd	Ne	Ne	Ne	Ne	S	Non	1	D	A	401	664	-	Lo	Lo
2019-11252	CE	Su	Non	Non	Su	Ac	SL	Non	0	Non	C	209	664, 703	IME, EC-RC-SC	Lo	Lo
2020-15127	CE	Nd	Ne	Ne	Su	Sy	SL	Non	1	D	SI	306	284	AE	Lo	Non
2021-32783	UA	Su	Ap	Ap	Ap	Sy	Un	Non	1	D	SI	610, 441	664	-	Lo	Lo
2021-38545	CE	Non	HW	HW	HW	Ac	SL	Non	1	D	C	-	-	-	H	Non
KubeEdge#1736	UA	Nd	Nd	R	Su	T	S	Co	1	De	A	-	-	-	-	-
2021-28166	CE	Ne	Ne	Ne	Ne	V	S	Non	1	D	A	476	703	PI	Lo	Lo
2020-35514	CE	Su	P	Su	P	Ac	SL	Non	1	D	SI	266	284	Pi	H	Lo
2020-8558	UA	Su	Se	Ne	Se	Ne	SL	Non	1	D	C	287, 420	284	CCE	Lo	Non
KubeEdge#2362	UA	Nd	Nd	Su	Su	Sy	S	L	1	D	A	-	-	-	-	-
Zetta#335	CE	Su	Su	Su	Se	Ne	Sy	Co	1	ReU	A				-	-
2020-8563	CE	Su	Su	Non	Su	V	SL	Co	0	Non	C	532	664	A/L-E, IME	Lo	Lo
2021-20218	CE	Nd	P	Su	P	Ac	SL	Non	No	D	SI	22	664		H	No
2014-5278	CE	Su	Su	Su	Ne	Ne	SL	Co	1	D	SI	-	-	-	Lo	Non
2020-8557	UA	Nd	Non	Non	Su	Sy	Un	ReU	0	Non	A	400	664	-	Lo	Lo
2020-13597	CE	Ne	Ne	Ne	Ne	Sy	SL	Co	1	Co	C	200, 201	664	IME	H	Lo
2021-21334	CE	Nd	Su	Su	Su	Ac	SL	Co	1	D	C	668	664	-	H	Lo
2021-21251	UA	Su	P	P	P	V	SL	Co	No	D	DI	22	664	-	Lo	Lo
2020-2211	CE	Su	P	P	P	V	SL	Co	1	D	SI	502	664	RME	Lo	Lo
2020-8566	UA	Su	P	Non	P	Ac	SL	No	0	Non	C	532	664	ALE, IME	Lo	Lo

Abbreviation Terms:
IE: Irreproducible execution condition, **UA:** Unknown application condition, **UE:** Unknown environment condition, **B:** Bad Testing, **CE:** Combinatorial Explosion, **R:** Resource, **AP:** API, **Su:** SUT, **D:** Driver, **Se:** Service, **Ne:** Network, **Nd:** Node, **HW:** Hardware, **Non:** None, **Ac:** Action, **V:** Value, **T:** Timing, **Sy**: System, **I:** Integrity, **S:** Signalled, **Un:** Unhandled, **SL:** Silent, **A:** Availability, **no:** No Information, **Co:** Configuration, **De:** Delay causing missing resource, **L:** Lack of Data, **ReB:** Resource Busy, **C:** Confidentiality, **ReU:** Resource Unavailable, **SI:** System Integrity, **DI:** Data Integrity, **Lo:** Low, **H:** High, **IME:** Information Management Errors, **RME:** Resource Management Errors, **ALE:** Audit/Logging Error, **PI:** Pointer Issues, **Pi:** Privilege Issues, **A/L-E:** Audit/Logging Errors, **RQ$_8$-A:** **RQ$_8$**- Attack Complexity, **RQ$_3$-F:** **RQ$_3$**- Failure Type, **RQ$_3$-D:** **RQ$_3$**- Detectability, **RQ$_8$-P:** **RQ$_8$**- Attack Privileges, **CCE:** Communication Channel Errors, **EC-RC-SC:** Error Conditions, Return Values, Status Codes.

and discussed at least one concrete case for each class. In practice, the first 30 vulnerabilities inspected at the beginning of the project had been reviewed by both the two authors to ensure common understanding. Further, randomly selected cases and unclear cases had been discussed. In total, about 50 vulnerabilities had been inspected by both authors. For a subset of the first 30 vulnerabilities there had been disagreement due to definition of common terminology and criteria, which lead the first author to re-classify, from scratch, all the 30 vulnerabilities till agreement was reached. For the remaining 20 randomly selected cases, the two authors were in agreement. Addressing $\mathbf{RQ}_7$ and $\mathbf{RQ}_8$ did not require any specific agreement between the authors because it relies on information available with the vulnerability report.

Table 3 provides the data collected for the vulnerabilities mentioned as examples in the following paragraphs.

3.2.1 $\boldsymbol{RQ}_1$: Why are Edge vulnerabilities not detected during testing?

To address this research question, we classify each vulnerability report according to the same five categories reported in Gazzola's work:

- *Irreproducible Execution Condition (IEC).* It indicates that the vulnerability cannot be identified at testing time because it is not feasible to reproduce the conditions under which it can be exploited. An example is Kubernetes vulnerability CVE-2021-3499 [112], which reports that Kubernetes is unable to apply multiple DNS firewall rules during egress communication (i.e., communication leaving the local network). Without knowing the specific firewall rules to apply during testing, it is unlikely to discover this vulnerability.
- *Unknown Application Condition (UAC).* It indicates that the security failure depends on an input that was not identified by the testing engineer because not specified in the documentation. An example is vulnerability CVE-2020-8565 [101], which reports that, with logging level 9, the system exposes administrator details by writing them in logs as plain text, including authorization and bearer token (i.e., an hexadecimal string used for requesting access to a resource). Since the availability of logging level 9 is not well documented [68], testing engineers may have overlooked it.
- *Unknown Environment Condition (UEC).* It indicates that the precondition or the type of input required for triggering the vulnerability depends on a characteristic of the environment (software environment or physical environment) that was not known to security engineers (e.g., because not well documented). An example is Kubernetes vulnerability report CVE-2020-8559 [99], which indicates that a malicious user can redirect update requests. This vulnerability has been likely not discovered at development time because of the limited documentation on redirect responses, which concerns the communication protocol.
- *Combinatorial Explosion (CE).* Sometimes, to detect a vulnerability at testing time, it is necessary to exercise the system with inputs derived by combining values belonging to different input partitions[3], for different input parameters or configurations. When the system is large, the combination of values belonging to

[3] An input partition is a input region with equivalent values, from a testing perspective [3].

different input partitions for different parameters and functions lead to a number of test cases that is very large and thus infeasible to be defined, executed, or verified (i.e., the number of test cases *explode*). Also, when inputs can have a complex structure adhering to a specific grammar (e.g., xpaths), testing different combinations of valid and invalid grammar tokens becomes challenging. Unfortunately, without details about the development budget for our case study subjects, it is not possible to determine a threshold above which it is impractical for software engineers to test different input (or grammar token) combinations. Therefore, we conservatively assume that combinatorial explosion is the cause of any vulnerability that can be triggered only with specific combinations of input parameters, independently from the number of parameters, input partitions, or grammar tokens, for the vulnerable function. Indeed, in large systems, it is common practice for engineers to limit testing cost by exercising only few combinations of inputs (e.g., by relying on the *weak equivalence class testing strategy* [3]). Please note that although functional testing approaches such as N-wise coverage [3] may have enabled engineers to address combinatorial explosion and discover vulnerabilities, the available information does not enable us to determine if such strategies had been applied in our case study subjects. Therefore, we simply report all the combinatorial cases together, independently of the strategy followed to test them. An example CE is provided by the Kubernetes vulnerability report CVE-2021-25737 [106]; it indicates that the user can redirect network traffic into a subnet, which is typically not allowed by the administrator. The vulnerable version of Kubernetes can prevent traffic redirection for Nodes and Pods but not for subnets created by a Node or Pod. Security engineers may have tested this features with Nodes and Pods but not with subnets.

– *Bad Testing (BT).* We consider a vulnerability to slip through the testing process because of *bad testing* when it is not possible to find a justification for the lack of testing effectiveness in terms of lack of feasibility (i.e., IEC), lack of documentation (i.e., UEC and UAC), or lack of test budget (i.e., CE). In practice, following the guidelines of Gazzola et al., anything not categorized in the above-mentioned scenarios is considered due to bad testing [41]. In practice, as for the study of Gazzola et al., we conservatively consider caused by bad testing only those cases where a basic security feature of the SUT is always not functioning as specified (e.g., when access to a feature is always granted, even if the username/password combination is wrong).

Our classification has been performed by reading each vulnerability report to determine the features that should be exercised to detect the vulnerability. Further, we inspected the available documentation to (1) determine UAC and UEC cases (they concern the lack of detailed documentation) and (2) to determine what are the possible input partitions. When available, we also inspected bug-fix commits to have a better understanding of the vulnerability. Although it is not possible to know the exact cause of each field failure without involving the actual developers of the frameworks, our investigation helps determining reasonable ones (i.e., causes that may not be true for the considered case study but might have been true for a system with the same characteristics).

3.2.2 $\mathbf{RQ}_2$: *What are the types of components involved in a security failure?*

This research question aims to characterize the components exercised when a security failure is observed. As mentioned in Section 3, this research question is divided into four:

- $\mathbf{RQ}_{2A}$: *What are the components manifesting an Edge security failure?*
- $\mathbf{RQ}_{2B}$: *What are the components that are in the state required to exploit an Edge vulnerability?*
- $\mathbf{RQ}_{2C}$: *What are the components that receive the inputs that trigger an Edge vulnerability?*
- $\mathbf{RQ}_{2D}$: *What are the faulty (i.e., vulnerable) Edge components?*

The above-mentioned RQs are addressed by tracing, for each vulnerability report, the types of components involved in the activities captured by $\mathbf{RQ}_{2A}$- $\mathbf{RQ}_{2D}$. We have refined the list of components introduced by Gazzola et al., which included resources, plugins, OS, drivers, and network. Our refined list of components includes additional elements that characterize Edge systems (see Section 2.1), which are API, Nodes, and hardware (i.e., the machine on which the software is running). Also, we explicitly indicate if the failure concerns the SUT (i.e., the Edge framework under test). We exclude the OS category from our analysis because the activity of the OS is generally invisible to the Edge frameworks and we did not identify any vulnerability related to it; further, OS-support tools are often part of Edge frameworks themselves.

All our components are described in the following:

- *Resources.* Resource refers to any software medium used to store data, for example files or databases. An example is given by Kubernetes vulnerability report CVE-2020-28914 [95], which indicates that a malicious user can access restricted folders (i.e., resources) with both read and write permissions using a guest account.
- *Drivers.* Driver indicate devices drivers for the operating system controlled by the Edge server controller (see Section 2.1).
- *Plugins.* A plugin is an add-on component or module that enhances the system's capabilities. An example is provided by Kubernetes vulnerability CVE-2021-31938 [109], which concerns the Kubernetes plugin *Helm*. Helm exchanges username and password without encryption, therefore, a malicious user may introduce a custom URI in the system configuration to steal the username and passwords of its users. In the case of $\mathbf{RQ}_{2A}$, it is Helm (i.e., the plugin) what experiences the effect of the vulnerability (i.e., receives username and password). For $\mathbf{RQ}_{2B}$, the component in the required state is a resource; precisely, a configuration file that contains the custom URI to exploit the vulnerability. For $\mathbf{RQ}_{2C}$, the component receiving the input that triggers the vulnerability is the Helm plugin. For $\mathbf{RQ}_{2D}$, the faulty component is the SUT, since it should not allow end-users to change the configuration files in which the Helm URI is located.
  Another case is provided by the docker vulnerability CVE-2021-39159 [114], where the faulty component is the plugin *matrix-media-repo*. The plugin matrix-media-repo minimizes the size of the images saved on the server side. However, accessing stored images from the database requires a decompression process; a

malicious user may upload special crafted images that exhaust the decompression process and cause a security failure (i.e., a denial-of-service) on the SUT (i.e., KubeEdge).

- *Software Under Test (SUT).* We introduced this term to indicate cases in which issues concern the Edge framework under test. An example is provided by vulnerability CVE-2021-34431 [111] in Docker, in which the faulty component is the Mosquitto [116] MQTT Broker (SUT, according to Figure 1). During the handshake process between the client and the server, a *CONNECT* packet should be sent from the client to the server only once. The server is responsible for processing the *CONNECT* request and reply; the presence of multiple *CONNECT* requests being sent to the server by a same client is considered a protocol violation which results in the client being disconnected. The vulnerability concerns Mosquitto, in which the disconnection of a client leads to a memory leak that may end-up into a denial-of-service. In the example, the node is the component affected by the effects of the vulnerability ($\mathbf{RQ}_{2A}$), the network protocol should be in a specific state (i.e., the CONNECT state) ($\mathbf{RQ}_{2B}$), the network receives the input which triggers the vulnerability ($\mathbf{RQ}_{2C}$), and the SUT (i.e., Mosquitto) is the faulty component ($\mathbf{RQ}_{2D}$).
- *Services.* Services are executable programs that provide the data required by the SUT. An example is provided by Kubernetes vulnerability report CVE-2019-11252 [91], which indicates that the services bound to loopback address (127.0.0.1) are accessible by other hosts on the network. Those services should only be accessible to local processes. In this case, these loopback services are the components experiencing the effects of the vulnerability ($\mathbf{RQ}_{2A}$).
- *Network.* Components implementing network-related functionalities (i.e., communication protocols, firewalls, and ports) belong to this category. An example is provided by the Kubernetes vulnerability report CVE-2021-28448 [108], which describes the incapability to enforce multiple firewall rules for DNS traffic during egress communication. In CVE-2021-28448, for $\mathbf{RQ}_{2A}$, the SUT is what experiences the effects of the security failure since its data could be shared with otherwise restricted URLs over the Internet (DNS filters are not working properly). For $\mathbf{RQ}_{2B}$, the network (specifically the network firewall) is the component in the state required to exploit a vulnerability. For $\mathbf{RQ}_{2C}$, it is the network what receives the input traffic exploiting the vulnerability. For $\mathbf{RQ}_{2D}$, it is the network the vulnerable component.
- *Node.* A Node is an execution environment; it includes a file system and all the programs and services running on it. In this category, we include also virtual machines and Pods. An example is provided by vulnerability CVE-2020-15127 [93] in Kubernetes; it concerns Pods leaking passwords to a phishing URI. In kubernetes, a container can be exported using two formats (i.e., *.OCI* and *.v2*). Importing a container from these images initiates dependency resolution through the Web. A malicious user can inject a phishing URL as a dependency to be resolved during the import of container; it will enable the malicious user to steal credentials. Importing an infected container image will thus result in credentials theft during dependency resolution. In the example, the newly deployed node is what it is compromised ($\mathbf{RQ}_{2A}$), the node is also what needs to be in the state that

requires resolving dependencies ($\mathbf{RQ}_{2B}$), the SUT (i.e., Kubernetes) is what receives the input to import the container from an image ($\mathbf{RQ}_{2C}$), Kubernetes (i.e., our SUT) is the faulty component ($\mathbf{RQ}_{2D}$).

- *API.* API indicates the components implementing the APIs used for controlling the Edge system (see Section 2.1). An example vulnerability is CVE-2021-32783 [110], which concerns the Contour controller API in Kubernetes. Typically, an access request from outside of the network is prohibited, therefore, the access is denied. However, the Contour controller is not capable of correctly handling multiple access requests thus resulting in a denial of service (DoS). In CVE-2021-32783, it is the SUT what is compromised after exploiting the vulnerability ($\mathbf{RQ}_{2A}$). Instead, it is the Contour API that is faulty ($\mathbf{RQ}_{2D}$), needs to be in the necessary state to exploit the vulnerability ($\mathbf{RQ}_{2B}$), and receives the input that triggers the vulnerability ($\mathbf{RQ}_{2C}$).
- *Hardware.* Hardware refers to the hardware components of the system, which include physical devices running the SUT (e.g., IoT devices, servers, or desktops) and network assets (e.g., routers and switches). An example is provided by vulnerability CVE-2021-38545 [113] in Raspberry Pi, which results in a *Glowworm* attack [119]. When speakers are connected to Raspberry Pi, voltage fluctuations caused by the use of speakers impact on the power supplied to the led of the Raspberry Pi module. If the led light is monitored, voltage fluctuations can be reconstructed and it is possible to reproduce the sound being played on the speakers [13]. In the example, the failure affects a hardware component ($\mathbf{RQ}_{2A}$); indeed, the led violates the implicit security requirement "it should not be possible to determine the sounds being played from light fluctuations". Further, the hardware should be in the necessary state (i.e., speakers being connected, $\mathbf{RQ}_{2B}$), the hardware is the component that receives the input (sound data) to trigger the vulnerability ($\mathbf{RQ}_{2C}$), and the hardware is the faulty component (i.e., it does not include a mechanism to avoid such light fluctuations, $\mathbf{RQ}_{2D}$).

Please note that not all the components mentioned above may be part of Edge framework distributions; indeed, only SUT, API, and Resources (e.g., configuration files) are released with Edge framework distributions. The other components (i.e., Drivers, Plugins, Services, Network, Node, Hardware) are usually developed by third-parties but are strongly coupled with an Edge framework and their CVEs provide references to such Edge framework. Examples of the second category of components follow. One example is CVE-2021-26928, which concerns the *service* BIRD daemon (it can be exploited to disrupt the integrity of Kubernetes). Another case is CVE-2020-13597, which concerns the *Network* layer of Calico and leads to information disclosure if IPv6 is enabled but unused. Last CVE-2021-38545, which concerns the *hardware* of Raspberry Pi.

3.2.3 $\mathbf{RQ}_3$: What kind of failures are observed when an Edge vulnerability is exploited?

Like Gazzola et al., for each vulnerability we determine failure type and detectability based on the description in the vulnerability report and bug fix commit, when available. Gazzola et al. determined category entries based on the taxonomies of Bondavali

and Simoncini [18], Aysan et al. [10], Avizienis et al. [9], Chillarege et al. [22], and Cinque et al. [24]. We extended their set with entries specific for our security context.

The *failure type* concerns how a failure *appears to an observer external to the system* [41]. We extended the set of failure types provided by Gazzola et al. (i.e., value, timing, or system) with two additional entries (i.e., action, and network). They are all described below.

- *Value.* Value failures occur when the system provides an output that does not match its specifications. In our context, they range from returning an illegal value (e.g, after exploiting an integrity vulnerability), to providing sensitive information (e.g., for a vulnerability concerning confidentiality).
- *Timing.* Timing failures include two cases: (1) the system takes longer than expected (according to specifications) to generate an output, (2) the system takes shorter than expected to generate output. An example is KubeEdge GitHub issue #1736 [62], which indicates that, during initialization, a Pod may try to allocate a storage volume according to configuration files that shall be provided by the Edge-core (i.e., the Edge server controller). Since the Pod is unable to find the configuration files in the directory, it hangs and results in a denial-of-service (i.e., a timing failure).
- *System.* System failures occur when the system crashes. An example is provided by the vulnerability report CVE-2021-28166 [107], which concerns Mosquitto communicating with an MQTT broker. CVE-2021-28166 indicates that an authenticated MQTT client can send a crafted packet CONNACK (connection Acknowledgment) to the broker thus causing a null pointer dereference that crashes the system (system failure).
- *Action.* Action failures consist of the system performing an illegal interaction with the environment. We introduced this category to compensate for the original categorization by Avizienis et al. [9] used by Gazzola et al., which considers the SUT as a black-box and excludes the possibility to observe other output interfaces rather than the ones with the end-user. To further clarify the difference between action failures and value failures, we report that a value failure occurs when a system output is expected (e.g., after an input or periodically) but the output data does not match specifications, an action failure occurs when the output is not expected at all. An example is the vulnerability report CVE-2020-35514 [96] of Kubernetes, which indicates that OpenShift, a containerization platform, fails to enforce restrictive write access policy for the Kubernetes *kubeconfig* file thus allowing an illegal modification (i.e., the action). Another case is Docker vulnerability CVE-2020-8564, which indicates that registry credentials are written into log files (i.e., the action) when Docker is configured with logging level 4.
- *Network.* Network failures concern any aspect of the network. Since networking components follow dedicated protocols, network failures (i.e., failing to comply with the protocol) are unlikely to belong to any category described above; for this reason, we introduced a specific category. An example is provided by the Kubernetes vulnerability report CVE-2020-8558 [98]; it describes a case in which services bound to the loopback address are accessible by other pods and contain-

ers on the local LAN network. Any other category different than *network failure* would not clearly capture the characteristics of such a failure.

The *detectability* attribute characterizes the difficulty of detecting the failure. Following Gazzola et al., we consider the categories *signaled*, *unhandled*, and *silent*. From the work of Gazzola et al., we exclude *self-healed* since Edge systems do not include any self-healing feature for security issues.

- *Signalled.* It concerns cases in which the system prompts an error message. This could primarily happen when an application encounters memory errors, prompting the user with an error message and asking for further actions. An example case is the KubeEdge GitHub report #2362, which indicates that the Edge device prompts an error because it is unable to connect with the Cloud through its API.
- *Unhandled.* A failure that the Edge system does not handle and that leads to a crash. The system does not detect the failure, while the user detects the uncontrolled crash of the application. An example is the GitHub issue #335 [144] of Zetta, which is about a memory overflow leading to a crash. It occurs when a dependency request is installed before the handler process starts, it leads to a slow but continuous memory consumption resulting in a crash.
- *Silent* A security failure that is not detected; consequently, the system operates with wrong parameters and values thus producing undesirable behaviors and output. This is the case of failures that are observable (e.g., the person who reported the bug was capable of observing them) but not automatically reported by the system as such (e.g., because implementing the logic to automatically determine if the system fails is not feasible since it relates to the oracle problem in software testing [12, 75]). An example is provided by the Kubernetes vulnerability report CVE-2020-8563 [100], which indicates that with logging level set to 4, the credentials of the vsphere controller are written into the controller log file as plain text. Only an end-user inspecting the log may notice such a security failure.

3.2.4 $\boldsymbol{RQ}_4$: What is the nature of the precondition enabling the attacker to exploit Edge vulnerabilities?

To address this research question, for each vulnerability, we keep track of the type of precondition that shall hold to enable exploiting the vulnerability, based on the description appearing in the vulnerability report. We identified the following categories:

- *Data.* What brings the system into a vulnerable state is a specific sequence of input data. A example is the vulnerability CVE-2020-15127 [93] presented earlier; it affects a Kubernetes Pod, which may leak passwords to a phishing URI while resolving malicious dependencies during the import of a container. In this case, the data consists of the phishing dependencies inserted by a malicious user.
- *Lack of Data.* What brings the system into the vulnerable state is the lack of an expected input (e.g., a missing initialization of a resource). It differs from *Data* since, in this case, the required data is not provided; in the case of *Data*, instead, the data is provided but with crafted values or in an unexpected order. An example is KubeEdge bug report #2362 [86], which indicates that the end-user cannot connect to the Kubernetes server (availability problem) because no

credentials are shared between the Cloud server and the Edge server. In this case the problem depends on a specific connection command not being automatically executed on the Cloud server.
- *Resource Busy.* It indicates that a required resource cannot be accessed because it is already busy. An example is provided by Kubernetes bug report #1017 [61], which indicates that two different go-routine requests for a resource already in use make the system unavailable.
- *Resource Unavailable.* It indicates that a required resource does not exist in the system. An example is Kubernetes vulnerability CVE-2020-8557 [97], which indicates that the Kubelet Edge device agent fails to manage the storage in a Pod; indeed, increasing the storage consumption may lead to writing data to the configuration files of a Kubelet agent resulting in compromising the Node. In this case, the unavailable resource is the file system storage.
- *System Configuration.* It indicates a misconfiguration of the system. An example is vulnerability CVE-2020-13597 [92] in Calico (a network security solution for containers); if a Pod is configured to work on IPv4 and meanwhile IPv6 is enabled and not being used, a specifically crafted request may cause the Pod to disclose information or cause a DoS.
- *Delay Causing Missing Resource.* It indicates the case in which a delay (e.g., in input, output, or module initialization) causes any resource to be missing (it differs from *Resource Unavailable* since in this case the missing resource is an output of the SUT). An example for such case was presented earlier, it concerns KubeEdge report #1736 [62], which indicates that, during the initialization of the SUT, a Pod tries to allocate storage volume using configuration files that should be created by the Edge-core. If the initialization of the Edge-core is delayed, then the pod is unable to find the configuration files in the directory and ends up with a denial of service.
- *None.* This case indicates that there is no precondition to be satisfied in order to exploit the vulnerability.

3.2.5 $\boldsymbol{RQ}_{5A}$: How many steps are required to exploit an Edge vulnerability?

To answer this research question we determine, by reading the vulnerability report, the number of steps required to exploit the vulnerability, once the system is in the state required to exploit the vulnerability. However, the type of action to be performed depends on the case study subject. Generally, a step is an action that can be described with a simple sentence using terminology that is well-understood in the domain. For example, the sentence *delete the content of the configuration file settings.xml* is a single step even if, in practice, implies opening a file first. For example, the Kubernetes vulnerability CVE-2021-20218 [103] reports a single step, consisting of executing the copy command on the Fabric8 plugin [35]. This step enables a malicious user to share restricted files and folders in the system. The docker vulnerability report CVE-2014-5278 [90], instead, describes a single step which consists of creating a new container with a name already assigned on the host. The vulnerability enables an attacker to intercept commands and control other containers with the same name.

3.2.6 $\mathbf{RQ}_{5B}$: *What is the nature of the input action enabling the attacker to exploit a vulnerability?*

This research question aims to characterize the types of inputs that enable a malicious user to exploit a vulnerability. We rely on the same categories reported for $\mathbf{RQ}_4$. An example concerning the *Data* category is that of Kubernetes vulnerability CVE-2021-21334 [105], which reports that an input request for cloning a container image (the name of the image is the required data) will result into the disclosure of information associated with the container image.

The category *None* should be used when no input is needed to exploit the vulnerability. This may be the case for vulnerabilities leading to the printout of credentials in log files without the need for further inputs from a malicious user.

3.2.7 $\mathbf{RQ}_6$: *What security properties are violated by Edge vulnerabilities?*

We address this research question by determining the security property that is violated when the vulnerability is successfully exploited. We consider availability, confidentiality, and integrity, which are the security properties described in most security standards (see Section 2.4). Concerning integrity, we distinguish between data integrity and system integrity. They are all described in the following:

- *Availability.* An example availability issue appears in KubeEdge bug report #1017 [61], which has been introduced previously. It concerns two go-routines trying to access, concurrently, a same web-socket. As a result, only one of the two routines succeeds; consequently, the availability of the function implemented by the failing routine is compromised.
- *Data Integrity.* Data-integrity restricts our focus on the integrity of the data stored by either the SUT or the environment in which the SUT is working. An example is provided by the Kubernetes vulnerability report CVE-2021-21251 [104]. It concerns the tarutils tool, which is used to extract compressed files. This vulnerability is a zip slip vulnerability, i.e., a vulnerability that enables an attacker to overwrite arbitrary files when the compressed file is packed in a specific manner.
- *System Integrity.* It concerns cases in which exploiting the vulnerability leads to a modification of the configuration of the system. An example is the CVE vulnerability CVE-2020-2211 [94], which concerns the *Jenkins Kubernetes CI/CD* plugin. The YAML parser in the plugin is not configured properly; consequently, it allows the upload of arbitrary file types, which leads to remote code execution therefore compromising the system integrity.
- *Confidentiality.* This category concern vulnerabilities affecting confidentiality. An example is the CVE vulnerability report CVE-2020-8566 [102], which concerns the *Ceph RADOS Block Device (RBD)*. RBD is the Kubernetes component for storage provisioning. When logging level is set to 4, RBD writes sensitive information (i.e., passwords) to the log file in plain text.

Violated security properties are reported also in NVD CVSS attributes (see Section 2); precisely, CVSS attributes capture the impact that a vulnerability has on each security property (i.e., None, Low, High). However, we do not have CVSS IDs for all

the vulnerabilities considered in our study but only for the ones collected from the CVE database. Further, CVSS attributes capture all the security properties that might be affected, which results in multiple security properties being likely violated by each vulnerability; in our analysis, instead, we report only one security property for each vulnerability, which we identify as either the security property that is easier to violate through an exploit (e.g., less steps to perform) or, if multiple properties can be violated with a same simple input, the security property that can be identified as being violated first. For example, the malicious modification of the configuration of the system (system integrity) may result in a Node not responding to requests (availability); in this case, although both system integrity and availability are violated, system integrity is the first property being violated. The reason for our choice is that, with our study, we aim to drive the implementation of software testing tools, which will likely discover scenarios that are short and easy to process; in other words, they will detect violations of security properties that are easier to trigger and report the first security being violated (without waiting for other effects).

3.2.8 $\boldsymbol{RQ_7}$: What faults cause Edge vulnerabilities?

In the following, we present the three different kinds of data collected to address $\mathbf{RQ}_7$:

- $\mathbf{RQ}_{7A}$: *What is the CWE vulnerability type?* We keep track of the CWE IDs associated to each vulnerability report. Although there is no guarantee that every CVE vulnerability report presents a set of CWE IDs capturing the vulnerability type, they are usually reported (for our case study subjects, 89.8% of the vulnerabilities present a CWE ID, see Section 4.7). The vulnerabilities without a CWE ID are not considered to address $\mathbf{RQ}_{7A}$.
- $\mathbf{RQ}_{7B}$: *What are the erronous software behaviours leading to Edge security failures?* For each CWE ID associated to a vulnerability, we inspect the *Research Concept* taxonomy and identify the corresponding pillars.
- $\mathbf{RQ}_{7C}$: *What are the developer mistakes leading to Edge vulnerabilities?* For each CWE ID associated to a vulnerability, we inspect the *Developer Concept* taxonomy and identify the corresponding pillars.

3.2.9 $\boldsymbol{RQ_8}$: How severe are Edge vulnerabilities?

For each CVE vulnerability, we inspect the corresponding entry in the NVD database and keep track of both the *NVD severity score* and the *CVSS entry*.

To discuss severity, we comment on the distribution of CVSS scores; for example, a high median for the CVSS score is a strong motivation for improvement in Edge security testing practices. Also, we report the percentage of vulnerabilities with a high impact on security properties.

To discuss the easiness of attacks, which should lead to easy test automation, we discuss the distribution of CVSS attributes *Attack Complexity* (Low/High) and *Privileges Required* (None/Low/High).

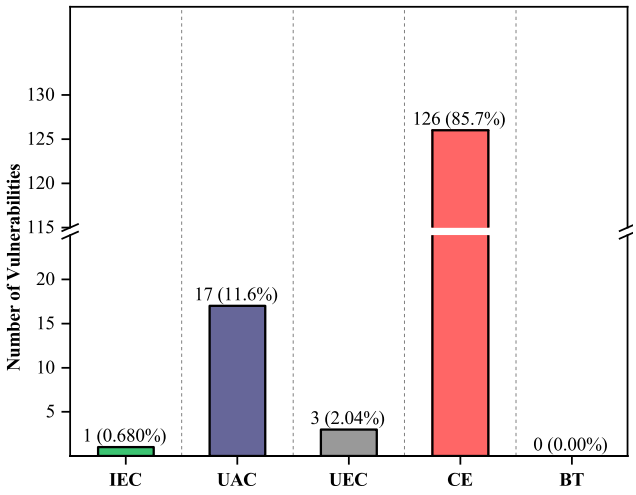


Fig. 4: $\mathbf{RQ}_1$: Why are Edge vulnerabilities not detected during testing?

4 Results

This section presents our findings; each research question (**RQ**) is discussed individually. In this section we do not provide example cases for the vulnerabilities investigated in our study because they are already described in Section 3.2; the reader can also refer to Table 3 to search for example cases through the manuscript.

For $\mathbf{RQ}_1$ to $\mathbf{RQ}_{5A}$, we also compare our results with those of Gazzola et al. [41]; $\mathbf{RQ}_4$ to $\mathbf{RQ}_8$, instead, were not studied by Gazzola et al.

4.1 $\mathbf{RQ}_1$: Why are Edge vulnerabilities not detected during testing?

Figure 4 presents our findings[4]; Combinatorial Explosion (CE) represents 85.7% of the vulnerabilities in our analysis, whereas Unknown Application Condition (UAC), Unknown Environment Condition (UEC), Irreproducible Execution Condition (IEC) and Bad Testing (BT) cover the 11.6%, 2.04%, 0.68%, and 0% of the cases, respectively. CE is the main reason for vulnerabilities not being detected at testing time (126 vulnerabilities), which is unsurprising given the complexity of Edge systems. Indeed, Edge systems are large and process inputs of different natures (e.g., Web forms, configuration files, network data). The second category is UAC, which indicates lack of appropriate documentation and might be due to the open source nature of our case study subjects. However, note that the development of most of our case study subjects is supported by professional software development companies and are used by many businesses (see Section 5.4), which minimizes this threat. Indeed, vulnerabilities due to bad documentation are low (i.e., only 11.6%). Similarly, UEC may be low because

[4] Please note that, to save space, in the barcharts appearing in Figure 4, we hid part of the Y-axis scale; the hidden part is highlighted with the symbol / /.

all the environment components (e.g., the OS) are widely used and well documented. Finally, in our analysis we encountered only one occurrence of IEC and no BT cases.

The trend for $\mathbf{RQ}_1$ is similar to the one observed for functional failures by Gazzola et al., except that UEC was ranked second in their study and we do not observe any bad testing case. In the study of Gazzola et al. the proportions observed for IEC, UAC, UEC, CE, and BT are 1.68%, 5.04%, 12.60%, 50.42%, and 30.25%, respectively. The larger proportion of UAC in our context is likely due to the complexity of Edge frameworks; indeed, the desktop applications considered by Gazzola et al. present a lower number of inputs, features, and configuration options that the Edge frameworks considered in our study. We believe that the likelihood of finding badly documented features is larger when the number of components and configuration options is large. Instead the lack of BT cases might be due to the fact that our case study subjects are software components with several years of development (e.g., KubeEdge is based on Kubernetes) during which trivial security issues slipping through the test process had been already detected. IEC cases, by definition, are expected to be limited in number; indeed, software inputs and environment conditions tend to be reproducible in the development environment.

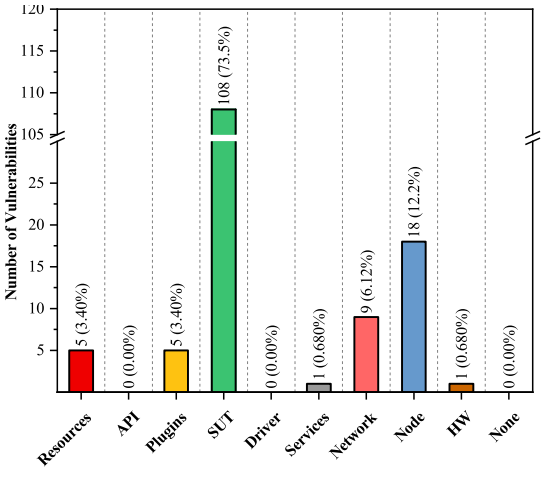
4.2 $\mathbf{RQ}_2$: What are the types of components involved in a security failure?

Figure 5 provides the distribution of each Edge component for the different sub-questions of $\mathbf{RQ}_2$. For all the sub-questions, SUT is the element with the highest number of entries, which is expected since we collected vulnerability reports concerning the SUT. However, the distribution of Edge components vary based on the sub-question considered, which indicate that Edge components are interlaced in cause-effect chains.

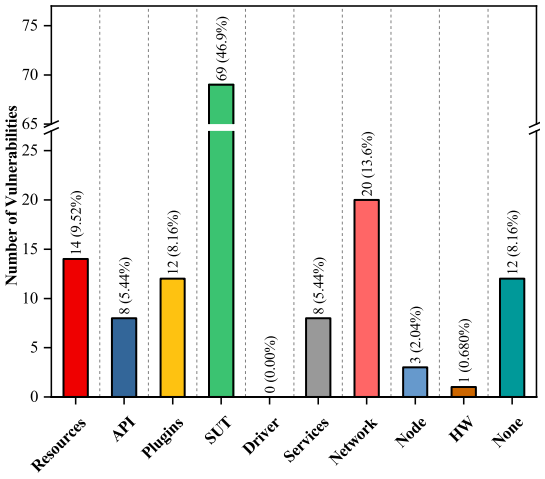
The vulnerable component ($\mathbf{RQ}_{2D}$) is generally the SUT (93 cases); however, (misconfigured) Network and Plugins are the second and third cause of security failures. We did not observe any vulnerability in Drivers and Nodes.

The consequences of vulnerabilities ($\mathbf{RQ}_{2A}$) mainly affect the SUT (108 cases) but also Nodes, Network, Plugins, Resources, and Services. The distribution for $\mathbf{RQ}_{2A}$ is different than the distribution observed for $\mathbf{RQ}_{2D}$; indeed, for $\mathbf{RQ}_{2A}$, we observe a larger number of SUT and Node cases along with a lower number for Plugins and Network. Such difference mainly depends on (i) Plugin faults impacting on the SUT and (ii) Network faults impacting on both Nodes and SUT.

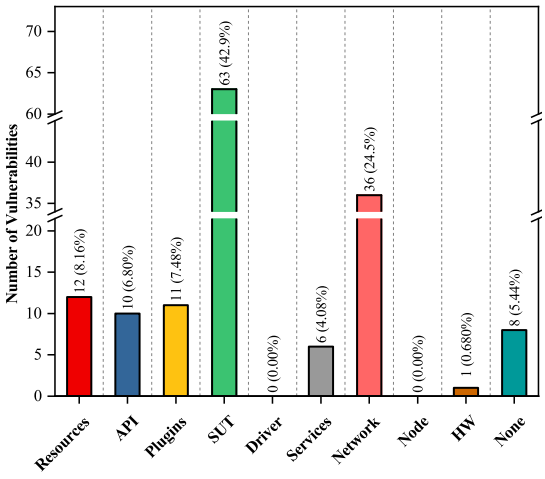
Concerning $\mathbf{RQ}_{2B}$, most of the vulnerabilities can be exploited (and, consequently, detected at testing time) only if some precondition holds, which is likely the reason why they are not detected at testing time (i.e., it is more difficult to spot a vulnerability if the system needs to be in a specific state). In 69 out of 147 cases, it is the SUT what needs to be in a specific state, which is often a specific configuration (e.g., vulnerability CVE-2020-8563, which requires the logging level to be set to 4). However, preconditions for exploitability may depend also on all the other components, except Drivers. The second and third components presenting preconditions for exploiting a vulnerability are Network and Resources, which are the primary means to provide inputs to Edge systems.



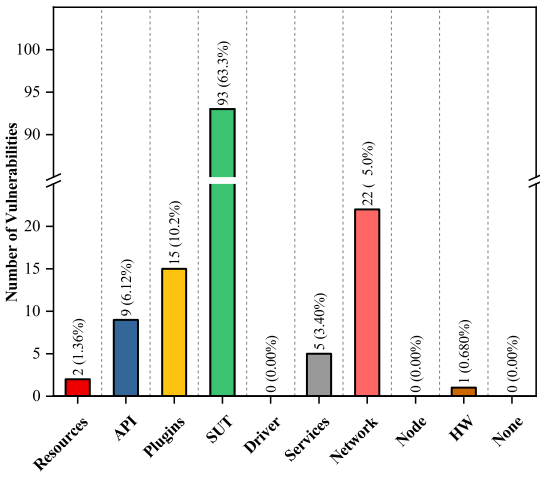
(a) **RQ**$_{2A}$: What are the components manifesting an Edge security failure?



(b) **RQ**$_{2B}$: What are the components that are in the state required to exploit an Edge vulnerability?



(c) **RQ**$_{2C}$: What are the components that receive the inputs that trigger an Edge vulnerability?



(d) **RQ**$_{2D}$: What are the faulty (i.e., vulnerable) Edge components?

Fig. 5: **RQ**$_2$: What are the types of components involved in a security failure?

As for $\mathbf{RQ}_{2C}$, the component that receives the largest number of inputs triggering a vulnerability is the SUT (63 cases), followed by Network (36), Resources (12), Plugins (11), APIs (10), Services (6), and Hardware (1). Unsurprisingly the SUT interface (e.g., Web page, Service, or API), which is the usual entry point for managing an Edge system, is the component with the largest number of entries. The other components, instead, follow the relevance of each input interface for the services provided by the SUT (i.e., Network is clearly more relevant than all the other components, which have the same importance). In eight cases, no input needs to be received by the software (see *None*); these are vulnerabilities related to logging, where the SUT periodically writes sensible information in log files.

A precise comparison with the results obtained by Gazzola et al. is complicated by the fact that their study does not separate $\mathbf{RQ}_2$ into four subquestions and considers a smaller set of component types. The main difference is that Resource was the component type mostly involved in failures (50%), while Plugins (3%), Services (6%), and Network (1%) had a more limited involvement. OS concerned 20% of the cases; OS cases had never been observd in our study (see Section 3.2.2). In our work, instead, we explicitly model the case of the SUT, which was ignored in the work of Gazzola et al.; concerning the other elements, the ones mostly involved in security failures, if we compute the average of the four RQs, are Network (14.8%), Plugins (7.31%), and Resources (5.61%). The difference in their distribution with respect to the work of Gazzola et al. is mostly due to the different nature of our context (i.e., networked Edge components instead of desktop applications).

4.3 $\mathbf{RQ}_3$: What kind of failures are observed when an Edge vulnerability is exploited?

Figure 6-A and -B present the distribution of the different types of security failures (left) and their detectability (right).

Concerning *failure type*, most of the vulnerabilities lead to Value failures (57 cases, 38.8%), which is expected since they include the effect of both authorization and integrity issues (see Section 3.2.7). The second frequent type of failures are Network failures (39 cases, 26.5%), which is expected since Edge frameworks mainly control devices over the network. Action failures are high (33 cases, 22.4%) because several vulnerabilities make the software perform illegal actions. System failures (usually crashes) are low (17 cases, 11.6%), likely because of the availability of static code analysis tools aiming at detecting such problems [54]. Timing failures (early or delayed response) are the lowest (one case, 0.68%).

Concerning *detectability*, the largest proportion of vulnerabilities (i.e., 127, 86.4%) leads to Silent failures, which is expected since this is the effect of a wide range of vulnerabilities, from authorization problems (e.g, letting malicious users to access private resources) to integrity ones (e.g., altering the content of a database). With much less entries, the second category is Unhandled failures (i.e., 11, 7.48%). Signalled failures have the least occurrences (i.e., 9, 6.12%), which is expected since it is difficult for an engineer to implement features capable of detecting the effect of vulnerabilities (e.g., functions that trigger an alarm in the presence of anomalous

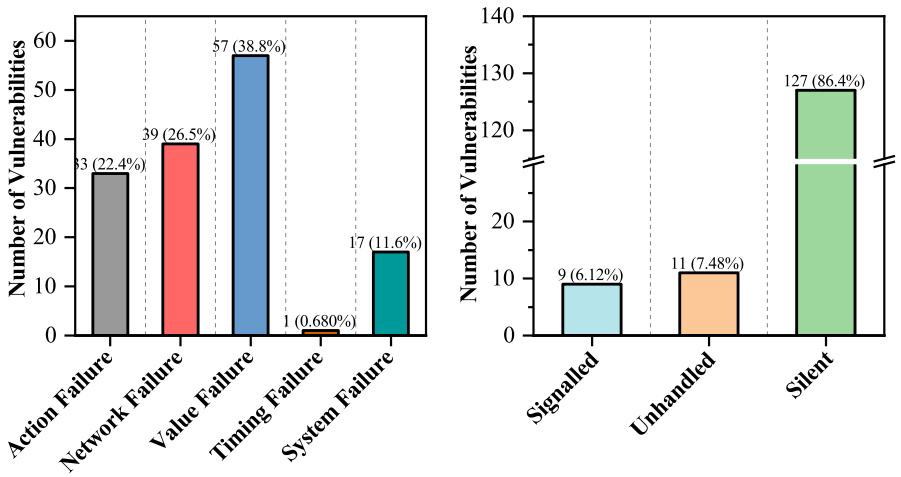


Fig. 6: $\mathbf{RQ}_3$: What kind of failures are observed when an Edge vulnerability is exploited?

data); only security failures leading to the lack of communication or causing memory allocation errors can be easily detected (see Section 3.2.3).

For both *failure type* and *detectability*, excluding cases introduced in our study (i.e., action failures and network failures), we observe the same rankings reported by Gazzola et al. In both the two studies, the ranking for failure type is (1st) Value failures, (2nd) System failures, and (3rd) Timing failures. For detectability, the ranking is (1st) Silent, (2nd) Unhandled, and (3rd) Signalled. However the distribution of the vulnerabilities for each ranked category differs across the two studies. Indeed, system failures are more frequent in desktop applications (33.7%, based on Gazzola et al. [41]) than in Edge systems (11.6%), possibly because Edge frameworks are more robust. Also, Value failures are more frequent in the study of Gazzola et al. (i.e., 61.5% VS 38.8%), possibly because in our study we observe also Action failures and Network failures (i.e., the total number of vulnerabilities is distributed across a larger number of categories). Silent failures, instead, are more frequent in Edge frameworks (our study, 86.4%) than in Gazzola et al. (i.e., 53%), likely because they reflect the effect of failure types not observed with desktop applications (i.e., Action and Network failures).

4.4 $\mathbf{RQ}_4$: What is the nature of the precondition enabling the attacker to exploit Edge vulnerabilities?

Figure 7 shows our results. In most of the cases (i.e., 92, 62.6%), the vulnerability can be exploited only if the system is in a specific configuration, which is expected since

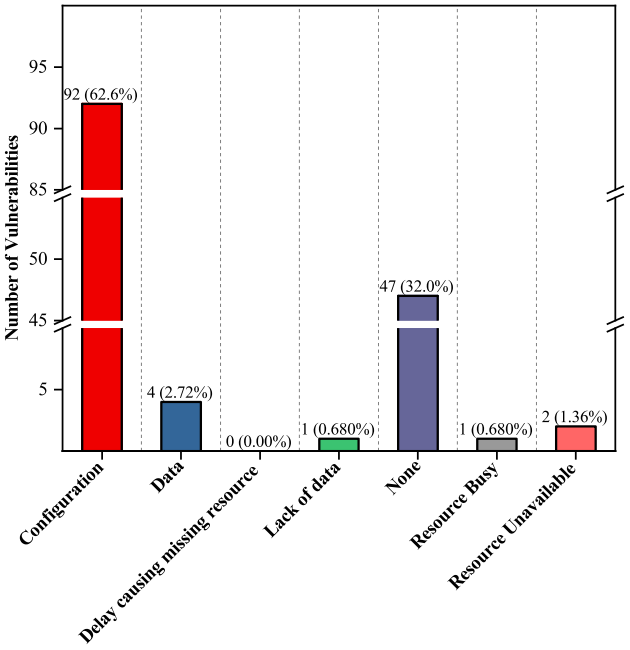


Fig. 7: $\mathbf{RQ}_4$: What is the nature of the precondition enabling the attacker to exploit Edge vulnerabilities?

Edge systems consist of many components that can be installed on different devices and require to be tuned according to the device characteristics and the service needs[5].

The second most frequent case, with 47 cases (32%), is the absence of any precondition to be fulfilled in order to exploit the vulnerability, which indicates that any configuration of the system exposes the vulnerability; this is not surprising since it is a sort of base case. Only few other vulnerabilities (eight in total) concern the other four cases (i.e., Data, Lack of data, Resource busy, Resource unavailable).

4.5 $\mathbf{RQ}_5$: What inputs enable exploiting Edge vulnerabilities?

Figure 8 presents the distribution of the number of steps required to exploit a vulnerability ($\mathbf{RQ}_{5A}$). We were unable to determine the number of steps required to exploit 20 vulnerabilities out of 147 because of the lack of detailed descriptions in the vulnerability reports and attached documents. For 119 vulnerabilities (81%), one step is sufficient to exploit the system (see Section 3.2.5 for examples), whereas two steps are required only in the case of three vulnerabilities. In eight cases (5.44%), no step is required to observe the effect of the vulnerability, they match the eight

[5] Please note that dedicated static analysis tools had been developed to simplify the configuration of Kubernetes [133]; therefore, we may expect that testing its security properties for all the available configurations is particularly challenging and error prone.

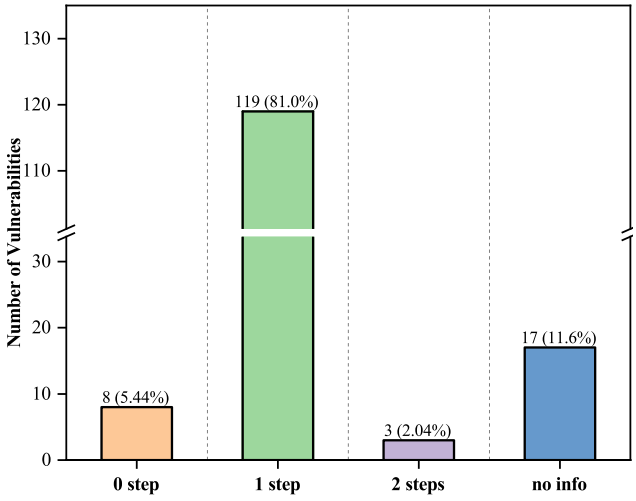


Fig. 8: $\mathbf{RQ}_{5A}$: How many steps are required to exploit an Edge vulnerability?

logging vulnerabilities reported in $\mathbf{RQ}_{2C}$ (i.e., the system periodically logs sensitive information).

The study of Gazzola et al., instead, reported a larger number of steps required to trigger a failure (median is two and 35% of the failures require at least three steps). We believe that this is mainly due to the nature of the software under analysis (e.g., desktop applications are more interactive than Edge frameworks).

Figure 9 provides the distribution of the input action types for the vulnerabilities considered in our study ($\mathbf{RQ}_{5B}$). The category with the largest number of entries is Data, which concerns any input provided to the software under test or its components. This is expected because Edge systems, like most software systems, generate outputs based on the data received as input; therefore, vulnerabilities are exploited by providing specific or crafted inputs to the system. Instead, only a few vulnerabilities (i.e., nine, 6.12%) can be exploited by changing a configuration file, which is expected since configuration files are generally not the main mean for end-users to interact with the system.

The other cases (i.e., Lack of data, Resource busy, Resource unavailable) are less frequent, possibly because they concern corner cases which may be difficult to spot (e.g., our case study subjects might be vulnerable but the vulnerabilities have not been discovered yet).

Unsurprinsingly, the eight cases not requiring any input (i.e., None) match the eight zero-step cases reported for $\mathbf{RQ}_{5A}$. These are the cases in which the SUT provides sensible information (e.g., login credentials) in log files, periodically.

In general, we can conclude that the inputs required to exploit an Edge vulnerability are simple. Indeed, most of the times one step is sufficient and the action to perform is about providing specific data values to the system.

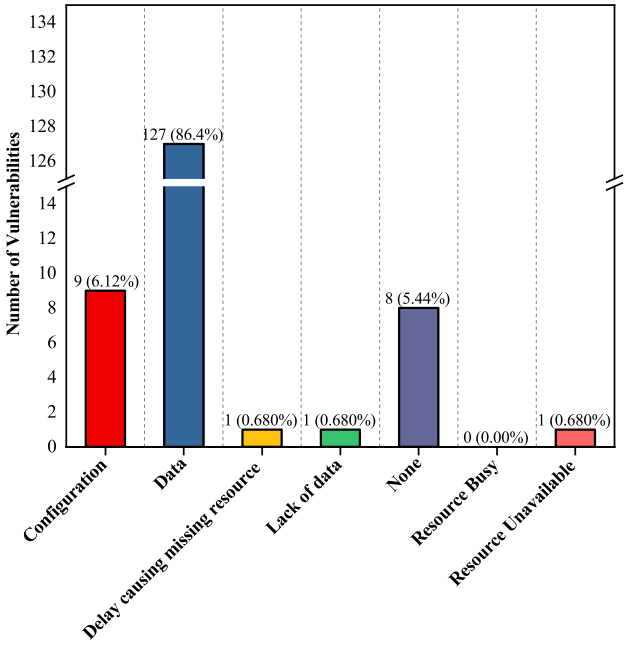


Fig. 9: **RQ**$_{5B}$: What is the nature of the input action enabling the attacker to exploit a vulnerability?

4.6 **RQ**$_6$: What security properties are violated by Edge vulnerabilities?

Figure 10 provides the distribution of security properties being violated by Edge vulnerabilities. Confidentiality has the highest number of occurrences in our study (81, 55.1%), whereas system integrity is the second most violated security property with 42 occurrences (28.6%). Data integrity and availability are observed with 7 (4.8%) and 17 (11.6%) occurrences, respectively.

Figure 11 provides the number of vulnerabilities affecting each security property, according to the NVD CVSS entries. For each security property, we report the number of vulnerabilities with High or Low impact on it. Also, we report the Total number of vulnerabilities concerning each security property. If we focus on the total number of vulnerabilities, we can notice that the ranking does not differ from the ones in Figure 10 (i.e., confidentiality is followed by integrity, while availability has the lowest number of cases) but the magnitude of the differences varies a lot. Indeed, based on CVSS data it is difficult to draw any conclusion (i.e., their difference is not significant, as reported in Section 5). Instead, by focusing on the vulnerability that is easier to exploit or that is violated first (i.e., our criteria, see Section 3.2.7), we can observe that Confidentiality is the security property that is more likely affected by vulnerabilities (Figure 10), which provides a clear direction for the development of test automation tools.

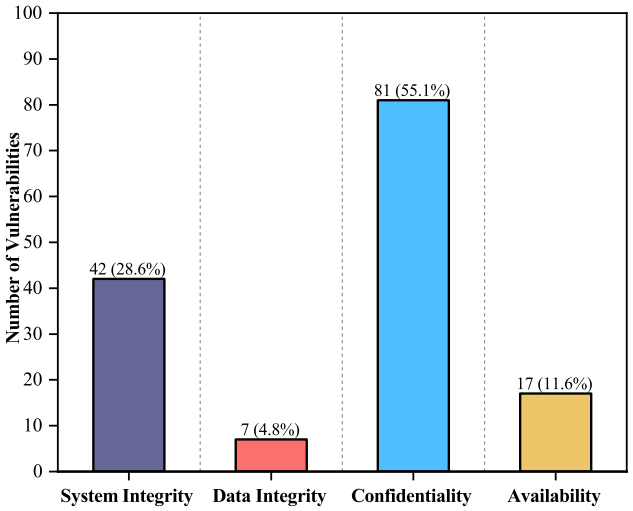


Fig. 10: $\mathbf{RQ}_6$: What security properties are violated by Edge vulnerabilities?

We verified that, for all the vulnerabilities, the security property that we selected matches one of the security properties reported by CVSS with the highest score (lower scores indicate that a security property violation is less noticeable). Such condition is particularly important in our context because testing tools should target the vulnerability with the highest impact, otherwise results might be perceived as irrelevant by end-users.

4.7 $\mathbf{RQ}_7$: What faults cause Edge vulnerabilities?

Figure 12, Figure 13, and Table 4, provide the distribution of *CWE developer concepts* (i.e., developer mistakes, collected to address $\mathbf{RQ}_{7C}$), *CWE Research Concepts pillars* (i.e., erroneous software behaviors due to the vulnerability, collected to address $\mathbf{RQ}_{7B}$), and CWE IDs (i.e., fault types, collected to address $\mathbf{RQ}_{7A}$), respectively. The CWE IDs for the CWE Research Concepts pillars are reported in Table 5. In the following, we first discuss the distribution of the most frequent developer mistakes and erroneous software behaviours, which helps prioritizing the target (input generation strategy) of security testing; we then discuss the distribution of CWE IDs, which helps understanding why testing practices in place aren't sufficient.

Our plots do not cover all the vulnerabilities in our study because of the limited information that can be retrieved from bug reports and CWE views. Precisely, among the 147 vulnerabilities in our study, 60 (40.8%) do not have an associated CWE developer concept. However, although the proportion of vulnerabilities with a CWE developer concept is contained, the proportion of vulnerabilities with CWE IDs and CWE research concepts is high; indeed, 132 out of 147 vulnerabilities (89.8%) have a CWE-ID assigned to them and 130 out of 147 vulnerabilities (88.4%) can be as-

Table 4: $\mathbf{RQ}_{7A}$: What is the CWE vulnerability type?We report the number of vulnerabilities belonging to each vulnerability type discovered in our investigation.

Occurrencies	CWE Number	Description
34	CWE-306	Missing Authentication for Critical Function
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory (‘Path Traversal’)
7	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor
6	CWE-532	Insertion of Sensitive Information into Log File
5	CWE-20	Improper Input Validation
4	CWE-400	Uncontrolled Resource Consumption
4	CWE-269	Improper Privilege Management
4	CWE-668	Exposure of Resource to Wrong Sphere
3	CWE-502	Deserialization of Untrusted Data
3	CWE-284	Improper Access Control
3	CWE-209	Generation of Error Message Containing Sensitive Information
3	CWE-94	Improper Control of Generation of Code (Code Injection)
3	CWE-918	Server-Side Request Forgery (SSRF)
3	CWE-770	Allocation of Resources Without Limits or Throttling
3	CWE-522	Insufficiently Protected Credentials
3	CWE-863	Incorrect Authorization
3	CWE-266	Incorrect Privilege Assignment
3	CWE-862	Missing Authorization
3	CWE-601	URL Redirection to Untrusted Site (’Open Redirect’)
3	CWE-250	Execution with Unnecessary Privileges
2	CWE-79	Improper Neutralization of Input During Web Page Generation (’Cross-site Scripting’)
2	CWE-295	Improper Certificate Validation
2	CWE-610	Externally Controlled Reference to a Resource in Another Sphere
2	CWE-78	Improper Neutralization of Special Elements used in an OS Command (’OS Command Injection’)
2	CWE-59	Improper Link Resolution Before File Access (’Link Following’)
2	CWE-476	NULL Pointer Dereference
2	CWE-789	Memory Allocation with Excessive Size Value
2	CWE-74	Improper Neutralization of Special Elements in Output Used by a Downstream Component (’Injection’)
1	CWE-669	Incorrect Resource Transfer Between Spheres
1	CWE-732	Incorrect Permission Assignment for Critical Resource
1	CWE-283	Unverified Ownership
1	CWE-23	Relative Path Traversal
1	CWE-287	Improper Authentication
1	CWE-420	Unprotected Alternate Channel
1	CWE-184	Incomplete List of Disallowed Inputs
1	CWE-798	Use of Hard-coded Credentials
1	CWE-312	Cleartext Storage of Sensitive Information
1	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
1	CWE-335	Incorrect Usage of Seeds in Pseudo-Random Number Generator (PRNG)
1	CWE-201	Insertion of Sensitive Information Into Sent Data
1	CWE-300	Channel Accessible by Non-Endpoint
1	CWE-434	Unrestricted Upload of File with Dangerous Type
1	CWE-1050	Excessive Platform Resource Consumption within a Loop
1	CWE-552	Files or Directories Accessible to External Parties
1	CWE-73	External Control of File Name or Path
1	CWE-372	Incomplete Internal State Distinction
1	CWE-61	UNIX Symbolic Link (Symlink) Following
1	CWE-215	Insertion of Sensitive Information Into Debugging Code
1	CWE-416	Use After Free
1	CWE-270	Privilege Context Switching Error
1	CWE-24	Path Traversal
1	CWE-401	Missing Release of Memory after Effective Lifetime
1	CWE-441	Unintended Proxy or Intermediary (’Confused Deputy’)
1	CWE-755	Improper Handling of Exceptional Conditions

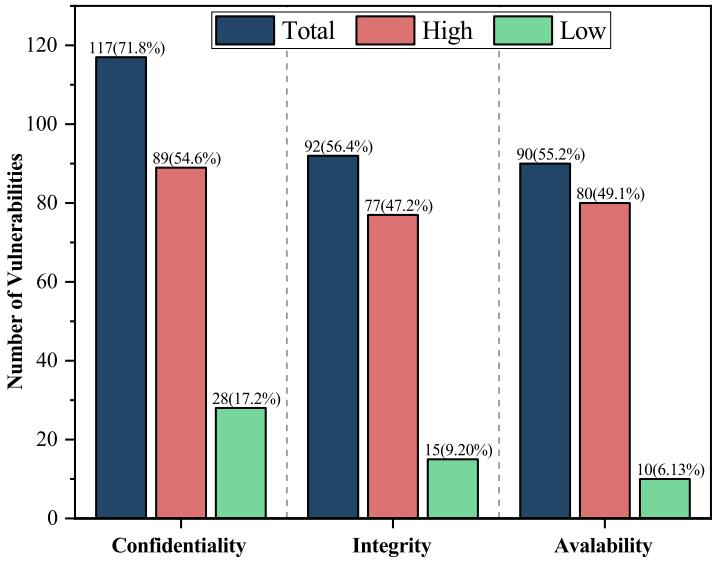


Fig. 11: $\mathbf{RQ}_6$: Number of vulnerabilities affecting each security property based on NVD's CVSS entries; in total (Total) and grouped by impact (High/Low)

sociated to a CWE research concept. Further, the results for $\mathbf{RQ}_{7C}$ are in line with those for $\mathbf{RQ}_{7B}$ and $\mathbf{RQ}_{7A}$ (see following paragraphs); therefore, our observations should hold for almost the whole set of vulnerabilities considered.

Although we analyzed 147 vulnerabilities in our study, the total number of research concepts appearing in Figure 13 is 160. Such difference depends on some vulnerabilities having more than one research concept associated to them (i.e., the software may behave in different invalid ways because of the vulnerability).

RQ$_{7C}$. By looking at the distribution of developer mistakes (Figure 12), we can observe that most of the vulnerabilities in the study are associated with the authentication mechanism (37 observations, 33.94%). Such result is in line with what observable from Table 4. Indeed, CWE-306 (Missing Authentication for Critical Function) has the largest number of occurrences; since CWE-306 concerns authorization to perform an action or access data, our finding is also line with $\mathbf{RQ}_6$ results (i.e., vulnerabilities concern Confidentiality, that is, users accessing data they are not authorized to access). More in general, still in line with the prevalence of confidentiality issues and authentication mechanism mistakes, we can observe that 42.6% of all the vulnerabilities with a CWE ID are related to Access Control[6]

The second place in the ranking provided by Figure 12 is taken by *Information management errors*, which have been observed 15 times (13.76%). Such observation is reflected in Table 4; indeed, Information management errors relate to the control

[6] CWE IDs related to Access Control are CWE-306, CWE-863, CWE-552, CWE-798, CWE-372, CWE-862, CWE- 269, CWE-266, CWE-283, CWE-250, CWE-532, CWE-732, CWE-522, CWE-287, CWE-420, CWE-284, CWE-300, CWE-295, CWE-270.

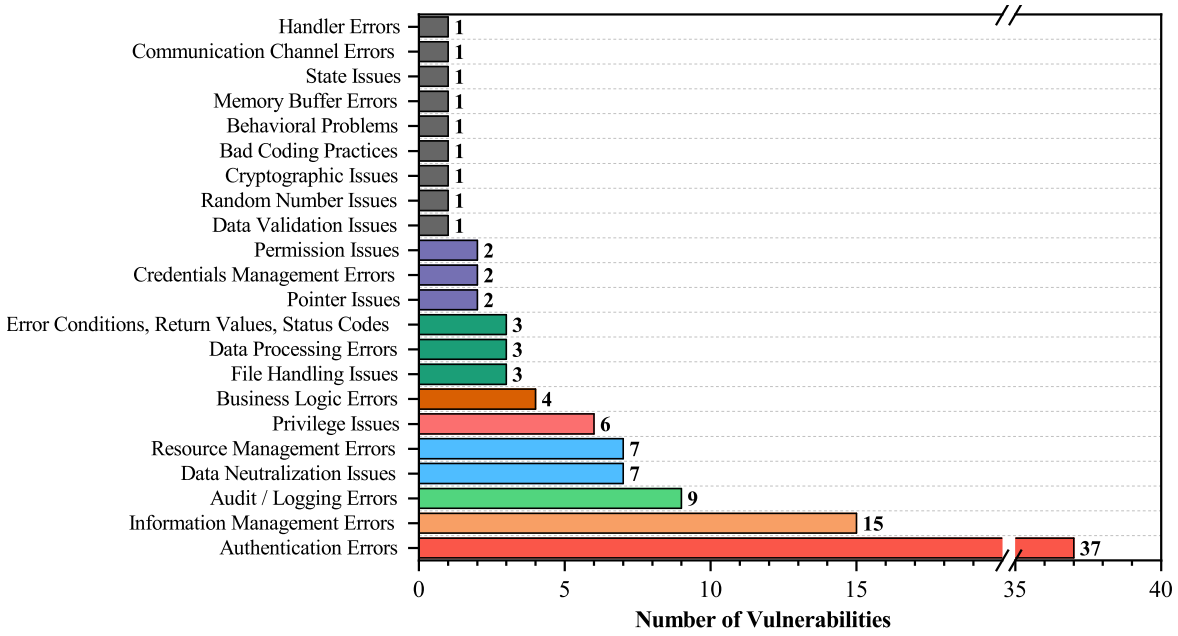


Fig. 12: **RQ**$_{7C}$: What are the developer mistakes leading to Edge vulnerabilities?

of resources, which concerns 52% of all the vulnerabilities with a CWE ID[7]. Among such CWE IDs, CWE-22 (Improper Limitation of a path name to a Restricted Directory) is ranked second in Table 4 with eight occurrences. The prevalence of vulnerabilities concerning the control of resources likely depends on the fact that Edge systems, especially the Edge controller, often manage files. Although some vulnerabilities about control of resources (i.e., path traversal vulnerabilities CWE-22 and CWE-24) can be detected by Web testing tools such as BurpSuite or OWASP Zap, the vulnerabilities considered in our analysis concern complex features, which are not fully supported by these tools. For example, path traversal is often the result of the extraction of a compressed file.

Logging errors, data neutralization, resource management errors, and privileges issues have been observed nine, seven, seven, and six times respectively.

RQ$_{7B}$. Figure 13 shows that the research concepts with the highest number of vulnerabilities are CWE-664 (Improper Control of a Resource Through its Lifetime) and CWE-284 (Improper Access Control) with 64 and 62 vulnerabilities, respectively, which is in line with our discussion above.

[7] CWE IDs related to the control of resources are CWE-22, CWE-863, CWE-552, CWE-312, CWE-434, CWE-372, CWE-601, CWE-184, CWE-94, CWE-610, CWE-441, CWE-200, CWE-22, CWE-668, CWE-74, CWE-23, CWE-20, CWE-24, CWE-250, CWE-502, CWE-532, CWE-669, CWE-732, CWE-522, CWE-73, CWE-400, CWE-209, CWE-918, CWE-201, CWE-1050, CWE-770, CWE-789, CWE-59, CWE-61, CWE-215, CWE-416, CWE-401.

CWE-707 (Improper Neutralization) is the third most frequent case (15 vulnerabilities), in line with the number of data-integrity issues ($\mathbf{RQ}_6$) and data neutralization mistakes (ranked fourth in the discussion for $\mathbf{RQ}_{7C}$, above), which are often caused by code injection or path traversal vulnerabilities. For example, the path traversal vulnerabilities reported in Section 3.2.7 can be exploited because the content of zip files is not verified.

CWE-703 (Improper Check or Handling of Exceptional Conditions) and CWE-693 (Protection Mechanism Failure) often lead to system crashes; indeed, they are often causing availability issues. CWE-710 (Improper Adherence to Coding Standards), CWE-691 (Insufficient Control Flow Management), and CWE-697 (Incorrect Comparison) are related to the quality of the software development procedures in place.

RQ$_{7A}$. Table 4 provides the detailed distribution of CWE IDs for our case study. Except for CWE-306, all the CWE IDs are assigned to less than ten vulnerabilities (median is two vulnerabilities for each CWE ID), which indicates that vulnerabilities are spread across vulnerability types and this may be a consequence of the large number of features implemented by Edge systems.

In addition to CWE-306 and CWE-22, already discussed above (see Paragraph $\mathbf{RQ}_{7C}$), other frequent CWE IDs are CWE-200, CWE-532, and CWE-20, which have been reported with 7, 6, and 5 occurrences in our results. CWE-532 and CWE-20 concern input neutralization issues (CWE-94, CWE-22, CWE-74, CWE-20, CWE-24, CWE-250, CWE-918, CWE-770, CWE-789, CWE-215, CWE-78, CWE-79, 14% of all the vulnerabilities with a CWE ID) and leakage of sensitive data (CWE-532, CWE-201, CWE-215, CWE-312, and CWE-209, 14%). Input neutralization issues can be detected using a wide range of tools (e.g., Metasploit [83] or Acunetix [48]); however, for the Edge systems under study, these vulnerabilities were not detected because they require the system to be in a specific state, which complicates testing. Some solutions for detecting data leakage exist [127]; however, they are mainly research prototypes, which is the reason why such vulnerabilities are not detected at development time. Leakage of sensitive data relates to the logging errors reported for $\mathbf{RQ}_{7C}$.

Memory issues are limited in number (i.e., nine, considering CWE-476, CWE-789, CWE-416, CWE-401, CWE-770); although some of these memory issues might be detected by means of static code analysis tools such as SonarQube [128] (it covers CWE-476, CWE-401, and CWE-416), we believe that they are not detected because they concern components implemented with the go-lang programming language [42], for which a limited set of static analysis tools are available [4, 45, 129]. Some cases concern bad coding practices (i.e., CWE-335, CWE-327, CWE-798, CWE-755). Tools like SonarQube may still help in identifying some of them (i.e., CWE-798, CWE-327, CWE-755); however, rules for the Go programming language are limited.

Table 5: Description of CWE Research Concepts (i.e., the erroneous software behaviours leading to security failures)

CWE: Research Concept	CWE
CWE-664: Improper Control of a Resource Through its Lifetime	CWE-283: Unverified Ownership
	CWE-863: Incorrect Authorization
	CWE-552: Files or Directories Accessible to External Parties
	CWE-798: Use of Hard-coded Credentials
	CWE-372: Incomplete Internal State Distinction
	CWE-862: Missing Authorization
	CWE-269: Improper Privilege Management
	CWE-266: Incorrect Privilege Assignment
	CWE-306: Missing Authentication for Critical Function
	CWE-295: Improper Certificate Validation
	CWE-250: Execution with Unnecessary Privileges
	CWE-532: Insertion of Sensitive Information into Log File
	CWE-732: Incorrect Permission Assignment for Critical Resource
	CWE-522: Insufficiently Protected Credentials
	CWE-287: Improper Authentication
	CWE-420: Unprotected Alternate Channel
	CWE-284: Improper Access Control
	CWE-300: Channel Accessible by Non-Endpoint
	CWE-270: Privilege Context Switching Error
CWE-284: Improper Access Control	CWE-863: Incorrect Authorization
	CWE-552: Files or Directories Accessible to External Parties
	CWE-798: Use of Hard-coded Credentials
	CWE-372: Incomplete Internal State Distinction
	CWE-862: Missing Authorization
	CWE-269: Improper Privilege Management
	CWE-266: Incorrect Privilege Assignment
	CWE-306: Missing Authentication for Critical Function
	CWE-283: Unverified Ownership
	CWE-532: Insertion of Sensitive Information into Log File
	CWE-732: Incorrect Permission Assignment for Critical Resource
	CWE-522: Insufficiently Protected Credentials
	CWE-287: Improper Authentication
	CWE-420: Unprotected Alternate Channel
	CWE-284: Improper Access Control
	CWE-300: Channel Accessible by Non-Endpoint
	CWE-270: Privilege Context Switching Error
	CWE-395: Use of NullPointerException Catch to Detect NULL Pointer Dereference
CWE-707: Improper Neutralization	CWE-94: Improper Control of Generation of Code ('Code Injection')
	CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
	CWE-74: Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')
	CWE-20: Improper Input Validation
	CWE-24: Path Traversal: '../filedir'
	CWE-250: Execution with Unnecessary Privileges
	CWE-918: Server-Side Request Forgery (SSRF)
	CWE-770: Allocation of Resources Without Limits or Throttling
	CWE-789: Memory Allocation with Excessive Size Value
	CWE-215: Insertion of Sensitive Information Into Debugging Code
	CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
	CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-703: Improper Check or Handling of Exceptional Conditions	CWE-476: NULL Pointer Dereference
	CWE-209: Generation of Error Message Containing Sensitive Information
	CWE-755: Improper Handling of Exceptional Conditions
CWE-693: Protection Mechanism Failure	CWE-327: Use of a Broken or Risky Cryptographic Algorithm
	CWE-312: Cleartext Storage of Sensitive Information
	CWE-798: Use of Hard-coded Credentials
	CWE-601: URL Redirection to Untrusted Site ('Open Redirect')
	CWE-184: Incomplete List of Disallowed Inputs
	CWE-335: Incorrect Usage of Seeds in Pseudo-Random Number Generator (PRNG)
CWE-710: Improper Adherence to Coding Standards	CWE-798: Use of Hard-coded Credentials
	CWE-476: NULL Pointer Dereference
	CWE-250: Execution with Unnecessary Privileges
CWE-691: Insufficient Control Flow Management	CWE-94: Improper Control of Generation of Code ('Code Injection')
	CWE-918: Server-Side Request Forgery (SSRF)
CWE-697: Incorrect Comparison	CWE-601: URL Redirection to Untrusted Site ('Open Redirect')
	CWE-184: Incomplete List of Disallowed Inputs

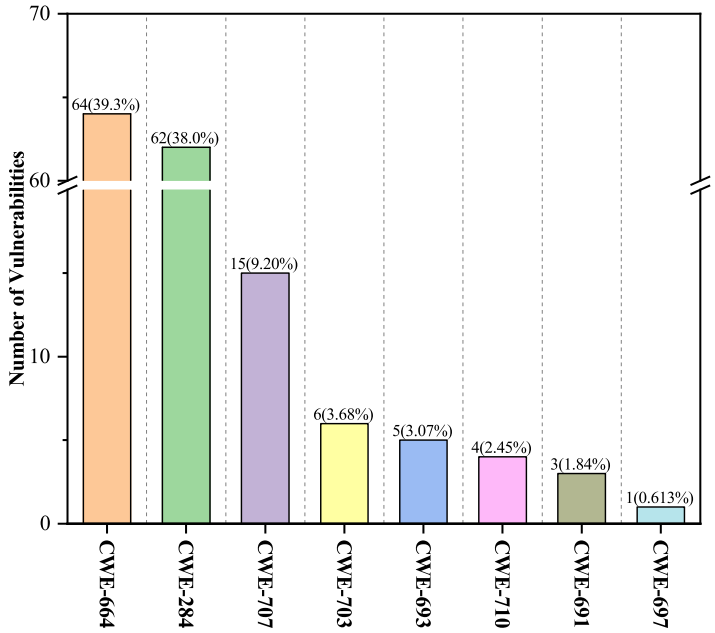


Fig. 13: **RQ**$_{7B}$: What are the erronous software behaviours leading to Edge security failures? See Table 5 for detailed descriptions.

4.8 **RQ**$_8$: How severe are Edge vulnerabilities?

Figure 14 shows the distribution of NVD severity score for the CVE vulnerabilities considered in our study; the median severity is 7.5, which indicates that more than half of the vulnerabilities have a high severity score (severity is considered high when the severity score is between 7.0 and 9.0, see Section 2.4).

Figure 15 provides the distribution of *Attack Complexity* values; the attack complexity is low for 85.7% of the cases, which indicates that it is relatively easy for a malicious user to exploit a vulnerability.

Figure 16 provides the distribution of the *Privileges Required* to exploit a vulnerability; high privileges are required for only 15 (10.2%) of the vulnerabilities, whereas 59 (40.1%) and 66 (44.9%) of the vulnerabilities can be exploited with low or no privileges at all. These numbers confirm the easiness for malicious actors to exploit Edge vulnerabilities, which increase the associated risks.

Further, Table 6 provides the percentage of vulnerabilities presenting a high, low, or no impact on Confidentiality, Availability, and Integrity, according to the NVD CVSS results. We can observe that more than half of the vulnerabilities present a high impact on at least one of the three security properties thus highlighting the need for improved security testing practices.

Based on the results above, we conclude that an improvement of Edge systems' testing practices is necessary.

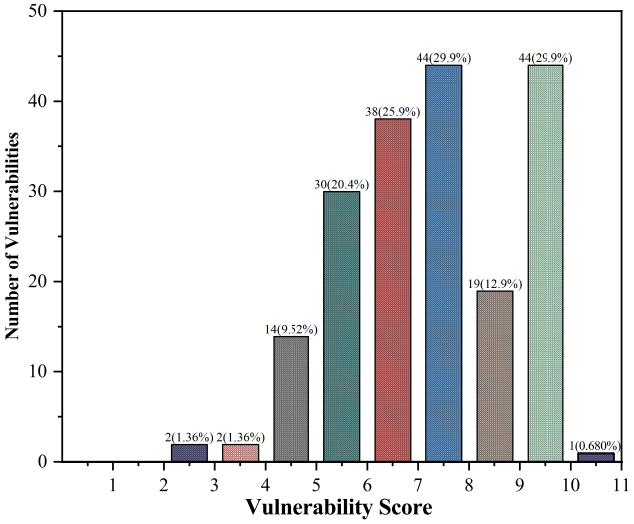


Fig. 14: $\mathbf{RQ}_8$: Distribution of NVD CVSS vulnerability scores

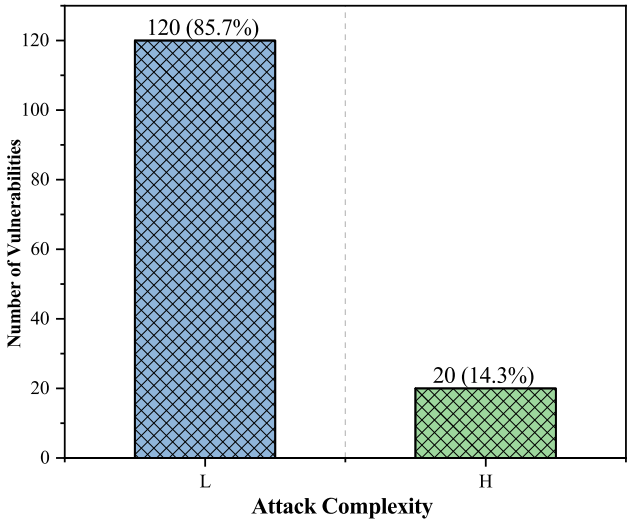


Fig. 15: Attack complexity (High - H, Low - L) for vulnerabilities in Edge frameworks, based on NVD CVSS entries

Table 6: Vulnerabilities' impact based on CVSS NVD scores.

	Confidentiality	**Integrity**	**Availability**
High	63.57%	55.00%	57.14%
Low	20.00%	10.71%	7.14%
None	16.43%	34.29%	35.71%

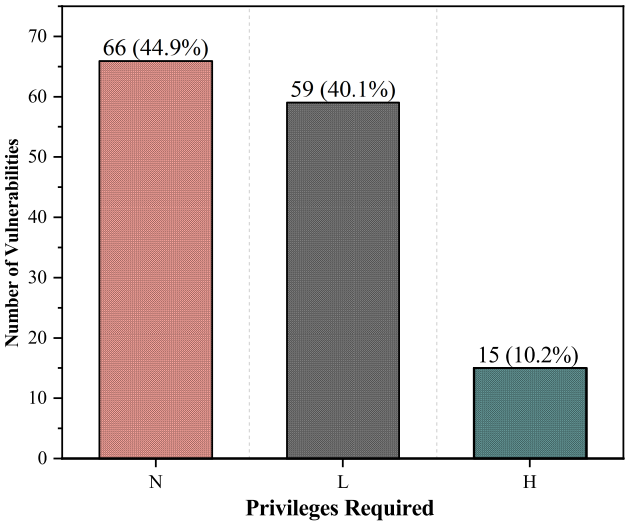


Fig. 16: Privileges required to exploit vulnerabilities in Edge frameworks, based on NVD CVSS entries (High - H, Low - L, None - N). Please note that *None* is the most critical situation since an attacker can exploit a vulnerability without any specific privilege on the system.

5 Threats to validity

5.1 Construct validity.

$\mathbf{RQ}_1$ to $\mathbf{RQ}_6$ might be affected by subjectivity in the manual classification. Indeed, the first author performed the manual classification after reading all the documentation available for each vulnerability. To minimize this risk, the first 30 vulnerabilities inspected at the beginning of the project had been reviewed with the second author to ensure common understanding. Further, randomly selected cases and unclear cases had been discussed. In total, about 50 vulnerabilities had been inspected by both authors. We also provide the classification results obtained for each vulnerability for further usage or independent analysis. $\mathbf{RQ}_{7A}$ to $\mathbf{RQ}_8$ are based on metrics (i.e., number of vulnerabilities for each CWE ID and CVSS score) that are commonly used in empirical studies [82].

5.2 Internal validity.

$\mathbf{RQ}_1$ to $\mathbf{RQ}_6$ results are derived from the inspection of vulnerability reports and documents linked in the vulnerability reports (e.g., documentation, patches). Incomplete or imprecise vulnerability descriptions may have affected our interpretation of results. We believe that the inspection of all the resources related to the vulnerabilities have mitigated this threat. $\mathbf{RQ}_{7A}$ to $\mathbf{RQ}_8$ are based on data provided by the CVE and

NVD repositories, which might be affected by mistakes (e.g., erroneous CWE identifier associated to a vulnerability). To mitigate this threat, the first author has read each CWE ID associated to the vulnerabilities investigated in our study, to ensure they were consistent with the vulnerability descriptions. Finally, the set of vulnerabilities reported for the frameworks selected for our study might be incomplete (e.g., the selected frameworks may not have been sufficiently used in the field to trigger all the vulnerabilities affecting them); this might be likely the case for K3OS and Zetta, which present only one vulnerability each. However, such threat should have a limited impact on our results because we do not aim to identify the less vulnerable framework but the characteristics of the vulnerabilities discovered in the field; vulnerabilities not discovered yet are out of the scope of our study. Please note that the low number of vulnerabilities reported for K3OS and Zetta unlikely reflects a higher degree of security for these two frameworks but it is likely the consequence of (1) a reduced code base with respect to KubeEdge and Mainflux (i.e., less code implemented, less vulnerabilities), (2) a limited user base (i.e., with less users, the number of vulnerabilities detected in the field is much more limited), and (3) a less rigorous security testing process than KubeEdge and Mainflux (see Section 2.2). Further, the low number of vulnerabilities reported for K3OS depends on our choice of not including Kubernetes vulnerabilities among K3OS total count (see Section 3.1). The code base that we considered for KubeEdge and Mainflux is larger than the one considered for K3OS and Zetta; indeed, when collecting KubeEdge vulnerabilities, we included vulnerabilities in dependencies (i.e., KubeEdge, Cri-o, Raspberry Pi, Mosquitto, and verneMQ, see Section 3.1), which leads to more than 1900k lines of code (LOC). For Mainflux, we collected also vulnerabilities concerning Docker components such as Containerd [25], which leads to more than 500k LOC. K3OS code, instead, includes 293k LOC, while Zetta 14K LOC. About the user base, if we rely on the number of forks on GitHub as a proxy to compare diffusion of frameworks, we observe that KubeEdge and Mainflux are the most widespread projects with 1500 and 587 forks, respectively, while K3OS and Zetta have less forks, 392 and 120, respectively. Based on the above, future work may concern assessing the relation between framework adoption and vulnerabilities being reported; for example, based on a security testing campaign for all the frameworks in our study aimed at determining how vulnerability distribution changes when extensive security testing is in place.

5.3 Conclusion validity.

Our study is purely observational; precisely, we compare the distribution of categorical variables not the effectiveness of different treatments. Therefore, we should ensure that the differences in the number of occurrences for each category are significant. For each RQ, to reject the null hypothesis *each category is equally likely* we performed a Pearson's Chi-squared goodness-of-fit test. Table 7 provides the results; for all our RQs, we reject the null hypothesis with p-value < 0.01. Please note that for $\mathbf{RQ}_6$ our manual analysis (i.e., what we plot in Figure 10), which identifies, for each vulnerability, only one violated security property (either the one that is easier to violate or the one that is violated first), leads to significant conclusions (see row

Table 7: Statistical significance of the differences across RQ categories (Chi-squared goodness-of-fit test)

RQ	p-value
$\mathbf{RQ}_1$	5.75e−86
$\mathbf{RQ}_{2A}$	4.61e−140
$\mathbf{RQ}_{2B}$	5.7e−48
$\mathbf{RQ}_{2C}$	3.19e−47
$\mathbf{RQ}_{2D}$	2.34e−101
$\mathbf{RQ}_3$ (Failure type)	1.02e−12
$\mathbf{RQ}_3$ (Detectability)	3.53e−41
$\mathbf{RQ}_4$	3.55e−75
$\mathbf{RQ}_{5A}$	7.99e−53
$\mathbf{RQ}_{5B}$	2.02e−19
$\mathbf{RQ}_6$	4.10e−19
$\mathbf{RQ}_6$ (NVD-Total)	0.1e−0
$\mathbf{RQ}_6$ (NVD-High)	0.6e−0
$\mathbf{RQ}_6$ (NVD-Low)	0.7e−2
$\mathbf{RQ}_{7A}$	7.44e−56
$\mathbf{RQ}_{7B}$	7.74e−51
$\mathbf{RQ}_{7C}$	6.0e−45
$\mathbf{RQ}_8$ (Attack Complexity)	2.87e−17
$\mathbf{RQ}_8$ (Priviliges Required)	7.70e−08

$\mathbf{RQ}_6$ in Table 7). Instead, the data derived from NVD's CVSS (i.e., what we plot in Figure 11) does not enable us to reject the null hypothesis neither by looking at the total counts (see row named $\boldsymbol{RQ}_6$ *NVD-Total* in Table 7) nor by looking at the vulnerabilities with the highest impact (see row $\boldsymbol{RQ}_6$ *NVD-High*). Indeed, as anticipated in Section 4.6, NVD's CVSS records usually report multiple security properties as being violated by each vulnerability. In practice, our choice makes the results more actionable in our context since it enables prioritizing the security feature to target. However, it might lead to oversimplification (a vulnerability may affect multiple security properties) and therefore should not be considered to draw general conclusions about the impact of vulnerabilities.

Another factor that may affect our conclusions is the distribution of faults per project. Indeed, two of our case study subjects include 98% of the vulnerabilities in our study: Kubedge (48.3%) and Mainflux (50.3%). If these two projects present different distributions for our RQ answers (e.g., the most frequent vulnerability type differ between them), our conclusions may not generalize. In practice, we need to determine if the answers provided to each RQ are equally likely to belong to both Kubedge and Mainflux. To this end, for each research question, we performed a Fisher's exact test [39]. The Fisher's exact test computes the probability (p-value) of observing the distribution of vulnerability results across our RQ choices[8], under the null hypothesis that each category is equally likely to appear in either Kubedge or Mainflux. We consider the null hypothesis to be rejected (i.e., Kubedge and Mainflux

[8] We use the term *choice* to indicate one of the possible answers that can be selected to address one RQ, for each vulnerability.

Table 8: Distribution of RQs answers for Mainflux (M) and KubeEdge (K).

RQ1			RQ2-A			RQ2-B			RQ2-C			RQ2-D			RQ3-A		
	K	M		K	M		K	M		K	M		K	M		K	M
IEC	1	0	**Resources**	1	4	**Resources**	5	9	**Resources**	4	8	**Resources**	1	1	**Policy Failure**	21	12
UAC	17	0	**API**	0	0	**API**	6	2	**API**	8	2	**API**	7	2	**Network Failure**	22	17
UEC	3	0	**Plugins**	5	0	**Plugins**	11	1	**Plugins**	9	2	**Plugins**	12	3	**Value Failure**	15	42
CE	50	74	**SUT**	40	67	**SUT**	18	50	**SUT**	16	45	**SUT**	37	55	**Timing Failure**	1	0
BT	0	0	**Driver**	0	0	**Driver**	0	0	**Driver**	0	0	**Driver**	0	0	**System Failure**	12	3
			Service	1	1	**Services**	6	1	**Services**	6	0	**Services**	4	0			
			Network	6	3	**Network**	9	11	**Network**	19	17	**Network**	9	13			
			Node	18	0	**Node**	3	0	**Node**	0	0	**Node**	0	0			
			HW	0	0	**HW**	1	0	**HW**	1	0	**HW**	1	0			
			None	0	0	**None**	12	0	**None**	7	1	**None**	0	0			

RQ3-B			RQ5-A			RQ4			RQ5-B			RQ6		
	K	M		K	M		K	M		K	M		K	M
Signalled	6	3	**Zero Step**	7	1	**Data(previous Input)**	4	0	**Data(previous Input)**	54	72	**System Integrity**	24	18
Unhandlled	6	4	**1 Step**	44	73	**Lack of Data**	1	0	**Lack of Data**	1	0	**Data**	6	1
Silent	59	67	**2 Step**	3	0	**Missing Node**	0	0	**Missing Node**	0	0	**Confidentiality**	28	52
			3 Step	0	0	**Resource Busy**	1	0	**Resource Busy**	0	0	**Availability**	13	3
			4+ Step	0	0	**Resource navailable**	2	0	**Resource navailable**	0	0			
			No Info	17	0	**Configuraiton**	28	62	**Configuraiton**	8	1			
						None	35	12	**None**	7	1			
									Delay Causing Missing	1	0			

RQ7-B			RQ8 (Distribution)			RQ8 (Attack Complexity)			RQ8 (Priviliges Required)		
	K	M		K	M		K	M		K	M
CWE-284: Improper Access Control	19	43	**0 - 1**	0	0	**Low**	50	70	**High**	43	16
CWE-664: Improper Control of a Resource Through its Lifetime	40	24	**1.1 - 2**	0	0	**High**	17	3	**Low**	5	10
CWE-697: Incorrect Comparison	1	0	**2.1 - 3**	0	2				**None**	19	47
CWE-693: Protection Mechanism Failure	2	3	**3.1 - 4**	1	2						
CWE-691: Insufficient Control Flow Management	1	2	**4.1 - 5**	8	5						
CWE-707: Improper Neutralization	8	7	**5.1 - 6**	16	20						
CWE-703: Improper Check or Handling of Exceptional Conditions	4	2	**6.1 - 7**	17	21						
CWE-435: Improper Interaction Between Multiple Correctly-Behaving Entities	0	0	**7.1 - 8**	14	23						
CWE-710: Improper Adherence to Coding Standards	2	2	**8.1 - 9**	9	45						
			9.1 - 10	2	1						

RQ7-A			RQ7-A			RQ7-A			RQ7-C		
	K	M		K	M		K	M		K	M
CWE-863	2	1	**CWE-444**	0	0	**CWE-918**	1	2	**State Issues**	1	0
CWE-552	0	1	**CWE-416**	1	0	**CWE-335**	1	0	**Data Processing Errors**	2	1
CWE-327	0	1	**CWE-270**	1	0	**CWE-20**	2	3	**Data Validation Issues**	1	0
CWE-312	0	1	**CWE-78**	1	1	**CWE-24**	1	0	**Data Neutralization Issues**	3	4
CWE-798	0	1	**CWE-787**	0	0	**CWE-283**	1	0	**Privilege Issues**	5	1
CWE-434	0	1	**CWE-401**	1	0	**CWE-250**	2	1	**Authentication Errors**	3	34
CWE-372	1	0	**CWE-79**	1	1	**CWE-502**	3	0	**File Handling Issues**	2	1
CWE-601	2	1	**CWE-290:**	0	0	**CWE-532**	5	1	**Pointer Issues**	2	0
CWE-184	1	0	**CWE-281**	0	0	**CWE-669**	1	0	**Business Logic Errors**	4	0
CWE-94	1	2	**CWE-256**	0	0	**CWE-732**	1	0	**Resource Management Errors**	7	0
CWE-610	2	0	**CWE-755**	1	0	**CWE-522**	2	1	**Audit / Logging Errors**	7	1
CWE-441	1	0	**CWE-201**	1	0	**CWE-306**	0	0	**Information Management Errors**	10	4
CWE-200	5	2	**CWE-300**	1	0	**CWE-73**	1	0	**Communication Channel Errors**	1	0
CWE-862	3	0	**CWE-295**	1	1	**CWE-287**	1	0	**Error Conditions, Return Values, Status Codes**	1	2
CWE-269	2	2	**CWE-1050**	1	0	**CWE-420**	1	0	**Random Number Issues**	1	0
CWE-266	3	0	**CWE-770**	3	0	**CWE-400**	2	2	**Cryptographic Issues**	1	0
CWE-306	2	32	**CWE-789**	2	0				**Bad Coding Practices**	1	0
CWE-22	4	4	**CWE-59**	1	1				**Behavioral Problems**	0	0
CWE-668	2	2	**CWE-61**	1	0				**Memory Buffer Errors**	0	0
CWE-74	2	0	**CWE-215**	1	0				**Credentials Management Errors**	0	2
CWE-23	1	0	**CWE-209**	1	2				**Permission Issues**	0	2
CWE-476	2	0	**CWE-284**	2	1				**Handler Errors**	0	1

have significantly different distributions for the different categories) if the p-value is below 0.05.

Table 8 reports the distribution of vulnerabilities for each RQ answer, for both Kubedge and Mainflux; Table 9 reports the p-values computed with the Fisher's exact test. We can observe that, for most of our RQs, it is not possible to reject the null hypothesis that each category is equally likely to appear in either Kubedge or Mainflux (p-value > 0.05), which indicates that the distribution of answers, for most of

Table 9: Statistical significance of the differences in RQ answers between Mainflux and KubeEdge, based on Table 8 (Fisher test)

RQ	p-value
$\mathbf{RQ}_1$	1
$\mathbf{RQ}_{2A}$	0.071
$\mathbf{RQ}_{2B}$	0.999
$\mathbf{RQ}_{2C}$	0.999
$\mathbf{RQ}_{2D}$	1
$\mathbf{RQ}_3$ (Failure type)	1
$\mathbf{RQ}_3$ (Detectability)	1
$\mathbf{RQ}_{5A}$	1
$\mathbf{RQ}_4$	0.476
$\mathbf{RQ}_{5B}$	0.035
$\mathbf{RQ}_6$	1
$\mathbf{RQ}_{7A}$	0.005
$\mathbf{RQ}_{7B}$	0.999
$\mathbf{RQ}_{7C}$	0.046
$\mathbf{RQ}_8$ (Distribution)	0.133
$\mathbf{RQ}_8$ (Attack Complexity)	1
$\mathbf{RQ}_8$ (Priviliges Required)	1

our RQs, do not present any pattern specific to any of the two frameworks. Therefore, we can conclude that most of our results are likely to generalize. In the following, we discuss the three RQs having a p-value below 0.05 (i.e., $\mathbf{RQ}_{5B}$, $\mathbf{RQ}_{7A}$, $\mathbf{RQ}_{7C}$). In the case of $\mathbf{RQ}_{5B}$, we observe that, for Mainflux, what enables the attacker to exploit a vulnerability is mainly input data (97.4% of the cases); instead, for KubeEdge, although input data remains the prevalent mean to exploit vulnerabilities (76.1%), vulnerabilities may be exploited also with no inputs (9.9%) or configuration options (11.3%). Although the difference in distribution is significant, it does not affect our conclusion, which is about focusing on input data generation to support testing; indeed, input data is the most frequent answer for both KubeEdge and Mainflux. In the case of $\mathbf{RQ}_{7A}$, we observe that, for Mainflux, 47.1% of the vulnerabilities concern *Missing Authentication for Critical Function (CWE-306)*, instead, for KubeEdge, the vulnerabilities are more uniformly spread across a larger set of vulnerability types. Still, although the difference in distribution is significant, it does not affect our conclusion for $\mathbf{RQ}_{7A}$, which is that the most frequent vulnerability types are the ones concerning *access control*[9] and *path traversal or control of resources*[10]. The percentage of *access control* vulnerabilities amounts to 37.8% for KubeEdge (31 out of 82) and 62.7% for Mainflux (42 out of 68). The percentage of vulnerabilities concerning *path traversal or control of resources* is 68.3% for KubeEdge and 39.7%

[9] Access control vulnerabilities are CWE-306, CWE-863, CWE-552, CWE-798, CWE-372, CWE-862, CWE- 269, CWE-266, CWE-283, CWE-250, CWE-532, CWE-732, CWE-522, CWE-287, CWE-420, CWE-284, CWE-300, CWE-295, and CWE-270.

[10] Vulnerabilities concerning path traversal or control of resources are CWE-863, CWE-552, CWE-312, CWE-434, CWE- 372, CWE-601, CWE-184, CWE-94, CWE-610, CWE-441, CWE-200, CWE-22, CWE-668, CWE-74, CWE-23, CWE-20, CWE-24, CWE-250, CWE-502, CWE-532, CWE-669, CWE-732, CWE-522, CWE-73, CWE-400, CWE-209, CWE-918, CWE- 201, CWE-1050, CWE-770, CWE-789, CWE-59, CWE-61, CWE-215, CWE-416, and CWE-401.

for Mainflux. Although their ranking is swapped (i.e., *access control* vulnerabilities are the most frequent for Mainflux but the second frequent for KubeEdge), they remain the two the most frequent vulnerability types for both the projects; therefore our observations may generalize to other projects. Finally, in $\mathbf{RQ}_{7C}$ the distribution of vulnerabilities is more spread out for KubeEdge while it concentrates manly on a single cause of errors for Mainflux. Indeed, 64.15% of the developer mistakes are authentication errors for Mainflux and only 5.6% for KubeEdge. In KubeEdge, the other frequent sources of problems are Data Neutralizaton Issues, Privilege Issues, Resource Management Errors, Logging Errors, and Information Management Errors, which cause 5.8%, 9.6%, 13.5%, 13.5%, and 19.2% of the vulnerabilities, respectively. In Mainflux, they cause 7.5%, 1.9%, 0%, 1.9%, and 7.6% of the vulnerabilities. In Mainflux, *Credentials Management Errors*, *Permission Issues*, and *errors in the management of Error Conditions, Return Values, Status Codes* have slightly higher frequencies (3.8%) than some of the four cases above. In the case of $\mathbf{RQ}_{7C}$, we believe that the difference in distribution between KubeEdge and Mainflux is in part related to the difference observed for $\mathbf{RQ}_{7A}$. Indeed, it is reasonable that the larger proportion of access control vulnerabilities observed in $\mathbf{RQ}_{7A}$ for Mainflux is related to the larger proportion of authentication errors observed for Mainflux. The mistakes leading to path traversal or control of resources, which are more frequent in KubeEdge, are likely more diverse. Also, the difference in distribution between KubeEdge and Mainflux might be due to a non-negligible proportion of vulnerabilities for which $\mathbf{RQ}_{7C}$ data is not available (60 vulnerabilities in total, 40.8%); for $\mathbf{RQ}_{7A}$, the proportion of missing vulnerabilities is lower, 10.20%, in total. To conclude, only the results of $\mathbf{RQ}_{7C}$ may not generalize. However, $\mathbf{RQ}_{7C}$ results are the least actionable; indeed, they do not enable us to derive any suggestion for the development of automated testing tools (see Section 6).

Finally, the results for KubeEdge and Mainflux may also generalize to K3OS and Zetta. In the case of K3OS, results should generalize because K3OS inherits all the Kubernetes vulnerabilities affecting KubeEdge. For Zetta, assuming that the low number of vulnerabilities found is due to a limited user base, we may observe, in case of a broader use of Zetta, a distribution of vulnerabilities similar to the one discussed above because Zetta includes components (e.g., the event broker, the pub-sub service, and the http-server) that, in a simplified manner, replicate the functionalities available in KubeEdge.

5.4 External validity.

We selected Edge frameworks that, based on our selection criteria, have an active user base, which indicates that they provide features that are necessary for the development of Edge systems For example, KubeEdge is used to manage nearly 100,000 edge nodes in unmanned toll stations across China [70]. Further, the selected frameworks include a range of features broad enough to support several contexts of use for Edge systems, including smart light, speed sensors' monitoring (e.g., vehicles'), smart home security, temperature sensing, and video streaming systems [65, 145]. Consequently, the vulnerabilities encountered in our investigation are likely repre-

sentative of the different types of vulnerabilities that might be encountered in Edge frameworks; indeed, every software feature may be vulnerable.

The type of security failures observed in the field depend not only on the features implemented by the software but also on the quality of the software security testing process in place. In our study, we considered only open source software; open source software is often developed by volunteers who may not be enforced to follow a quality assurance process. However, this is not the case for KubeEdge, Mainflux, and K3OS because their development is supervisioned by private companies that have invested effort towards test automation for these frameworks (see Section 2.2). The development process of KubeEdge, the largest system considered in our study, relies on code review activities (e.g., contributions are revised by senior members[11]) and two security teams [63, 64] that audit the system and respond to reports of security issues. Further, KubeEdge is based on Kubernetes, whose development team includes a group of security experts [67]. Mainflux is developed and maintained by Mainflux Labs, which is a for-profit technology company; considering that Mainflux Labs developed a test suite and a benchmark for Mainflux, and that Mainflux Labs provides auditing services, we assume the development process behind Mainflux to be no different than the one adopted for other commercial Edge software. Similar to Mainflux is the case of K3OS, which is part of Rancher, a framework developed by the open source software development company Suse [131]. Among the frameworks selected for our study, only Zetta is not supported by a for-profit organization but only volunteers; therefore, the conclusions drawn for Zetta may not generalize to commercial software solutions. However the impact of this threat is limited because Zetta provides only 1 of the 147 vulnerabilities investigated in our study (see Table 2, Page 16).

Given the growing popularity of Edge systems, the number of Edge vulnerabilities to be studied might increase and vary; therefore, larger replications of our study will be possible in the future.

6 Discussion and lessons learned

Our study aims to support the development of testing automation techniques that discover vulnerabilities in Edge systems.

RQ$_1$ indicates that security vulnerabilities slip through the testing process not because of bad testing but because of other reasons, which are *Combinatorial explosion*, *Unknown environment conditions*, *Unknown application conditions*, *Irreproducible execution conditions*. Software faults (and therefore vulnerabilities, which are a specific type of fault) that slip through the testing process because of the reasons above are defined as *field intrinsic* by to Gazzola et al. [41]. To identify such faults, Gazzola et al. propose to rely on field-based testing, which concerns performing testing activities directly in the production environment. Field-based testing might be adopted also to discover field-intrinsic vulnerabilities. A recent survey [15] identifies three field-based testing approaches: *online testing*, where test cases are executed directly on the

[11] see https://kubeedge.io/en/docs/community/membership/

software instance used in production, *offline testing*, where test cases are executed on a separated software instance running in the production environment, and *ex-vivo testing*, where test cases are executed in-house (i.e., in the development environment) but using data collected from the field.

Field-based testing solutions differ for the approach adopted, the software properties under test (i.e., functional, robustness, security), the test generation strategy (specification-based, structure-based, fault-based, and reusing pre-existing test cases), the environment in which test cases are generated (i.e., in-house, in-house with field data, or in-the-field), the criterion adopted to trigger test cases (i.e., periodically, after a specific event, after a request, based on a policy, when a function is used, after system reconfiguration, after environment change, after module change/insertion/removal), the resources required (e.g., user inputs, memory, logs, test data), and the types of oracles (i.e., domain-dependent or domain-independent).

The number of available field-based testing techniques targeting software security is limited, seven out of 80 papers appearing in the above-mentioned survey [15]. Two papers propose a technique that works offline [30,31], five papers concern online testing [14, 16, 32, 47, 146]. Six techniques are specification-based [14, 16, 30, 31, 32, 47], one is fault-based [146]. They are activated by three different types of triggers: a policy [16, 32], the execution of a certain functionality defined either at run-time [30] or before [31, 47], and the deployment of a new module [14, 146]. Unfortunately, these seven field-based security testing approaches cannot be applied to test Edge frameworks; indeed, four of them address problems in online service compositions [14, 16, 32, 146], one approach targets only integer overflows [47], which were not observed in our analysis, two approaches [30, 31] concern offline testing (i.e., they test sibling processes with modified configurations), which is infeasible with large service (e.g., Edge controller) or with embedded devices running Edge nodes. New field-based testing solutions for Edge security testing thus need to be developed.

Since $\mathbf{RQ}_1$ indicates that most of the vulnerabilities are not discovered at testing time because of combinatorial explosion (i.e., the infeasibility to exercise the Edge framework under all the possible execution conditions), we believe that field-based testing might be an ideal solution since it might be implemented by developing techniques that identify the conditions in which testing automation should be triggered (e.g., when observing combination of inputs not tested in-house). To further support our suggestion, we joined the results obtained for $\mathbf{RQ}_1$ and $\mathbf{RQ}_4$, which enables us to report that 84 out of 126 (67%) CE vulnerabilities present a specific combination of configuration parameters as precondition (i.e., they can be exploited only if a specific configuration is in place). Such number indicates that, by activating field-based testing whenever the system is executed with a configuration not tested in-house, we may discover up to 57% (i.e., 84 out of 147) Edge vulnerabilities.

Based on $\mathbf{RQ}_2$ results, we conclude that the SUT is the component that (a) is usually faulty, (b) receives the inputs that trigger the vulnerability, (c) presents the preconditions required for the vulnerability to be exploitable, and (d) shows failures. Therefore, testing techniques should focus on the SUT interfaces, typically command line utilities, API, or Web interfaces.

$\mathbf{RQ}_3$ results indicate that most of the security failures are silent; also, the majority includes *Value* (38.8%) and *Action* failures (22.4%). Therefore, approaches looking

for crashes are not sufficient to support the identification of Edge vulnerabilities, which prevents the adoption of most fuzz testing approaches [81]. Fuzz testing tools (e.g., AFL [139]) usually rely on evolutionary search algorithms to generate test inputs by modifying previously generated inputs that demonstrated to be effective in improving a target metric (e.g., code coverage). Fuzz testing is normally used to either identify inputs leading to crashes or memory errors (e.g., use after free, out of bounds accesses, memory leaks); although memory errors might be indicators of vulnerabilities leading to value or timing failures (e.g., accessing private data or causing denial of service), without manual inspection it is not possible to determine if a memory error is exploitable as a vulnerability (i.e., if it breaks security properties) [53]. Therefore, fuzz testing can't be used to automatically detect Edge vulnerabilities resulting in value errors. It is therefore necessary to identify solutions addressing the oracle problem (i.e., the problem of automatically determining if a test output is correct [12]); in this regard, metamorphic testing might be an option since it has shown successful results when applied to test the security of Web systems [75]. Metamorphic security testing concerns specifying properties (called metamorphic relations) that relate the outputs generated by a set of source inputs and a set of follow-up inputs derived from them. Source inputs are sequences of legal Web interactions (e.g., HTTP requests) collected using a Web crawler. Follow-up inputs are generated by altering source inputs as an attacker would do. Metamorphic relations enable engineers to avoid implementing test assertions to verify that test inputs lead to specific test outputs [75]; indeed, metamorphic relations enable testing a software with any test input and automatically verifying the correctness of the software outputs. One alternative solution that enables engineers to automatically verify software outputs consists of relying on executable formal specifications (e.g., assertions verifying method post-conditions and used in property-based testing [38]); however, such solution is generally infeasible for Edge systems because software projects usually lack executable formal specifications because they are expensive to produce and maintain. For such reason, engineers manually implement test assertions that are specific for the inputs exercised by a test case. Instead, recent work has shown that it is possible to define generic metamorphic relations that can discover a broad range of vulnerabilities and can be reused across software systems because they process system inputs [20]; assertions, instead, are typically implemented within low-level software functions and, therefore, can't be reused across systems. Like assertions, metamorphic relations enable detecting silent failures (i.e., failures that can be detected only by verifying the correctness of the output data generated by the system). An example of how metamorphic security testing enables engineers to test a software system without implementing test assertions for every test inputs follows. With metamorphic relations, bypass authorization vulnerabilities can be detected by verifying if a URL provided by the Web interface of a user leads to a different response page when requested by a user whose Web interface does not provide the same URL. If the two users receive the same response page then the second user had been able to bypass the authorization schema [75]. Thanks to the use of Web crawlers, such metamorphic relation can be tested with any URL provided by a Web system thus enabling the exhaustive testing of all the available URLs. Since manually deriving test assertions

should be based on user-specific access policies, such exhaustive testing is infeasible without metamorphic relations.

The results of $\mathbf{RQ}_{5A}$ indicate that, once the system reaches the state required to trigger the vulnerability, one input action (one step) is generally sufficient to exploit a vulnerability. In addition, $\mathbf{RQ}_4$ indicates that, usually, it is a specific system configuration what enables exploiting the vulnerability. Therefore, it should be feasible to automate security testing for Edge systems. Indeed, once a configuration to be tested is identified, it might be sufficient to exercise the system with all the possible single (one step) actions not with long action sequences, which should result in a quicker testing process. Further, it should be feasible to thoroughly test the system.

The results of $\mathbf{RQ}_{5B}$ indicate that most of the inputs triggering vulnerabilities are data, which means that even brute force approaches like fuzzing might be sufficient to exploit vulnerabilities; however, the oracle problem needs to be addressed (e.g., through metamorphic relations, as suggested above).

The results of $\mathbf{RQ}_6$ show that 55% of the vulnerabilities concern confidentiality. Since confidentiality failures are about accessing sensible resources and do not affect the state of the system, we believe that isolation techniques, which are difficult to implement, are not needed when testing for confidentiality problems. Consequently, field-based testing solutions focusing on confidentiality will not need to integrate solutions that ensure isolation. Further, based on $\mathbf{RQ}_1$ results, we suggested to automatically trigger field-based testing when observing new configurations not tested in-house. After joining $\mathbf{RQ}_6$ and $\mathbf{RQ}_4$ data, we determined that 72 out of 81 confidentiality vulnerabilities depend on a specific configuration of the system (i.e., they were likely not detected because the specific configuration they depend on was not considered). Therefore, we can speculate that a field-based testing approach that focuses on confidentiality issues and is triggered by untested configurations might feasibly detect a large proportion of Edge vulnerabilities (i.e., 72 out of 147, 49%).

The results for $\mathbf{RQ}_{7A}$ to $\mathbf{RQ}_{7C}$ provide further directions for the implementation of testing automation techniques. The results of $\mathbf{RQ}_{7A}$ indicate that most of the vulnerabilities concern CWE-306 (Missing Authentication for Critical Function), which indicates that it is necessary to develop methods to automatically determine what are the functions that should require authentication. Authorization problems (i.e., CWE-284 in $\mathbf{RQ}_{7B}$) are frequent and, unfortunately, covered by existing field testing approaches only in the case of Web services [14, 16, 32, 146]. However, related work has shown that it is feasible to detect authentication and authorization problems with metamorphic security testing [75].

Input neutralization issues are often due to improper exception handling, which may indicate the need for better robustness testing.

Finally, leakage of sensitive data, memory issues, and bad coding practices might be detected through improved static code analysis tools; however, the evaluation of the effectiveness and extensibility of existing tools go beyond the scope of this paper. For that, we refer the reader to a recent empirical evaluation of Web-based systems [34], which has shown that exploratory manual penetration testing is more effective than automated static analysis tools in detecting severe vulnerabilities (e.g., the ones in the *OWASP Top Ten* list [121] related to *Security Logging and Monitoring Failures*, like *CWE 532 - Insertion of Sensitive Information into Log File*, which is

a form of information leakage). Automated static analysis tools, instead, detect the largest number of vulnerabilities, overall.

To summarize, since we observed that vulnerabilities are likely not discovered at testing time because of combinatorial explosion, we suggest researchers to introduce new security testing techniques for Edge systems that aim to address such problem ($\mathbf{RQ}_1$). The need for improved testing is motivated by the fact that the Edge vulnerabilities detected in-the-field are severe and easy to exploit (see $\mathbf{RQ}_8$). To minimize the number of vulnerabilities discovered by the end-users (or by malicious users), we suggest the development of field-based testing techniques that are triggered when the system is executed with a configuration not tested in-house ($\mathbf{RQ}_4$). The feasibility of such techniques should be facilitated by most vulnerabilities requiring only one input step to be exploited (i.e., testing techniques don't have to derive long input sequences, $\mathbf{RQ}_{5A}$); further, plain input data is sufficient to exploit most of them ($\mathbf{RQ}_{5B}$). Such techniques should target the interfaces of Edge frameworks not the components they rely upon (e.g., network or drivers, $\mathbf{RQ}_2$). Further, field-based security testing techniques shall focus on confidentiality, which concerns a large portion of the cases ($\mathbf{RQ}_6$); based on our results, field-based security testing techniques targeting confidentiality and triggered in the presence of untested configurations should be able to address 49% of the vulnerabilities. Since most vulnerabilities lead to silent, value failures ($\mathbf{RQ}_3$), researchers need to address the oracle problem (i.e., vulnerabilities are unlikely detected by looking for crashes); however, metamorphic testing might be a feasible solution since it has been successfully applied to detect authentication and authorization problems, which are among the most frequent types of vulnerabilities and developer mistakes ($\mathbf{RQ}_7$). Finally, till new approaches are not developed, we suggest developers of Edge frameworks to increase the effort put into testing of configurations; especially their effect on confidentiality.

7 Related work

To the best of our knowledge, our work is the first to report on vulnerabilities affecting Edge frameworks. Related work concerns empirical studies of software vulnerabilities, which we summarize below.

A recent survey of empirical studies on software failures indicates that their typical workflow includes six steps, which match our workflow: Define problem scope (see Section 3), Collect defect reports and supplementary data (see Sections 3.1 and 3.2), Analyze bug characteristics (see $\mathbf{RQ}_1$ to $\mathbf{RQ}_6$ and $\mathbf{RQ}_8$), Perform root cause analysis (see $\mathbf{RQ}_{7A}$ to $\mathbf{RQ}_{7C}$), Report results (see Section 4), Discuss impact and recommendations for industry (see Section 6). The survey is based on 52 papers; however, only five of them focus on software vulnerabilities [17, 28, 71, 82, 140]. Further, none of the selected papers aim to discuss the feasibility of performing field-based testing, which was instead the aim of Gazzola et al. [41].

Bavota et al. analyzed the vulnerabilities affecting the Android OS [71, 82]. Their study investigates type and evolution of vulnerabilities, the most common CVSS vectors, the Android subsystems mostly affected by vulnerabilities, and the time required to fix them. Similar to our results, Bavota's study show that vulnerabilities affecting

access control and privileges (i.e., CWE pillars CWE-664 and CWE-284 in our case, see $\mathbf{RQ}_{7B}$) are the most frequent ones. However, in their analysis, memory errors take the second place, which is not the case for us, likely because of the different nature of these two types of software. Indeed, Android includes an OS layer that takes care of handling also the memory at kernel level, which is not the case for Edge systems where a third party OS layer (excluded from vulnerability reports) takes care of handling the memory. The layers mainly affected by Android vulnerabilities are the kernel, the native libraries, and the application layer, which is in line with our findings where the SUT, Plugins, and APIs are among the mostly affected components in Edge systems. Somehow, these results show that the core components (i.e., SUT for our analysis and kernel for Bavota's) are the ones affected by most of the vulnerabilities, possibly because they implement most of the core software features. Their take-out lessons mostly concern the improvement of coding practices while we focus on a complementary aspect, i.e., the development of testing automation tools.

Blessing et al. [17] analyzed the vulnerabilities affecting eight cryptographic libraries (i.e., OpenSSL, GnuTLS, Mozilla NSS, WolfSSL, Botan, Libgcrypt, LibreSSL, and BoringSSL). They collected data from multiple sources (i.e., NVD, CVE and OpenCVE). Their findings suggest that vulnerabilities in cryptographic libraries are mainly due to the following CWE weaknesses: exposure of sensitive information, improper input validation, numeric errors, memory buffer issues, resource management errors, and cryptographic issues. Expectedly, the distribution of these weaknesses differ from the ones reported in our paper (e.g., memory buffer issues count for 20% of the cryptographic cases while in our paper they are less than 1%); indeed, Edge frameworks and cryptographic libraries present a very different nature.

Zaman et al., compared faults affecting two types of non-functional properties for the Firefox Web-browser, which are security and performance [140]. Different from our work, they do not aim to study the reasons why faults are not detected at testing time but they focus on the fault-fixing process and report about the time required to fix these faults, the number of developers involved in the fix, and the complexity of the fix (number of lines and files modified). Similarly, Catolino et al. discuss the distribution of different types of faults, the time before assignment/response/change, the duration of the bug fixing process, and the topics related to different fault types [19]; such information does not help designing automated testing tools, which is our purpose. Tan et al. report on the faults affecting three popular open-source systems in 2014: the Mozilla Web-browser [117], the Apache HTTP Server [5], and the Linux operating system's kernel [72]. Their discussion of security vulnerabilities is limited; indeed, they report that semantic bugs are the main cause of security vulnerabilities but they do not report any finer grained characterization. Further, they report that availability is violated slightly more than confidentiality and integrity; however, the data set is older than ours and their systems are different in nature.

Cottrell et al. report on the frequency, type, and severity of vulnerabilities affecting hardware and software robotic components [28]. They report that vulnerabilities are more frequent in software (92.6%) than in hardware (7.4%) components, which is in line with our findings. They do not explicitly rely on CWE vulnerability types; however, they report that software vulnerabilities mainly concern Memory management (32.4%), Input sanitization (24.1%), Authorization/Authentication

(22.5%), Denial of Service (19.6%), Cryptography (7.3%), Insecure default settings (7.3%), Dependency management (6%), Directory traversal (2.5%), Hard-coded secrets (2.4%). Such distribution of vulnerability types different from ours; we believe that the difference is mainly due to the nature and maturity of the software considered. For example, memory management issues have a limited impact in in Edge frameworks (see CWE-789, CWE-1050, CWE-416, CWE-401 in Table 4, which count for 3.4% of the total), likely because Edge frameworks delegate memory management to widely adopted open-source OSs. Input sanitization issues affect both robotics and Edge systems; however, they are less frequent in Edge systems (see CWE-707, 9.20% in Figure 13, Page 42). Authorization and authentication issues are more frequent in Edge frameworks because it is a key feature of Web-based distributed systems (see 38% for CWE-284 in Figure 13). The frequency of denial of service (i.e., availability) issues is in line with our findings (see Figure 10). Finally, in robotics systems, severity is considered either high or critical for more than 50% of the software vulnerabilities, a result that is similar to ours (see Section 4.8), which indicates that improved security testing solutions are necessary across fields, not only Edge systems.

Austin et al. empirically evaluated the effectiveness of different security testing approaches in detecting vulnerabilities of Web-based content management systems (CMS) [8]. They compared four approaches: exploratory manual penetration testing, systematic manual penetration testing, static analysis, and automated penetration testing (i.e., dynamic program analysis). Their results show that different approaches detect different vulnerabilities; precisely, static analysis detects mainly code injection vulnerabilities, systematic manual penetration testing detects audit and input validation vulnerabilities, automated penetration testing detects information leaks but leads to a high false positive rate for other types of vulnerabilities. Finally, static analysis accurately detects unsafe code and lack of null checks but leads to a high false positive rate for other types of vulnerabilities. CMS share a subset of the features of Edge systems (e.g., Web interfaces, interaction with databases); therefore, the results of Austin et al. confirm that testing these systems is complex (i.e., different approaches are required). Also, it shows that existing test automation tools are affected by a high false positive rate which may limit their adoption.

Zahid et al. recently conducted a survey on risk management approaches for cyber-physical systems (CPS), including IoT systems [138]. Their findings show that availability and integrity are of major concern for CPS, in contrast to Cloud systems where access control, integrity, and auditability are the most-studied quality attributes. Unsurprisingly, since Edge frameworks inherit several characteristics of Cloud computing frameworks, we observe a higher impact of access control and integrity issues rather than availability ones.

Tabrizchi et al., in a recent survey on security challenges in Cloud computing [132], list the architectural solutions that might be adopted to ensure security properties; further, they list the security threats affecting Cloud systems, according to literature. The provided list of threats includes path traversal attacks, code injections, authentication issues, abuse of functionalities, resource manipulation, denial of service, and data breaches. All the threats identified by Tabrizchi et al. match the vulnerabilities reported in our study; such result is not surprising since Cloud systems share many commonalities with Edge systems. However, Tabrizchi et al. do not provide any so-

lution for software testing automation neither provide suggestions for prioritizing the testing of vulnerabilities based on their frequency or criticality, which we do, instead. Similar to the work of Tabrizchi et al., the work of Ardagna et al. [6] provides another taxonomy of Cloud security solutions but is more dated. Their work shows that the number of automated testing solutions was limited; however, they do not provide any direction for future work.

8 Conclusion

We presented an empirical study of the security vulnerabilities affecting Edge frameworks. Our objective is to support the development of automated software testing techniques targeting software security in Edge frameworks. Our work is motivated by the increasing relevance of the Edge paradigm, which ensures low latency for several data-intensive applications (e.g., video streaming, video conferencing, video surveillance, naval services). This is the case for our industry partner, SES, a world-leading satellite operator.

We selected Edge frameworks with reported vulnerabilities and an active user base. We have manually read all the vulnerability reports and processed CWE and CVSS data reported in the CVE and NVD databases. We investigated eleven research questions that concern aspects influencing the development of automated testing tools (i.e, weaknesses in the testing process, types of components involved in a security failure, type of failures observed, steps required to exploit a vulnerability, nature of preconditions and inputs leading to a successful exploit, security properties being violated, frequent vulnerability types, software behaviours and developer mistakes associated to these vulnerabilities, severity).

Our results show that the large number of features implemented by Edge frameworks result in a combination of configuration options that often prevent the detection of vulnerabilities. Vulnerabilities are often due to implementation errors in the Edge software but their consequences affect both the software itself, the network configuration, and the controlled nodes. Confidentiality is the security property mostly affected by these security vulnerabilities, which can be easily exploited (in one step). Half of these vulnerabilities have a high NVD severity score, which highlights the need for their timely detection. We identify field-based testing (i.e., performing testing activities directly in the production environment) as a possible solution to address these vulnerabilities, which is facilitated by the prevalence of confidentiality problems (i.e., testing in the field is unlikely to affect the functioning of the system). Our future work will concern the definition of such solutions based on our findings.

Acknowledgments

This work has been supported by SES [126] and the Luxembourg National Research Fund (FNR) under the project INSTRUCT (IPBG19/14016225/INSTRUCT [73]).

Declarations

Data Availability

The authors declare that the data supporting the findings of this study are available at the following URL `https://zenodo.org/record/7826981`.

Funding and/or Conflicts of interests/Competing interests

The authors declare that they have no conflict of interest.

References

1. Alvin Jude: How will 5G and edge computing transform the future of mobile gaming? `https://www.ericsson.com/en/blog/2021/3/5g-edge-computing-gaming`. Last Accessed: 2023
2. Alwarafy, A., Al-Thelaya, K.A., Abdallah, M., Schneider, J., Hamdi, M.: A survey on security and privacy issues in edge-computing-assisted internet of things. IEEE Internet of Things Journal **8**(6), 4004–4022 (2021). DOI 10.1109/JIOT.2020.3015432
3. Ammann, P., Offutt, J.: Introduction to software testing - 2nd Edition. Cambridge University Press (2016)
4. Analysis Tools team: Static analysis tools for GO. `https://analysis-tools.dev/tag/go`. Last Accessed: 2022
5. Apache foundation: `https://www.apache.org/`. Last Accessed: 2022
6. Ardagna, C.A., Asal, R., Damiani, E., Vu, Q.H.: From security to assurance in the cloud: A survey. ACM Computing Surveys (CSUR) **48**(1), 1–50 (2015)
7. ARM: Microcontrollers and infrastructure manufacturer. `https://www.arm.com/`. Last Accessed: 2022
8. Austin, A., Holmgreen, C., Williams, L.: A comparison of the efficiency and effectiveness of vulnerability discovery techniques. Information and Software Technology **55**(7), 1279–1288 (2013). DOI https://doi.org/10.1016/j.infsof.2012.11.007. URL `https://www.sciencedirect.com/science/article/pii/S0950584912002339`
9. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE transactions on dependable and secure computing **1**(1), 11–33 (2004)
10. Aysan, H., Punnekkat, S., Dobrin, R.: Error modeling in dependable component-based systems. In: 2008 32nd Annual IEEE International Computer Software and Applications Conference, pp. 1309–1314. IEEE (2008)
11. Bai, T., Pan, C., Deng, Y., Elkashlan, M., Nallanathan, A., Hanzo, L.: Latency minimization for intelligent reflecting surface aided mobile edge computing. IEEE Journal on Selected Areas in Communications **38**(11), 2666–2682 (2020)
12. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE Transactions on Software Engineering **41**(5), 507–525 (2015)
13. Ben Nassi, Yaron Pirutin, Tomer Cohen Galor, Yuval Elovici, and Boris Zadov: `https://www.nassiben.com/glowworm-attack`. Last Accessed: 2022
14. Bertolino, A., Angelis, G.D., Frantzen, L., Polini, A.: The plastic framework and tools for testing service-oriented applications. In: Software Engineering, pp. 106–139. Springer (2007)
15. Bertolino, A., Braione, P., De Angelis, G., Gazzola, L., Kifetew, F., Mariani, L., Orrù, M., Pezzè, M., Pietrantuono, R., Russo, S., Tonella, P.: A Survey of Field-based Testing Techniques. ACM Computing Surveys **54**(5) (2021). DOI 10.1145/3447240
16. Bertolino, A., De Angelis, G., Kellomaki, S., Polini, A.: Enhancing service federation trustworthiness through online testing. Computer **45**(1), 66–72 (2011)
17. Blessing, J., Specter, M.A., Weitzner, D.J.: You really shouldn't roll your own crypto: An empirical study of vulnerabilities in cryptographic libraries. arXiv preprint arXiv:2107.04940 (2021)
18. Bondavalli, A., Simoncini, L.: Failure classification with respect to detection. In: [1990] Proceedings. Second IEEE Workshop on Future Trends of Distributed Computing Systems, pp. 47–53. IEEE (1990)

19. Catolino, G., Palomba, F., Zaidman, A., Ferrucci, F.: Not all bugs are the same: Understanding, characterizing, and classifying bug types. Journal of Systems and Software **152**, 165–181 (2019). DOI https://doi.org/10.1016/j.jss.2019.03.002. URL https://www.sciencedirect.com/science/article/pii/S0164121219300536
20. Chaleshtari, N.B., Pastore, F., Goknil, A., Briand, L.C.: Metamorphic testing for web system security. IEEE Transactions on Software Engineering (2023). Accepted, available at https://arxiv.org/abs/2208.09505
21. Chen, E.Y., Pei, Y., Chen, S., Tian, Y., Kotcher, R., Tague, P.: Oauth demystified for mobile application developers. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14, p. 892–903. Association for Computing Machinery, New York, NY, USA (2014). DOI 10.1145/2660267.2660323. URL https://doi.org/10.1145/2660267.2660323
22. Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M.Y.: Orthogonal defect classification-a concept for in-process measurements. IEEE Transactions on software Engineering **18**(11), 943–956 (1992)
23. ci4rail: Computing Intelligence for Rail and Public Transport. http://www.ci4rail.com. Last Accessed: 2022
24. Cinque, M., Cotroneo, D., Kalbarczyk, Z., Iyer, R.K.: How do mobile phones fail? a failure data analysis of symbian os smart phones. In: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), pp. 585–594. IEEE (2007)
25. Cloud Native Computing Foundation: https://github.com/containerd/containerd. Last Accessed: 2023
26. Clound Native Computing Foundation: https://www.cncf.io/. Last Accessed: 2022
27. Common Vulnerability Scoring System: https://www.first.org/cvss/. Last Accessed: 2022
28. Cottrell, K., Bose, D.B., Shahriar, H., Rahman, A.: An empirical study of vulnerabilities in robotics. In: 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), pp. 735–744 (2021). DOI 10.1109/COMPSAC51774.2021.00105
29. CVE Numbering Authorities (CNA): https://www.cve.org/ProgramOrganization/CNAs. Last Accessed: 2022
30. Dai, H., Murphy, C., Kaiser, G.: Configuration fuzzing for software vulnerability detection. In: 2010 International Conference on Availability, Reliability and Security, pp. 525–530. IEEE (2010)
31. Dai, H., Murphy, C., Kaiser, G.E.: Confu: Configuration fuzzing testing framework for software vulnerability detection. In: Security-Aware Systems Applications and Software Development Methods, pp. 152–167. IGI Global (2012)
32. De Angelis, G., Bertolino, A., Polini, A.: (role) cast: A framework for on-line service testing. In: International Conference on Web Information Systems and Technologies, vol. 2, pp. 13–18. SCITEPRESS (2011)
33. Dempsey, K., Shah, N., Arnold, C., Johnston, J.R., Jones, A.C., Orebaugh, A., Scholl, M., Stine, K.: NIST Special Publication 800-137 Information Security. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-137.pdf. Last Accessed: 2022
34. Elder, S., Zahan, N., Shu, R., Metro, M., Kozarev, V., Menzies, T., Williams, L.: Do I really need all this work to find vulnerabilities? Empirical Software Engineering **27**(6), 154 (2022). DOI 10.1007/s10664-022-10179-6. URL https://doi.org/10.1007/s10664-022-10179-6
35. Fabric8 Maven Plugin: https://maven.fabric8.io. Last Accessed: 2022
36. Fayad, M., Schmidt, D.C.: Object-oriented application frameworks. Commun. ACM **40**(10), 32–38 (1997). DOI 10.1145/262793.262798. URL https://doi.org/10.1145/262793.262798
37. Felderer, M., Büchler, M., Johns, M., Brucker, A.D., Breu, R., Pretschner, A.: Security testing: A survey. In: Advances in Computers, vol. 101, pp. 1–51. Elsevier (2016)
38. Fink, G., Bishop, M.: Property-based testing: A new approach to testing for assurance. SIGSOFT Softw. Eng. Notes **22**(4), 74–80 (1997). DOI 10.1145/263244.263267. URL https://doi.org/10.1145/263244.263267
39. Fisher, R.A.: On the interpretation of x2 from contingency tables, and the calculation of p. Journal of the Royal Statistical Society **85**(1), 87–94 (1922). URL http://www.jstor.org/stable/2340521
40. Gavan Murphy: Asset Tracking – Living on the Edge. https://www.iottechnews.com/news/2022/nov/09/asset-tracking-living-on-the-edge/. Last Accessed: 2023

41. Gazzola, L., Mariani, L., Pastore, F., Pezze, M.: An exploratory study of field failures. In: 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), pp. 67–77. IEEE (2017)
42. Google: Go lang. https://go.dev. Last Accessed: 2022
43. Gopalakrishna, N., Anandayuvaraj, D., Detti, A., Bland, F., Rahaman, S., Davis, J.C.: "if security is required": Engineering and security practices for machine learning-based iot devices. In: 2022 IEEE/ACM 4th International Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT), pp. 1–8. IEEE Computer Society, Los Alamitos, CA, USA (2022). DOI 10.1145/3528227.3528565. URL https://doi.ieeecomputersociety.org/10.1145/3528227.3528565
44. Hagar, J.D.: IoT System Testing: An IoT Journey from Devices to Analytics and the Edge. Apress (2002)
45. Honnef, D.: Staticcheck: static analysis tool for the go programming language. "https://staticcheck.io/". Last Accessed: 2022
46. Huawei: http://www.huawei.com. Last Accessed: 2022
47. Hui, Z.W., Huang, S., Ji, M.Y.: A runtime-testing method for integer overflow detection based on metamorphic relations. Journal of Intelligent & Fuzzy Systems **31**(4), 2349–2361 (2016)
48. Invicti: Acunetix. https://www.acunetix.com/plp/web-vulnerability-scanner/?utm_term=acunetix&utm_campaign=1077471751&utm_content=55423374169&utm_source=Adwords&utm_medium=cpc&gclid=EAIaIQobChMIjbm99ZTI9gIVgxoGAB1IsAK3EAAYASAAEgJo0PD_BwE. Last Accessed: 2022
49. ISO: ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary. ISO/IEC/IEEE 24765:2017(E) pp. 1–541 (2017). DOI 10.1109/IEEESTD.2017.8016712
50. Jin, X., Katsis, C., Sang, F., Sun, J., Kundu, A., Kompella, R.: Edge security: Challenges and issues (2022). DOI 10.48550/ARXIV.2206.07164. URL https://arxiv.org/abs/2206.07164
51. K3OS: K3OS Automated Test Suite. https://github.com/rancher/k3os/blob/master/scripts/test. Last Accessed: 2022
52. K3OS: K3OS Edge Computing Framework. https://k3os.io/. Last Accessed: 2022
53. Koziol, J.: Charlie Miller Reveals His Process for Security Research (2010). URL https://resources.infosecinstitute.com/topic/how-charlie-miller-does-research/
54. Kube-score: Static code analysis for kubernetes object definitions. https://kube-score.com/. Last Accessed: 2022
55. KubeEdge: KubeEdge Deployment using Keadm. https://kubeedge.io/en/docs/setup/keadm/. Last Accessed: 2022
56. KubeEdge: KubeEdge Development Process. https://kubeedge-docs.readthedocs.io/en/latest/getting-started/contribute.html. Last Accessed: 2022
57. KubeEdge: KubeEdge Edge Computing Framework. https://kubeedge.io/en/. Last Accessed: 2022
58. KubeEdge: KubeEdge End-To-End Test Suite. https://github.com/kubeedge/kubeedge/tree/master/tests/e2e. Last Accessed: 2022
59. KubeEdge: KubeEdge GitHub issue tracker. https://github.com/kubeedge/kubeedge/issues. Last Accessed: 2022
60. KubeEdge: KubeEdge Integration Test Suite. https://github.com/kubeedge/kubeedge/tree/master/tests/integration. Last Accessed: 2022
61. KubeEdge: KubeEdge Issue 1017. https://github.com/kubeedge/kubeedge/issues/1017. Last Accessed: 2022
62. KubeEdge: KubeEdge Issue 1736. https://github.com/kubeedge/kubeedge/issues/1736. Last Accessed: 2022
63. KubeEdge: KubeEdge Security Team. https://github.com/kubeedge/community/tree/master/security-team. Last Accessed: 2022
64. KubeEdge: KubeEdge Sig-Security Team. https://github.com/kubeedge/community/tree/master/sig-security. Last Accessed: 2022
65. KubeEdge Edge framework examples: KubeEdge. https://kubeedge.io/en/docs/developer/device_crd/. Last Accessed: 2022
66. Kubernetes: Kubernetes pods. https://kubernetes.io/docs/concepts/workloads/pods/. Last Accessed: 2022

67. Kubernetes: Kubernetes Security Special Interest Group. https://github.com/kubernetes/community/tree/master/sig-security. Last Accessed: 2022
68. Kubernetes: Logging in Kubernetes. https://github.com/kubernetes/community/blob/master/contributors/devel/sig-instrumentation/logging.md. Last Accessed: 2022
69. Kubernetes: Open-source system for automating deployment, scaling, and management of containerized applications. https://kubernetes.io. Last Accessed: 2022
70. Kubernetes: Test Report on KubeEdge's Support for 100,000 Edge Nodes. https://kubeedge.io/en/blog/scalability-test-report/. Last Accessed: 2022
71. Linares-Vásquez, M., Bavota, G., Escobar-Velásquez, C.: An empirical study on android-related vulnerabilities. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 2–13 (2017). DOI 10.1109/MSR.2017.60
72. Linux foundation: https://www.kernel.org/. Last Accessed: 2022
73. Luxembourg National Research Fund: INSTRUCT - INtegrated Satellite – TeRrestrial Systems for Ubiquitous Beyond 5G CommunicaTions. https://instruct-ipbg.uni.lu/. Last Accessed: 2022
74. Mai, P.X., Goknil, A., Shar, L.K., Pastore, F., Briand, L.C., Shaame, S.: Modeling security and privacy requirements: a use case-driven approach. Information and Software Technology **100**, 165–182 (2018). Available at https://orbilu.uni.lu/handle/10993/35498
75. Mai, P.X., Pastore, F., Goknil, A., Briand, L.C.: MCP: A security testing tool driven by requirements. In: ICSE'19, pp. 55–58 (2019). DOI 10.1109/ICSE-Companion.2019.00037
76. MainFlux: Consulting and Security Audits. https://mainflux.com/consulting.html. Last Accessed: 2022
77. Mainflux: Mainflux. https://github.com/mainflux/mainflux/issues. Last Accessed: 2022
78. MainFlux: Mainflux Benchmark. https://github.com/mainflux/benchmark. Last Accessed: 2022
79. Mainflux Framework: Mainflux. https://mainflux.com/. Last Accessed: 2022
80. Malik, J., Pastore, F.: Replicability package. https://zenodo.org/record/7826981. DOI 10.5281/zenodo.7826981. Last Accessed: 2023
81. Manes, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The Art, Science, and Engineering of Fuzzing: A Survey. IEEE Transactions on Software Engineering **47**(11), 2312–2331 (2021). DOI 10.1109/TSE.2019.2946563
82. Mazuera-Rozo, A., Bautista-Mora, J., Linares-Vásquez, M., Rueda, S., Bavota, G.: The android os stack and its vulnerabilities: an empirical study. Empirical Software Engineering **24**(4), 2056–2101 (2019)
83. Metasploit: Metasploit edge computing framework. https://www.metasploit.com". Last Accessed: 2022
84. Microsoft: Accelerating IoT solution development and testing with Azure IoT Device Simulation. https://azure.microsoft.com/pl-pl/blog/accelerating-iot-solution-development-and-testing-with-azure-iot-device-simulation/. Last Accessed: 2022
85. Microsoft: Visual Studio Code Kubernetes Tools. https://marketplace.visualstudio.com/items?itemName=ms-kubernetes-tools.vscode-kubernetes-tools. Last Accessed: 2022
86. MITRE: https://github.com/kubeedge/kubeedge/issues/2362. Last Accessed: 2022
87. MITRE: Common Vulnerabilities and Exposures project. https://cve.mitre.org/cve/. Last Accessed: 2022
88. MITRE: Common Weaknesses Enumeration project. https://cwe.mitre.org. Last Accessed: 2022
89. MITRE Corporation: https://www.mitre.org. Last Accessed: 2022
90. MITRE: CVE-2014-5278: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-5278. Last Accessed: 2022
91. MITRE: CVE-2019-11252: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11252. Last Accessed: 2022
92. MITRE: CVE-2020-13597: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-13597. Last Accessed: 2022

93. MITRE: CVE-2020-15157: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15157. Last Accessed: 2022
94. MITRE: CVE-2020-2211: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-2211. Last Accessed: 2022
95. MITRE: CVE-2020-28914: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-28914. Last Accessed: 2022
96. MITRE: CVE-2020-35514: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-35514. Last Accessed: 2022
97. MITRE: CVE-2020-8557: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8557. Last Accessed: 2022
98. MITRE: CVE-2020-8558: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8558. Last Accessed: 2022
99. MITRE: CVE-2020-8559: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8559. Last Accessed: 2022
100. MITRE: CVE-2020-8563: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8563. Last Accessed: 2022
101. MITRE: CVE-2020-8565: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8565. Last Accessed: 2022
102. MITRE: CVE-2020-8566: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8566. Last Accessed: 2022
103. MITRE: CVE-2021-20218: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-20218. Last Accessed: 2022
104. MITRE: CVE-2021-21251: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-21251. Last Accessed: 2022
105. MITRE: CVE-2021-21334: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-21334. Last Accessed: 2022
106. MITRE: CVE-2021-25737: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-25737. Last Accessed: 2022
107. MITRE: CVE-2021-28166: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-28166. Last Accessed: 2022
108. MITRE: CVE-2021-28448: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-28448. Last Accessed: 2022
109. MITRE: CVE-2021-31938: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31938. Last Accessed: 2022
110. MITRE: CVE-2021-32783: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-32783. Last Accessed: 2022
111. MITRE: CVE-2021-34431: CVE-2021-34431. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-34431. Last Accessed: 2022
112. MITRE: CVE-2021-3499: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3499. Last Accessed: 2022
113. MITRE: CVE-2021-38545: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-38545. Last Accessed: 2022
114. MITRE: VE-2021-39159: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-39159. Last Accessed: 2022
115. Mosenia, A., Jha, N.K.: A comprehensive study of security of internet-of-things. IEEE Transactions on Emerging Topics in Computing **5**(4), 586–602 (2017). DOI 10.1109/TETC.2016.2606384
116. Mosquitto: https://mosquitto.org. Last Accessed: 2022
117. Mozilla foundation: https://www.mozilla.org. Last Accessed: 2022
118. MQTT: https://mqtt.org/. Last Accessed: 2022
119. Nassi, B., Pirutin, Y., Galor, T., Elovici, Y., Zadov, B.: Glowworm attack: Optical tempest sound recovery via a device's power indicator led. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS '21, p. 1900–1914. Association for Computing Machinery, New York, NY, USA (2021). DOI 10.1145/3460120.3484775. URL https://doi.org/10.1145/3460120.3484775
120. National Vulnerability Database: https://nvd.nist.gov. Last Accessed: 2022
121. OWASP: OWASp Top Ten. https://owasp.org/www-project-top-ten/. Last Accessed: 2022
122. Rancher: Rancher container management. https://rancher.com/. Last Accessed: 2022

123. SES Luxembourg: SES broadcasting services. https://www.ses.com/find-service/broadcasters. Last Accessed: 2022
124. SES Luxembourg: SES connectivity for commercial aviation. https://www.ses.com/find-service/commercial-aviation. Last Accessed: 2022
125. SES Luxembourg: SES connectivity for commercial maritime. https://www.ses.com/find-service/commercial-maritime. Last Accessed: 2022
126. SES Luxembourg: SES, leading satellite operator. https://ses.com/. Last Accessed: 2022
127. Shabtai, A., Elovici, Y., Rokach, L.: A Survey of Data Leakage Detection and Prevention Solutions. Springer Publishing Company, Incorporated (2012)
128. SonarQube: https://www.sonarqube.org/. Last Accessed: 2022
129. Sonarsource: Sonarsource tools for GO. "https://rules.sonarsource.com/go". Last Accessed: 2022
130. Stankovic, J.A.: Research directions for the internet of things. IEEE internet of things journal **1**(1), 3–9 (2014)
131. Suse: Suse software. https://www.suse.com. Last Accessed: 2022
132. Tabrizchi, H., Kuchaki Rafsanjani, M.: A survey on security challenges in cloud computing: issues, threats, and solutions. The journal of supercomputing **76**(12), 9493–9532 (2020)
133. The Chief I/O: 7 Static Analysis Tools to Secure and Build Stable Kubernetes Clusters . https://thechief.io/c/editorial/7-static-analysis-tools-to-secure-and-build-stable-kubernetes-clusters/. Last Accessed: 2022
134. Todd Erdley: How Edge Computing Unleashes Innovation in Live Streaming? https://www.tvtechnology.com/opinion/how-edge-computing-unleashes-innovation-in-live-streaming. Last Accessed: 2023
135. VerneMQ Broker: Vernemq. https://vernemq.com/. Last Accessed: 2022
136. Xiao, Y., Jia, Y., Liu, C., Cheng, X., Yu, J., Lv, W.: Edge computing security: State of the art and challenges. Proceedings of the IEEE **107**(8), 1608–1631 (2019). DOI 10.1109/JPROC.2019.2918437
137. Yomo Framework: Yomo. https://yomo.run/. Last Accessed: 2022
138. Zahid, M., Inayat, I., Daneva, M., Mehmood, Z.: Security risks in cyber physical systems—a systematic mapping study. Journal of Software: Evolution and Process **33**(9), e2346 (2021). DOI https://doi.org/10.1002/smr.2346. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2346
139. Zalewski, M.: American Fuzzy Lop: a security- oriented fuzzer (2020). URL http://lcamtuf.coredump.cx/afl/
140. Zaman, S., Adams, B., Hassan, A.E.: Security versus performance bugs: A case study on firefox. In: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, p. 93–102. Association for Computing Machinery, New York, NY, USA (2011). DOI 10.1145/1985441.1985457. URL https://doi.org/10.1145/1985441.1985457
141. Zetta: Zetta Automated Test Suite. https://github.com/zettajs/zetta/tree/master/test. Last Accessed: 2022
142. Zetta.: Zetta Edge Computing Framework. https://github.com/zettajs/zetta/wiki/Overview. Last Accessed: 2022
143. Zetta: Zetta GitHub bug reports. https://github.com/zettajs/zetta/issues. Last Accessed: 2022
144. Zetta: Zetta Issue 335. https://github.com/zettajs/zetta/issues/335. Last Accessed: 2022
145. Zetta Edge framework examples: https://www.zettajs.org/projects/. Last Accessed: 2022
146. Zhang, J.: An approach to facilitate reliability testing of web services components. In: 15th International Symposium on Software Reliability Engineering, pp. 210–218. IEEE (2004)