PhD-FSTM-2025-085

The Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 25/09/2025 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITE DU LUXEMBOURG
EN INFORMATIQUE

by

Jahanzaib Malik
Born on 13 July 1994 in Quetta (Pakistan)

ML-DRIVEN FIELD-BASED SECURITY TESTING OF SOFTWARE CONFIGURATION UPDATES IN MODERN SATCOM SYSTEMS

Dissertation defense committee
Dr. Fabrizio Pastore, Dissertation Supervisor
Assoc. Professor, University of Luxembourg
Dr. Domenico Bianculli, Member
Assoc. Professor, University of Luxembourg
Dr. Symeon CHATZINOTAS, Chairman
Professor, University of Luxembourg
Dr. Joel Grotz, SES Supervisor
Senior Manager, SES Luxembourg
Dr. Mathias Payer, Vice Chairman
Assoc. Professor, EPFL
Dr. Stefano Russo, Member
Professor, Università di Napoli Federico II

Acknowledgement

I wish to convey my profound appreciation to my supervisor, Prof. Fabrizio Pastore, whose steadfast mentorship, incisive counsel, and constant encouragement illuminated every stage of this research. His scholarly expertise, patience, and perceptive critiques not only refined the scope of my work but also sharpened its academic rigor.

My deepest gratitude goes to my parents, whose limitless faith in my abilities, countless sacrifices, and unflagging moral support fueled my determination. Their love has been an enduring source of resilience, inspiring me to navigate obstacles and pursue excellence.

I am equally indebted to my thesis committee and reviewers for their discerning questions and constructive observations, which significantly enhanced the clarity and depth of this dissertation. I extend heartfelt thanks to my colleagues in the Software Verification and Validation group for their stimulating discussions, collaborative spirit, and willingness to lend a helping hand whenever challenges arose.

I also express sincere appreciation to the members at the Interdisciplinary Centre for Security, Reliability and Trust (SnT) and to SES Luxembourg for providing outstanding facilities and essential financial support; your contributions were indispensable to the successful execution of this study.

Finally, I remain grateful to everyone who, in ways large or small, shaped this research—whether through illuminating conversations, insightful suggestions, or simple words of encouragement. Your collective generosity has left an indelible imprint on these pages.

This thesis stands as a testament to the concerted efforts and benevolence of many individuals and institutions. Your unwavering confidence in me and dedication to my success will forever be remembered with sincere gratitude.

Jahanzaib Malik

University of Luxembourg

12 June 2025

ML-driven field-based security testing of software configuration updates in modern SATCOM systems

Jahanzaib Malik

Abstract

Satellite communication (SATCOM) plays a critical role in everyday life, enabling vital services ranging from fast internet and disaster recovery to IoT connectivity. Rapid and cost-effective reconfiguration of existing infrastructure, particularly in satellite systems where new deployments are expensive and long, is essential to meet dynamic operational demands. A solution to achieve such an objective is the use of edge technologies with Software Defined Networks (SDNs), since the former enhance responsiveness and efficiency by processing data closer to end users, while the latter decouple network control from data forwarding, thus enabling swift adaptation to evolving scenarios. These solutions make SATCOM systems highly reconfigurable, but make their field configuration hardly foreseeable before deployment, with the risk of having vulnerable field configurations that may compromise dependability and system security, which is our focus in this work. To address these challenges, this dissertation presents three main contributions. The first contribution is an empirical study of 147 vulnerabilities in four widely used edge computing frameworks to understand why security issues persist. The study reveals that the complexity of edge environments makes exhaustive in-house testing impractical, and that many vulnerabilities affect confidentiality and are observed with configurations partially tested before deployment. These findings motivate the development of in-the-field testing approaches. The second contribution is the definition of FISTS, a field-based security testing approach for software-defined network (SDN) configuration updates. FISTS probes the network before and after updates and integrates a data analysis pipeline with unsupervised machine learning to detect anomalies. Evaluated with real and simulated SATCOM data, FISTS achieves high precision and recall, demonstrating its effectiveness and scalability. The third contribution is the extension of FISTS by incorporating human-in-the-loop feedback and weakly supervised learning. We evaluated 814 pipelines integrating 37 anomaly detection algorithms using 300 datasets. Our results show that combining expert feedback with algorithms like HBOS and OCSVM yields the best performance in terms of both recall and efficiency. These findings are an empirically well-founded practical solution to address a key problem for SATCOM systems.

Contents

1 Introduction **1**

1.1 Context and Motivation 1

1.2 Research Contributions 3

1.3 Dissemination 5

1.4 Dissertation Outline 5

2 Background and Related Work **7**

2.1 Edge Computing 7

2.1.1 Architecture of Edge Systems 8

2.1.2 Testing of Edge Systems 9

2.2 Field Failures 10

2.3 Security Testing Glossary 11

2.4 Related work on Empirical Studies on Software Vulnerabilities 14

2.5 Related work on SDN configurations 17

2.6 Network Scanning with NMAP 19

2.7 Data Clustering 20

2.8 Anomaly Detection 20

2.9 Machine Learning with Human-in-the-Loop 23

2.10 Weakly- and Semi- Supervised Anomaly Detection 24

3 Empirical Evaluation of Vulnerabilities in Edge Frameworks for SATCOM **27**

3.1 Introduction 27

3.2 Study Design 29

3.2.1 Data collection 33

3.2.2 Analysis Method 38

3.3 Results . . . 50
3.3.1 $\mathbf{RQ}_1$: Why are Edge vulnerabilities not detected during testing? . . . 50
3.3.2 $\mathbf{RQ}_2$: What are the types of components involved in a security failure? . . . 52
3.3.3 $\mathbf{RQ}_3$: What kind of failures are observed when an Edge vulnerability is exploited? . . . 54
3.3.4 $\mathbf{RQ}_4$: What is the nature of the precondition enabling the attacker to exploit Edge vulnerabilities? . . . 55
3.3.5 $\mathbf{RQ}_5$: What inputs enable exploiting Edge vulnerabilities? . . . 57
3.3.6 $\mathbf{RQ}_6$: What security properties are violated by Edge vulnerabilities? . . . 59
3.3.7 $\mathbf{RQ}_7$: What faults cause Edge vulnerabilities? . . . 62
3.3.8 $\mathbf{RQ}_8$: How severe are Edge vulnerabilities? . . . 66
3.4 Threats to validity . . . 69
3.4.1 Construct validity. . . . 69
3.4.2 Internal validity. . . . 69
3.4.3 Conclusion validity. . . . 71
3.4.4 External validity. . . . 75
3.5 Discussion and lessons learned . . . 76
3.6 Conclusion . . . 80

4 FISTS: Field Based Security Testing for SDN **82**
4.1 Introduction . . . 82
4.2 FISTS . . . 83
4.2.1 FISTS Steps 1 and 2: Traffic Generation and Response Monitoring . . . 84
4.2.2 FISTS Step 3: Host Matching and Comparison . . . 86
4.2.3 FISTS Step 4: Preprocessing . . . 89
4.2.4 FISTS Step 5: Vulnerability Prioritization . . . 89
4.2.5 FISTS usage . . . 91
4.3 Empirical Evaluation . . . 91
4.3.1 Experiment Setup . . . 92
4.3.2 RQ1: Pruning effectiveness . . . 94
4.3.3 RQ2: Best FISTS pipeline . . . 97
4.3.4 RQ3 - Accuracy of host matching module . . . 99

4.3.5 RQ4 - Scalability of approach . . . 100
4.3.6 Threats to validity . . . 101
4.4 Conclusion . . . 102

5 Anomaly Detection Algorithms for Field-based Security Testing of Updated SDN Configurations 103
5.1 Introduction . . . 103
5.2 Improved FISTS . . . 104
5.2.1 pFISTS-H: pFISTS with HITL . . . 105
5.2.2 pFISTS-W: Weakly Supervised pFISTS with HITL . . . 107
5.3 Empirical Evaluation . . . 109
5.3.1 Assessed sorting procedures . . . 110
5.3.2 Subjects of the study . . . 112
5.3.3 RQ1: Pruning effectiveness . . . 113
5.3.4 RQ2: Best FISTS pipeline . . . 116
5.3.5 RQ3 - Scalability of sorting procedures . . . 118
5.3.6 Discussion . . . 120
5.3.7 Threats to validity . . . 121
5.4 Conclusion . . . 122

6 Conclusion and Future Prospects 124
6.1 Conclusion . . . 124
6.2 Future Prospects . . . 125

List of Figures

2.1 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). 26

3.1 Research Questions: Objectives, Data analyzed, and Information derived. 30
3.2 Activity diagram for our approach in the manuscript 34
3.3 **RQ**$_1$: Why are Edge vulnerabilities not detected during testing? 51
3.4 **RQ**$_2$: What are the types of components involved in a security failure? 53
3.5 **RQ**$_3$: What kind of failures are observed when an Edge vulnerability is exploited? . . 54
3.6 **RQ**$_4$: What is the nature of the precondition enabling the attacker to exploit Edge vulnerabilities? 56
3.7 **RQ**$_{5A}$: How many steps are required to exploit an Edge vulnerability? 57
3.8 **RQ**$_{5B}$: What is the nature of the input action enabling the attacker to exploit a vulnerability? 58
3.9 **RQ**$_6$: What security properties are violated by Edge vulnerabilities? 59
3.10 **RQ**$_6$: Number of vulnerabilities affecting each security property based on NVD's CVSS entries; in total (Total) and grouped by impact (High/Low) 60
3.11 **RQ**$_{7C}$: What are the developer mistakes leading to Edge vulnerabilities? 63
3.12 **RQ**$_{7B}$: What are the erroneous software behaviors leading to Edge security failures? See Table 3.5 for detailed descriptions. 66
3.13 **RQ**$_8$: Distribution of NVD CVSS vulnerability scores 67
3.14 Attack complexity (High - H, Low - L) for vulnerabilities in Edge frameworks, based on NVD CVSS entries 68

3.15 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. 68

4.1 Overview of FISTS. It shows FISTS steps along with example outputs. The left side shows a network being tested by FISTS before and after reconfiguration. The NSCR output is exemplified by the provided table. 84
4.2 Dataset creation procedure and misconfiguration scenarios 85
4.3 FISTS's host matching algorithm. 88
4.4 Execution time of FISTS algorithms 101

5.1 Pseudo code for pFISTS 105
5.2 Pseudo code for pFISTS-H. Vector operators follow Python syntax. 105
5.3 Pseudo code for pFISTS-W. 108
5.4 Overview of Data Generation 110
5.5 Execution time of FISTS sorting procedures 121
5.6 Execution time of sorting procedures based on pFISTS-H 122
5.7 Execution time of sorting procedures based on pFISTS-W 123

List of Tables

2.1 Port states determined by Nmap. 19

3.1 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 3.2.1). 35
3.2 Vulnerabilities selected for each case study subject 36
3.3 Data collected for the vulnerabilities described in Section 3.2.2 37
3.4 $\mathbf{RQ}_{7A}$: What is the CWE vulnerability type?We report the number of vulnerabilities belonging to each vulnerability type discovered in our investigation. 61
3.5 Description of CWE Research Concepts (i.e., the erroneous software behaviours leading to security failures) 65
3.6 Vulnerabilities' impact based on CVSS NVD scores. 67
3.7 Statistical significance of the differences across RQ categories (Chi-squared goodness-of-fit test) 70
3.8 Distribution of RQs answers for Mainflux (M) and KubeEdge (K). 72
3.9 Statistical significance of the differences in RQ answers between Mainflux and KubeEdge, based on Table 3.8 (Fisher test) 73

4.1 Example execution data collected by NMAP for an initial configuration (O: open, C: closed) 85
4.2 Example execution data collected by NMAP for an updated configuration 85
4.3 Example of NSCR state changes 86
4.4 Prioritized NCSR report example 86
4.5 Results for synthetic, real, and realistic dataset with and without pruning 96

5.1 Abbreviations and Full Names 111
5.2 Results for synthetic, real, and realistic dataset with and without pruning 114

5.3 Distribution for synthetic. real. and realistic dataset with and without pruning 115
5.4 Table for Statistical Significance 120

Abbreviations

ADD	Active Anomaly Detection
API	Application Programming Interface
ARM	Advanced RISC Machine
BDD	Binary Decision Diagram
BT	Bad Testing
CBLOF	Clustering Based Local Outlier Factor
CNCF	Cloud Native Computing Foundation
CE	Combinatorial Explosion
CFP	Consecutive False Positive
COF	Connectivity-Based Outlier Factor
CONNACK	Connection Acknowledgment
CPS	Cyber Physical System
CVSS	Common Vulnerability Scoring System
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DC	Debugging Cost
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
DPF	Dynamic Data Plane Function
FISTS	Field Based Security Testing for Software Defined Networks
FP	False Positive
FtDD	Flow Table Decision Diagram
GNN	Graph Neural Network
HAC	Hierarchical Agglomerative Clustering
HBOS	Histogram-Based Outlier Score
HITM	Human in the Loop

HMC	Host Matching and Comparison
HSA	Header Space Analysis
HTTP	Hypertext Transfer Protocol
IEC	Ir-reproducible Execution Condition
IF	Isolation Forest
IoT	Internet of Things
IP	Internet Protocol
K3OS	Kubernetes
KM	K-means
KNN	k-Nearest Neighbors
KPCA	Kernel Principal Component Analysis
LAN	Local Area Network
LOF	Local Outlier Factor
LUNAR	Unifying Local Outlier Detection Methods via Graph Neural Networks
MAC	Media Access Control Address
MQTT	Message Queuing Telemetry Transport
Nmap	Network Mapper
NSCR	Network State Change Report
NVD	National Vulnerability Database
OCSVM	One Class Sub Vector Machine
ODL	Open Day Light
OF	Open Flow
OS	Operating System
OWASP	Open Worldwide Application Security Project
PCA	Principal Component Analysis
RQ	Research Question
SATCOM	Satellite Communication
SC	Stopping Condition
SDN	Software Defined Network
SD-WAN	Software Defined Wide Area Network
SES	Satellite Communication System
SHAC	Sorted Hierarchical Agglomerative Clustering

SIF	Sorted Isolation Forest
SKN	Sorted K-Means
SKNN	Sorted k-Nearest Neighbors
SMB	Server Message Block
SOTA	State-of-the-Art
SUT	Software Under Test
SYN	Synchronize
TCM-KNN	Transductive Confidence Machines for K-Nearest Neighbors
TCP	Transmission Control Protocol
TELCO	Telecommunication
UAC	Unknown Application Condition
UDP	User Datagram Protocol
UEC	Unknown Execution Condition
UML	Unified Modeling Language
URL	Uniform Resource Locator
VM	Virtual Machine
WAN	Wide Area Network
WS	Weakly Supervised
WSAD	Weakly Supervised Anomaly Detection
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Context and Motivation

Communication sectors, including satellite communication (SATCOM [1]), are critical for our daily life and thus need to evolve to enable new services for demanding customers while containing costs. Example SATCOM services include fast internet for aviation [2], navy [3], and land [4], disaster recovery [5], connection of remote communities [6], DTH broadcasting [7], IoT connectivity [8].

What enables the delivery of high-quality (e.g., high speed, low delay) services over an infrastructure that is costly to grow (e.g., due to significant expenses related to satellite deployment) is the quick and efficient reconfiguration of existing infrastructure components. Satellites play a crucial role in global connectivity, providing wide-area coverage and enabling communication in remote and hard-to-reach regions. However, deploying new satellite infrastructure is capital-intensive and has prolonged development cycles. Therefore, efficiently managing and optimizing the utilization of satellite resources is essential.

Edge computing [9] complements satellite systems by processing data closer to the end users, significantly reducing latency, bandwidth usage, and enhancing service responsiveness. Usage of edge infrastructure includes but is not limited to smart agriculture applications for optimizing farming practices, autonomous vehicle operations requiring immediate real-time data analysis, disaster management and emergency response through rapid processing of sensor and drone data, remote telemedicine enabling immediate healthcare support, maritime and offshore operational optimization, and industrial automation facilitating predictive maintenance and operational efficiency.

Software-Defined Networking (SDN) [10] is an innovative approach that transforms how networks

are designed and managed by separating the intelligence that directs traffic from the physical devices that forward it. This separation allows network administrators to control the entire infrastructure through a single, centralized interface rather than configuring each device individually. With SDN, policies and adjustments can be applied instantly and consistently across the network, improving responsiveness to changing demands. By turning network management into a software-driven process, SDN promotes flexibility, simplifies maintenance, and paves the way for more agile, automated operations.

SATCOM relies on both Edge stack and SDN to deliver its services, nevertheless; both present issues. In SDN, re-configurations are frequent; they may concern both the control and the application plane [11], [12]. Unfortunately, the presence of multiple applications (e.g., firewall, router) interacting with the data packets flowing through the SDN may lead to inconsistent or conflicting configurations, as reported in empirical studies[13], [14]. We refer to such situations with the generic term of *SDN misconfigurations*.

This PhD thesis aims to study and address the configuration problems introduced by Edge and SDN solutions through three contributions. As the first contribution of this thesis, we conducted a detailed empirical study that focuses on the reasons for vulnerabilities in open-source edge systems. The study focus on the nature of failure, pre-conditions, components involved, probable cause of earlier identification, severity, steps for reproducibility, affected security properties. The study reveals that the complexity of edge environments makes exhaustive in-house testing impractical, and that many vulnerabilities affect confidentiality and are observed with configurations partially tested before deployment.

As second contribution of this thesis, we propose FIeld-based Security Testing of SDN Configurations Updates (FISTS); a field-based testing approach that works in four steps. In the first step, probe the network before and after the reconfiguration. In the second step, we automatically match the hosts identified with the two network scans. In the third step, we prioritize the inspection of the scanned hosts by leveraging the results of anomaly detection algorithms. In the fourth step, engineers inspect the prioritized list of hosts and stops when the list does not present any more vulnerabilities. An engineer inspects the prioritized list of hosts and stops when the list does not present any more vulnerabilities. FISTS underlying idea can also be leveraged by engineers to monitor other issues in edge systems (i.e., Identifying affected nodes).

We conducted an empirical assessment with different datasets of network updates to determine the best configuration for FISTS by comparing results achieved with and without pruning, along with

different anomaly detection algorithms. Further, we demonstrate the accuracy of our host matching component. Last, we report on the scalability of our network scanning method, and the selected anomaly detection algorithms, by reporting on the time required to monitor and process 400 network nodes.

As third contribution of this thesis, we extended FISTS to incorporate Human-in-the-loop (HITL) and Weakly Supervised (WS) approach in FISTS, which utilize feedback from cybersecurity analyst in the learning process. The original implementation of FISTS leverages a subset of state-of-the-art anomaly detection algorithms. We conducted an empirical evaluation with different datasets of network updates to determine the best configuration for FISTS. Specifically, we compared the results achieved using 37 different anomaly detection algorithms combined with multiple configuration options (i.e., pruning and stopping conditions).

Beyond assessing FISTS, our empirical evaluation acts as a benchmark for a large set of anomaly detection algorithms, thus being of scientific interest for the broader academic community. Further, our findings could be helpful for sectors that may face similar challenges in the near future because they just started transitioning to software-defined systems (e.g., software-defined vehicles [15]).

This PhD work has been conducted in the context of the project INSTRUCT (INtegrated Satellite – TeRrestrial Systems for Ubiquitous Beyond 5G CommunicaTions)[1], funded by Luxembourg's *Fonds National de la Recherche (FNR), Grant IPBG19/14016225/INSTRUCT*. INSTRUCT is an industry-led research partnership between *SES* [16], leader in global content connectivity solutions, and the *Interdisciplinary Center for Security, Reliability and Trust (SnT), University of Luxembourg*. INSTRUCT envisions to create a ubiquitous, intelligent, self-organized and secured satellite–terrestrial integrated system exploiting ground–breaking SATCOM technologies. Leveraging the INSTRUCT project, during the PhD journey, SES engineers were met periodically. In our meetings, SES engineers provided additional context for our research problem, and validated the assumptions and feasibility of the solutions proposed in this PhD thesis. Further, they provided guidance and feedback for the benchmark datasets considered in our work.

1.2 Research Contributions

Building on the limitations of model based testing solution and challenges posed by evolving SATCOM networks, this thesis aims to identify what are the reasons for failures in complex systems including

[1] https://instruct-ipbg.uni.lu/

edge components and aims to address the testing of reconfigurations, which we found to be a critical cause, with a focus on SDNs; which are common in SATCOM. Further, a large portion of this thesis focuses on the development of an automated testing method to verify security properties of SDN based SATCOM systems.

This work investigates the following research questions.

QUESTION 1: What are the reasons for field failures in modern SATCOM systems including edge components?

To address this research question we conducted an empirical study based on 8 research question which help identify nature of vulnerabilities that affect such systems. What components are involved, what are the trigger conditions, and number of steps required to reproduce vulnerability?

QUESTION 2: Is Field-based testing solution capable of detecting vulnerabilities in the reconfiguration of SDN network?

To address this research question, we focus on the development of an automated approach called FISTS: Field based security testing for software defined networks. We capture the state of the network before and after the configuration of the system to create a representative dataset, that include information about devices on the network and their port states. Further, we used unsupervised learning algorithms and an automated sorting procedure to automate the inspection of devices to identify vulnerabilities.

QUESTION 3: What are the best anomaly detection solutions to detect vulnerabilities in field-based security testing of SDNs?

A number of anomaly detection algorithms are available and each may perform differently. To assess how state-of-the-art anomaly detection algorithms perform to detect such vulnerabilities; we extend FISTS to implement 7 new anomaly detection algorithms and a total of 37 new procedures. We also implement a feedback loop which incorporates expert feedback during the inspection of the network to iteratively learn anomalies in an incremental way. The two new techniques developed are Human-in-the-Loop (HITL) and Weakly Supervised (WS).

1.3 Dissemination

Below is the list of articles as an outcome of PhD thesis, presented alongside venues where work has been published.

- Malik, J., Pastore, F. An empirical study of vulnerabilities in edge frameworks to support security testing improvement. Empir Software Eng 28, 99 (2023). https://doi.org/10.1007/s10664-023-10330-x
- J. Malik and F. Pastore, "Field-Based Security Testing of SDN Configuration Updates" in IEEE Transactions on Reliability, doi: 10.1109/TR.2025.3531654
- J. Malik and F. Pastore, "Anomaly Detection Algorithms for Field-based Security Testing of Updated SDN Configurations" submitted to IEEE Transactions on Reliability

1.4 Dissertation Outline

Chapter 2 presents background information related to edge computing, testing of edge systems, field failure, Software Defined Networks (SDN), Network Scanning with NMAP, data clustering, and anomaly detection.

Chapter 3 presents the results of our empirical study which focuses on the identification of vulnerabilities that affect edge platforms. The focus is mainly on the identification of vulnerabilities that affect edge system therefore addressing **Research Question 1**. To identify vulnerabilities we selected 4 edge frameworks (K3OS, Zetta, Mainflux, and KubeEdge) from among the total of 14 frameworks, based on criteria (most active community and open-source). CVE database and GitHub bug reports are inspected in order to identify appropriate answers to the research questions. 147 vulnerabilities belonging to 4 selected frameworks are inspected in the study. A total of 8 research questions (few with multiple sub-questions) are use to identify the cause, affect, nature, and severity of each vulnerability. We address 8 research questions: (1) Why are Edge vulnerabilities not detected during testing? (2) What are the types of components involved in a security failure? (3) What kind of failures are observed when an Edge vulnerability is exploited? (4) What is the nature of the precondition enabling the attacker to exploit Edge vulnerabilities? (5) What Inputs Enable Exploiting Edge Vulnerabilities? (6) What Security Properties are Violated by Edge Vulnerabilities? (7) What Faults Cause Edge Vulnerabilities? (8) How Severe are Edge Vulnerabilities?

Chapter 4 describes our work on the development of FISTS: FIeld-based Security Testing of SDN Configurations Updates, directly addressing **Research Question 2**. We focus on the reconfigurations

of network services in the satellite communication sector, and target security requirements, which are often hard to verify; for example, although connectivity may function properly, confidentiality may be broken by packets forwarded to a wrong destination. We analyze FISTS based on 4 research questions: (1) Does pruning improve FISTS results? (2) What FISTS pipeline leads to the best results? (3) How does the host matching component impact on FISTS results? (4) How does FISTS scale?

In Chapter 5, we perform an extension of FISTS to leverage multiple anomaly detection algorithms i.e., HBOS, OCSVM, CBLOF, COF, LUNAR, PCA, and KPCA; implemented in weakly supervised and human-in-the-loop, directly addressing **Research Question 3**. Specifically, we perform an extensive assessment with a large set of anomaly detection algorithms integrated into FISTS, for a total of 37 sorting procedures, combined with options controlling pruning and stopping conditions, led to a total of 814 FISTS configurations being assessed. We evaluate results based on 3 research questions: (1) Does pruning improve the results obtained by the new sorting procedures integrated into FISTS? (2) What FISTS pipeline leads to the best results? (3) How do FISTS pipeline scale?

Chapter 2

Background and Related Work

In this chapter we provide a brief overview of Edge technology, Field Failures, Glossary of terminologies, related work on SDN configurations, NMAP tool, Data Clustering, Anomaly Detection, Machine Learning with Human-in-the-loop and Weakly and Semi Supervised Anomaly Detection.

2.1 Edge Computing

The Edge computing paradigm has been introduced to enable data transfer with extremely low latency for real-time services, bandwidth optimization, scalability, improved reliability, and lower operational cost. Well known services relying on Edge computing include, for example, E-sports [17], live streams broadcasts [18], [19], package tracking [20], and internet connectivity services for cruise lines [21] and aviation [22].

The development of services leveraging the Edge paradigm is supported by Edge frameworks; well known examples are KubeEdge [23], Yomo [24], K3os [25], and Mainflux [26]. 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* [27]. 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 [28]. 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.

2.1.1 Architecture of Edge Systems

Figure 2.1 provides a generic architecture of an Edge system. In Figure 2.1, 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 [29]; 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 2.1, 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 2.1, 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 2.1).

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 [30] 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 [31].

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.

2.1.2 Testing of Edge Systems

Edge frameworks are tested according to standard software engineering practices [32]. 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 [33]). The open-source frameworks considered in our study (i.e., KubeEdge, MainFlux [26], K3OS [25], and Zetta) are supported by private companies. KubeEdge is supported by the Cloud Native Computing Foundation and 27 additional private companies (e.g., ARM [34], Huawei [35], ci4rail [36]); 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 [37]. However, since industry participation in open source projects does not provide any guarantee about software security [38], 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 [39], integration [40] and system test cases [41]. Also, KubeEdge's development process includes on code review activities (e.g., contributions are revised by senior members[1]) and two security teams [42], [43] 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 [44]. Mainflux includes automated test cases and a dedicated benchmark [45]; further, MainFlux Labs perform security audits [46]. Finally, both K3OS [47] and Zetta [48] 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

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

missing [49]. 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 [50]; 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 [51]) 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 [52], [53], 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.

2.2 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. [54], 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 [55]–[59], 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.

In this thesis, we do not target faults affecting the functional properties (as done by Gazzola et al. [54]) 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. We address eight research questions instead of four.

2.3 Security Testing Glossary

Below, we provide definitions for security terminology appearing in the thesis; 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" [60].* 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 [61], 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)* [62] is a database managed by the Mitre corporation [63]. It lists publicly disclosed vulnerabilities. The CVE list is enumerated and managed by the CVE Numbering Authorities (CNA) [64]. 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 [63]. 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 [60]:

- *Confidentiality* concerns *"preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information"* [60].
- *Integrity* concerns *"guarding against improper information modification or destruction, and includes ensuring information non-repudiation and authenticity"* [60].

- *Availability* concerns *"ensuring timely and reliable access to and use of information"* [60].

NVD. The *National Vulnerability Database (NVD)* is the U.S. government repository of vulnerability data [65]. 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 [66]. 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).

By identifying anomalies in network data, we aim at detecting configuration updates that are faulty and, consequently, introduce vulnerabilities. In this section, we present related (verification of SDN configuration) and background approaches (network scanning with NMAP, data clustering, and anomaly detection).

2.4 Related work on Empirical Studies on Software Vulnerabilities

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 3.1), Collect defect reports and supplementary data (see sections 3.2.1 and 3.2.2), 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 3.3), Discuss impact and recommendations for industry (see section 3.5). The survey is based on 52 papers; however, only five of them focus on software vulnerabilities [67]–[71]. 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. [54].

Bavota et al. analyzed the vulnerabilities affecting the Android OS [68], [69]. 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. [71] 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 [67]. 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 [72]; 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 [73], the Apache HTTP Server [74], and the Linux operating system's kernel [75]. 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 [70]. 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 are 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 Edge frameworks (see CWE-789, CWE-1050, CWE-416, CWE-401 in Table 3.4, which count for 3.4% of the total), likely because Edge frameworks delegate memory management to widely adopted open-source OS's. Input sanitization issues affect both robotics and Edge systems; however,

they are less frequent in Edge systems (see CWE-707, 9.20% in Figure 3.12, Page 66). 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 3.12). The frequency of denial of service (i.e., availability) issues is in line with our findings (see Figure 3.9). 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 3.3.8), 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) [76]. 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 [77]. 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 [78], 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 solution 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. [79] 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.

2.5 Related work on SDN configurations

Approaches for the verification of SDN configurations rely on either model analysis or testing [80], as explained below.

Model-based approaches can be used to verify that SDN systems implement the provided configuration by relying on model-checking strategies to identify conflicts [80]. For example, Saied et al. automatically detect and correct misconfigurations in SDN switches [81]. Initially, they collect configuration data to construct formal models representing both intended and actual network behaviors using network switch flows. Then, they rely on model checking to identify discrepancies between these models, assess their impact, and align actual operations with intended configurations. Sadauoi et al., detect misconfiguration in OpenFlow flow rules utilizing a flow table decision diagram (FtDD) created from the flow rules of all the switches on the network [82]. They parse all possible paths that a packet could take in the given network topology to detect conflicting rules. Another approach relies on flow rules to create a binary decision diagram (BDD) that is processed by relying on Computational Tree Logic (CTL) queries to detect intra-rule misconfigurations [83]. Pan et al., instead, present a dedicated algorithm that relies on intervals derived from flow rules to check for redundant rules and minimize the total number of rules [84]. Le Brun et al., instead, automatically derive inputs (e.g., network packets) from the configuration itself (e.g., to request a whitelisted URL) to test if the SDN leads to the expected outcomes (e.g., URL can be accessed) [85].

The main limitation of the approaches above is that they rely on a model of the system, which should be either manually produced by the engineer or derived from the configurations provided to the SDN software. Both cases are expensive, indeed, the former directly requires engineers to produce a model, the latter requires the tool to be tightly integrated with the SDN software (e.g., it should process the proprietary format of commercial tools such as Versa [86]); such integration might be expensive to maintain. In addition to that, a problem that may affect model-based verification approaches relying

on model checking is that they require detailed modelling of the system (what is not modelled is not tested) and furthermore, may not detect problems visible only when the system is executed (e.g., because of delays during transmission).

Because of the above, automated testing approaches not relying on SDN modelling might be more adequate to address our problem. In the context of SDN testing, researchers have explored the use of fuzz testing [87], which led to the development of DELTA [88], BEADS [89], FragScapy [90], and AFLNET [91]. Although such fuzzers do not directly address the problem of testing SDN misconfigurations, they might be leveraged to generate traffic in place of NMAP. DELTA generates anomalous traffic by altering valid data packets. Most of the vulnerabilities detected by DELTA result from the modification of control flow messages, which is out of scope for our project; however, DELTA includes a Host agent that is useful for launching some attacks initiated by hosts (it generates network flows by creating new TCP connections or by using existing utilities, such as Tcpreplay). BEADS aims at discovering failures that may be triggered by the presence of malicious SDN components (applications or controllers) and therefore its applicability is out of scope since, as mentioned in the Introduction section, SATCOM providers assume that all the components installed on a SATCOM SDNs can be trusted and they are not interested in testing them. For the same reason, approaches for the identification of adversarial data planes are out of scope as well [92]. FragScapy is a command-line tool that can be used to generate 'fragroute-like' tests using Scapy [93]. FragScapy is useful to collect network data that can't be collected by NMAP in the presence of firewalls, but, alone, does not enable detecting misconfigurations. AFLNet targets implementations of network protocols thus targeting a problem different than ours.

In addition to FragScapy, other tools can generate test traffic and might be considered in the future to complement NMAP in FISTS; we describe them in the following. ATPG [94] generates packets to test network availability and verify packet latency along with the consistency of the data plane with respect to configuration specifications; BUZZ [95] focuses on policy implementations; Monocle [96] checks inconsistencies in the data plane; sPing [97] relies on packet injection to discover network loops, black holes, and link layer information; RuleScope [98] inspects SDN forwarding; SDN traceroute [99] is a packet-tracing tool for measuring paths in SDN networks; sTrace [100] is a packet-tracing tool for large, multi-domain SDN networks; FLOWGUARD [101] detects firewall policy violations; Libra [102] detects loops, black holes, and other reachability failures; HSA (Header Space Analysis) [103] is a protocol-agnostic framework to identify reachability failures, forwarding loops, and traffic isolation; FlowTest and GraphPlan [104] tackle stateful and dynamic data plane

Table 2.1: Port states determined by Nmap.

Port State	**Scan Type**	**Description**
Open	All	An application is actively accepting TCP connections, UDP datagrams, or SCTP associations on this port.
Closed	All	The port is accessible (it receives and responds to Nmap), but there is no application listening on it.
Filtered	All	NMAP cannot determine whether the port is open because packet filtering (e.g., firewall) prevents responses to reach NMAP.
Unfiltered	ACK	The port is accessible, but Nmap is unable to determine whether it is open or closed
Open|Filtered	UDP, IP, FIN, NULL, Xmas	NMAP is unable to determine whether a port is open or filtered because the selected scan type does not enable getting any response from the port.
Closed|Filtered	IP ID idle	NMAP is unable to determine whether the port is closed or filtered.

functions (DPFs), and policy requirements. To conclude, testing approaches aim to detect problems that are complementary to our (e.g., high latency, network loops); however, some of them might be leveraged to collect additional data to extend FISTS in the future. Feasible options include FragScapy to complement NMAP limitations, and collect latency data for anomaly detection; however, NMAP remains the best choice for monitoring available ports.

2.6 Network Scanning with NMAP

In this thesis, we rely on Network Mapper (NMAP) which is an open-source network scanning utility. NMAP is the industry-wide standard for exploring computer networks. NMAP offers various network scanning techniques and strategies which enable professionals to discover network hosts and identify running services. For every scanned host, NMAP reports the state of the probed ports; the possible cases are shown in Table 2.1.

Since service providers (e.g., SATCOM companies) may rely on proprietary SDN protocols that may change over time, we test the SDN in a black-box manner. Specifically, we do not rely on the NMAP options for OpenFlow (e.g., openflow-info script). The goal is not to test the SDN itself but to test the running configuration, and identifying the possibility of it being in a vulnerable state.

2.7 Data Clustering

Data clustering involves the partitioning of a dataset into groups, or clusters, based on the similarity among the data points within each group [105]–[107]. Clustering algorithms differ for the strategy adopted to partition the dataset; their performance depends on the characteristics of the dataset [106]. Among traditional clustering algorithms [108], hierarchical algorithms (e.g., HAC [109] and BIRCH [110]) perform well when clusters have a tree-shaped form, partition-based algorithms (e.g., K-means [111] and PAM [112]) assume that clusters have the form of hyper-spheres, density-based algorithms (e.g., DBSCAN [113], OPTICS [114], and Mean Shift [115]) do not make assumptions on the clustering shape but assume that points belonging to a same dense region belong to the same cluster. Below, we detail the approaches considered in this paper (i.e., HAC and K-means). *Hierarchical agglomerative clustering* (HAC) is a bottom up approach in which each data point starts in its own cluster; it iteratively merges pairs of clusters thus producing a dendrogram. The input of HAC is a matrix capturing the distance between every pair of data point. Grouping aims at minimizing one objective function; in our work we rely on the *ward* linkage method, which minimizes the variance of the clusters being merged [116]. *K-means* is the most popular clustering algorithm; it works by iteratively assigning data points to the nearest cluster centroid and recalculating the centroids until convergence. The objective is to minimize the within-cluster sum of squares, making data points within the same cluster as similar as possible while maximizing dissimilarity between clusters.

Both HAC and K-means do not automatically determine how many clusters should be generated. To determine the number of clusters that best separate our dataset we rely on Silhouette analysis [117], which is a state-of-the-practice solution for this purpose. Precisely, we execute our clustering algorithms multiple times, with a number of clusters ranging from 2 to N (in our experiments we set N to 50). According to Silhouette analysis we select the clustering output having the largest number of clusters with a max Silhouette coefficient above the Silhouette score. The Silhouette coefficient is computed for every data point in a cluster as the difference between the average distance from the data points within a cluster and the data points in the closest cluster normalized by the largest of the two. The Silhouette score is the average Silhouette index across all the datapoints in a cluster.

2.8 Anomaly Detection

Anomaly detection refers to the process of identifying and flagging data points that significantly differ from the expected or normal behavior within a given dataset [118], [119]. Specifically, we

aim to identify outliers within tabular data. A recent survey classifies fundamental outlier detection approaches into nearest-neighbor-based, projection-based, and clustering-based [119]. In this paper, we consider one representative for each category, which are Local Outlier Factor (LOF) [120] and Isolation Forest (IF) [121], for the first two categories. We selected these two algorithms because they are accurate (as reported in the survey) and largely adopted by data analysts (e.g., included in well-known libraries). Clustering-based methods, instead, rely on popular clustering algorithms (e.g., k-means, see Section 2.7) but differ for their rationales, which are: (a) anomalies do not belong to any cluster, (b) anomalies are away from the cluster centroids, (c) anomalies belong to small or sparse clusters [118], [119]. In Section 4.2.4, we introduce two approaches (SKM and SHAC) that rely on (c). Below, we describe the state-of-the-art anomaly detection approaches that we selected for FISTS.

Isolation Forest (IF) [121] is a powerful method for anomaly detection in high-dimensional datasets. It is based on the intuition that anomalies are often isolated from the majority of data points and can, therefore, be identified quicker. The IF algorithm constructs a binary tree-like structure known as an *Isolation Tree* by randomly selecting features and splitting the data at random values within those features. This partitioning process is repeated recursively until each data point is isolated within a leaf or all the data points in a leaf have the same values. Anomalies are then identified as those data points that require fewer splits to isolate, making IF particularly effective at detecting outliers or anomalies within complex and high-dimensional datasets. This algorithm has gained popularity for its speed and accuracy in anomaly detection tasks, making it a valuable tool in various domains, including fraud detection, network security, and quality control.

K-Nearest Neighbor (KNN) is a machine learning algorithm used for both classification and anomaly detection. In KNN, the K represents the number of nearest neighbors considered when making a prediction for a new data point. The KNN intuition that data points are likely similar to those in their proximity led to multiple anomaly detection approaches. One approach flags as anomalous those data points with a K-th neighbor having a distance above a threshold d [122]; another approach flags as anomalous the N points with the longest distance from the K-th neighbor[123]. However, both approaches require identifying a threshold, which can be challenging; further, the latter does not consider the local density of the neighborhood (e.g., the K-th neighbor might be far away, but the others are not) and did not lead to successful result in our experiments in Section 4.

Local Outlier Factor (LOF) [120] takes a density-based approach, considering the local neighborhood density of each data point. It computes the average distance of a point from its K nearest neighbors, which quantifies how isolated a point is within its immediate surroundings and is used

as an anomaly measure. High LOF values suggest that a point is in a less dense neighborhood and is likely an outlier. LOF identifies as anomalous those data points with an anomaly score above a threshold (default is 1.5). Building on LOF, in our FISTS paper, we introduced *Sorted KNN (sKNN)*, which sorts dataset entries based on the anomaly score computed by LOF.

Histogram-based Outlier Score (HBOS) [124] uses histograms to estimate the density of data points in each feature. For each feature (in our context, the state of a specific port), it constructs a histogram (either with static or dynamic bin widths) and calculates outlier scores based on the portion of data points are in these histograms, as follows

$$\text{HBOS}(p) = \sum_{i=0}^{d} \log\left(\frac{1}{\text{hist}_i(p)}\right) \tag{2.1}$$

with p being the data point to assess, d the number of features, $\text{hist}_i(p)$ the number of items in the histogram that contains p.

One-Class Support Vector Machines (OCSVMs) [125] consider anomalies as rare deviations from the distribution of datapoints in space. OCSVM establishes this boundary by maximizing the margin around the normal instances, thereby creating a *normalcy region*. OCSVM first constructs an hyperplane by solving an optimization problem that maximizes the margin between the data points and the origin while allowing some flexibility for outliers. This hyperplane then serves as the decision boundary for classifying new data points as normal or anomalous.

Cluster-Based Local Outlier Factor (CBLOF) [126] determines outlier scores by leveraging clusters obtained from a clustering algorithm (usually k-means). It categorizes clusters into small and large based on a specified threshold (8, in our experiments). For large clusters, the anomaly score is calculated by multiplying the distance of the data point from the cluster's centroid by the total number of data points in that cluster. For small clusters, the anomaly score is determined by multiplying the distance to the centroid of the nearest large cluster by the number of data points in the small cluster to which the data point belongs. In Section 4, instead, we proposed leveraging hierarchical agglomerative clustering (HAC) [109] and k-means [111] to group data points, and then, for anomaly detection, prioritize the inspection of data points belonging to smaller clusters; we called our approaches sorted HAC (SHAC) and sorted k-means (SKM).

Connectivity-Based Outlier Factor (COF) [127] computes an anomaly (i.e., outlier) score by comparing the distance of each data point from its nearest neighbors with the distances of those neighbors to their own nearest neighbors. First, COF calculates the sum of the distances between a

data point and its k nearest neighbors. Then, it divides the obtained distance by the average distance sum for its k neighbors. A high score indicates that the data point is relatively isolated compared to its neighbors, suggesting it is an outlier.

LUNAR [128] is a state-of-the-art graph-based anomaly detection approach. It uses a one-layer graph neural network (GNN) by constructing a graph where each data sample is represented as a node. Directed edges connect each node to its k nearest neighbors in the dataset. For each target node, LUNAR learns its anomaly score by aggregating information from these neighboring nodes. Unlike other GNN approaches, LUNAR builds its own graph from feature-based tabular data rather than using pre-existing graph datasets. It also inputs the distances between nodes and their nearest neighbors, rather than using raw feature vectors, making it more adaptable. Additionally, LUNAR employs a learnable message aggregation function to combine neighbor information, providing more flexibility compared to fixed aggregation methods used in many other GNNs.

Principal Component Analysis (PCA) [129] performs linear dimensionality reduction by projecting data onto a lower-dimensional subspace defined by the directions (principal components) that capture the most variance in the data. Outliers can be detected by examining how far points are from the expected structure in this lower-dimensional space, especially on the axes associated with smaller variance. The farther a point is from the central cluster in this reduced space, the more likely it is an outlier.

Kernel Principal Component Analysis (KPCA) [130] is an extension of Principal Component Analysis (PCA) that uses kernel methods to capture nonlinear relationships in the data. Instead of applying PCA directly to the original data, KPCA first maps the data into a higher-dimensional space using a kernel function, such as the Gaussian kernel.

2.9 Machine Learning with Human-in-the-Loop

Human-in-the-Loop (HITL) approaches for machine learning leverage feedback from the end user to improve the results of a machine learning model [131]–[133]. The most notable approach is *active learning*, which consists of improving the learning process by selectively choosing the most important data points to label thus reducing labeling costs [134]. For example, active learning has been applied to perform intrusion detection by leveraging Transductive Confidence Machines for K-Nearest Neighbors (TCM-KNN) [135]. TCM-KNN leverages the Kolmogorov-Smirnov test to determine uncertain datapoints, which are then presented to engineers for labeling. Results show that

99.7% accuracy can be achieved by labeling 40 out of 500 data points; however, based on our results, other anomaly detection methods may enable engineers to inspect less data points [136]. Other work integrates active learning with built-in attack graphs [137], which are not available in our context.

Other applications of HITL to anomaly detection aim at incorporating analyst feedback during the anomaly investigation process to adjust the ranking of the anomaly detector and reduce the number of false positives. This is the case for *Active Anomaly Discovery (AAD)*, which presents an interactive data exploration loop where, within each iteration, the algorithm selects a data instance to present to the expert as a potential anomaly, and the expert labels it as anomalous or nominal [138]. AAD incorporates such feedback by adjusting the parameters of the anomaly detection algorithm to prioritize confirmed anomalies. Implementations include updating a set of weights applied to an ensemble of histogram density estimators [138] or isolation forest trees[139]. Experiments with real-world data confirm that AAD enables detecting 80% more anomalies that traditional anomaly detection algorithms. Other AAD approaches tune the OCSVM learned parameters [140] or improve OCSVM variants by presenting to engineers both likely anomalous data points and data points that are uncertain [141].

Some works leverage the availability of historical data. In the context of wireless IoT network anomaly detection, successful applications include leveraging an existing dataset **KDD1999** to train XGBoost, which is then used to label the data under analysis, and collect expert feedback until reaching 99% precision and recall [142]. Other authors leverage historical data to improve the performance of OCSVM in cellular network anomaly detection [143].

2.10 Weakly- and Semi- Supervised Anomaly Detection

Weakly Supervised Anomaly Detection (WSAD) leverages existing labeled data to learn unlabeled data [144]. WSAD approaches address incomplete supervision, inexact supervision, and inaccurate supervision; in our context, we can leverage labels provided by human experts when inspecting a subset of data points, which matches the case of incomplete supervision. Several weakly supervised approaches addressing incomplete supervision leverage human-in-the-loop, and they correspond to the AAD approaches described in Section 2.9.

Besides WSAD approaches that leverage HITL, the literature reports on generic weakly supervised machine learning approaches that do not take advantage of expert inputs to address generic machine learning problems, not only anomaly detection. They are known as semi-supervised approaches [145].

In the following paragraphs, we discuss the feasibility of leveraging such approaches to perform anomaly detection in our context, using a small subset of labeled data points. In our discussion, we follow the classification provided in a recent survey [145]: wrapper methods, intrinsic methods, and transductive methods.

Wrapper methods leverage supervised classifiers to learn from unlabeled data; for example, the *self-training method* [146], which we leverage in this paper. The *self-training method*, first, trains a classifier on the available labeled data, then, uses the classifier to predict unlabeled data points, last, it adds the most confident predictions to the labeled data set. The process is iterated till all the data is labeled. Other wrapper methods work on *multiclass problems* (out of context in our work), or *leverage ensemble classifiers*, which require a large number of labeled data points that are not available in our context (e.g., it is too expensive to ask developers to label 10% of the data points, as suggested in related studies [147]). *Unsupervised preprocessing methods* include feature extraction from unlabeled data (i.e., what achieved by PCA and KPCA in Section 2.8 and included in our study), *cluster-then-label approaches*, which relate to the clustering approaches in Section 2.8 and included in our study, and *pre-training methods*, which consist of pre-training the internal layer of DNN-based approaches, but DNNs tend to perform poorly with tabular data [148] and we therefore excluded them from our investigation.

Intrinsic methods consist of variants of supervised-learning methods whose objective function takes into account unlabeled samples, they can be classified as maximum-margin methods, perturbation methods, and manifold methods. *Maximum-margin* methods (e.g., SVM-based or Gaussian-based) attempt to maximize the distance between the labeled data point and a decision boundary but their application to our context would require a costly, sufficiently large labeled subset of data points (e.g., 10% [149]). *Perturbation methods* alter labeled inputs during training, and they typically leverage DNNs thus being unlikely successful in our context for the reasons provided in the previous paragraph. *Manifold methods* aim to regularize or approximate the manifold so that data points with the same labels are close on a manifold; although not largely used in anomaly detection yet, the manifold regularization extension of OCSVM represents an interesting application for future work [150].

Transductive methods are also called *graph-based* methods; they either identify cuts to separate differently labeled data points or propagate their labels based on proximity concepts. The state of the art approach for graph-based anomaly detection (i.e., LUNAR) was described in Section 2.

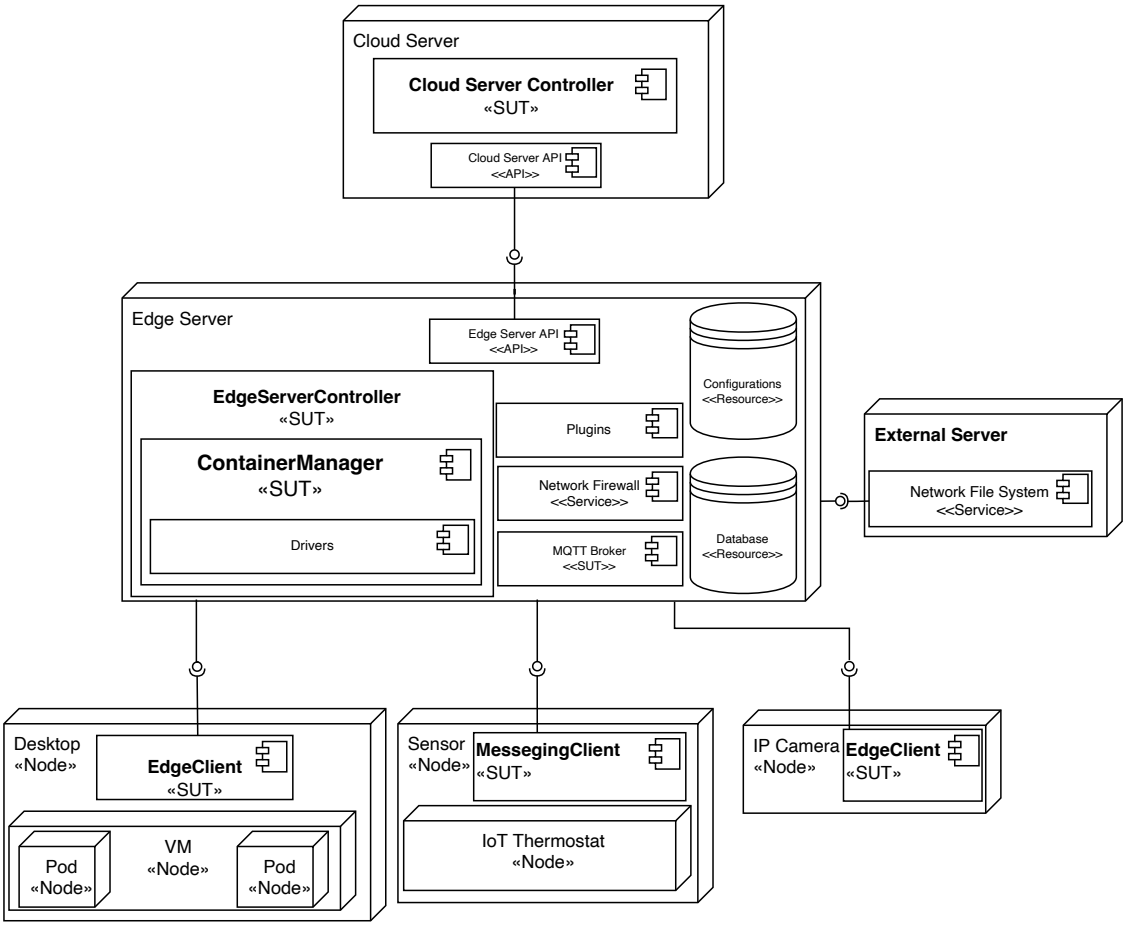


Figure 2.1: UML deployment diagram capturing the architecture of Edge systems. Ball and socket notation is used to distinguish between the component providing a service (ball) and the component relying on the service (socket).

Chapter 3

Empirical Evaluation of Vulnerabilities in Edge Frameworks for SATCOM

3.1 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 [151]. 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. [54].

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 [69], we analyzed the distribution of CWE [152] identifiers (i.e., types of weaknesses leading to

Edge vulnerabilities reported in the Common Vulnerabilities and Exposures (CVE) database [62]. Also, we studied their severity, based on the CVSS entries of the National Vulnerability Database (NVD) [65].

In total, we defined eight research questions. We surveyed 263 bug reports concerning four Edge frameworks (Mainflux [26], K3OS [25], KubeEdge [153], and Zetta [154]). 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 ($\mathbf{RQ}_6$). Confidential-

ity 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 [155]. 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 [156].

3.2 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.

This study addresses *eleven research questions* (including sub-quesitons), 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 3.1 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 3.1, we organize our RQs according to their objectives (i.e., identifying the

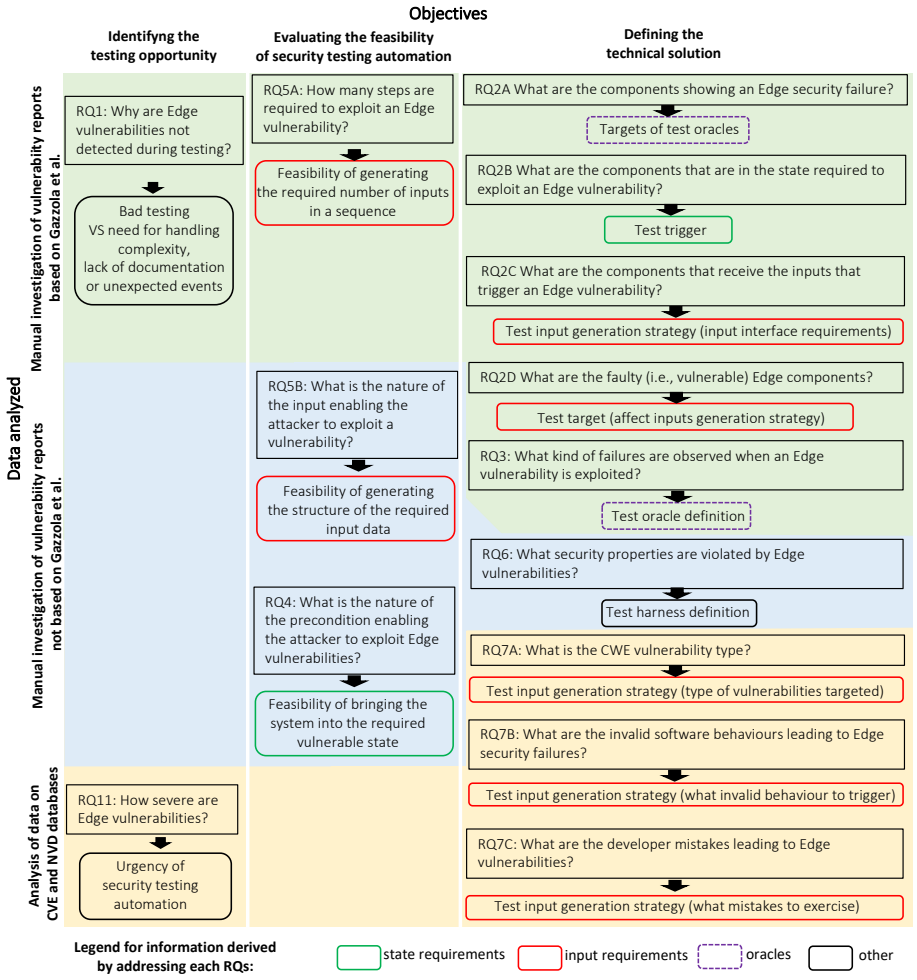


Figure 3.1: Research Questions: Objectives, Data analyzed, and Information derived.

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 [157], [158]. 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).

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:

- **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 2.3). 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 $\mathbf{RQ}_5$ into two separate questions:

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

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

$\mathbf{RQ}_6$**:** *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[1] 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.

$\mathbf{RQ}_7$**:** *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 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 erroneous software behaviors leading to Edge security failures?*

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

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

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

To evaluate the importance of improving Edge security testing approaches, **RQ$_8$** discusses severity based on NVD CVSS scores (see Section 2.3); severity analysis provides an indication about the urgency for automated security testing approaches.

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

3.2.1 Data collection

Figure 3.2 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 [159].

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 3.1, 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* [23] is the framework with the largest number of users providing comments in the issue

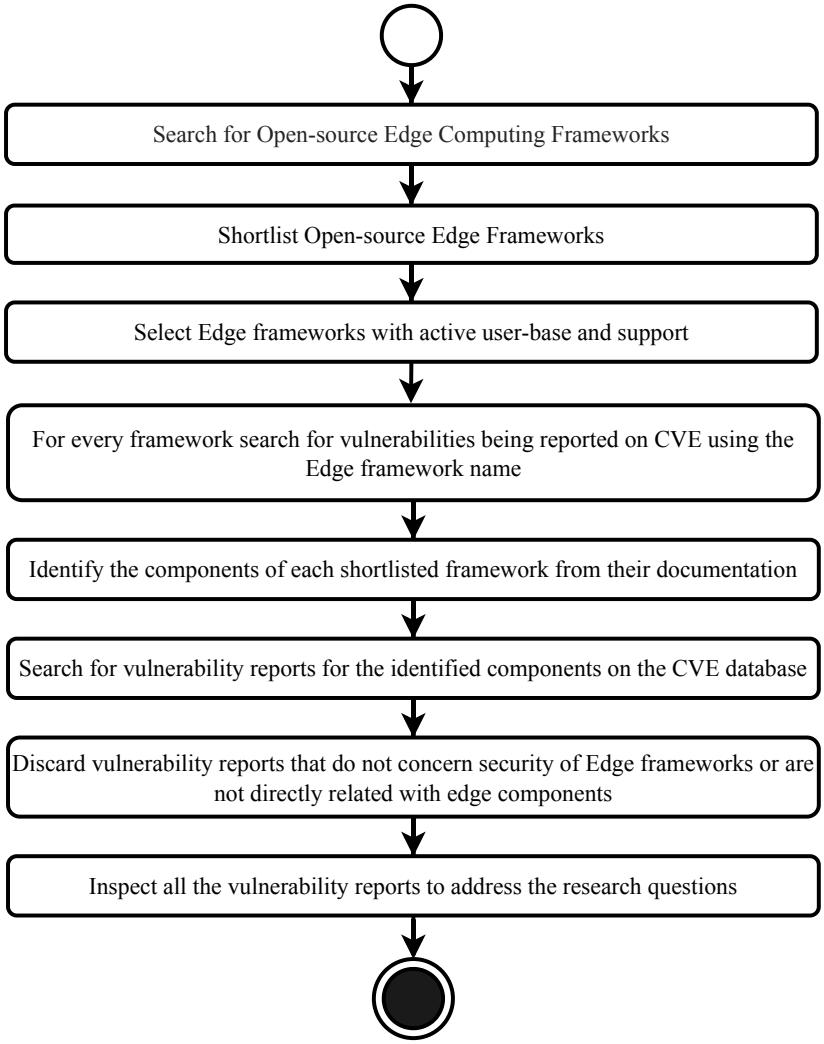


Figure 3.2: Activity diagram for our approach in the manuscript

tracker, probably because it is the most widely adopted one. It is developed as an open-source project by Cloud Native Computing Foundation (CNCF) [160]. It is an open-source product built upon Kubernetes [161], 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 [153] and CVE database [62], respectively.

Mainflux [26] 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 [162].

Zetta [154] is an open-source, Web-based Edge framework which provides connectivity to different

Table 3.1: 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 3.2.1).

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	✓	✓	✓

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 [163].

K3os [25] is an open-source Edge framework designed to work in low resource environments with the capability of being managed through a light-weight Kubernetes dashboard called *k3s*. For example, it is used by Rancher, a multi-Cloud container management platform [164].

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 [165] and VerneMQ [166]), 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

Table 3.2: 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*.

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 3.2 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).

Column *Vulnerabilities* in Table 3.2 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 [167] in Kubernetes [161], which concerns the Microsoft Visual Studio Code Kubernetes tool [168]. 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 3.2). 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 3.2), not the number of Edge vulnerabilities selected at the end of the process.

Table 3.3: Data collected for the vulnerabilities described in Section 3.2.2

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.

3.2.2 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 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 3.3 provides the data collected for the vulnerabilities mentioned as examples in the following paragraphs.

$\mathbf{RQ}_1$: 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 [169], 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 [170], 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., a hexadecimal string used for requesting access to a resource). Since the availability of logging level 9 is not well documented [171], 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 [172], 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[2], for different input parameters or configurations. When the system is large, the combination of values belonging to 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

[2] An input partition is a input region with equivalent values, from a testing perspective [173].

cost by exercising only few combinations of inputs (e.g., by relying on the *weak equivalence class testing strategy* [173]).

Please note that although functional testing approaches such as N-wise coverage [173] 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 [174]; 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 [54]. 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).

$\mathbf{RQ_2}$: 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 3.1, this research question is divided into four:

- $\mathbf{RQ_{2A}}$: *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?*

The above-mentioned RQs are addressed by tracing, for each vulnerability report, the types of components involved in the activities captured by **RQ**$_{2A}$- **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 [175], 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 [167], 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 **RQ**$_{2A}$, it is Helm (i.e., the plugin) what experiences the effect of the vulnerability (i.e., receives username and password). For **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 **RQ**$_{2C}$, the component receiving the input that triggers the vulnerability is the Helm plugin. For **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 [176], 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 [177] in Docker, in which the faulty component is the Mosquitto [165] MQTT Broker (SUT, according to Figure 2.1). 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 [178], 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 [179], 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 [180] 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 [181], 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 [182] in Raspberry Pi, which results in a *Glowworm* attack [183]. 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 [184]. 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 dis-

tributions; 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.

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 [55], Aysan et al. [56], Avizienis et al. [57], Chillarege et al. [58], and Cinque et al. [59]. 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* [54]. 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 [185], 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 [186], 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. [57] 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 [187] 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 [188]; it describes a case in which services bound to the loopback address are accessible by other pods and containers 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 [189] 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 [61], [190]).

  An example is provided by the Kubernetes vulnerability report CVE-2020-8563 [191], 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.

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. An example is the vulnerability CVE-2020-15127 [180] 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 [192], 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 [193], 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 [194], 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 [195] 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 [185], 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.

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 [196] reports a single step, consisting of executing the copy command on the Fabric8 plugin [197]. This step enables a malicious user to share restricted files and folders in the system. The docker vulnerability report CVE-2014-5278 [198], 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.

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 [199], 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.

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 2.3). 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 [193], 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 [200]. 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 [201], 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 [202], 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 Chapter 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).

$\mathbf{RQ}_7$: 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 3.3.7). The vulnerabilities without a CWE ID are not considered to address $\mathbf{RQ}_{7A}$.
- $\mathbf{RQ}_{7B}$: *What are the erroneous software behaviors leading to Edge security failures?* For each CWE ID associated to a vulnerability, we inspect the *Research Concept* taxonomy and identify the corresponding pillars.

- **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.

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).

3.3 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 3.2.2; the reader can also refer to Table 3.3 to search for example cases through the manuscript.

For **RQ**$_1$ to **RQ**$_{5A}$, we also compare our results with those of Gazzola et al. [54]; **RQ**$_4$ to **RQ**$_8$, instead, were not studied by Gazzola et al.

3.3.1 RQ$_1$: Why are Edge vulnerabilities not detected during testing?

Figure 3.3 presents our findings[3]; 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).

[3]Please note that, to save space, in the barcharts appearing in Figure 3.3, we hid part of the Y-axis scale; the hidden part is highlighted with the symbol //.

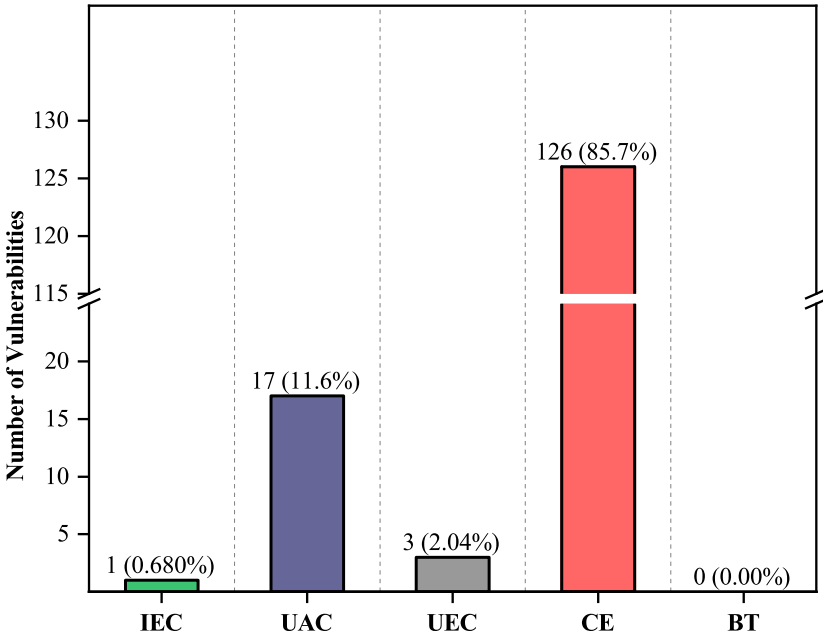


Figure 3.3: $\mathbf{RQ}_1$: Why are Edge vulnerabilities not detected during testing?

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 3.4.4), which minimizes this threat. Indeed, vulnerabilities due to bad documentation are low (i.e., only 11.6%). Similarly, UEC may be low because 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.

3.3.2 $\mathbf{RQ}_2$: What are the types of components involved in a security failure?

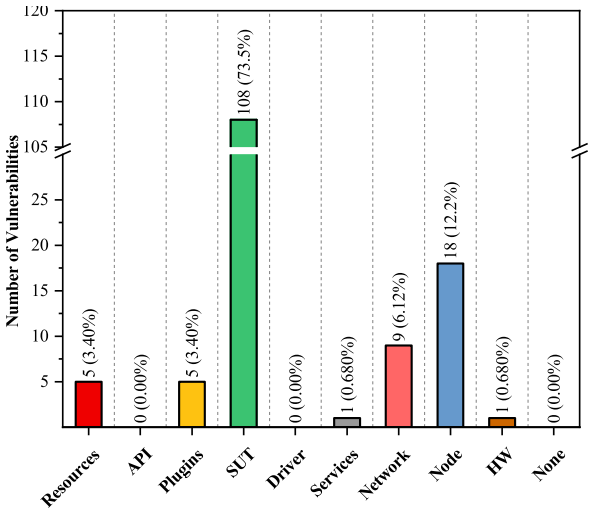
Figure 3.4 provides the distribution of each Edge component for the different sub-questions of $\mathbf{RQ}_2$. For all the sub-questions, SUT is the element with the highest number of entries, which is expected since we collected vulnerability reports concerning the SUT. However, the distribution of 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.

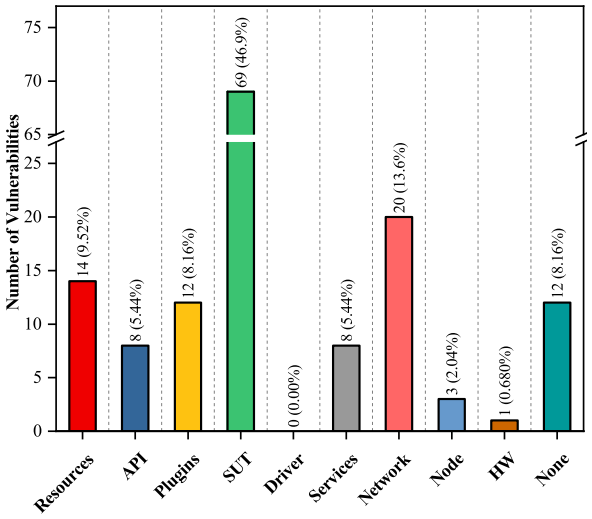
The consequences of vulnerabilities ($\mathbf{RQ}_{2A}$) mainly affect the SUT (108 cases) but also Nodes, Network, Plugins, Resources, and Services. The distribution for $\mathbf{RQ}_{2A}$ is different than the distribution observed for $\mathbf{RQ}_{2D}$; indeed, for $\mathbf{RQ}_{2A}$, we observe 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.

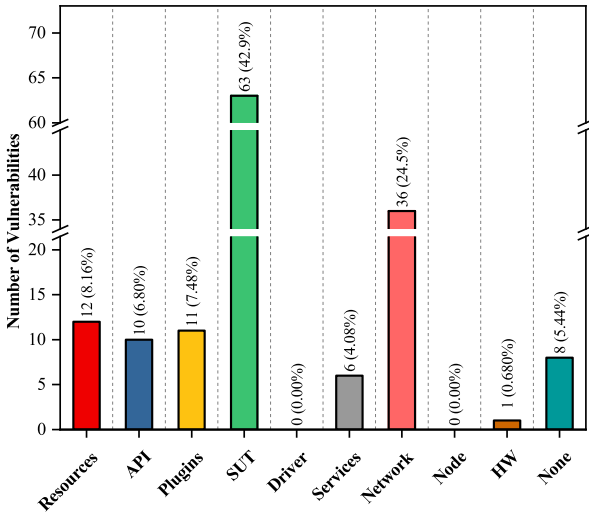
As for $\mathbf{RQ}_{2C}$, the component that receives the largest number of inputs triggering a vulnerability is the SUT (63 cases), followed by Network (36), Resources (12), Plugins (11), APIs (10), Services (6), and Hardware (1). Unsurprisingly the SUT interface (e.g., Web page, Service, or API), 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



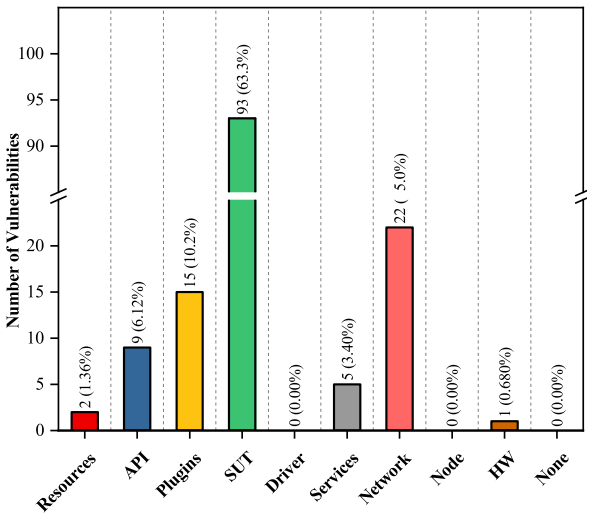
(a) **RQ**$_{2A}$: What are the components manifesting an Edge security failure?



(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?

Figure 3.4: **RQ**$_2$: What are the types of components involved in a security failure?

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).

3.3.3 $\mathbf{RQ}_3$: What kind of failures are observed when an Edge vulnerability is exploited?

Figure 3.5-A and -B present the distribution of the different types of security failures (left) and their detectability (right).

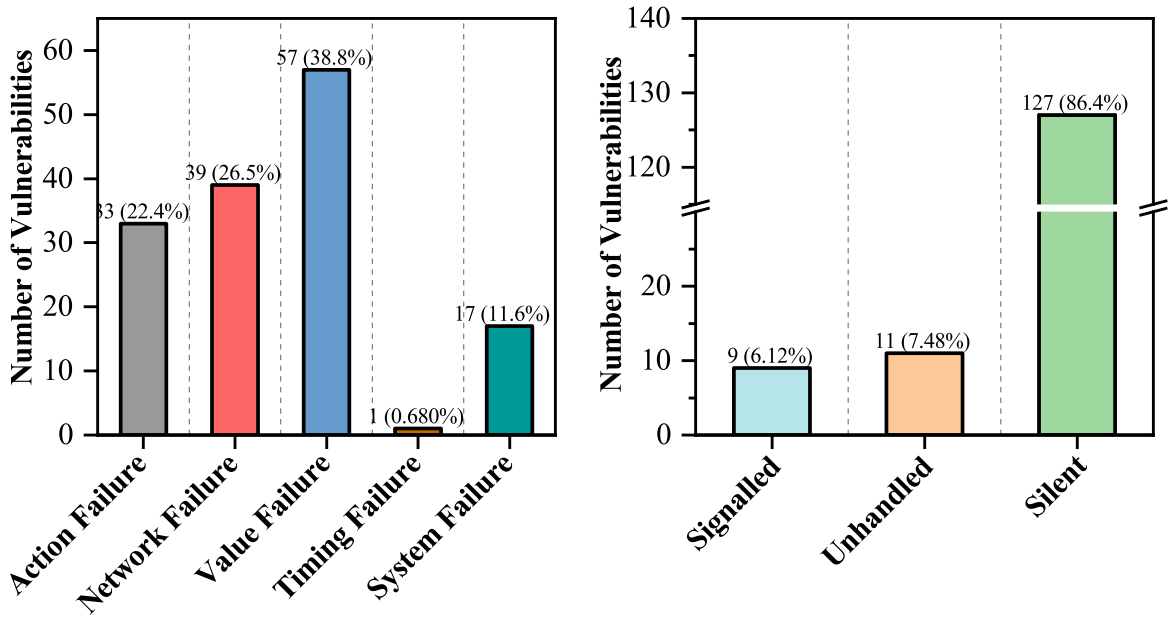


Figure 3.5: $\mathbf{RQ}_3$: What kind of failures are observed when an Edge vulnerability is exploited?

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 3.2.2).

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 [203]. 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 data); only security failures leading to the lack of communication or causing memory allocation errors can be easily detected (see section 3.2.2).

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. [54]) 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).

3.3.4 RQ_4: What is the nature of the precondition enabling the attacker to exploit Edge vulnerabilities?

Figure 3.6 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 Edge systems consist of many components that can be installed on different devices and require to be tuned according to the device

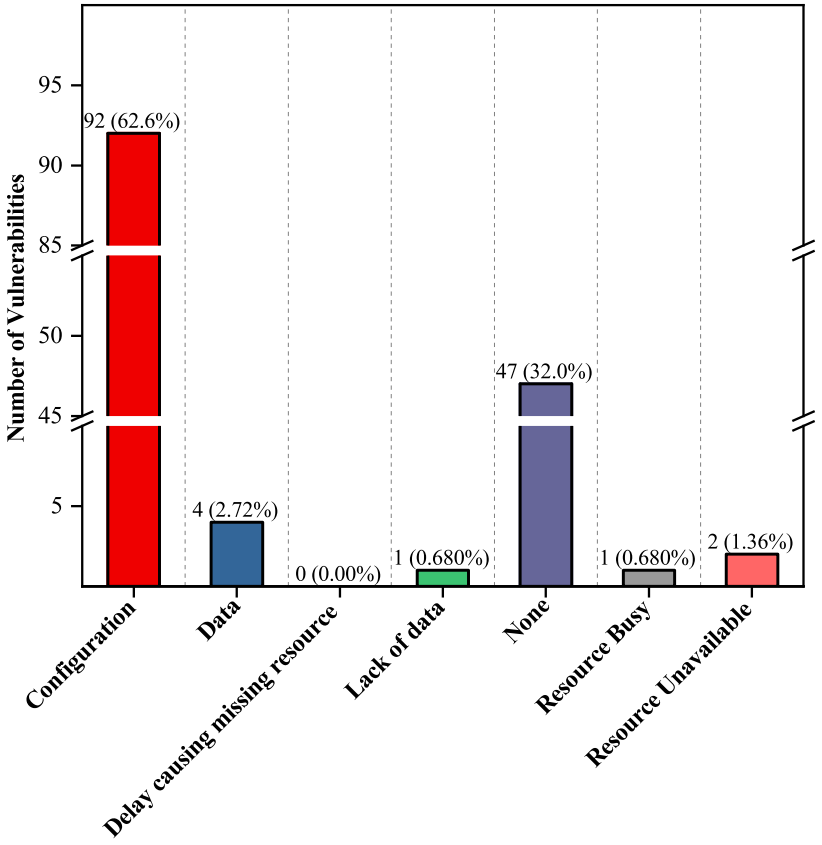


Figure 3.6: $\mathbf{RQ}_4$: What is the nature of the precondition enabling the attacker to exploit Edge vulnerabilities?

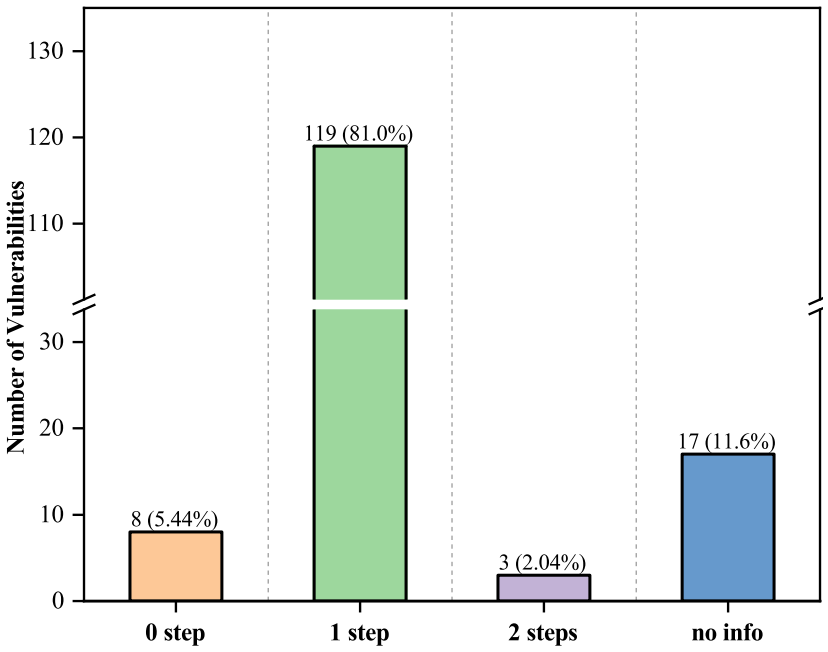


Figure 3.7: $\mathbf{RQ}_{5A}$: How many steps are required to exploit an Edge vulnerability?

characteristics and the service needs[4]; therefore, we may expect that testing its security properties for all the available configurations is particularly challenging and error prone..

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).

3.3.5 $\mathbf{RQ}_5$: What inputs enable exploiting Edge vulnerabilities?

Figure 3.7 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 3.2.2 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 logging vulnerabilities reported in $\mathbf{RQ}_{2C}$ (i.e., the system periodically logs sensitive information).

[4]Please note that dedicated static analysis tools had been developed to simplify the configuration of Kubernetes [204]

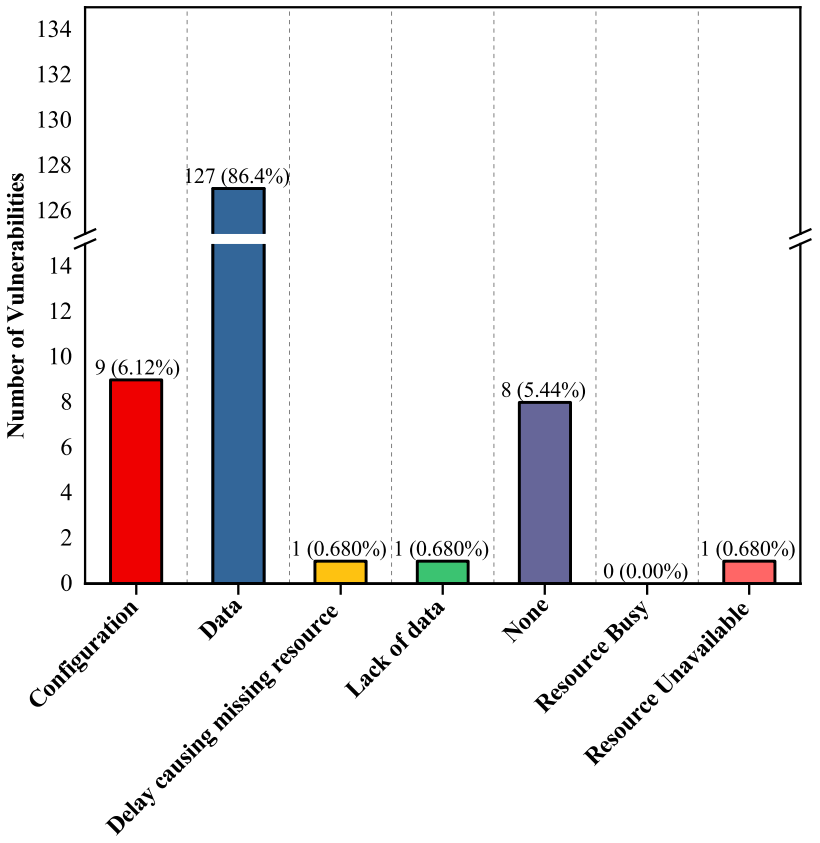


Figure 3.8: $\mathbf{RQ}_{5B}$: What is the nature of the input action enabling the attacker to exploit a vulnerability?

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 3.8 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).

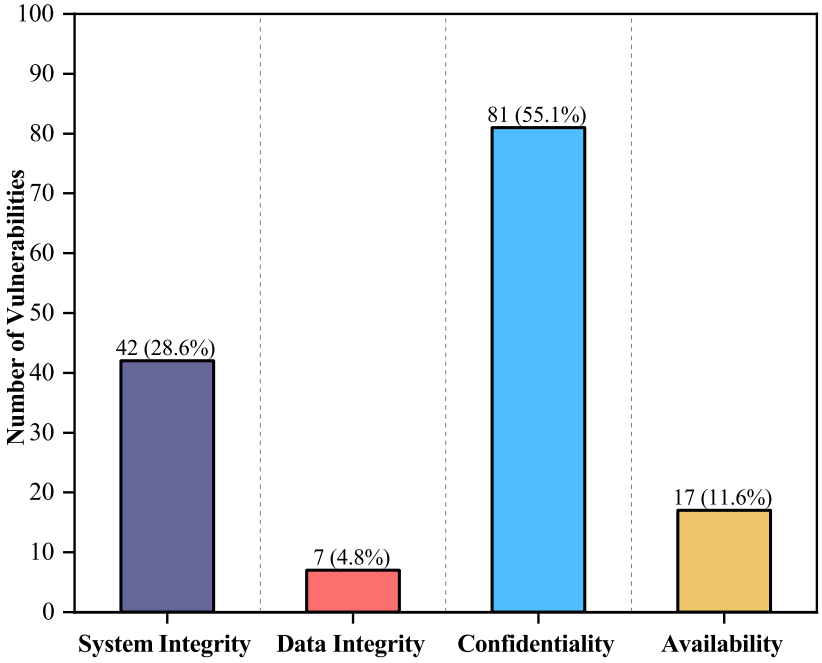


Figure 3.9: **RQ**$_6$: What security properties are violated by Edge vulnerabilities?

Unsurprinsingly, the eight cases not requiring any input (i.e., None) match the eight zero-step cases reported for **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.

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

Figure 3.9 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 3.10 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 3.9 (i.e., confidentiality is followed by integrity, while availability

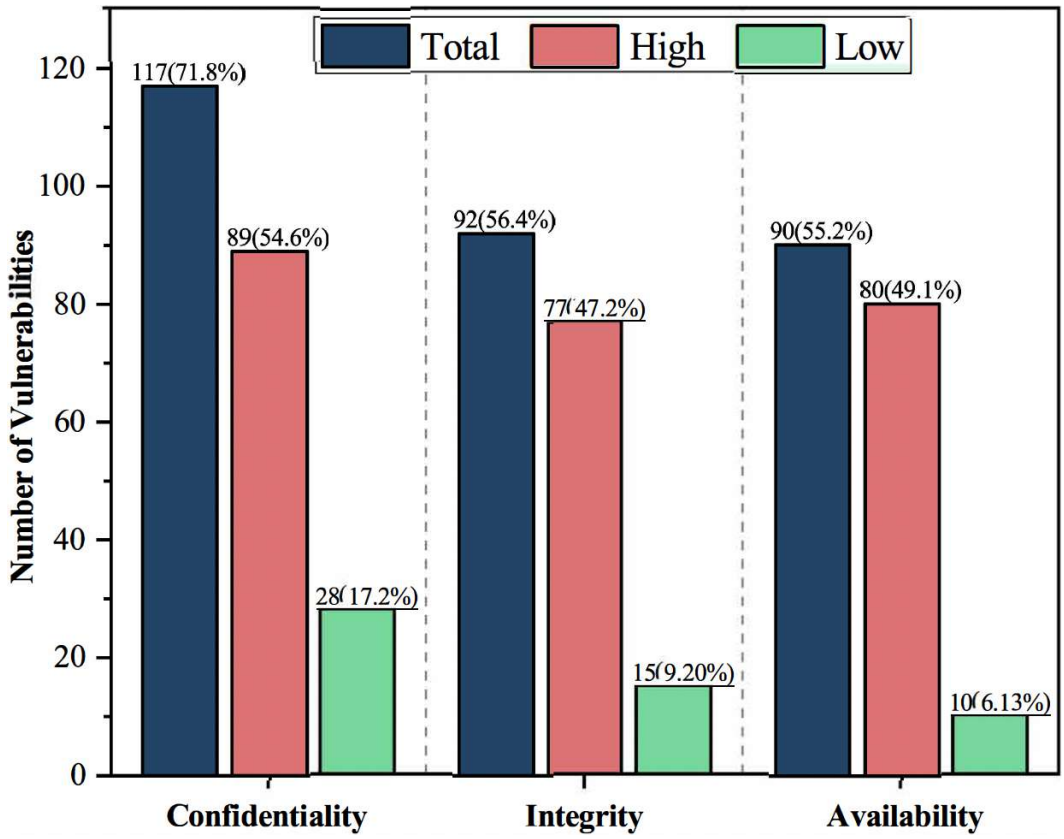


Figure 3.10: **RQ**$_6$: Number of vulnerabilities affecting each security property based on NVD's CVSS entries; in total (Total) and grouped by impact (High/Low)

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 3.4). Instead, by focusing on the vulnerability that is easier to exploit or that is violated first (i.e., our criteria, see section 3.2.2), we can observe that Confidentiality is the security property that is more likely affected by vulnerabilities (Figure 3.9), which provides a clear direction for the development of test automation tools.

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.

Table 3.4: $\mathbf{RQ}_{7A}$: What is the CWE vulnerability type?We report the number of vulnerabilities belonging to each vulnerability type discovered in our investigation.

Occurrences	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

3.3.7 RQ_7: What faults cause Edge vulnerabilities?

Figure 3.11, Figure 3.12, and Table 3.4, 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 3.5. In the following, we first discuss the distribution of the most frequent developer mistakes and erroneous software behaviors, 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 associated 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 3.12 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).

$\mathbf{RQ}_{7C}$. By looking at the distribution of developer mistakes (Figure 3.11), 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 3.4. 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

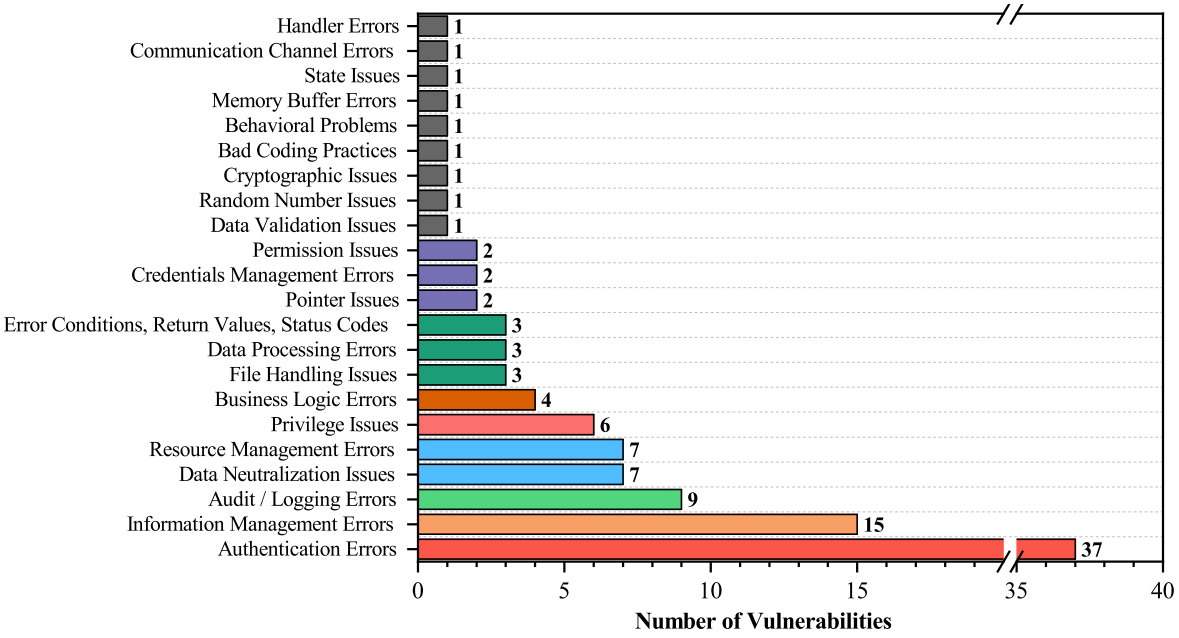


Figure 3.11: $\mathbf{RQ}_{7C}$: What are the developer mistakes leading to Edge vulnerabilities?

Control[5]

The second place in the ranking provided by Figure 3.11 is taken by *Information management errors*, which have been observed 15 times (13.76%). Such observation is reflected in Table 3.4; indeed, Information management errors relate to the control of resources, which concerns 52% of all the vulnerabilities with a CWE ID[6]. Among such CWE IDs, CWE-22 (Improper Limitation of a path name to a Restricted Directory) is ranked second in Table 3.4 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.

[5]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.

[6]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.

RQ$_{7B}$. Figure 3.12 shows that the research concepts with the highest number of vulnerabilities are CWE-664 (Improper Control of a Resource Through it's Lifetime) and CWE-284 (Improper Access Control) with 64 and 62 vulnerabilities, respectively, which is in line with our discussion above.

CWE-707 (Improper Neutralization) is the third most frequent case (15 vulnerabilities), in line with the number of data-integrity issues (**RQ**$_6$) and data neutralization mistakes (ranked fourth in the discussion for **RQ**$_{7C}$, above), which are often caused by code injection or path traversal vulnerabilities. For example, the path traversal vulnerabilities reported in Section 3.2.2 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 3.4 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 **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 [205] or Acunetix [206]); 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 [207]; 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 **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

Table 3.5: 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

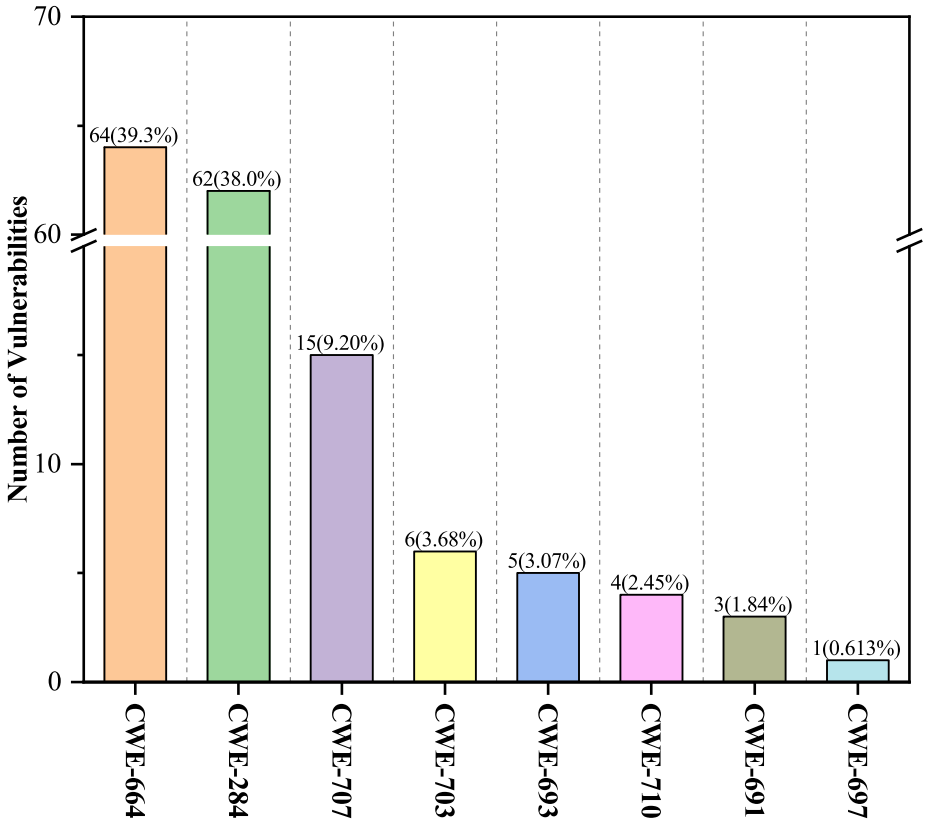


Figure 3.12: **RQ**$_{7B}$: What are the erroneous software behaviors leading to Edge security failures? See Table 3.5 for detailed descriptions.

code analysis tools such as SonarQube[208] (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 [209], for which a limited set of static analysis tools are available [210]–[212]. 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. esearch concept in our study for all the study subjects.

3.3.8 RQ$_8$: How severe are Edge vulnerabilities?

Figure 3.13 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 2.3).

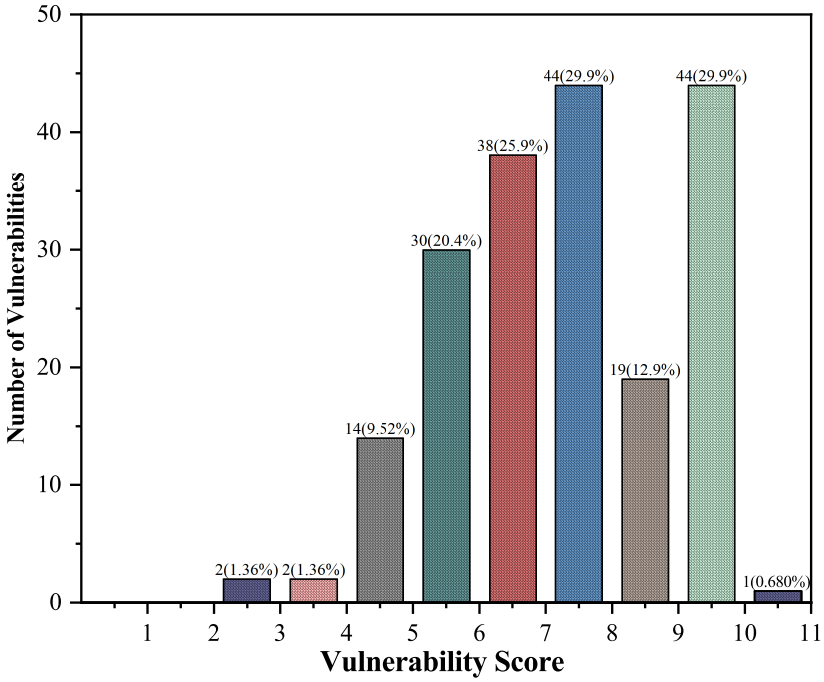


Figure 3.13: **RQ**$_8$: Distribution of NVD CVSS vulnerability scores

Table 3.6: 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%

Figure 3.14 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 3.15 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 3.6 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.

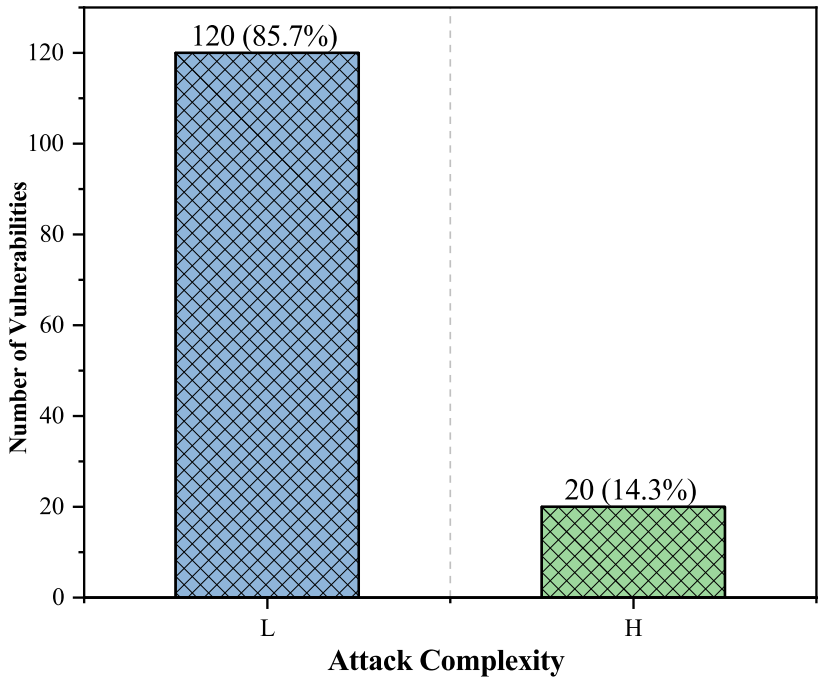


Figure 3.14: Attack complexity (High - H, Low - L) for vulnerabilities in Edge frameworks, based on NVD CVSS entries

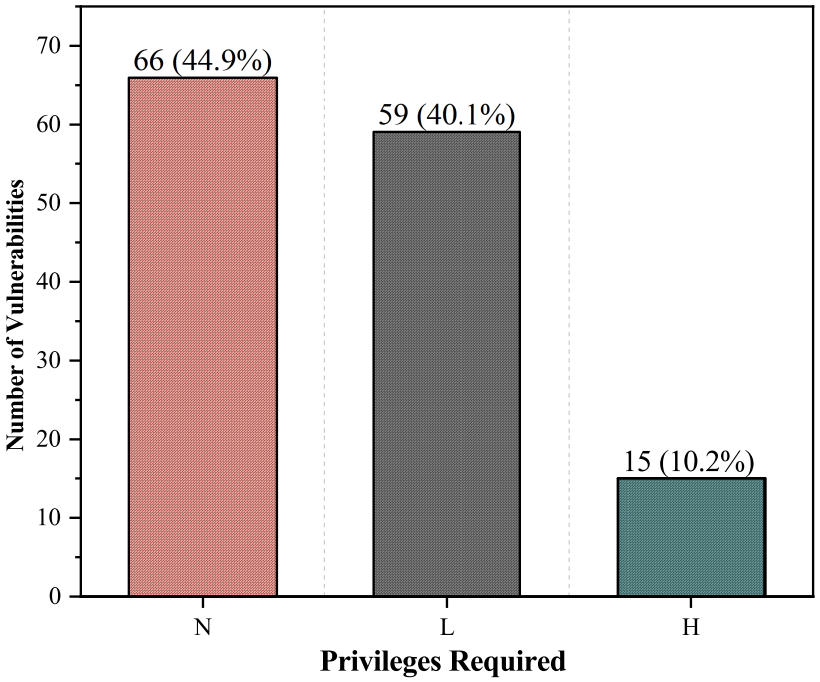


Figure 3.15: Privileges required to exploit vulnerabilities in Edge frameworks, based on NVD CVSS entries (High - H, Low - L, None - N). Please note that *None* is the most critical situation since an attacker can exploit a vulnerability without any specific privilege on the system.

3.4 Threats to validity

3.4.1 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 [69].

3.4.2 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 2.1.2). Further, the low number of vulnerabilities reported for K3OS depends on our choice of

Table 3.7: 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

not including Kubernetes vulnerabilities among K3OS total count (see section 3.2.1). 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, seesSection 3.2.1), which leads to more than 1900k lines of code (LOC). For Mainflux, we collected also vulnerabilities concerning Docker components such as Containerd[213], 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.

3.4.3 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 3.7 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 3.9), 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 $\mathbf{RQ}_6$ in Table 3.7). Instead, the data derived from NVD's CVSS (i.e., what we plot in Figure 3.10) 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 3.7) nor by looking at the vulnerabilities with the highest impact (see row $\boldsymbol{RQ}_6$ *NVD-High*). Indeed, as anticipated in section 3.3.6, 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 [214]. The Fisher's exact test computes the probability (p-value) of observing the distribution of vulnerability results across our RQ choices[7], 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 have significantly different distributions for the different categories) if the p-value is below 0.05.

Table 3.8 reports the distribution of vulnerabilities for each RQ answer, for both Kubedge and Mainflux; Table 3.9 reports the p-values computed with the Fisher's exact test. We can observe that,

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

Table 3.8: 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

Table 3.9: Statistical significance of the differences in RQ answers between Mainflux and KubeEdge, based on Table 3.8 (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

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 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*[8] and *path traversal or control of resources*[9]. 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% 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 mainly 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 Neutralization 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 3.5).

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

[8] 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.

[9] 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.

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.

3.4.4 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 [215]. 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 [216], [217]. Consequently, the vulnerabilities encountered in our investigation are likely representative 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 2.1.2). 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[10]) and two security teams [42], [43] 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 [44]. 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 [37]. 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

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

solutions. However the impact of this threat is limited because Zetta provides only 1 of the 147 vulnerabilities investigated in our study (see Table 3.2).

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.

3.5 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. [54]. 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 [155] identifies three field-based testing approaches: *online testing*, where test cases are executed directly on the 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 [155]. Two papers propose a technique that works offline [218], [219], five papers concern online testing [220]–[224]. Six techniques are

specification-based [218]–[223], one is fault-based [224]. They are activated by three different types of triggers: a policy [221], [222], the execution of a certain functionality defined either at run-time [218] or before [219], [223], and the deployment of a new module [220], [224]. 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 [220]–[222], [224], one approach targets only integer overflows [223], which were not observed in our analysis, two approaches [218], [219] 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 [225]. Fuzz testing tools (e.g., AFL [226]) 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) [227]. 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 [61]); in this regard, metamorphic testing might be an option since it has shown successful results when applied to test the security of Web systems [190]. 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 [190]; 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 [228]); 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 [229]; 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 [190].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 [220]–[222], [224]. However, related work has shown that it is feasible to detect authentication and authorization problems with metamorphic security testing [190].

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 [230], 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 [231] 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 minimize the number of vulnerabilities discovered by the 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, untill 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.

3.6 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).

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 behaviors 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).

Chapter 4

FISTS: Field Based Security Testing for SDN

4.1 Introduction

What enables the delivery of high-quality (e.g., high speed, low delay) services over an infrastructure that is costly to grow (e.g., because of costs related to satellite deployment) is the quick reconfiguration of the infrastructure, which enables reusing components (e.g., satellites) for different purposes. In the communication sector, a key technology to achieve such reconfiguration capabilities are Software Defined Networks (SDNs).

In the previous chapter (Chapter 3), we established that security vulnerabilities within edge systems predominantly affect confidentiality, consequently leading to authentication and authorization issues, especially during the reconfiguration processes of Software Defined Networks (SDNs). Reconfiguration emerged as a particularly critical challenge, primarily because it influences a wide range of components, each necessitating unique and specific testing approaches. Building upon these insights, this chapter is focused on the development of an approach that is capable of detecting such vulnerable situations during the reconfiguaiton of SDN.

To address the problem above, we propose FIeld-based Security Testing of SDN Configurations Updates (FISTS), a field-based testing approach that works in four steps. In the first step, we scan the SDN network, before and after a configuration change, with a predefined set of data packets (e.g., the ones generated by the NMAP security scanner [232]) and collect response data to determine the state (e.g., open ports) of the reachable hosts[1]. In the second step, we automatically match the hosts

[1]We use the term host to refer to a generic machine on the network, whether it is a server, router, switch, or end-user

identified with the two network scans (e.g., a same host changed IP) to determine changes in their state. In the third step, we prioritize the inspection of the scanned hosts by leveraging the results of anomaly detection algorithms. Specifically, our approach can leverage well known anomaly detection algorithms such as isolation forest [233] and local outlier factor [120]; further, we propose two solutions for the prioritization of anomalies that are based on KNN [123], HAC [109] and k-means [111].

As a result, the prioritized list includes items likely affected by vulnerabilities on top; note that in this context, a vulnerability is a specific host state (e.g., port 8080 is reachable) due to an SDN misconfiguration (e.g., a stateful firewall SDN application had not been set up). In the fourth step, engineers inspect the prioritized list of hosts and stops when the list does not present any more vulnerabilities; we empirically determined a threshold for the number of consecutive false positives to be observed before stopping.

We conducted an empirical assessment with different datasets of network updates to determine the best configuration for FISTS by comparing results achieved with and without pruning, along with different anomaly detection algorithms. Further, we demonstrate the accuracy of our host matching component. Last, we report on the scalability of our network scanning method, and the selected anomaly detection algorithms, by reporting on the time required to monitor and process 400 network nodes.

4.2 FISTS

FISTS relies on the intuition that security vulnerabilities induced by an SDN configuration change (hereafter, *SDN reconfiguration*) might be detected by identifying port state changes that look anomalous if compared to the other port state changes. FISTS implements a field-based testing approach that probes the network before and after an SDN reconfiguration to determine port states, and then, after identifying port state changes, even in the presence of re-assigned IP and MAC addresses, executes anomaly detection algorithms to determine the vulnerable changes that affect specific hosts.

FISTS can detect reconfigurations leading to security vulnerabilities due to erroneous port states including the one exemplified in Figure 4.1 (see section 4.2.1 for other cases). The original SDN configuration shows a Server Message Block (SMB) server (operates on port 445) connected to switch 3 and a high-priority flow rule on switch 2 that, to increase confidentiality by reaching only the subnet with the SMB server, forwards all traffic for port 445 to switch 3. In an update, the engineers move the

terminal.

SMB server to a dedicated switch (i.e., switch 6) but forget to update the rule on switch 2 to forward SMB traffic to the new destination switch 6 instead of switch 3. Consequently, all the traffic for the SMB server is routed to the wrong subnet, which poses serious confidentiality issues even if packets may be re-routed to reach switch 6 (not shown in the picture). Note that Figure 4.1 assumes proactive SDN flows (i.e., packets flow is preconfigured); although reactive SDN flows (i.e., rules are created as packets come into the switch) may prevent such mistakes, they cannot be used in several context (e.g., because of confidentiality requirements preventing the free routing of packets).

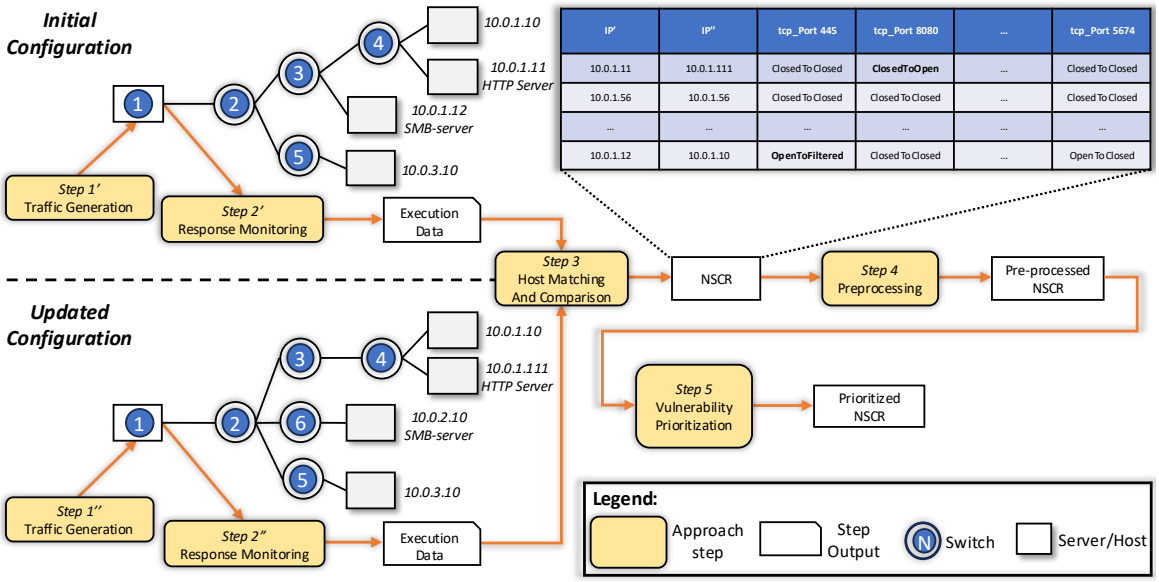
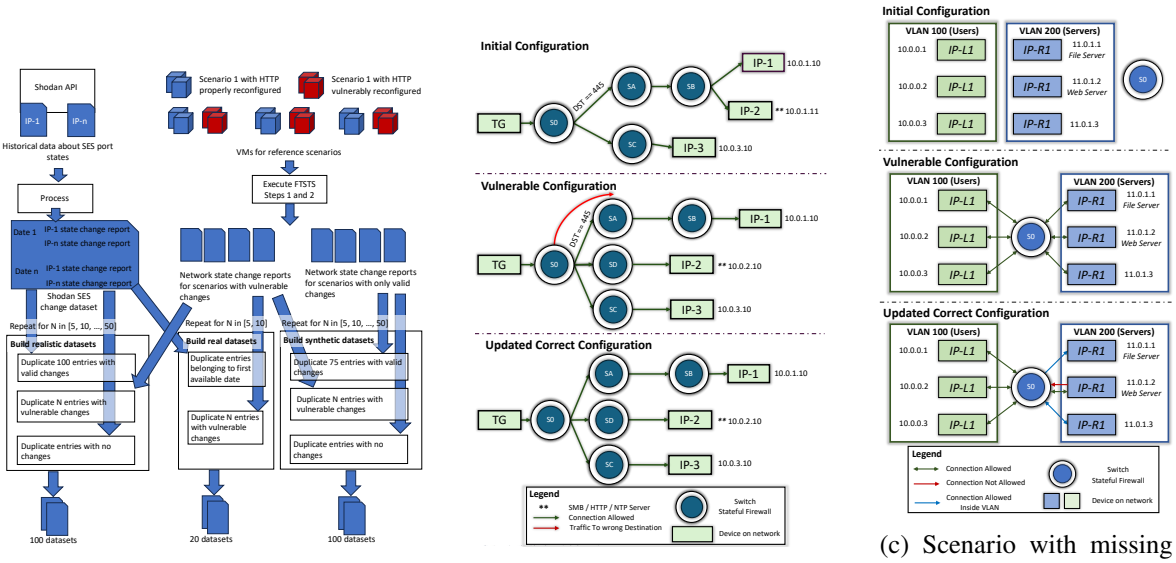


Figure 4.1: Overview of FISTS. It shows FISTS steps along with example outputs. The left side shows a network being tested by FISTS before and after reconfiguration. The NSCR output is exemplified by the provided table.

FISTS works in four steps, depicted in Figure 4.1 and described below. Note that both Step 1 and Step 2 are performed twice, before and after an SDN reconfiguration.

4.2.1 FISTS Steps 1 and 2: Traffic Generation and Response Monitoring

In Step 1, FISTS generates network traffic for reconnaissance purposes, the responses generated by the hosts in the network are collected in Step 2 and processed to produce information about the port states of each host in the network. Our current implementation of FISTS relies on NMAP for both Step 1 and Step 2 because NMAP can generate data for network reconnaissance (i.e., FISTS Step 1) and produces scan result files in XML format (i.e., FISTS Step 2). We rely on NMAP TCP SYN and UDP scans. However, other tools might be integrated in the future to extend our capabilities; for example,



(a) Datasets creation procedure (b) Scenario with redundant rule (c) Scenario with missing firewall rules

Figure 4.2: Dataset creation procedure and misconfiguration scenarios

Table 4.1: Example execution data collected by NMAP for an initial configuration (O: open, C: closed)

IP	MAC	tcp 22	udp 22	tcp 123	udp 123	tcp 445	...	tcp 8080	udp 8080
10.0.1.56	00:1A:2B:3C:4D:5E	C	C	C	**O**	C	...	C	C
10.0.1.12	08:15:23:42:67:94	C	C	C	C	**O**	...	C	C
10.0.1.11	2A:9F:C7:0E:3D:81	C	C	C	C	C	...	**O**	C
10.0.3.10	5C:73:2F:A1:6E:B0	**O**	C	C	C	C	...	C	C

we may rely on FragScapy [90] to collect additional information masked by stateful firewalls. At a high-level, the result of Step 2 is a dataset where each data entry provides the following information for one host in the network:

- MAC address
- IP address
- for all the ports,
  - ⇒ whether they are UDP or TCP
  - ⇒ if they are in one of the following states: Open, Closed, Filtered, Open|Filtered

Tables 4.1 and 4.2 provide examples of data entries collected for the two configurations appearing

Table 4.2: Example execution data collected by NMAP for an updated configuration

IP	MAC	tcp22	udp22	tcp123	udp123	tcp445	...	tcp8080	udp8080
10.0.1.56	00:1A:2B:3C:4D:5E	C	C	C	**C**	C	...	C	C
10.0.2.10	08:15:23:42:67:94	C	C	C	C	**F**	...	C	C
10.0.1.111	2A:9F:C7:0E:3D:81	C	C	C	C	C	...	**O**	C
10.0.3.10	5C:73:2F:A1:6E:B0	**O**	C	C	C	C	...	C	C

Table 4.3: Example of NSCR state changes

Initial	Updated	Label
Closed	Open	ClosedToOpen
Open	Closed	OpenToClosed
Open|Filtered	Closed	OpenFilteredToClosed

Table 4.4: Prioritized NCSR report example

IP'	IP"	tcp22	udp22	tcp445	udp445	tcp8080	...	udp5674	Anomaly Sc
10.0.1.12	10.0.2.10	ClosedToClosed	ClosedToClosed	**OpenToFiltered**	ClosedToClosed	ClosedToClosed	...	ClosedToClosed	0.97
10.0.1.22	10.0.1.22	ClosedToClosed	ClosedToClosed	ClosedToClosed	ClosedToClosed	ClosedToClosed	...	**OpenToFiltered**	0.93
10.0.1.56	10.0.1.56	ClosedToClosed	ClosedToClosed	ClosedToClosed	ClosedToClosed	ClosedToClosed	...	ClosedToClosed	0.82
10.0.3.10	10.0.3.10	ClosedToClosed	ClosedToClosed	ClosedToClosed	ClosedToClosed	ClosedToClosed	...	ClosedToClosed	0.77
...	...	...	...	...	...	...	...	...	...
10.0.1.11	10.0.1.111	ClosedToClosed	ClosedToClosed	ClosedToClosed	ClosedToClosed	**OpenToOpen**	...	ClosedToClosed	0.39

in Figure 4.1. To enable anomaly detection, features should be consistent across entries; hence, for every port, we capture information for both the TCP and the UDP protocol as separate features (i.e., columns).

4.2.2 FISTS Step 3: Host Matching and Comparison

In Step 3, the Host Matching and Comparison (HMC) module processes the data collected before (by Step 2') and after (by Step 2") a reconfiguration to (1) determine what data entries belong to a same host in the two configurations and (2) determine what are the port state changes for each host.

Determining what data entries belong to a same host is necessary because SDN reconfigurations may change the IP or MAC address of one host (e.g., a server) without changing anything else. Not noticing hosts with addresses being changed may lead to incorrect identification of port state changes. For example, in Figure 4.1, the IP of the SMB server may change from 10.0.1.12 to 10.0.2.10 (MAC address remains the same); without noticing that the SMB sever has been simply moved to another IP address, FISTS may interpret all the open ports (e.g., 445) of IP 10.0.1.12 as becoming closed and port 445 of IP 10.0.2.10 as becoming filtered, which may misguide the FISTS anomaly detection algorithm used in Step 5 or the end-user processing FISTS results. Similarly, a host with a Web server has been moved from IP 10.0.1.11 to IP 10.0.1.111, without noticing that the host is likely the same, we may interpret all the ports of IP 10.0.1.11 as becoming closed and port 8080 of IP 10.0.1.111 as becoming open, which may lead to false positives (e.g., such bulk changes of port states may be reported as anomalous by an anomaly detection algorithm).

Since hosts with a change in their IP or MAC address may also present changes in their port states (e.g., the SMB server in our running example); it is not possible to simply look at matching port

states to identify the hosts assigned to a different IP or MAC address. Therefore, we have developed a dedicated greedy algorithm; its pseudo-code appears in Figure 4.3. Our matching algorithm performs $n \times m$ comparisons between the n data entries from the initial configuration and the m data entries from the updated configuration (Lines 4 and 5 in Figure 4.3). For each data entry pair, it determines if the IP, MAC address, and state of each port match (Lines 6 to 19). Such information is used to assign a matching score to each pair, as follows:

$$matching\_score = score\_ip + score\_mac + score\_ports$$

The value of $score\_ip$ is set to 0.2 when two IPs match, otherwise it is set to 0. The value of $score\_mac$ is set to 0.2 when MAC addresses match, otherwise it is set to 0. The value of $score\_ports$ is computed by considering all the ports reported as not closed in at least one the two entries, and by multiplying the proportion of matching port states by 0.6. The algorithm then sorts all the pairs based on their matching score (Line 25) and stores the matched pairs till all the IPs in the two datasets had been considered (Lines 27 to 37). Our choice for the values assigned to $score\_ip$, $score\_mac$, and $score\_ports$ enables ensuring that a host that, in the updated configuration, changes IP or MAC and a limited subset of its ports is still matched with the correct entry in the initial configuration. Unmatched IPs correspond to hosts being added or removed from the network and are thus matched with a generated entry having null IP (Lines 38 to 47).

Our matching algorithm leads to pairs of IPs, each modeling a single host. They are used by FISTS to construct, using the execution data collected by Step 3, a table, called *Network State Change Report* (NSCR), that reports, for each host, its port state changes. In the NSCR, each port state change is captured by a label that joins the name of the port state in the initial and the updated configuration for the selected port; Table 4.3 shows a few examples of how port state change labels are generated from specific port states in the initial and updated configuration. In total, we have 16 different label (i.e., 4 labels for the initial state times 4 labels for the updated state). An example NSCR is shown in Figure 4.1, each column contains information about the state change observed for one port working with a specific protocol (UDP or TCP). Please note that we also track ports that do not change their state (e.g., *tcp_Port 445* in the first row of Figure 4.1's table, labeled as *ClosedToClosed*).

In the presence of firewalls, port scanning with NMAP will lead to most or all ports being shown as filtered (see Table 2.1). However, this behavior does not compromise the applicability of FISTS. Indeed, service providers testing their own infrastructure can configure scanning nodes appropriately, if needed. For example, in the presence of firewalls that prevent responses from subnets, if the

Require: $DS_{initial}$, data set collected from the initial SDN configuration
Require: $DS_{updated}$, data set collected from the updated SDN configuration
Ensure: $MATCH$, a table whose rows contain the values of two matching entries from $DS_{initial}$ and $DS_{updated}$, plus a match score

▷*Compute a matching score for all the possible pairs of entries*
$assigned\_initial \leftarrow$ empty set
$assigned\_updated \leftarrow$ empty set
$scored \leftarrow$ empty list
for e_n in initial configuration set **do**
  for e_m in updated configuration set **do**
    if IP address of e_n and e_m match **then**
      $score\_ip \leftarrow 0.2$
    end if
    if MAC address of e_n and e_m match **then**
      $score\_mac \leftarrow 0.2$
    end if
    ▷*Compute score_ports*
    $matches \leftarrow 0$
    $ports \leftarrow 0$
    for any port p being not closed in either e_n and e_m **do**
      $ports \leftarrow ports + 1$
      if state of port p in e_n and e_m match **then**
        $matches \leftarrow matches + 1$
      end if
    end for
    $score\_ports \leftarrow \frac{matches}{ports} * 0.6$
    ▷*Score of the entry pair*
    $matching\_score \leftarrow score\_ip + score\_mac + score\_ports$
    $scored \leftarrow scored \cup \langle e_n.IP, e_m.IP, matching\_score \rangle$
  end for
end for
sort $scored$ based on $score$
$MATCH \leftarrow$ empty set
for $\langle e_n.IP, e_m.IP, matching\_score \rangle$ in $scored$ **do**
  ▷*Check if e_n is not already matching another entry*
  if $e_n.IP$ in $assigned\_initial$ **then**
    continue
  end if
  ▷*Check if e_m is not already matching another entry*
  if $e_m.IP$ in $assigned\_updated$ **then**
    continue
  end if
  ▷*Add the matching pair to $MATCH$*
  $MATCH \leftarrow MATCH \cup \langle e_n.IP, e_m.IP, matching\_score \rangle$
  ▷*Update the list of processed IPs*
  $assigned\_initial \leftarrow assigned\_initial \cup e_n.IP$
  $assigned\_updated \leftarrow assigned\_updated \cup e_m.IP$
end for
for e_n in initial configuration set **do**
  if $e_n.IP$ not in $assigned\_initial$ **then**
    $MATCH \leftarrow MATCH \cup \langle e_n.IP, null\_IP, 1.0 \rangle$
  end if
end for
for e_m in updated configuration set **do**
  if $e_m.IP$ not in $assigned\_updated$ **then**
    $MATCH \leftarrow MATCH \cup \langle null\_IP, e_n.IP, 1.0 \rangle$
  end if
end for

Figure 4.3: FISTS's host matching algorithm.

scanning node is on the WAN, NMAP will report that all the subnet nodes have filtered ports; however, if after a configuration change, a subnet provides access from WAN to an SMB server on 8080, FISTS will report *FilteredToOpen*, which is a state change (note that without a firewall we

would observe *ClosedToOpen*), which may still enable anomaly detection. If access to some services of a given private subnet is provided only to other private subnets, multiple scanning nodes might be used (e.g., one per subnet). Further, approaches relying on packet fragmentation such as hping [234], FragScapy [90], and some NMAP extensions [235] enable port scanning in the presence of firewalls and can be integrated into FISTS.

4.2.3 FISTS Step 4: Preprocessing

In Step 4, the NSCR is preprocessed to enable the application of anomaly detection algorithms. Indeed, such algorithms usually work with numerical data while NSCR entries contain textual categorical data. To transform our data, we cannot apply integer encoding (i.e., replace each unique category label with an integer) because it would introduce an arbitrary ordering across labels (e.g., 'OpenToClosed' being closer to 'OpenToFiltered' than to 'ClosedToOpen)', which is a bad practice [236]; therefore, we apply one-hot encoding. Since one-hot encoding replaces each column with a number of boolean columns matching the number of distinct possible values, we derive 16 columns for each TCP and UDP port column.

In Step 4, FISTS may also perform an additional, optional, task, which consists of *pruning* entries that do not present any change in their state. Our intuition is that reconfigurations not introducing any change in the port states of a host unlikely introduce a vulnerability affecting such host because the set of open, filtered, and closed ports remains the same, and thus the host works as in the previous configuration. Instead, keeping such entries may increase the likelihood of false positives; for example, in the presence of several hosts without port state changes, Isolation Forest may create several trees where all the sampled data entries match, have a minimal distance from the root, and are thus erroneously reported as anomalous.

4.2.4 FISTS Step 5: Vulnerability Prioritization

In Step 5, FISTS sort all the entries in the NSCR according to their likelihood of being affected by a vulnerable configuration, such likelihood is captured by an anomaly score generated by an anomaly detection algorithm. An example output is shown in Table 4.4.

Our key intuition is that the prioritization of all the hosts enables the definition of a human-in-the-loop process that overcomes the limitations of existing anomaly detection approaches relying on predefined thresholds. Indeed, existing anomaly detection approaches select a subset of data entries as anomalous based on some arbitrary thresholds, which are defined either for the distance between

dataset features [123] or for the generated anomaly scores [120], [121]. However, since features may change from context to context (e.g., different SATCOM providers may change different sets of ports in different SDN reconfigurations), thresholds that are successful in one case study may not work with others. Instead, it would be better to enable the end-user stop inspecting anomalous hosts when there is evidence that what reported by FISTS does not help discovering vulnerabilities; we achieve such objective by using as stopping condition (SC) the number of consecutive false positives (CFP) observed, if $FP \geq SC$ the engineer can stop inspecting hosts.

By relying on the number of consecutive false positives as a stopping condition, unlike related work, we do not assume that anomalies might be identified with the same distance thresholds in different case study subjects, but we look for evidence that hosts with an anomaly score below a certain value aren't vulnerable, which is achieved by verifying that there are SC consecutive false positives. In our empirical evaluation, we assess the performance of different configurations of FISTS, with different SCs.

We call the approach above *FISTS hosts selection* and it has the objective of reducing SDN maintenance costs by enabling engineers to inspect only a subset of the potentially vulnerable hosts. However, companies delivering critical services may require their engineers to inspect all the hosts, to ensure the absence of any vulnerability. In such case, engineers should inspect all the hosts in the order provided by FISTS, we call such approach *hosts prioritization*. When applied for hosts prioritization, FISTS should enable engineers identify all the vulnerable hosts at the beginning of their investigation, thus minimizing the time required to discover and fix vulnerabilities.

To sort NSCR hosts, FISTS should rely on anomaly detection algorithms producing an anomaly score that can be used for prioritization. Unfortunately, well-known algorithms for anomaly detection (i.e., IF and LOF) select a subset of dataset entries as anomalous instead of sorting dataset entries according to their likelihood of being anomalous. However, internally, they rely on an anomaly score to sort dataset entries and then select the ones with the highest anomaly score; therefore, we defined two approaches that rely on such internal information. We call *Sorted Isolation Forest (SIF)* our approach relying on the anomaly score generated by the IF algorithm. We call *Sorted KNN (SKNN)*, the algorithm that sorts dataset entries based on the anomaly score computed by LOF, which consists of the average pairwise distance computed using a KNN approach.

Although effective, IF and LOF aren't the only approaches capable of extracting anomaly information from a dataset. Precisely, since some vulnerabilities may present similar effects (i.e., they lead to the same port changes), we believe that clustering algorithms can detect hosts with similar changes,

and thus enable the detection of hosts with similar vulnerabilities. However, clustering algorithms do not sort data entries; but vulnerable configurations tend to affect a small subset of the hosts (e.g., because they are introduced by corner cases), and thus we can prioritize the inspection of hosts based on the size of the cluster they belong to. Precisely, clusters with few items are more likely due to a vulnerable configuration and should thus be inspected first. Within a cluster, entries are inspected in a random order.

To perform clustering, we rely on K-means and HAC (see Chapter 2). We selected K-means because it is a standard baseline clustering approach. We selected HAC because our preprocessed dataset results from the application of one-hot encoding, and thus the distance between two data entries would capture the number of port state changes that do not match between two entries; in other words, hosts with the same port state changes will be very close. Since we assume that a vulnerable configuration affects hosts in a same way (i.e., port states change similarly), a hierarchical clustering algorithm that builds clusters by joining hosts that are close to each other, seems an appropriate fit for the identification of clusters with vulnerable hosts. We refer to our strategy to prioritize hosts based on clustering algorithms as *sorted K-means* (SKM) and *sorted HAC* (SHAC).

4.2.5 FISTS usage

The FISTS approach is implemented as a pipeline; a *FISTS pipeline consists* of all the components automating Steps 1 to 5. A pipeline configuration is set by (1) enabling/disabling the pruning component, (2) selecting the anomaly detection algorithm to be used (i.e., SIF, SKNN, SKM, SHAC), and, if needed, setting its hyper-parameters (i.e., K for SKNN) (3) selecting the stopping condition (i.e., the number of false positives after which engineers should stop inspecting results, if engineers aim to apply FISTS for *hosts selection*).

4.3 Empirical Evaluation

We performed an empirical evaluation that aims at addressing the following research questions (RQs):

RQ1. The pruning tasks in FISTS preprocessing step (i.e., ignoring hosts not affected by any state change) may influence the execution of anomaly detection algorithms. For example, without pruning, a density-based algorithm like LOF or SKNN may end-up computing anomaly scores that are very similar for non-vulnerable hosts with no state changes and for hosts with one (vulnerable) state change (same density), thus leading to prioritized hosts where several false positives may appear before a

true positive. Further, as discussed in section 4.2.3, the absence of pruning may affect IF as well. Therefore, we aim to determine if, overall, the best results are observed with or without pruning.

RQ2. The performance of a FISTS pipeline may depend on the characteristics of the SDN under test; for example, the total number of hosts with valid and invalid state changes and the number of ports with valid/invalid state changes may lead to different performance results for the FISTS anomaly detection algorithms and, consequently, may affect what would be the best stopping condition. We aim to assess what is the FISTS pipeline leading to the best results with different datasets.

RQ3. We aim to assess the accuracy of the host matching component and its impact on FISTS results (e.g., the proportion of false alarms due to inaccurate host matching).

RQ4. We aim to report on the scalability of the approach for a network having the characteristics (e.g., configured ports), which we expect to be representative of SATCOM organizations. We aim to assess both network probing and the anomaly detection component.

4.3.1 Experiment Setup

We target SATCOM providers that rely on SDNs for network communications. To perform experiments that reflect production conditions and, at the same time, share our datasets without breaking confidentiality agreements, we considered public data about SES networks. Although SES cannot disclose the nature (e.g., SDN or traditional) of the networks monitored by Shodan, SES engineers been confirmed that they share similar characteristics (number of nodes, open ports) with SES-managed SDN networks.

In our context, a dataset is a file with the same data included in a Network State Change Report (NSCR); in practice, a dataset simulates the result of executing FISTS Steps 1 to 3 and enables assessing FISTS anomaly detection performance. We created three groups of datasets (called real, synthetic, and realistic) following the process depicted in Fig 4.2a and described below.

To create our datasets, we collected from Shodan, a public database with information about public Internet IPs (e.g., open ports, network provider name), the history of port state changes for IPs belonging to SES. Shodan periodically scans all the IPs on the Internet and keeps track of the open ports; because of such characteristic, Shodan data can be used to simulate the data produced by the NMAP component in FISTS and enable a large-scale assessment of FISTS.

We analyzed Shodan data collected in the period between April 2021 and May 2023; it concerns 405 SES hosts (we assume each IP belongs to a distinct host). We determine port state changes by comparing port states collected in subsequent scans. We ended up with a change dataset indicating,

for every day scanned by Shodan, state change information for every monitored IP. The max number of hosts with port state changes in a day is 13, the min, max, and average number of ports with a modified state are 0, 13, and 1.61, respectively.

Since Shodan does not monitor all the networks managed by SES or competing companies (e.g., private corporate networks) it does not enable assessing FISTS with data capturing the characteristics of all the possible contexts in which it could be applied, such as a larger number of host with port state changes, or a type of port change different than the one observed in the Shodan dataset. Further, Shodan does not distinguish between vulnerable and valid port changes.

To address the limitations above, to build our datasets, we extended the data collected by Shodan with additional data, as described below.

Synthetic datasets. We aim at ensuring that four scenarios considered relevant by SATCOM operator are detected by FISTS. Three scenarios are similar to the one depicted in Figure 4.1, where a flow rule is not deleted in a reconfiguration; these three scenarios differ for the port targeted by the flow rule (one for NTP port 123, one for SMB port 445, and one for HTTP port 8080). For clarity, Figure 4.2b shows what would be the correct updated configuration in this case, and what is, instead, the vulnerable updated configuration. The fourth scenario (Figure 4.2c) captures a situation where, in the updated configuration, a subnet is connected to a switch but a stateful firewall that should enable only the WebServer in the subnet to be accessed from the outside is not configured.

We created 100 datasets including configuration changes belonging to the four reference scenarios. To this end, we first relied on virtual machines to create the four reference scenarios. For all of them, we used FISTS (Steps 1 and 2) to test both the vulnerable reconfiguration and a valid reconfiguration; then we generated the NSCR (Step 3).

We populated 100 datasets as follows. First, we simulated, in each dataset, 72 hosts with a correct reconfiguration, by copying entries randomly selected from the NSRCs belonging to valid reconfigurations; the number 72 results from duplicating 12 times the 6 valid changes in the monitored scenarios. Please note that, to simulate different hosts, we replace the IP address of the copied entries with an IP not appearing yet in the dataset[2]. Then, we introduced into the datset a number of vulnerable hosts by duplicating entries for hosts with vulnerable changes (we took them from the the NSCRs belonging to vulnerable reconfigurations). Since the number of vulnerable hosts in a system may

[2]We repeat such procedure every time we copy an entry, also for building the other datasets; for brevity, we will not report such clarification further. Whenever we indicate that we copy an entry, we mean that we copy the entry and update the IP address appropriately.

vary, we constructed 100 datasets, each having between 5 and 50 vulnerable hosts (we generated 10 datasets for each number of vulnerable hosts in the range, in steps of five). Finally, in each dataset, we reached a total of 405 entries by duplicating entries without state changes from the NSCRs belonging to correct reconfigurations (we updated the IP addresses to match SES hosts).

Real datasets. To assess FISTS with real data, we generated 20 datasets using the data collected by Shodan; we considered the first available date for each IP. Based on Shodan's data, these datasets include 10 entries with correct configuration changes (the number of modified ports range from 5 to 50) and 395 entries without any change. Then, to obtain datasets with diverse vulnerable entries, we introduced, in each of the 20 dataset, a number of entries with vulnerable configurations by replicating the process adopted for the synthetic datasets (i.e., copying the vulnerable entries from the synthetic NSCRs) but considering a number of violations that varies between 5 and 10; we limit the number of vulnerable entries to 10 because it is very unlikely to observe a higher proportion of vulnerable state changes (with 10 correct changes, 10 vulnerable changes account for 50%).

Realistic datasets. We created 100 datasets to assess FISTS using NSCRs with a number of valid state changes that is larger than what reported in daily Shodan scans, to simulate what happens in private corporate networks not monitored by Shodan.

We constructed each dataset by copying 100 randomly selected entries with changes from the Shodan change dataset. Then, we copy N entries with vulnerable changes from the NSCRs belonging to synthetic scenarios with vulnerable changes. We considered a number of N vulnerable entries between 5 and 50, and created 10 dataset files for each, thus ending up with 100 datasets, in total.

Finally, for each dataset, we sampled and copied additional entries with no state change from the Shodan dataset, till we reached a total of 405 entries in each file.

4.3.2 RQ1: Pruning effectiveness

Experiment Design

RQ1 concerns the effectiveness of FISTS's pruning solution (i.e., removal of entries with no state changes). To address RQ1, we compared the results obtained by FISTS pipelines with pruning enabled and FISTS pipelines without pruning. Precisely, since our datasets simulate the results of FISTS Steps 1 to 3, the evaluated pipelines consist of FISTS Steps 4 and 5.

We considered all the four different anomaly detection approaches integrated into FISTS: SIF,

SHAC, SKM, SKNN; for SKNN, we considered different configurations for K (i.e., 10, 15, 20, and 25). In total, we selected seven FISTS algorithm configurations. We executed the selected FISTS anomaly detection algorithm configurations, with and without pruning, ending up with 14 FISTS anomaly detection algorithm executions. Finally, to assess *FISTS hosts selection*, as stopping condition (SC), we considered a number of false positives ranging from 1 to 10, in steps of 1, and the case in which 20 false positives should be observed before stopping the inspection; in total, we have 11 configurations for SC. It leads to 154 (14×11) FISTS pipelines being assessed.

Further, as baseline, we considered also cases in which the datasets are processed by the state-of-the art algorithms IF and LOF; for LOF we considered the same values for K considered for SKNN (i.e., 10, 15, 20, and 25). This leads to 5 additional pipelines, for a total of 159 anomaly detection pipelines assessed in our experiments.

To deal with randomness, we executed each anomaly detection pipeline 40 times on each dataset. Since, in total, we have 220 datasets, it leads to a total of 1.399.200 runs ($159 \times 40 \times 220$).

FISTS produces as output a prioritized NSCR, with hosts being prioritized according to their anomaly score (highest first). Since, for each dataset, we know what are the vulnerable hosts, we can determine false positives (i.e., hosts inspected before reaching the stopping condition that are not vulnerable) and true positives (i.e., hosts inspected before reaching the stopping condition that are vulnerable).

To assess *FISTS hosts selection*, given a FISTS's prioritized NSCR, we can determine the number of false and true positive observed. False and true positives enable us to compute precision, recall, and F1 score according to standard formula.

To assess *FISTS hosts prioritization*, we count the number of hosts to inspect before identifying all the true positives, which enables determining how quickly a configuration helps engineers detect all the faults. We refer to such metric as *debugging cost (DC)*. By definition, FISTS pipelines differing only for SC, will lead to the same debugging cost.

To perform our assessment, we consider each dataset separately because different FISTS pipelines may perform differently with different datasets.

To determine if, overall, pruning is beneficial for FISTS, we compare the best result obtained with any FISTS pipeline, in each dataset, with and without pruning. Pruning is beneficial if it enables achieving better metrics in all the datasets.

Table 4.5: Results for synthetic, real, and realistic dataset with and without pruning

	Synthetic								Real								Realistic							
	Without Pruning				With Pruning				Without Pruning				With Pruning				Without Pruning				With Pruning			
Algorithm	**SC**	**Prec.**	**Rec.**	**DC**	**SC**	**Prec.**	**Rec.**	**DC**	**SC**	**Prec.**	**Rec.**	**DC**	**SC**	**Prec.**	**Rec.**	**DC**	**SC**	**Prec.**	**Rec.**	**DC**	**SC**	**Prec.**	**Rec.**	**DC**
SIF	1	**0,94**	0,88	41,11	1	**0,94**	0,90	**35,48**	20	**0,32**	1,00	**9,99**	20	**0,42**	1,00	**14,49**	20	**0,37**	0,96	**46,52**	20	**0,39**	0,93	53,91
SIF	20	0,50	**0,94**	41,11	20	0,50	**0,94**	**35,48**	20	0,32	**1,00**	**9,99**	20	0,42	**1,00**	**14,49**	20	0,37	**0,96**	**46,52**	20	0,39	**0,93**	53,91
IF	N/A	**0,00**	**0,00**	N/A	N/A	**0,06**	0,17	N/A	N/A	**0,00**	0,34	N/A	N/A	**0,59**	0,97	N/A	N/A	**0,00**	0,00	N/A	N/A	**0,53**	0,51	N/A
SHAC	1	**0,92**	0,75	247,88	1	**0,92**	0,77	50,00	1	**0,38**	0,50	10,40	20	**0,42**	1,00	15,00	20	**0,00**	0,00	353,53	1	**0,94**	0,84	47,80
SHAC	20	0,46	**0,76**	247,88	20	0,46	**0,80**	50,00	20	0,32	**1,00**	10,40	20	0,42	**1,00**	15,00	20	0,00	**0,00**	353,53	20	0,48	**0,87**	47,80
SKM	1	**0,92**	0,76	298,66	1	**0,92**	0,69	60,79	1	**0,34**	0,45	10,01	20	**0,42**	1,00	15,12	20	**0,00**	0,00	365,52	1	**0,94**	0,83	49,81
SKM	20	0,47	**0,78**	298,66	20	0,43	**0,75**	60,79	20	0,32	**1,00**	10,01	20	0,42	**1,00**	15,12	20	0,00	**0,00**	365,52	7	0,70	**0,84**	49,81
LOF-10	N/A	**1,00**	**0,83**	N/A	N/A	**1,00**	0,83	N/A	N/A	**0,32**	**0,67**	N/A	N/A	**0,00**	**0,00**	N/A	N/A	**0,11**	**0,10**	N/A	N/A	**1,00**	**0.75**	N/A
LOF-15	N/A	**0,75**	**0.88**	N/A	N/A	**0,79**	0,88	N/A	N/A	**0,32**	**0,67**	N/A	N/A	**0,00**	**0,00**	N/A	N/A	**0,07**	**0,15**	N/A	N/A	**1,00**	**0.70**	N/A
LOF-20	N/A	**0,62**	**0,91**	N/A	N/A	0,65	**0,91**	N/A	N/A	**0,32**	**0,67**	N/A	N/A	**0,00**	**0,00**	N/A	N/A	**0,07**	**0,20**	N/A	N/A	**1,00**	**0.78**	N/A
LOF-25	N/A	**0,55**	**0,93**	N/A	N/A	0,58	**0,93**	N/A	N/A	**0,32**	**0,67**	N/A	N/A	**0,00**	**0,00**	N/A	N/A	**0,07**	**0,25**	N/A	N/A	**1,00**	**0.72**	N/A
SKNN-10	1	**0,94**	0,87	156,25	1	**0,94**	0,88	59,18	20	**0,32**	1,00	11,25	20	**0,42**	1,00	17,50	1	**0,00**	0,00	363,27	1	**0,89**	0,65	36,31
SKNN-10	20	0,47	**0,88**	156,25	20	0,43	**0,96**	59,18	20	0,32	**1,00**	11,25	20	0,42	**1,00**	17,50	1	0,00	**0,00**	363,27	20	0,48	**0,98**	36,31
SKNN-15	1	**0,94**	0,92	56,01	1	**0,95**	0,93	36,44	20	**0,32**	1,00	10,85	20	**0,42**	1,00	15,50	1	**0,00**	0,00	363,26	1	**0,90**	0,66	30,92
SKNN-15	20	0,50	**0,96**	56,01	20	0,51	**0,95**	36,44	20	0,32	**1,00**	10,85	20	0,42	**1,00**	15,50	1	0,00	**0,00**	363,26	20	0,51	**1,00**	30,92
SKNN-20	1	**0,94**	0,92	37,85	1	**0,94**	0,89	37,98	20	**0,32**	1,00	10,85	20	**0,42**	1,00	15,50	1	**0,00**	0,00	361,53	1	**0,91**	0,66	**30,65**
SKNN-20	20	0,50	**0,96**	37,85	20	0,50	**0,92**	37,98	20	0,32	**1,00**	10,85	20	0,42	**1,00**	15,50	1	0,00	**0,00**	361,53	20	0,51	**1,00**	**30,65**
SKNN-25	1	**0,94**	0,92	**34,91**	1	**0,94**	0,86	41,19	20	**0,32**	1,00	10,85	20	**0,42**	**1,00**	15,50	1	**0,00**	0,00	361,63	1	**0,91**	0,66	31,19
SKNN-25	20	0,50	**0,96**	**34,91**	20	0,49	**0,88**	41,19	20	0,32	**1,00**	10,85	20	0,42	**1,00**	15,50	1	0,00	**0,00**	361,63	20	0,50	**1,00**	31,19

Results

Tables 4.5 report results obtained by the FISTS pipelines having the highest precision, recall, and F1 score, for each FISTS algorithm. Precisely, for each anomaly detection algorithm, we report on the SC leading to the highest precision, recall, and F1 score (best values are bold, algorithms where SC is not applicable have all the values bold). We underline the best performance value obtained in each dataset. We also report the debugging cost (DC) of each algorithm (best dataset value in bold).

For *hosts selection*, the best results observed with the synthetic dataset without pruning are 1.00 (precision), 0.96 (recall), 0.93 (F1 score); with pruning, we observe, 1.00 (precision), 0.96 (recall), 0.94 (F1 score). Pruning slightly improves F1 score (0.93 VS 0.94), the other metrics are the same. The best results for the real dataset without pruning are 0.38 (precision), 1.00 (recall), 0.48 (F1 score); with pruning, we observe, 0.59 (precision), 1.00 (recall), 0.59 (F1 score). Pruning improves precision (0.38 VS 0.59) and F1 score (0.48 VS 0.59), recall is maximal in both. The best results for the realistic dataset without pruning are 0.37 (precision), 0.96 (recall), 0.53 (F1 score); with pruning, we observe, 1.00 (precision), 1.00 (recall), 0.88 (F1 score). Pruning improves precision (0.37 VS 1.00), recall (0.96 VS 1.00) and F1 score (0.53 VS 0.88). Concluding, we can affirm that pruning improves FISTS hosts selection results.

For *hosts prioritization*, we focus on DC. The best result (i.e., lowest average number of hosts to be inspected) for the synthetic dataset without pruning is 34.91, with pruning is 35.48; it shows that without pruning, engineers inspect 0.57 anomalies less, on average. For the real dataset, the best result without pruning is 9.99, with pruning is 14.49; it shows that without pruning, engineers inspect 4.5 anomalies less, on average. For the realistic dataset, the best result without pruning is 46.52, with

pruning is 30.65; it shows a trend that differs from the other two datasets, indeed, without pruning, engineers inspect 15.96 anomalies more, on average. Although in two datasets the lack of pruning slightly reduced the number of anomalies to be inspected, overall, because of what observed in the realistic dataset, pruning leads to a higher reduction of debugging cost. Indeed, considering all the datasets, FISTS without pruning leads to 25.23 DC, while FISTS with pruning leads to 18.47 DC.

Concluding, we suggest relying on pruning to perform anomaly detection with FISTS.

4.3.3 RQ2: Best FISTS pipeline

Experiment Design

We rely on the same data collected for RQ1. But, since RQ1 shows that pruning leads to best results, we focus on anomaly detection pipelines with pruning.

For hosts selection, the best pipelines for a dataset are the ones that maximize recall (and significantly differ from other pipelines) and, among the ones with highest recall, maximize precision (and significantly differ from other pipelines). Alternatively, F1 score can be considered to identify the best pipelines. Since hosts selection leads to the inspection of a subset of hosts, potentially overlooking vulnerabilities, maximizing recall (i.e., the proportion of faulty configurations detected) is a necessity because engineers aim at detecting all the vulnerabilities, to have a secure system. Since precision is the proportion of inspected hosts that are faulty, maximizing precision implies that engineers minimize the inspection of hosts that are valid. Consequently, once recall is maximized, the approach that maximizes precision is the one that enable using engineers time most effectively.

For hosts prioritization, the best pipelines for a dataset are the ones minimizing DC (i.e., they help identifying all the vulnerabilities with the minimal number of anomalies to be inspected) and showing significant difference from the others.

Ideally, the best pipelines (i.e., the ones that we suggest for their use with FISTS) are the ones belonging to the intersection of the sets of best pipelines observed with the different datasets.

We compute the significance of the difference between results observed with datasets having the same number of anomalies by computing p-values with the Mann–Whitney U test [237], a non parametric method; we report a significant difference when p-values are below 0.05.

Results

We aim to identify the subset of pipelines that perform best in all the datasets.

For hosts selection, with the synthetic dataset, the best performing approach is SKNN-10 with SC=20 (recall is 0.96); however, it does not significantly differ from SKNN-15 with SC=20 (recall is 0.95) and SKNN-20 with SC=20 (recall is 0.92). Further, SKNN-15 with SC=20 and SKNN-20 with SC=20 have a significantly higher precision. Other approaches whose recall does not significantly differ from SKNN-10 with SC=20 are SIF SC=20 (recall is 0.94), LOF 20 (recall is 0.91), and LOF 25 (recall is 0.93). For the real dataset, all the pipelines except the ones based on LOF and IF reach maximum recall. The poorer performance observed for LOF and IF (recall is zero for LOF and below 1 for IF) is likely due to the fact that, since the real dataset includes only few changes (vulnerable and correct), what happens is that all the changes (correct and vulnerable) may look similar (i.e., are very close if we apply a distance metric). Consequently, approaches that simply select a subset of ports based on a threshold on the anomaly score (e.g., LOF) perform much worse than approaches that sort all the anomalies (i.e., SIF, SHAC, SKM, SKNN) and provide some tolerance for classification mistakes (i.e., they use SC=20). Indeed, SC equal to 20 enables FISTS pipelines to include the vulnerable hosts in their selection, instead, lower SC values doesn't (please note that other values for SC do not appear in the table because SC ≤ 10 does not enable selecting any vulnerable host).

The IF result is nevertheless interesting because, despite not being able to achieve max recall, it has a better precision than other pipelines. However, the pipelines performing best in both the synthetic and the real dataset are SKNN-10 with SC=20, SKNN-15 with SC=20, SKNN-20 with SC=20, and SIF with SC=20. Finally, in the realistic dataset, among SIF and SKNN-based pipelines, the best recall results (0.86 and 0.87) are those obtained by SKNN-based pipelines, with the best precision (0.51) achieved by SKNN-15 with SC=20 and SKNN-20 with SC=20. Concluding the best FISTS pipelines are SKNN-15 with SC=20 and SKNN-20 with SC=20, with SKNN-15 with SC=20 to be favoured since it had slightly better results in the synthetic dataset.

For hosts prioritization, recall that we ignore the configuration value assigned to SC because all hosts are inspected and we aim at determining how quickly all the vulnerabilities are found. With the synthetic dataset, the best result is achieved by SIF with 35.48 hosts to be inspected on average. However, such result does not significantly differ from that of SKNN-15 (36.44) and SKNN-20 (37.98). For the real dataset, SIF still performs the best (14.49), with SKNN-15 and SKNN-20 performing similarly (15.50); also, as observed for hosts selection, clustering algorithms perform well (SHAC's DC is 15, SKM's is 15.12), which indicates that they perform well when only a few state changes are observed. Finally, for the realistic dataset, the best results are observed with SKNN-15 (30.92) and SKNN-20 (30.65), they both perform significantly better than SIF (53.91). Concluding, also for hosts

prioritization we suggest relying on SKNN-15 and SKNN-20.

Concluding, our results show that the approach proposed in this paper (i.e., sorting KNN results and relying on thresholds over false positives) outperform state-of-the-art approaches (i.e., IF and LOF). Also, relying on SKNN-15 and SKNN-20 enables achieving best results for both hosts selection and prioritization. Since our datasets include vulnerabilities affecting reachability (Scenarios 1, 2, and 3) and confidentiality (Scenarios 1, 2, 3, and 4), the high recall achieved by the best pipeline enables us to conclude that FISTS empower engineers in effectively assessing both security properties.

4.3.4 RQ3 - Accuracy of host matching module

Experiment Design

We aim to assess the accuracy of the host matching component and its impact on FISTS results. Since Shodan does not provide information about what hosts have changed their IP or MAC address from one scan to the other, we created additional datasets where we introduced changes in the IP and MAC addresses of hosts and verify if FISTS properly process them. Precisely, we followed the process described in section 4.2.1 to create 100 real datasets with 5 to 50 vulnerable hosts (10 datasets for each different number of vulnerable hosts). From these 100 datasets, we conducted two experiments (namely EXP1 and EXP2), deriving 100 *original NSCR realistic datasets* in each, as follows. In EXP1, for each real dataset, we randomly sampled 40 hosts and changed their IP address for the updated configuration, we also sampled 40 other hosts and changed their MAC address. In EXP2, we randomly sampled 40 hosts and changed both their IP and MAC address. For each experiment, we thus obtained *modified NSCR realistic datasets*. Finally, for each *modified NSCR realistic dataset*, we derived execution data files for the initial and updated configuration, to be processed by our *host matching and comparison* component.

Since we know what the expected NSCR is, we can count true and false positives. A true positive is an NSCR entry generated by our algorithm that matches what in the corresponding modified NSCR realistic dataset. A false positive is an NSCR entry not present in the corresponding dataset. We assess our approach in terms of accuracy.

Results

The execution of our algorithm on the 100 data file pairs led to perfect accuracy. Further, the distribution of entries having changes in both port state and IP or MAC address, in EXP1, ranges

within 0 and 35 for each modified NSCR realistic dataset file (median is 12.5), in EXP2, it ranges between 22 and 44 (median is 38.5). Given our 100% accuracy, it demonstrates FISTS HMC algorithm effectiveness with different numbers of port state changes, even in the presence of simultaneous changes of IP and MAC (i.e., EXP2).

4.3.5 RQ4 - Scalability of approach

Experiment Design

Three are the steps that mainly affect the scalability of the approach: Step 1 - Traffic Generation, Step 2 - Response Monitoring, Step 5 - Vulnerability Prioritization. Executing Step 3 and Step 4 on execution data collected from 405 IPs takes less than two seconds, therefore, we exclude them from our investigation for RQ4.

Since Steps 1 and 2 are coupled (i.e., implemented with NMAP), we assess them together. Given that, for security reasons, we cannot scan the SES network directly, we simulated multiple scans of a network with 405 IPs and report on execution time. Precisely, we executed NMAP to independently scan 1000 ports of three hosts in the updated configuration in Fig 4.2b; we rely on a simulation with VMs. We repeated the execution 400 times, to collect enough data points (each data point captures the execution time to scan 3 IPs). To simulate multiple scans of a network with 405 IPs, we randomly sampled 135 data points ($135 \times 3 = 405$), and repeated the sampling 100 times. We discuss the distribution of execution time, to demonstrate the scalability of FISTS Steps 1 and 2 in the context of SATCOM providers. Specifically, although SDNs can be quickly reconfigured every few seconds, in our reference context, they are modified based on customer requirements (e.g., companies buying satellite communication channels), which happens few times in a week, during dedicated management windows of 15 to 30 minutes.

For Step 5, what affects the scalability of the approach is the time taken by the anomaly detection algorithms. We compare the different algorithms by reporting the time taken to process the different datasets considered for RQ1 and RQ2.

Results

The time taken by NMAP to scan 1000 ports in 405 IPs ranges between 1816.7 and 1821.6 with a median of 1819.54 seconds (30.32 minutes), which is acceptable since it fits in a typical maintenance window for networks. Further, the scan can be parallelized thus reducing the total time required;

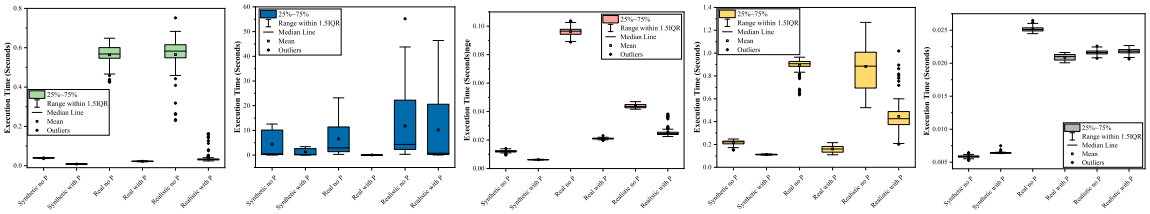
however, although we suggest relying on parallel scanning, what may limit its adoption is the network bandwidth dedicated to maintenance operations, which vary across companies. Last, the time required to scan different sets of IPs is very similar (mix and max are 1816.73s and 1821.62s, respectively), thus suggesting results might be confirmed by repetitions of the experiment in other contexts.

Figures 4.4a to 4.4e show the execution time taken by SHAC, SKM, LOF, IF/SIF, and SKNN on all the datasets. SIF and IF take the same execution time because SIF relies on IF, but simply selects all the sorted datapoints instead of a subset.

Our plots show that the use of pruning reduces execution time; indeed, the median execution time observed with pruning (see *with P* in the figures) is lower than what observed without pruning (see *no P* in the figures) in all the cases except the synthetic dataset for SKNN. In the synthetic dataset for SKNN, the execution time is so low (between 0.005 and 0.010 seconds), that the better result obtained without pruning may be due to a slight change in the background tasks running on the machine used for experiments. Concluding, the results observed for execution time provide an additional argument in favor of the adoption of pruning, as we already suggested when addressing RQ1.

All the approaches, except SKM, process each dataset in less than 1.4 seconds, thus showing that they scale well. SKM, instead, takes up to 47 seconds in the case of the realistic dataset with pruning. However, one minute to process one NSCR dataset is an acceptable time since it's an order of magnitude less than what required to collect the data by scanning the network. SKNN, which is the approach selected in RQ2 takes less than 0.3 seconds, thus showing that it is also among the quickest ones.

Figure 4.4: Execution time of FISTS algorithms



(a) SHAC. (b) SKM. (c) LOF. (d) IF/SIF. (e) SKNN.

4.3.6 Threats to validity

For *internal validity*, we manually inspected the results (i.e., labeled datapoints) for a subset of the datasets. For *construct validity*, we relied on metrics (precision, recall) commonly adopted to assess machine learning algorithms, and a metric (debugging cost) that is an indirect measure of effort. For

conclusion validity, we relied on non-parametric statistical tests. For generalizability, we constructed datasets that have characteristics (number of hosts, port numbers) similar to those of large SATCOM providers. However, the results (precision, recall) obtained with unsupervised machine learning algorithms may vary across network environments. To address this threat, we considered datasets presenting a varying proportion of hosts with port or IP changes. For vulnerabilities, we constructed scenarios that shall be considered invalid in any context (i.e., messages forwarded to the wrong subnet) but we also considered a scenario that is likely vulnerable (i.e., a whole subnet with only one server becomes fully reachable), based on feedback from SATCOM experts, although it might be considered valid in some peculiar contexts. Concluding, although we expect replicability in SATCOM networks similar to ours, repeatability in different contexts can't be granted.

4.4 Conclusion

In this paper, we addressed the problem of automatically detecting configuration updates for software defined networks (SDNs) that introduce vulnerabilities. Model-based approaches that process configuration files of SDN software are unlikely applicable in industry because commercial software with proprietary formats is used. Therefore, we propose FIeld-based Security Testing of SDN Configurations Updates (FISTS), a black-box, field-based testing approach that collects data about the port states of hosts on the network before and after a configuration update, and then relies on an anomaly detection algorithm to sort hosts based on the probability of being vulnerable. Also, we propose a strategy to stop inspecting hosts when there is evidence that not inspected hosts are unlikely vulnerable. Further, we integrated a solution to determine if a host present in the initial configuration changed IP or MAC address as a result of the configuration update. Finally, we also suggest an anomaly detection approach (SKNN) that relies on sorting the results produced by the K-Nearest Neighbour algorithm to identify anomalous hosts.

Our empirical assessment of FISTS shows that best results are obtained with SKNN and pruning, with precision, recall, and F1 score reaching peaks of 0.95, 1.00, and 0.94, respectively. Also, they enable detecting all the anomalies injected in our datasets (up to 50) by inspecting up to 37.98 hosts on average. Further, we demonstrated the accuracy of the host matching algorithm, and the scalability of FISTS, which can scan 405 IPs in 30 minutes (parallelizable), and processes the collected data in less than a minute. Our replication package and results are available online [238].

Chapter 5

Anomaly Detection Algorithms for Field-based Security Testing of Updated SDN Configurations

5.1 Introduction

Software Defined Networks (SDNs) significantly enhance network management by providing dynamic, centralized control that enables rapid reconfiguration and adaptability. However, this centralized flexibility introduces unique security vulnerabilities, particularly pronounced during network reconfiguration phases. The criticality of secure SDN reconfigurations is especially evident in Satellite Communications (SATCOM), where any security oversight can severely disrupt crucial services, impacting both civilian and military operations.

In Chapter 4, we introduced FIeld-based Security Testing of SDN Configuration Updates (FISTS), a methodology developed to identify and prioritize vulnerabilities induced by SDN reconfiguration activities. Through systematic scanning, host state matching, anomaly detection, and prioritized inspection, FISTS effectively aids engineers in swiftly pinpointing misconfigurations and security flaws.

Despite its effectiveness, FISTS exhibits limitations related to limited number of algorithms inspected and expert feedback was not incorporated in the learning process. To overcome these constraints, this chapter introduces significant enhancements to the FISTS. Specifically, we extend our prior work by evaluating 33 additional prioritization strategies. These strategies encompass: (a) integration of seven state-of-the-art anomaly detection algorithms previously unexplored, (b) a newly

developed active anomaly discovery procedure, termed pFIST-H and pFISTS-W, designed to augment the effectiveness of existing anomaly detection algorithms, which integrates self-training methods and human-in-the-loop feedback mechanisms, and (c) also created more comprehensive dataset which leads to evaluation of more practical real world use-cases.

We compare and analyze the performance of 37 distinct anomaly detection algorithms—comprising the 33 newly integrated procedures and the four best-performing algorithms from our previous investigation—under various configuration settings, such as different pruning strategies and stopping conditions. This comprehensive study enables us to identify optimal configurations, further enhancing the precision, scalability, and overall effectiveness of FISTS in safeguarding SDNs against security vulnerabilities during reconfiguration processes.

5.2 Improved FISTS

This section describes the technical improvements introduced into the vulnerability prioritization algorithm of FISTS (hereafter, pFISTS), which are: supporting an additional set of anomaly detection algorithms, integrating a dedicated HITL algorithm (hereafter, pFISTS-H), and leveraging a dedicated weakly supervised learning algorithm with HITL (hereafter, pFISTS-W). These extensions are key to support our extensive empirical assessment of anomaly detection algorithms for field-based testing of SDNs, which is the core contribution of this paper.

Figure 5.1 provides an overview of pFISTS. It receives as input the anomaly detection algorithm selected by the cybersecurity analyst, the degree of user interaction chosen by the analyst (i.e., HITL, HITL with weakly supervised training, or none) and the preprocessed NSCR, which is a table where each row captures information about one host, with columns capturing the state of each port, after one-hot-encoding (see Section 4.2).

pFISTS, first (Line 1 in Figure 5.1), executes the anomaly detection algorithm to obtain the *prioritized pNSRC*, which results from adding to each row of the pNSRC the anomaly score computed by the anomaly detection algorithm and sorting it accordingly.

pFISTS, then, operates according to the degree of interaction chosen by the cybersecurity analyst. Specifically, it either executes pFISTS-H on the prioritized NSCR (Line 3 in Figure 5.1), or executes pFISTS-W on the prioritized NSCR (Line 7), or just returns the prioritized NSCR to the analyst (Line 10). Both pFISTS-H and pFISTS-W include instructions to provide the prioritized NSCR to the analyst, but within their respective HITL loop. As for the original FISTS implementation, the analyst

Require: $Algo$, anomaly detection algorithm selected by the end-user
Require: $InteractivityChoice$, type of interactivity with end user
Require: $pNSCR$, preprocessed NSCR
Execute *Algo* and obtain the *prioritized pNSRC*
if $InteractivityChoice$ is *HITL* **then**
    Execute pFISTS-H on *prioritized pNSCR*
    Return
end if
if $InteractivityChoice$ is *weakly supervised* **then**
    Execute pFISTS-W on *prioritized pNSCR*
    Return
end if
Provide *prioritized pNSCR* to analyst
Return

Figure 5.1: Pseudo code for pFISTS

Require: $ppNSCR$, prioritized NSCR (most anomalous datapoints appear first)
$counter \leftarrow 0$
while $counter < 10$ **do**
    Present *ppNSCR[counter]* to the analyst and collect *feedback* (true label)
    $labels[counter] \leftarrow feedback$
    $counter \leftarrow counter + 1$
end while
$labeledNSCR \leftarrow ppNSCR[0:10] + labels[counter]$
$labeledNSCR \leftarrow labeledNSCR \cup ppNSCR[-10:]$ labeled as $Not\ anomalous$
Train logistic regression model (*LR*) on $labeledNSCR$
$unlabeledNSCR \leftarrow ppNSCR \setminus labeledNSCR$
Predict probability score for $unlabeledNSCR$ using *LR*
Sort $unlabeledNSCR$ based on estimated probabilities (most anomalous first)
$counter \leftarrow 0$
while $unlabeledNSCR$ not empty **do**
    $host \leftarrow$ pop the first item from $unlabeledNSCR$
    Provide $host$ to analyst and collect feedback
    $labels[counter] \leftarrow feedback$
    if prediction is incorrect **then**
        $labeledNSCR \leftarrow labeledNSCR \cup unlabeledNSCR[0:counter] + labels$
        $unlabeledNSCR \leftarrow unlabeledNSCR \setminus labeledNSCR$
        Train *LR* on $labeledNSCR$
        Predict probability score for $unlabeledNSCR$ using *LR*
        Sort $unlabeledNSCR$ based on estimated probabilities (most anomalous first)
    end if
    $counter \leftarrow counter + 1$
end while

Figure 5.2: Pseudo code for pFISTS-H. Vector operators follow Python syntax.

inspects the provided hosts one-by-one and, in all the cases, when operating for *hosts selection*, the analyst is expected to stop inspecting hosts once a given number of false positives is observed. Below, we describe pFISTS-H and pFISTS-W.

5.2.1 pFISTS-H: pFISTS with HITL

Inspired by AAD approaches (see Section 2.9), we propose to iteratively provide anomalous hosts to the analyst and collect her feedback (i.e., whether the provided data points are anomalous or not) to improve anomaly detection results. To enable the assessment of different algorithms with HITL, instead of developing an approach tailored to a given anomaly detection algorithm, we propose an

approach that can be integrated with any anomaly detection algorithm. Specifically, we compute an initial anomaly score for all the data points using an anomaly detection algorithm selected by the end user and then leverage the logistic regression algorithm to fine-tune the anomaly score based on the analyst's feedback. The logistic regression algorithm works by computing a linear combination of input features for a data point to estimate, through a logistic function, the probability that the data point belongs to a class (in our case, that it is vulnerable). In pFISTS, the probability computed by the logistic regression algorithm is then used to re-sort hosts thus leveraging the analyst feedback.

Our rationale is based on the observation that an anomaly detection algorithm can make mistakes in three situations. First, since anomaly detection algorithms typically treat the majority of (similar) data points as non-anomalous, if there are more vulnerable hosts than valid ones, then the anomaly score would somehow be flipped, and non-vulnerable hosts would be presented first, which is undesirable. Second, we expect that some company-specific network changes (e.g., all Web services are executed on port 8080 instead of 80) may introduce changes that are spread through a subset of hosts (e.g., 10 Web servers out of 400 hosts) thus resulting in multiple anomalies being erroneously reported by the anomaly detection algorithm. Third, certain algorithms may perform better than others on specific cases, which is the reason why ensemble machine learning methods combine the prediction of different models to compute accurate results. Logistic regression can address all these three cases. Indeed, logistic regression, when computed using the anomaly score alone, could rearrange (e.g., flip) the initial predictions made by the anomaly detection algorithm. Further, and more importantly, when the logistic regression algorithm is applied to both the anomaly score and all the features of the NSCR it could learn how specific cases (e.g., port 80 becoming close and port 8080 open) shall affect the score computation (in our example above, lower the anomaly score). Last, logistic regression can be used to build an ensemble model that leverages, as features, the predictions of different anomaly detection algorithms. In our empirical assessment, we separately consider these different cases.

pFISTS-H is shown in Figure 5.2. Lines 1 to 6 provide a special treatment for the first ten data points. Indeed, since logistic regression is a supervised method, we use the first ten data points returned by the anomaly detection algorithm to train the logistic regression model; specifically, we present these data points to the analyst in the order provided by the anomaly detection algorithm and collect feedback (i.e., true or false, indicating if they are vulnerable or not). Further, since the first ten data points are likely vulnerable cases, we also consider the last ten data points but assume they are correctly scored as not vulnerable. The first ten and last ten datapoints are thus used to build a labeled dataset used to initially train the logistic regression model (Lines 7 to 9). The trained model is then

used to predict the probability score for the unlabeled data (Line 11) and sort the data (Line 12).

The algorithm, then, iteratively pops the first (i.e., most anomalous) host from the unlabeled data, presents it to the analyst, and collects feedback. Based on feedback, pFISTS-H determines whether retraining of the logistic regression model is necessary. Retraining occurs when the feedback does not match the prediction, that is, (a) the analyst indicates that the host is vulnerable, but the probability score is lower than 0.5, or (b) the analyst indicates that the host is not vulnerable, but the probability score is higher than 0.5. For retraining, the labeled dataset is extended with the unlabeled data inspected so far (Line 19) and it is used to train again the logistic regression model which, in turn, is then used to predict the probability for the remaining unlabeled data (Line 22) and sort it accordingly (Line 23). If the prediction is correct, to save computational power, the logistic regression model is not retrained.

The algorithm provided in Figure 5.2 applies logistic regression to the anomaly score of one algorithm and the other features of the NSCR. Applying logistic regression to derive an ensemble method based on HITL is conceptually similar; the key differences are (1) for each host, the anomaly scores of multiple anomaly detection algorithms are provided within the prioritized NSCR, (2) for the first ten datapoints, the score of only one algorithm (i.e., the first in the user selection) is used for prioritization, (3) training of logistic regression is performed using the anomaly score of multiple algorithms instead of one.

5.2.2 pFISTS-W: Weakly Supervised pFISTS with HITL

In our work, we leverage weakly supervised learning with incomplete supervision through self-training, which consists of initially training the model on the labeled data, then use it to predict labels for the unlabeled data, which are then used to retrain the model (see Section 2.10). Our rationale is that anomaly detection algorithms are good at identifying extreme cases (i.e., hosts that are very likely to be vulnerable or not vulnerable because they contain atypical or no port changes) but under perform when the separation between anomalous and not anomalous cases is not clear; therefore, we aim to leverage unsupervised learning to determine those unclear cases and rely on HITL to appropriately prioritize them.

Figure 5.3 provides the pseudo-code for pFISTS-W. It consists of a first loop performing weakly supervised leaning (Lines 6 to 15) and a second loop performing HITL (Lines 18 to 29).

We leverage logistic regression as the underlying model to determine how confidently anomalous and non-anomalous hosts are separated by the anomaly detection algorithm. Specifically, we create

Require: $ppNSCR$, prioritized NSCR (most anomalous datapoints appear first)
Ensure: A prioritized list of hosts is shown to the analyst
1: Initialize logistic regression model and other variables
2: #Initialize the labeledNSCR with the likely correct predictions
3: $labeledNSCR \leftarrow labeledNSCR \cup ppNSCR[0:10]$ labeled as $True$
4: $labeledNSCR \leftarrow labeledNSCR \cup ppNSCR[-10:] + False * 10$
5: $NSCR \leftarrow ppNSCR \cup labeledNSCR$
6: **while** $ppNSCR$ not empty **do**
7: Train logistic regression model on $labeledNSCR$
8: Predict labels and probabilities for $ppNSCR$
9: Identify confidently predicted points
10: **if** no confident predictions **then**
11: Break the loop
12: **end if**
13: Add confidently predicted points and their prediction to $labeledNSCR$
14: Remove confidently predicted points from $ppNSCR$
15: **end while**
16: Provide to analyst all datapoints confidently predicted as anomalous
17: Sort $ppNSCR$ based on last predicted probabilities
18: **while** $ppNSCR$ not empty **do**
19: $host \leftarrow$ pop the first item from $ppNSCR$
20: Provide $host$ to analyst and collect feedback
21: Add $host$ and $feedback$ to $processed$
22: **if** prediction is incorrect **then**
23: #Add the collected feedbacks to labeled NSCR
24: $labeledNSCR \leftarrow labeledNSCR \cup processed$
25: Train *LR* on $labeledNSCR$
26: Update predictions for $ppNSCR$
27: Sort $ppNSCR$ based on the predicted probabilities
28: **end if**
29: **end while**
30: Provide to analyst all datapoints confidently predicted as not anomalous

Figure 5.3: Pseudo code for pFISTS-W.

an initial *labeled NSCR* using the the first and the last ten data points in the prioritized NSCR as representatives for vulnerable and non-vulnerable hosts (Lines 3 to 5). We then train the logistic regression model using such labeled dataset (Line 7), predict the probability score (i.e., likelihood of being anomalous) for the unlabeled items in the prioritized NSCR (Line 8), and identify the data points most confidently predicted. Confidently predicted data points are those with a probability score higher than 90% (likely vulnerable) or lower than 10% (likely not vulnerable). Those data points are then added to the labeled NSCR (Line 13), which is then used to retrain the logistic regression model in the next iteration of the algorithm (Line 7), if any data point in the NSCR remains to be predicted (Line 6). The weakly supervised loop stops not only when there are no more data points to be predicted (unlikely situation), but also when the logistic regression can't make any confident prediction (Line 11), which means that we have also identified all the items for which the anomaly detection algorithm is unsure.

pFISTS-W then provides to the analyst all the data points confidently predicted as anomalous (Line 16) because they likely need to be inspected urgently. Then, if there are items that could not be confidently predicted using the weakly supervised pass (i.e., $ppNSCR$ *not empty*, Line 18), pFISTS-W

relies on the HITL pass to leverage the analyst's feedback to prioritize them. Precisely, after sorting the remaining data points based on the last predicted probabilities (Line 27), pFISTS-W follows pFISTS-H approach and iteratively provides the most anomalous host to the analyst and collects feedback (Line 21); if the collected feedback does not match the prediction made by the logistic regression (same criteria of pFISTS-H), it adds the data points with feedbacks collected so far to the labeled dataset (Line 24), retrains the logistic regression model (Line 25), updates the predictions for unlabeled data points (Line 26), and sorts them based on the prediction score (Line 27). The iteration continues till all the data points resulting in not confident predictions are provided to the analyst; in practice, the HITL pass shall help providing those data points in the best order possible (i.e., anomalies first).

Last, pFISTS-W provides the analyst with the data points that were confidently predicted as not anomalous (Line 30) because they may still include some anomalous changes to be inspected.

5.3 Empirical Evaluation

All the configuration options available in FISTS (i.e., prioritization algorithm, interactivity choice, stopping condition, and pruning) enable the end-user to configure what we call a FISTS pipeline. In this Section, we aim at comparing the effectiveness and scalability of different FISTS pipelines. Note that we distinguish between FISTS sorting procedure (i.e., the combination of prioritization algorithm and interactivity choice) and FISTS pipelines (i.e., what results from all the options). We aim to address the following research questions (RQs):

RQ1. Does pruning improve the results obtained by the new sorting procedures integrated into FISTS? Previous work has shown that pruning helps improving anomaly detection precision and recall (Section 4.3; however, the newly integrated sorting procedures may lead to different results.

RQ2. What FISTS pipeline leads to the best results? The performance of a FISTS pipeline may depend on the characteristics of the SDN being tested. For instance, the proportion and number of hosts invalid state changes can lead to different performance outcomes for a FISTS sorting procedure, which, in turn, may also affect the effectiveness of a stopping condition. We aim to evaluate what new FISTS pipelines achieve the best results across different datasets and compare results with the best pipelines identified in our previous work.

RQ3. How do FISTS pipeline scale? Our previous work demonstrated that FISTS data collection (Steps 1, 2, 3, and 4) is feasible within the maintenance window of SATCOM networks. We thus aim to evaluate the scalability of the anomaly detection process, for networks that are representative of

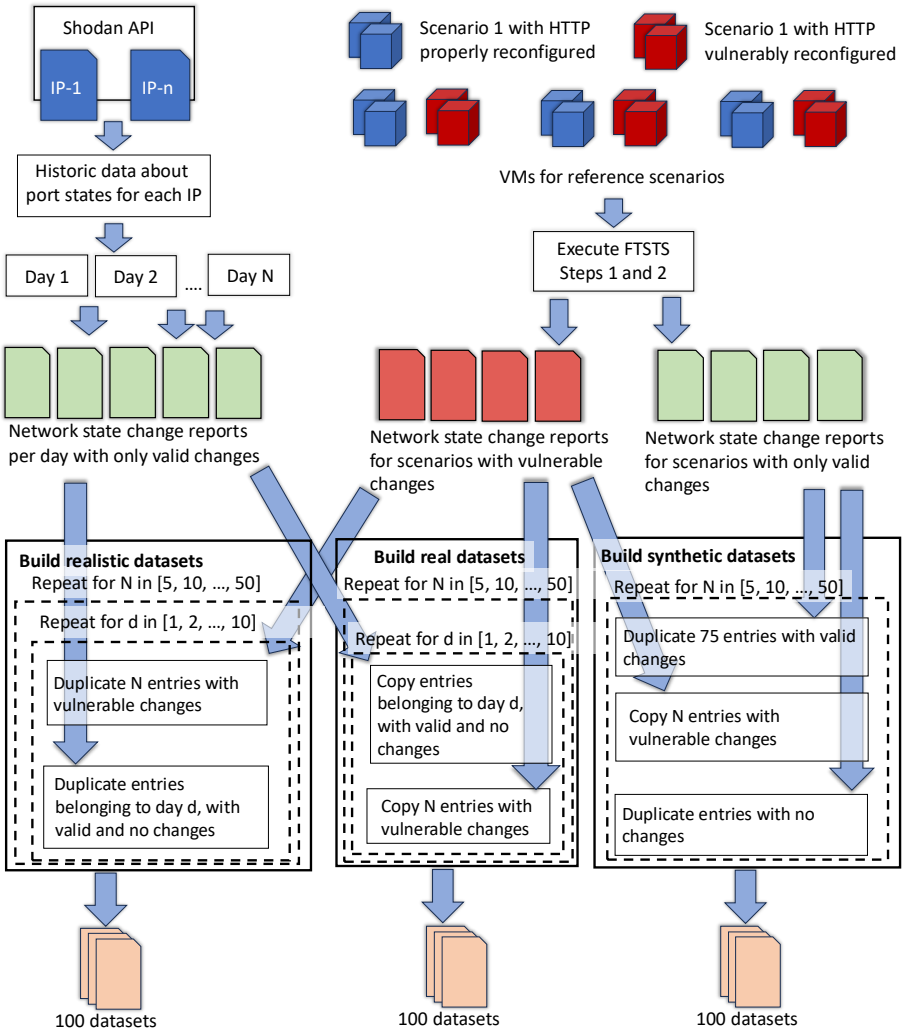


Figure 5.4: Overview of Data Generation

SATCOM organizations.

5.3.1 Assessed sorting procedures

For our experiments we considered FISTS sorting procedures resulting from the combination of different anomaly detection algorithms with pFISTS-H and pFISTS-W. Specifically, we considered the best performing algorithm in our previous study, which is sKNN, and the anomaly detection algorithms presented in Section 2.8, which are CBLOF, COF, LUNAR, OCSVM, HBOS, PCA, and KPCA.

Table 5.1 provides, for each sorting procedure considered in our experiment, an identifier and a description of the components it integrates. The dash symbol is used to separate information on the algorithms used to compute anomaly scores (on the left) from information on the use of pFISTS-H

Table 5.1: Abbreviations and Full Names

Algorithm	**Description**
sPCA	Anomaly score computed by PCA.
sCOF	Anomaly score computed by COF.
sKPCA	Anomaly score computed by KPCA.
sHBOS	Anomaly score computed by HBOS.
sLUNAR	Anomaly score computed by LUNAR.
sOCSVM	Anomaly score computed by OCSVM.
sCBLOF	Anomaly score computed by CBLOF.
sKNN10	Anomaly score from KNN, configured with *K=10*.
sKNN15	Anomaly score from KNN, configured with *K=15*.
sKNN20	Anomaly score from KNN, configured with *K=20*.
sKNN25	Anomaly score from KNN, configured with *K=25*.
sPCA−H	Anomaly score from PCA, combined with pFISTS-H
sCOF−H	Anomaly score from COF, combined with pFISTS-H.
sHBOS−H	Anomaly score from HBOS, combined with pFISTS-H.
sOCSVM−H	Anomaly score from OCSVM, combined with pFISTS-H.
sKNN15−H	Anomaly score from KNN (K=15), combined with pFISTS-H.
sPf−H	Anomaly score from PCA, combined with pFISTS-H, using all features for logistic regression.
sCf−H	Anomaly score from COF, combined with pFISTS-W, using all features for logistic regression.
sHf−H	Anomaly score from HBOS , combined with pFISTS-H, using all features for logistic regression.
sKf−H	Anomaly score from KNN (K=15), combined with pFISTS-H, using all features for logistic regression.
sOf−H	Anomaly score from OCSVM , combined with pFISTS-H, using all features for logistic regression.
sOH−H	Anomaly score from OCSVM and HBOS, combined with pFISTS-H.
sOHP−H	Anomaly score from OCSVM, HBOS, and PCA, combined with pFISTS-H.
sOHPK−H	Anomaly score from OCSVM, HBOS, PCA, and KNN (K=15), combined with pFISTS-H.
sPCA−W	Anomaly score from PCA, combined with pFISTS-W.
sCOF−W	Anomaly score from COF, combined with pFISTS-W.
sHBOS−W	Anomaly score from HBOS, combined with pFISTS-W.
sOCSVM−W	Anomaly score from OCSVM, combined with pFISTS-W.
sKNN15−W	Anomaly score from KNN (K=15), combined with pFISTS-W.
sPf−W	Anomaly score from PCA, combined with pFISTS-W, using all features for logistic regression.
sCf−W	Anomaly score from COF, combined with pFISTS-W, using all features for logistic regression.
sHf−W	Anomaly score from HBOS, combined with pFISTS-W, using all features for logistic regression..
sKf−W	Anomaly score from KNN (K=15), combined with pFISTS-W, using all features for logistic regression.
sOf−W	Anomaly score from OCSVM, combined with pFISTS-W, using all features for logistic regression.
sOH−W	Anomaly score from OCSVM and HBOS, combined with pFISTS-W.
sOHP−W	Anomaly score from OCSVM, HBOS, and PCA, combined with pFISTS-W.
sOHPK−W	Anomaly score from OCSVM, HBOS, PCA, and KNN (K=15), combined with pFISTS-W.

or pFISTS-W (on the right). A lowercase f before the dash indicates that the logistic regression algorithm in pFISTS-H or pFISTS-W leverages not only the anomaly score but all the features (i.e., port state changes) available in the *NSCR*. The instances of pFISTS-H and pFISTS-W that work on an algorithm ensemble present the initial letter of the integrated algorithms on the left of the dash symbol (e.g., *sOHP-W*).

In total, we considered 37 different sorting procedures that, combined with 11 stopping conditions (we considered all the values from 1 to 10 and 20, as in our previous study), and with pruning being enabled or disabled. It led to 814 (i.e., $37 \times 11 \times 2$) FISTS pipelines being assessed.

5.3.2 Subjects of the study

In Chapter 4 our focus is on SATCOM providers that use SDNs for network communications. To conduct experiments that mirror production conditions while still being able to share our datasets without violating confidentiality agreements, we utilized public data about SES networks available on the Shodan network scan repository [239].

For our experiments, we created three types of datasets (i.e., real, synthetic, and realistic) following the process illustrated in Fig. 5.4; each dataset contains the same data as an NSCR. Among datasets in current study, synthetic datasets have similar characteristics; whereas, real and realistic datasets have been created differently by monitoring multiple days for Shodan.

To create the *synthetic dataset*, we monitored three distinct instances of an erroneous split of a subnet resulting in a redundant rule, and a correct reconfiguration for the same scenario (shown in Fig. 4.1, see Chapter 4 for more detail). We also simulated a scenario with a correct and incorrect reconfiguration of a subnet, with and without configuring firewall rules, respectively. We then duplicated the entries belonging to properly and improperly (i.e., vulnerable) configured hosts to build 100 distinct datasets, with a varying number of invalid changes (from 5 to 50 in steps of 5).

For the *real dataset*, different from Chapter 4, to increase diversity, we analyzed, instead of one day, the first ten days monitored by Shodan in January 2025, which concern 280 SES hosts, assuming each IP belongs to a distinct host. We determined port state changes by comparing port states collected in subsequent scans. We built a different dataset from each monitored day. Further, we selected N anomalous entries belonging to the synthetic data and combined them with the valid entries belonging to a specific Shodan daily scan; different from our previous study, we also consider cases where the number of anomalous changes is higher than the number of valid changes.

For the *realistic dataset*, we followed the same procedure of the real dataset but we duplicate valid entries from Shodan till reaching 405 hosts thus better balancing the number of vulnerable and legal changes. The main difference between the real and realistic dataset is that the real dataset includes less hosts with valid changes.

5.3.3 RQ1: Pruning effectiveness

Experiment Design

To address RQ1, we executed all our 814 FISTS pipelines and compared the results obtained by FISTS pipelines with pruning and FISTS pipelines without pruning.

To deal with randomness, we executed each FISTS pipeline 40 times on each dataset. Since, in total, we have 300 datasets, it leads to a total of 9,768,000 runs (814 * 40 * 300).

FISTS produces as output a prioritized NSCR, with hosts prioritized according to their anomaly score (highest first). Since, for each dataset, we know which are the vulnerable hosts, we can determine false positives (i.e., hosts that are not vulnerable but are inspected before reaching the stopping condition) and true positives (i.e., inspected hosts that are vulnerable).

To assess *FISTS hosts selection*, given a prioritized NSCR, we can determine the number of false and true positives observed. False and true positives enable us to compute precision, recall, and F1 score according to standard formula. We assess our pipelines using recall (because our ultimate goal is to identify all the vulnerabilities) and F1 score (because it captures how an approach balances precision and recall). We do not discuss precision because it might be misleading (e.g., high but no vulnerability is discovered).

To assess *FISTS hosts prioritization*, we count the number of hosts to inspect before identifying all the true positives, which enables determining how quickly a configuration helps engineers detect all the faults. We refer to such metric as *debugging cost (DC)*. By definition, FISTS pipelines that differ only for SC will lead to the same debugging cost.

To perform our assessment, we consider each dataset type separately because different FISTS pipelines may perform differently with different datasets.

To determine if, overall, pruning is beneficial for FISTS, we compare the distribution of recall and F1 score obtained by FISTS pipelines, in each dataset, with and without pruning.

Results

Table 5.2 reports the results obtained by the FISTS pipelines. Precisely, for each sorting procedure, we report the SC leading to the highest recall and F1 score. We also report the debugging cost (DC) of each pipeline.

To discuss RQ1, we report in Table 5.3 the distribution of recall, F1 score, and DC observed with and without pruning. For recall and F1 score, there is no difference for the best result (in bold)

Table 5.2: Results for synthetic, real, and realistic dataset with and without pruning

	Synthetic								Real								Realistic							
	Without Pruning				With Pruning				Without Pruning				With Pruning				Without Pruning				With Pruning			
Algorithm	**SC**	**Rec.**	**F1.**	**DC**	**SC**	**Rec.**	**F1.**	**DC**	**SC**	**Rec.**	**F1.**	**DC**	**SC**	**Rec.**	**F1.**	**DC**	**SC**	**Rec.**	**F1.**	**DC**	**SC**	**Rec.**	**F1.**	**DC**
sCBLOF	20	0.93	0.55	61.04	20	0.91	0.64	58.9	20	0.96	0.78	38.43	20	0.95	0.77	37.81	20	0.94	0.66	89.91	20	0.95	0.67	92.4
sCBLOF	20	0.93	0.55	61.04	1	0.88	0.91	58.9	20	0.96	0.78	38.43	20	0.95	0.77	37.81	1	0.94	0.93	89.91	1	0.95	0.95	92.4
sCOF	20	0.92	0.64	38.85	20	0.99	0.68	29.58	20	1	0.81	31.17	20	1	0.81	30.62	20	1	0.7	27.5	20	1	0.7	27.5
sCOF	1	0.87	0.9	38.85	1	0.97	0.96	29.58	8	0.96	0.84	31.17	8	0.97	0.84	30.62	1	1	0.97	27.5	1	1	0.97	27.5
sCOF−W	20	0.65	0.39	320.18	20	0.31	0.2	104.98	7	0.98	0.86	30.03	8	0.99	0.85	29.76	1	1	0.97	27.5	1	1	0.97	27.5
sCOF−W	20	0.65	0.39	320.18	20	0.31	0.2	104.98	20	1	0.82	30.03	20	1	0.81	29.76	20	1	0.7	27.5	20	1	0.7	27.5
sCf−W	20	0.4	0.22	279.7	20	0.28	0.18	93.84	20	0.96	0.81	30.07	20	0.96	0.81	29.97	20	1	0.74	27.5	20	1	0.74	27.5
sCf−W	1	0.15	0.24	279.7	1	0.14	0.23	93.84	7	0.95	0.82	30.07	3	0.86	0.82	29.97	1	1	0.97	27.5	1	1	0.97	27.5
sCf−H	1	0.14	0.22	83.84	1	0.13	0.2	92.27	1	0.81	0.86	30.16	1	0.82	0.86	29.78	1	0.96	0.95	38.82	1	0.96	0.95	38.82
sCf−H	20	0.39	0.22	83.84	20	0.17	0.12	92.27	20	1	0.81	30.16	20	1	0.81	29.78	20	0.96	0.67	38.82	20	0.96	0.67	38.82
sCOF−H	1	0.06	0.11	91.43	1	0.06	0.11	72.61	1	0.72	0.79	31.07	1	0.72	0.78	30.69	1	0.91	0.9	54.3	1	0.91	0.9	54.3
sCOF−H	20	0.3	0.18	91.43	20	0.16	0.12	72.61	20	1	0.81	31.07	20	1	0.81	30.69	20	0.91	0.63	54.3	20	0.91	0.63	54.3
sHBOS	1	1	0.97	27.5	1	0.81	0.87	52.82	20	0.97	0.79	35.88	20	0.97	0.79	35.89	1	1	0.97	27.5	1	1	0.97	27.5
sHBOS	1	1	0.97	27.5	1	0.81	0.87	52.82	20	0.97	0.79	35.88	20	0.97	0.79	35.89	1	1	0.97	27.5	1	1	0.97	27.5
sHBOS−W	20	0.99	0.6	59.4	20	0.63	0.4	81.08	20	0.98	0.79	36.24	20	0.98	0.79	36.25	20	0.06	0.05	291.6	20	0.06	0.05	291.6
sHBOS−W	20	0.99	0.6	59.4	2	0.35	0.45	81.08	20	0.98	0.79	36.24	20	0.98	0.79	36.25	1	0.06	0.1	291.6	1	0.06	0.1	291.6
sHf−W	20	0.27	0.18	266.12	20	0.49	0.3	104.23	20	0.89	0.74	35.69	20	0.89	0.74	35.7	20	1	0.73	27.5	20	1	0.73	27.5
sHf−W	1	0.21	0.31	266.12	1	0.21	0.31	104.23	20	0.89	0.74	35.69	20	0.89	0.74	35.7	1	1	0.97	27.5	1	1	0.97	27.5
sHf−H	1	0.48	0.61	279.81	1	0.26	0.38	92.13	8	0.97	0.83	30.15	8	0.97	0.83	30.16	1	0.96	0.95	38.82	1	0.96	0.95	38.82
sHf−H	20	0.48	0.41	279.81	20	0.43	0.32	92.13	20	1	0.81	30.15	20	1	0.81	30.16	20	0.96	0.67	38.82	20	0.96	0.67	38.82
sHBOS−H	1	0.06	0.1	296.6	1	0.09	0.16	107.97	9	0.94	0.78	36.24	9	0.94	0.78	36.25	1	0.06	0.1	291.6	1	0.06	0.1	291.6
sHBOS−H	20	0.06	0.05	296.6	20	0.24	0.2	107.97	20	0.98	0.79	36.24	20	0.98	0.79	36.25	20	0.06	0.05	291.6	20	0.06	0.05	291.6
sKNN10	20	0.88	0.62	148.57	20	0.96	0.58	60.66	20	1	0.81	33.07	20	1	0.81	33.12	20	0.89	0.62	151.81	20	0.89	0.62	152.06
sKNN10	1	0.87	0.91	148.57	1	0.87	0.91	60.66	8	0.99	0.84	33.07	8	0.99	0.84	33.12	1	0.88	0.91	151.81	1	0.88	0.91	152.06
sKNN15	20	0.96	0.66	55.94	20	0.94	0.66	39.01	20	1	0.81	30.8	20	1	0.81	30.77	20	0.99	0.69	41.26	20	0.99	0.69	40.76
sKNN15	1	0.92	0.93	55.94	1	0.92	0.93	39.01	1	0.57	0.64	30.8	1	0.57	0.64	30.77	1	0.99	0.97	41.26	1	0.99	0.97	40.76
sKNN15−W	2	0.53	0.62	146.12	1	0.49	0.61	63.38	8	0.99	0.85	31.01	8	0.99	0.85	31.01	2	0.88	0.85	137.83	2	0.88	0.85	137.24
sKNN15−W	20	0.85	0.52	146.12	20	0.89	0.52	63.38	20	1	0.81	31.01	20	1	0.81	31.01	20	0.88	0.6	137.83	20	0.88	0.6	137.24
sKf−W	20	0.75	0.48	172.16	20	0.78	0.43	78.35	20	0.95	0.81	30.64	20	0.95	0.81	30.65	20	1	0.73	31.47	20	1	0.73	27.96
sKf−W	1	0.48	0.61	172.16	1	0.41	0.54	78.35	1	0.6	0.67	30.64	1	0.6	0.67	30.65	1	0.98	0.96	31.47	1	0.99	0.97	27.96
sKf−H	1	0.63	0.75	218.81	1	0.66	0.77	75.82	1	0.81	0.84	29.24	1	0.82	0.84	29.24	1	0.95	0.94	52.89	1	0.95	0.94	52.39
sKf−H	1	0.63	0.75	218.81	1	0.66	0.77	75.82	8	1	0.85	29.24	8	1	0.85	29.24	1	0.95	0.94	52.89	1	0.95	0.94	52.39
sKNN15−H	20	0.83	0.6	156.01	20	0.84	0.58	71.47	20	1	0.81	30.94	20	1	0.81	30.94	20	0.89	0.63	67.69	20	0.9	0.62	68.22
sKNN15−H	1	0.83	0.88	156.01	1	0.79	0.85	71.47	1	0.68	0.73	30.94	1	0.68	0.73	30.94	1	0.89	0.9	67.69	1	0.89	0.9	68.22
sKNN20	20	0.96	0.66	38.03	20	0.91	0.64	39.74	20	1	0.81	31.28	20	1	0.81	31.28	20	1	0.7	27.5	20	1	0.7	27.5
sKNN20	1	0.92	0.93	38.03	1	0.88	0.91	39.74	8	0.92	0.79	31.28	8	0.92	0.79	31.28	1	1	0.97	27.5	1	1	0.97	27.5
sKNN25	20	0.96	0.66	34.96	20	0.88	0.63	42.73	20	1	0.81	32.41	20	1	0.81	32.41	20	1	0.7	27.5	20	1	0.7	27.5
sKNN25	1	0.92	0.93	34.96	1	0.86	0.9	42.73	20	1	0.81	32.41	20	1	0.81	32.41	1	1	0.97	27.5	1	1	0.97	27.5
sKPCA	20	0.65	0.26	377.45	20	0.48	0.25	126.53	20	0.99	0.8	38.94	20	0.99	0.8	39.04	1	0.27	0.39	401	1	0.26	0.38	401
sKPCA	20	0.65	0.26	377.45	20	0.48	0.25	126.53	20	0.99	0.8	38.94	20	0.99	0.8	39.04	20	0.27	0.24	401	20	0.26	0.24	401
sLUNAR	20	0.57	0.38	266.26	20	0.95	0.46	93.97	20	1	0.8	35.7	20	1	0.8	35.84	20	0.55	0.37	288.42	20	0.58	0.37	286.66
sLUNAR	1	0.51	0.63	266.26	1	0.52	0.65	93.97	1	0.51	0.64	35.7	1	0.51	0.64	35.84	1	0.5	0.62	288.42	1	0.5	0.62	286.66
sOCSVM	20	0.99	0.69	28.14	20	0.96	0.67	32.47	20	0.98	0.8	38.2	20	0.98	0.8	38.01	20	1	0.7	27.5	20	1	0.7	27.5
sOCSVM	1	0.99	0.97	28.14	1	0.94	0.94	32.47	20	0.98	0.8	38.2	8	0.81	0.74	38.01	1	1	0.97	27.5	1	1	0.97	27.5
sOCSVM−W	20	0.57	0.35	379.77	20	0.87	0.51	93.83	20	1	0.82	33.69	20	1	0.83	32.82	20	1	0.7	27.5	20	1	0.7	27.5
sOCSVM−W	1	0.3	0.41	379.77	2	0.42	0.53	93.83	20	1	0.82	33.69	20	1	0.83	32.82	1	1	0.97	27.5	1	1	0.97	27.5
sOf−W	1	0.27	0.39	245.2	1	0.26	0.37	99.09	1	0.4	0.5	35.23	1	0.43	0.52	34.31	1	1	0.97	27.5	1	1	0.97	27.5
sOf−W	20	0.36	0.25	245.2	20	0.65	0.36	99.09	20	0.9	0.74	35.23	20	0.92	0.76	34.31	20	1	0.74	27.5	20	1	0.74	27.5
sOf−H	1	1	0.97	30.74	1	1	0.97	30.96	1	0.4	0.5	35.23	1	0.43	0.52	34.31	1	1	0.97	27.5	1	1	0.97	27.5
sOf−H	20	1	0.7	30.74	20	1	0.7	30.96	20	0.9	0.74	35.23	20	0.92	0.76	34.31	20	1	0.74	27.5	20	1	0.74	27.5
sOH−W	2	0.33	0.41	370.86	2	0.42	0.53	93.33	20	1	0.82	33.55	20	1	0.83	32.68	1	1	0.97	27.5	1	1	0.97	27.5
sOH−W	20	0.61	0.37	370.86	20	0.88	0.51	93.33	20	1	0.82	33.55	20	1	0.83	32.68	20	1	0.7	27.5	20	1	0.7	27.5
sOHP−W	20	1	0.68	36.14	20	0.99	0.67	40.47	20	0.94	0.81	33.13	20	0.95	0.83	32.18	1	1	0.97	27.5	1	1	0.97	27.5
sOHP−W	20	1	0.68	36.14	20	0.99	0.67	40.47	20	0.94	0.81	33.13	20	0.95	0.83	32.18	20	1	0.74	27.5	20	1	0.74	27.5
sOHPK−W	20	1	0.68	36.14	20	0.99	0.67	40.47	20	0.94	0.81	33.13	20	0.95	0.83	32.18	1	1	0.97	27.5	1	1	0.97	27.5
sOHPK−W	1	0.49	0.63	36.14	1	0.49	0.63	40.47	20	0.94	0.81	33.13	20	0.95	0.83	32.18	1	1	0.97	27.5	1	1	0.97	27.5
sOHPK−H	20	0.86	0.63	120.38	20	0.82	0.6	65.04	20	0.99	0.8	35.7	20	1	0.8	35.01	20	0.91	0.63	54.3	20	0.91	0.63	54.3
sOHPK−H	1	0.85	0.89	120.38	1	0.81	0.87	65.04	20	0.99	0.8	35.7	20	1	0.8	35.01	1	0.91	0.9	54.3	1	0.91	0.9	54.3
sOHP−H	1	0.37	0.5	191.09	1	0.25	0.38	105.09	2	0.49	0.58	35.98	2	0.54	0.62	35.33	1	0.91	0.9	54.3	1	0.91	0.9	54.3
sOHP−H	1	0.37	0.5	191.09	1	0.25	0.38	105.09	20	0.98	0.79	35.98	20	0.99	0.8	35.33	1	0.91	0.9	54.3	1	0.91	0.9	54.3
sOH−H	20	0.47	0.37	191.09	20	0.28	0.24	105.09	20	0.98	0.79	36.05	20	0.99	0.8	35.4	20	0.91	0.63	54.3	20	0.91	0.63	54.3
sOH−H	1	0.37	0.5	191.09	1	0.25	0.38	105.09	20	0.98	0.79	36.05	20	0.99	0.8	35.4	1	0.91	0.9	54.3	1	0.91	0.9	54.3
sOP−W	1	0.49	0.63	36.14	1	0.49	0.63	40.47	20	0.94	0.81	33.13	20	0.95	0.83	32.18	1	1	0.97	27.5	1	1	0.97	27.5
sOP−W	20	1	0.68	36.14	20	0.99	0.67	40.47	20	0.94	0.81	33.13	20	0.95	0.83	32.18	20	1	0.74	27.5	20	1	0.74	27.5
sOP−H	20	0.47	0.37	191.09	20	0.28	0.24	105.09	20	0.98	0.79	35.98	20	0.99	0.8	35.33	20	0.91	0.63	54.3	20	0.91	0.63	54.3
sOP−H	1	0.37	0.5	191.09	1	0.25	0.38	105.09	2	0.49	0.58	35.98	2	0.54	0.62	35.33	1	0.91	0.9	54.3	1	0.91	0.9	54.3
sOCSVM−H	20	0.47	0.37	191.09	20	0.28	0.24	105.09	20	0.97	0.79	36.98	20	0.98	0.79	36.38	20	0.91	0.63	54.3	20	0.91	0.63	54.3
sOCSVM−H	1	0.37	0.5	191.09	1	0.25	0.38	105.09	20	0.97	0.79	36.98	20	0.98	0.79	36.38	1	0.91	0.9	54.3	1	0.91	0.9	54.3
sPCA	20	0.86	0.62	44.81	20	0.84	0.61	49.09	20	0.97	0.79	36.25	20	0.97	0.79	36.15	20	1	0.7	27.5	20	1	0.7	27.5
sPCA	1	0.85	0.89	44.81	1	0.83	0.88	49.09	20	0.97	0.79	36.25	20	0.97	0.79	36.15	1	1	0.97	27.5	1	1	0.97	27.5
sPCA−W	20	0.45	0.28	289.94	20	0.84	0.37	102.74	20	0.91	0.77	35.01	20	0.91	0.77	34.99	20	1	0.74	27.5	20	1	0.74	27.5
sPCA−W	5	0.36	0.37	289.94	20	0.84	0.37	102.74	20	0.91	0.77	35.01	20	0.91	0.77	34.99	1	1	0.97	27.5	1	1	0.97	27.5
sPf−W	20	0.97	0.65	47.42	20	0.97	0.65	48.71	20	0.9	0.77	34.99	20	0.91	0.77	34.99	20	1	0.74	27.5	20	1	0.74	27.5
sPf−W	1	0.49	0.63	47.42	1	0.49	0.63	48.71	20	0.9	0.77	34.99	20	0.91	0.77	34.99	1	1	0.97	27.5	1	1	0.97	27.5
sPf−H	1	0.53	0.65	252.47	1	0.21	0.33	95.37	9	0.98	0.82	31.46	9	0.98	0.82	31.42	1	0.96	0.95	38.82	1	0.96	0.95	38.82
sPf−H	1	0.53	0.65	252.47	1	0.21	0.33	95.37	1	0.66	0.71	31.46	1	0.66	0.71	31.42	1	0.96	0.95	38.82	1	0.96	0.95	38.82
sPCA−H	1	0.06	0.1	296.32	20	0.21	0.18	109.33	20	0.98	0.8	36.4	20	0.98	0.8	36.36	1	0.06	0.1	291.6	1	0.06	0.1	291.6
sPCA−H	1	0.06	0.1	296.32	1	0.09	0.16	109.33	9	0.94	0.79	36.4	9	0.94	0.79	36.36	1	0.06	0.1	291.6	1	0.06	0.1	291.6
sCOF−W	20	0.65	0.39	320.18	20	0.31	0.2	104.98	7	0.98	0.86	30.03	8	0.99	0.85	29.76	1	1	0.97	27.5	1	1	0.97	27.5
sCOF−W	20	0.65	0.39	320.18	20	0.31	0.2	104.98	20	1	0.82	30.03	20	1	0.81	29.76	20	1	0.7	27.5	20	1	0.7	27.5
sHAC	20	0.76	0.58	249.53	20	0.79	0.58	50.25	20	1	0.81	31.97	20	1	0.81	32	20	0.98	0.69	35.59	20	0.98	0.69	34.98
sHAC	1	0.75	0.83	249.53	1	0.75	0.83	50.25	1	0.60	0.67	31.97	1	0.59	0.66	32	1	0.98	0.95	35.59	1	0.98	0.95	34.98
sIF	1	0.88	0.91	42.26	1	0.89	0.92	36.82	20	0.99	0.80	35.41	20	0.99	0.80	35.42	1	1	0.97	27.50	1	1	0.97	27.50
sIF	20	0.93	0.65	42.26	20	0.93	0.65	36.82	20	0.99	0.80	35.41	20	0.99	0.80	35.42	20	1	0.70	27.50	20	1	0.70	27.50
sKM	1	0.76	0.83	306.90	1	0.70	0.80	58.58	1	0.63	0.70	31.98	1	0.62	0.70	32.07	1	0.98	0.95	35.59	1	0.98	0.95	34.98
sKM	1	0.76	0.83	306.90	1	0.70	0.80	58.58	8	0.94	0.82	31.98	8	0.94	0.82	32.07	1	0.98	0.95	35.59	1	0.98	0.95	34.98

Table 5.3: Distribution for synthetic. real. and realistic dataset with and without pruning

	Synthetic						Real						Realistic					
	Without Pruning			With Pruning			Without Pruning			With Pruning			Without Pruning			With Pruning		
Algorithm	**Rec.**	**F1.**	**DC**	**Rec.**	**F1.**	**DC**	**Rec.**	**F1.**	**DC**	**Rec.**	**F1.**	**DC**	**Rec.**	**F1.**	**DC**	**Rec.**	**F1.**	**DC**
MAX	**1**	**0.97**	379.77	**1**	**0.97**	126.53	**1**	**0.86**	38.94	**1**	**0.86**	39.04	**1**	**0.97**	401	**1**	**0.97**	401
Q3	0.92	0.675	266.23	0.88	0.68	99.09	0.99	0.81	35.98	1	0.82	35.70	1	1	54.3	1	0.97	54.3
Q2	0.65	0.61	164.09	0.68	0.56	74.22	0.97	0.80	33.55	0.97	0.80	32.97	0.98	0.88	35.59	0.99	0.88	34.98
Q1	0.39	0.37	44.81	0.28	0.33	48.71	0.91	0.78	31.17	0.91	0.77	31.01	0.91	0.67	27.5	0.91	0.67	27.5
MIN	0.06	0.05	**27.5**	0.06	0.11	**29.58**	0.4	0.5	**29.24**	0.43	0.52	**29.24**	0.06	0.05	**27.5**	0.06	0.05	**27.5**

observed with and without pruning; quartiles are also very similar thus indicating that pruning does not necessarily improve sorting procedures performance. However, some differences are observed for the DC score (i.e., number of hosts to inspect in order to detect all the vulnerabilities), where the presence of pruning leads to lower DC values (see quartiles) for the synthetic dataset, which is expectable since pruning eliminates some entries. However, pruning does not improve the best achievable results, indeed, without pruning the best FISTS pipeline is *sHBOS*, which leads to DC=27.5; instead, with pruning, the best result is obtained with sCOF, which leads to DC=29.58, a higher (and thus worse) value. More precisely, we can report that several sorting procedures present worse results when executed with pruning, for all the stopping conditions; they are sHBOS, sH-W, sHf-H, sKNN15, sKNN20, sKNN25, sOCSVM, sO-H, sOH-H, sOHP-H, sOHKP-H, sOP-H, sHAC, and sKM. Our previous results have shown a more positive effect of pruning; this happened likely because of the nature of the algorithms previously considered. Indeed, previous work included Isolation Forest, which leverages random selection of features to split, and the presence of less data points likely makes the algorithm more effective. Similarly, we included the HAC and K-means clustering algorithms, where a reduced number of items may change clustering results. Since, in the current real dataset we also consider cases with a larger proportion of vulnerable changes and realistic dataset generation is also different (more comprehensive with multiple days of collected data; see 5.3.2), this may lead to differences between the two studies. sIF in case of realistic dataset has significantly lower DC (27.5) compared to what we had in the previous study (DC=46.52 without and DC=53.91 with Pruning; Section 4.5). sHAC in case of realistic dataset without pruning in the former study had 0 precision and recall; on the contrary, in the current study sHAC achieved recall of 0.98 with DC=35.59. With pruning sHAC improved compared to what we had earlier (DC=47.80 and DC=34.98). sKM in case of realistic dataset also exhibits the same behavior as of sHAC without pruning (0 precision and recall), but exhibits improved results in current study (DC=365.52 vs DC=35.59); also improved DC with pruning (DC=49.81 vs DC=34.98). The improved behavior of sIF, sKM, and sHAC is associated with the dataset generation process.

Concluding, since the use of pruning may negatively affect some sorting procedures, we can't

suggest on the use of pruning with every algorithm; therefore, we aim to identify the best FISTS pipeline (i.e., address RQ2) considering all the available FISTS pipeline results, including the ones without pruning.

5.3.4 RQ2: Best FISTS pipeline

Experiment Design

We rely on the same data collected for RQ1. For hosts selection, the best pipelines for a dataset are the ones that maximize recall.

F1 score shall also be discussed, to identify the best pipelines among those that perform similarly. For host prioritization, the best pipelines are those that minimize DC.

Ideally, the best pipelines (i.e., the ones that we suggest for their use with FISTS) shall perform well in all the different datasets.

Results

For hosts selection in synthetic datasets without pruning, the best performing pipelines are sOf-H and sHBOS with SC=1 (recall is 1.00, F1 score is 0.97) also having higher F1 score. With pruning, the best performing approach is sOf-H with SC=1 (recall is 1.00, F1 score is 0.97). However, a number of other approaches reach a high recall; these are sOCSVM (SC=1 and 20) and sH-W (SC=20), without pruning, and sOP-W (SC=20), sCOF (SC=1), sOHP-W (SC=20), and sOHPK-W (SC=20) with pruning. Such small differences might be due to randomness factors. However, all pipelines with SC=20 lead to a lower F1 score than pipelines with SC=1 (by definition).

In the real datasets, with and without pruning, the best performing pipelines (recall is 1.00) are sCOF, sCOF-H, sCOF-W, sCf-H, sHBOS-f-H, sKNN (with K equal to 10, 15, 20, 25), sLUNAR, sPf-H, sKNN-H, sOCSVM-W, sOH-W, sKNN-f-H, and sHAC using SC=20. Among them, the highest F1 score is obtained by sCOF-W, sOCSVM-W, and sOH-W, thus showing that a weakly supervised algorithm (pFISTS-W) help improving results (i.e., reducing false positives in this case).

In realistic datasets, with and without pruning, the best performing pipelines in terms of recall (1.0) are sHf-W (SC=1 and SC=20), sCOF (SC=1 and SC=20), sCOF-W (SC=1 and SC=20), sCf-W (SC=1 and SC=20), sHBOS (SC=1 and SC=20), sHf-W (SC=1 and SC=20), sKNN20 (SC=1 and SC=20), sKNN25 (SC=1 and SC=20), sOCSVM (SC=1 and SC=20), sOCSVM-W (SC=1 and SC=20), sOf-W (SC=1 and SC=20), sOf-H (SC=1 and SC=20), sOH-W (SC=1 and SC=20), sOHP-W

(SC=1 and SC=20), sOHPK-W (SC=1 and SC=20), sOP-W (SC=1 and SC=20), sPCA (SC=1 and SC=20), sPCA-W (SC=1 and SC=20), sPf-W (SC=1 and SC=20), sKf-W (SC=20), and sIF (SC=1 and SC-20). For all of them, the best F1 score (0.97) is obtained with SC equal to 1.

The pipelines that lead to the best recall for both the real and realistic dataset are sCOF, sCOF-W, sKNN20, sKNN25, sOCSVM-W, sOH-W, with SC=20. Among them, sCOF-W, sOCSVM-W, and sOH-W have the highest F1 score in the realistic dataset. Unfortunately, none of them works best also on the synthetic dataset. The main reason for the lack of pipelines working best in all the three datasets is that although the three datasets present similar vulnerable hosts, the valid hosts in the synthetic dataset are different from the ones in the real and realistic, because the latter two come from Shodan. Precisely, valid hosts in the synthetic datasets have state changes that mainly affect the same ports of vulnerable hosts; in the real and realistic dataset vulnerable host tend to present state changes affecting ports that differ from the ones changes in non-vulnerable hosts.

However, if we look at average performance across datasets we can observe that, with pruning, the highest average recall (best values, up to two decimals of precision) is obtained by sCOF with SC=20 (recall=0.995, F1 score=0.73) and sOCSVM-f-H with SC=20 (recall=0.993, F1 score=0.73). The highest average recall without pruning, instead, is obtained by sHBOS with SC=20 (recall=0.999, F1 score=0.73), sOCSVM-f-H with SC=20 (recall=0.993, F1 score=0.73), and sOCSVM with SC=20 (recall=0.992, F1 score=0.73). The highest average F1 scores are obtained by sCOF with SC=1 with pruning (0.87), sOf-H with SC=1 with pruning (0.865), and sOf-H with SC=1 without pruning (0.865) but they have a recall between 0.84 and 0.85 (i.e., 14% less vulnerabilities detected), and therefore we find it most appropriate to suggest relying on approaches maximizing recall. sKNN20, the best algorithm according to our previous work, still presents a recall that is high (i.e., 0.988, without pruning) but lower than the one of the best algorithms identified in the current study.

For *hosts prioritization*, we discuss the results in Table 5.2 ignoring the configuration value assigned to SC because all hosts are inspected and we are interested in determining how quickly all the vulnerabilities are found (i.e., DC score).

For the synthetic dataset without pruning, the best performing approach is sHBOS (DC=27.5). With pruning, the best performing approach is sCOF (DC=29.58).

For the real dataset the best pipeline, with and without pruning is sKNN-f-H (DC=29.24 in both cases). For the realistic dataset with and without pruning, several pipelines achieve the best DC (27.5), they are: sCOF, sCOF-W, sCf-W, sHBOS, sHf-W, sKNN(with K equal to either 20 or 25), sOCSVM, sOCSVM-W, sOf-W, sOf-H, sOH-W, sOHP-W, sOHPK-W, sOP-W, sPCA, sPCA-W, sPf-W and sIF.

sIF show significantly improved performance with the DC=27.5. sKM and sHAC show improved performance compared to the earlier study with and without pruning, but not good enough compared to other anomaly detection pipelines with best DC=34.98.

For hosts prioritization, it is not possible to identify a pipeline that works best with all the datasets. However, if we consider the average DC score obtained across the three datasets (not reported for space reasons), we observe that sCOF with pruning leads to the lowest cost (i.e., DC=29.58), followed by sHBOS without pruning (i.e., DC=30.29). Other effective methods are sOf-H with pruning (DC=30.92), sOf-H without pruning (DC=31.15), and sOCSVM without pruning (DC=31.28).

Concluding, sCOF with SC=20 and pruning leads to the best result for hosts prioritization and second-best for hosts selection, sHBOS with SC=20 and no pruning leads to the best result for hosts selection and second-best for hosts prioritization, but differences are minimal. For hosts selection, other algorithms configured with SC=20 perform similar to sCOF and sHBOS; they are sOf-H (with or without pruning) and sOCSVM without pruning. pFISTS-H and pFISTS-W help improving the performance of single sorting procedures within a dataset (i.e., for each dataset, there is always one approach with HITL or Weakly supervised among the best performing approaches); further, the pFISTS-H-based sOf-H performs among the best across all datasets. Ensemble methods are also among the top pipelines for both hosts prioritization and selection, but they do not outperform the best ones. Last, despite the sometimes small differences on average performance, across pipelines with SC=20, the above-mentioned pipelines (i.e., sCOF, sOf-H, sHBOS, and sOCSVM) differ significantly (for $\alpha = 0.05$) and practically (i.e., considering effect size) from other pipelines in at least one dataset (Table 5.4).

5.3.5 RQ3 - Scalability of sorting procedures

Experiment Design

We aim to compare sorting procedures based on the time taken to prioritize hosts. We consider the time taken to generate results for RQ1 and RQ2. For pFISTS-H and pFISTS-W approaches, we take into account only the time required to process results, not the time taken by human experts to investigate and label instances. However, this shall not affect the correctness of the conclusions made on the feasibility of our approach; indeed, for all the pipelines, it's desirable that the investigation of anomalous hosts is performed within a maintenance window, which means that the time taken by all the automated sorting procedures shall leave enough time for manual inspection. Consequently, we can compare pFISTS-H and pFISTS-W with other sorting procedures without taking into account the

time taken by human experts to investigate results because, anyway, it's needed by all the pipelines.

Results

Our results show that the use of pruning reduces execution time; indeed, the median execution time observed with pruning is lower than what observed without pruning in all the cases except for sKNN, sCBLOF, sCOF-H, sCf-H, where, however, the medians with and without pruning are similar (but the max and third quartile are lower for pruning).

All the approaches, except sCBLOF, sCOF, sLUNAR, sPf-H, sOHP-H, sOHPK-H, sKM and all weakly supervised (-W) approaches, process each dataset in less than 1.5 seconds; they do not present any practical difference among each other, and do not impact on the testing time.

Among the slower approaches, the slowest is sCBLOF, which takes up to 85 seconds and sKM which takes up to 57 seconds in the case of the realistic dataset without pruning (median 3 and 5 seconds respectively). However, sCBLOF worst-case processing time is acceptable because the network maintenance window is 15 to 30 minutes, and parallelized network scans may reduce data collection time to 10 minutes (Chapter 4); consequently, investing up to a couple of minutes for data analysis and leaving the rest of the maintenance window for the investigation of the most suspicious hosts is deemed acceptable by practitioners.

Interestingly, the set of slowest approaches includes also sCOF, which is part of the best FISTS pipelines based on RQ2 results. sCOF takes up to 14 seconds in case of realistic dataset without pruning (median 8 seconds). Given that, on average, sCOF leads to avoiding the inspection of less than one host compared to sHBOS (i.e., 29.58 VS 30.29, see RQ2), a larger time required for data processing may nullify the advantage obtained during anomaly inspection. Therefore, although 15 seconds for data processing are acceptable, sHBOS shall be preferred for both hosts selection and prioritization, especially for larger datasets.

Despite being among the slowest set of sorting procedures, all pFISTS-W approaches take less than 10 seconds.

The slowest are the ones based on sCf-W, which show execution times similar to sCOF; indeed, sCf-W takes up to 13 seconds in the case of realistic dataset without pruning (median 0.1 sec) and sCOF-W takes up to 12.3 seconds in case of realistic dataset without pruning (median 1.9 sec).

Concluding, except for sCBLOF, which may not scale well to larger datasets, all the investigated sorting procedures are sufficiently quick to be used in production. However, since sCOF is slower than other best performing pipelines in RQ2, we suggest leveraging sHBOS, which performs best for

Table 5.4: Table for Statistical Significance

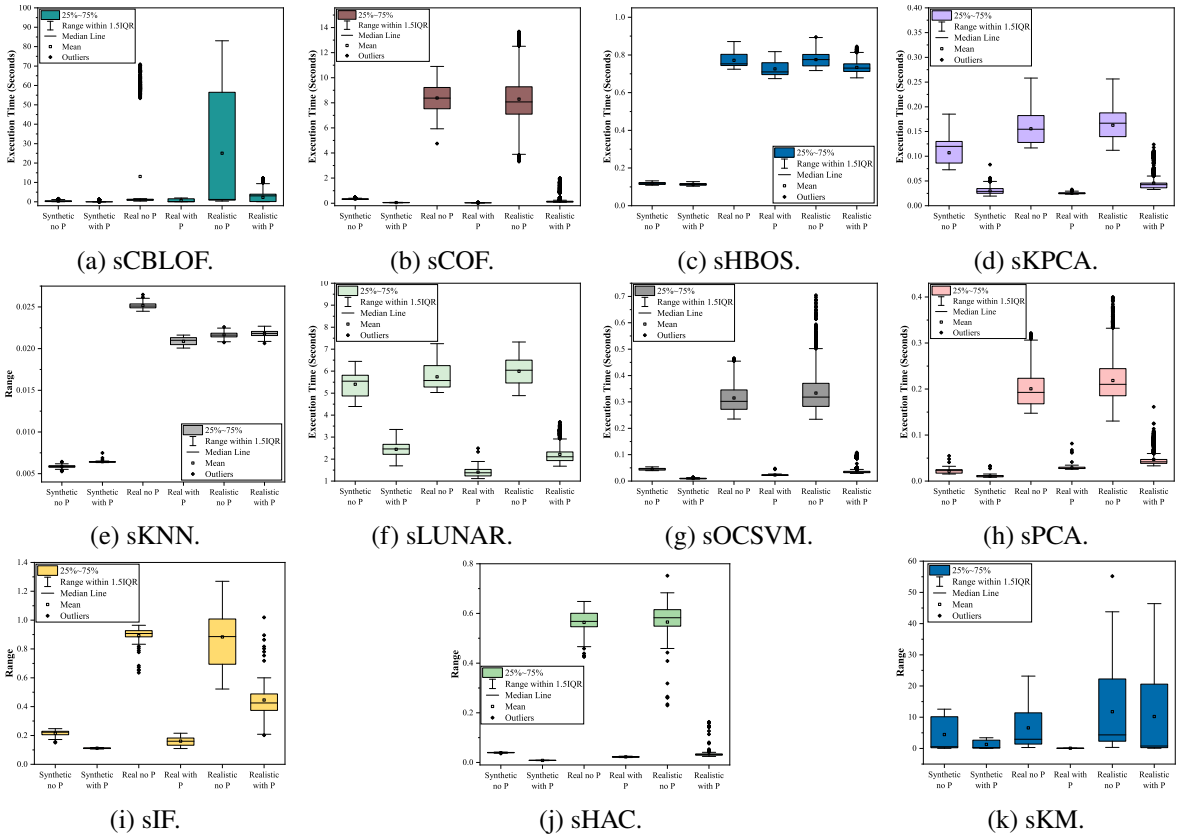
	p-Value $>$ 0.05			0.43 $<$ Effect size $<$ 0.54		
Algo	**Real**	**Realistic**	**Synthetic**	**Real**	**Realistic**	**Synthetic**
sCOF α	sKNN15*, SKf-H, sHf-H, sPf-H, sLunar, sCf-H, sKNN(10/15.20.25)	sP-W, sO-W, sOH-W, sOHP-W, sOHPK-W, sKf-W, sHf-W, sPf-W, sOf-W, sOf-H, sHBOS, sOCSVM, sPCA, sKNN20/25, sOP-H, sOP-W, SC-W	NA	sO-W, sKNN15-W, sOH-W, sOHPK-W, sH-W, sP-H, sH-H, sOH-H, sOHP-H, sKPCA, sOP-H, sHBOS, sPCA, sO-H, sCBLOF, sCOF-W, sOCSVM	sKf-H, sHf-H, sPf-H, sKNN15	NA
sOf-H α	sOf-W	sP-W, sO-W, sOH-W, sOHP-W, sOHPK-W, sKf-W, sHf-W, sPf-W, sOf-W, sHBOS, sPCA, sOCSVM, sCOF, sC-W, sKNN20/25, sOP-W	NA	sOf-W, sP-W, sPf-W	sKf-H, sHf-H, sPf-H, sKNN15, sKf-H, sHf-H, sPf-H	NA
sHBOS β	sO-W, sOH-W, sPCA, sCBLOF, sC-W	sP-W, sO-W, sOH-W, sOHP-W, sOHPK-W, sHf-W, sPf-W, sOf-W, sOf-H, sOCSVM, sPCA, sCOF, sC-W, sKNN20/25, sOP-W	sOf-H	sO-W, sK15-W, sH-W, sOH-W,sP-H, sO-H, sK-H, sH-H, sOH-H, sOHP-H, sOHPK-H, sKf-H, sHf-H, sPf-H, sOCSVM, sPCA, sCBLOF, sCOF, sKPCA, sLUNAR, sOP-W, sC-W, sC-H, sKNN10/15/20/25, sOP-H	sKf-W, sKf-H, sHf-H, sPf-H, sKNN15	NA
sOCSVM β	sOCSVM-H	sP-W, sO-W, sK-W, sOH-W, sOHP-W, sOHPK-W, sHf-W, sPf-W, sOf-W, sOf-H, sHBOS, sPCA, sCOF, sKNN20/25, sOP-W, sOP-H	NA	sOCSVM-W, sKNN-W, sHBOS-W, sOH-W, sP-H, sO-H, sKNN-H, sHBOS-H, sOH-H, sOHP-H, sOHPK-H, sKf-H, sHf-H, sPf-H, sHBOS, sPCA, sCBLOF, sCOF, sKPCA, sLUNAR, sKNN10/15/20/25, sOP-H, sC-W, sC-H	sKf-W, sKf-H, sHf-H, sPf-H, sKNN15	NA
sOf-H β	sPf-W, sOf-W,	sP-W, sK-W, sOH-W, sOHP-W, sOHPK-W, sHf-W, sPf-W, sHBIS, sOCSVM, sPCA, sCOF, sC-W, sKNN20/25, sOP-W	sHBOS	sHf-W, sPf-W, sOf-W	sKf-W, sKf-H, sHf-H, sPf-H, sKNN15	NA

Notes:
$\alpha : WithPruning(SC = 20), \quad \beta : WithoutPruning(SC = 20).$

hosts selection and second-best for hosts prioritization.

5.3.6 Discussion

On average, across our three datasets, the best approaches for hosts selection and prioritization are sHBOS and sCOF, with sHBOS being more scalable and thus preferable. However, we observe that the integration of human-in-the-loop with sOf-H enables an effective use of pruning thus leading to a similar recall (i.e, 0.993 for sOf-H VS 0.999 for sHBOS) and debugging cost (i.e., 30.92 for sOf-H with pruning VS 30.29 for sHBOS without pruning) with lower execution time (i.e., less than 0.1 seconds VS up to 0.8 for sHBOS), which could be useful for larger datasets (e.g., 40,000 nodes). Approaches based on pFISTS-W that integrate pruning, ensemble methods, human-in-the-loop, and weakly-supervised approach leads to best performance when the differences between valid and invalid hosts are minimal (e.g., all hosts present a limited set of ports that could be open), they are sOP-W, sOHP-W, and sOHPK-W.



(a) sCBLOF. (b) sCOF. (c) sHBOS. (d) sKPCA.

(e) sKNN. (f) sLUNAR. (g) sOCSVM. (h) sPCA.

(i) sIF. (j) sHAC. (k) sKM.

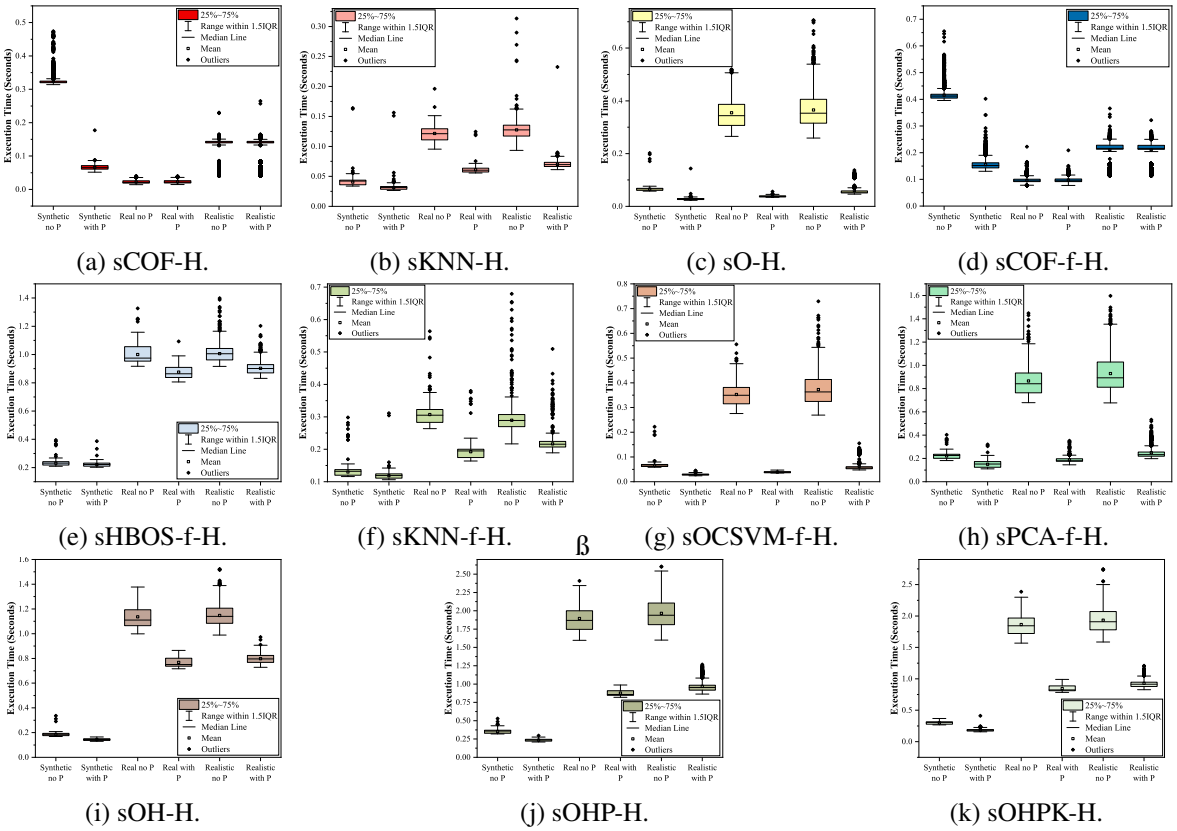
Figure 5.5: Execution time of FISTS sorting procedures

5.3.7 Threats to validity

To address threats to *external validity*, we utilized a comprehensive approach by including diverse datasets in our analysis. These datasets encompass real-world network data, thereby ensuring the generalizability of our findings beyond synthetic or narrowly defined scenarios. This diverse dataset selection aids in capturing various network behaviors and anomalies, significantly mitigating the risk of overfitting to a specific dataset or network scenario.

In addressing *construct validity*, we carefully selected metrics that are widely acknowledged and consistently employed within the anomaly detection research community. Specifically, we focused on metrics such as recall and the F1 score. These metrics effectively capture the essential performance aspects of anomaly detection models, ensuring that our evaluation aligns well with the intended theoretical constructs. By choosing commonly validated and accepted metrics, we further mitigate the risk of misrepresenting or inadequately capturing the effectiveness of our proposed methodologies.

Regarding *conclusion validity*, our conclusions are strongly supported by a robust statistical basis. The findings are derived from a considerable volume of experimental data, specifically 1200 data points



(a) sCOF-H. (b) sKNN-H. (c) sO-H. (d) sCOF-f-H.

(e) sHBOS-f-H. (f) sKNN-f-H. (g) sOCSVM-f-H. (h) sPCA-f-H.

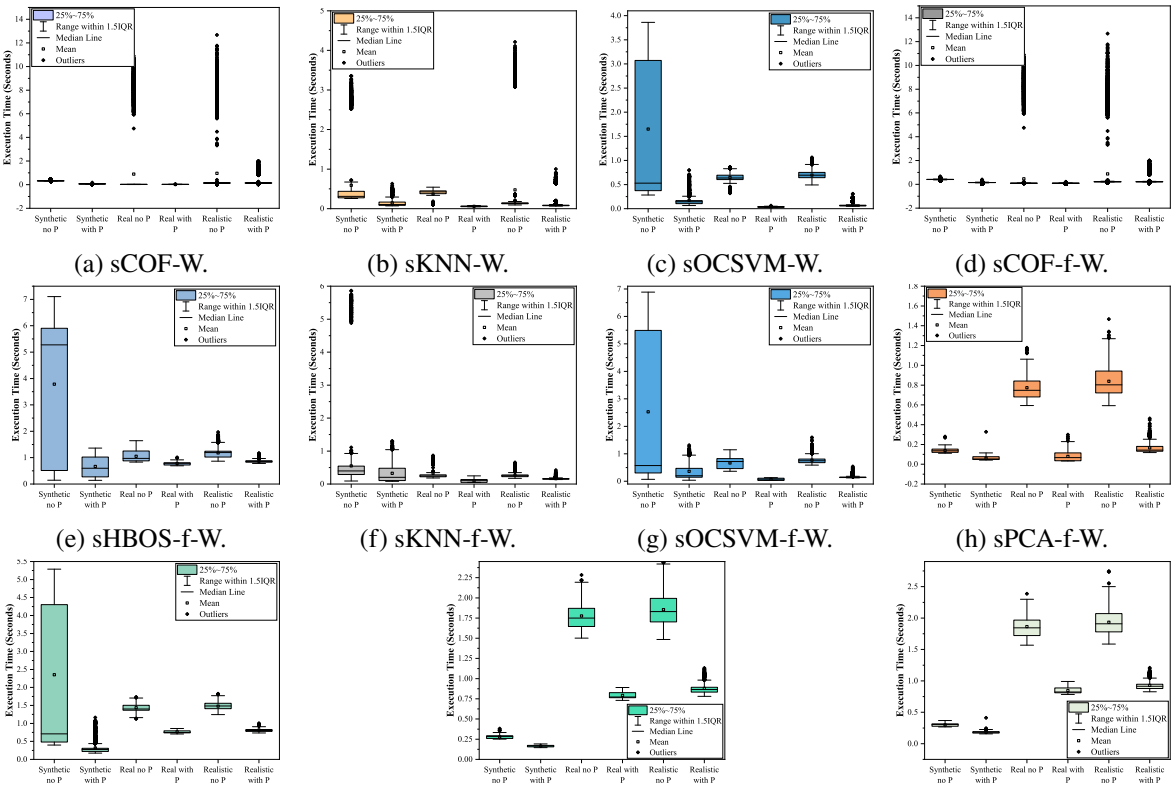
(i) sOH-H. (j) sOHP-H. (k) sOHPK-H.

Figure 5.6: Execution time of sorting procedures based on pFISTS-H

resulting from 40 independent executions across 300 datasets. This large sample size strengthens the statistical power and reliability of our results. Moreover, we utilized rigorous statistical analyses, specifically the chi-square test, to affirm the significance of our findings. Our statistical assessment indicates a clear and significant difference between groups achieving recall values of 0.99 and 0.98, under the assumptions inherent in binomial distribution analyses. This comprehensive statistical approach provides confidence in the reliability and reproducibility of our conclusions.

5.4 Conclusion

We address the identification of SDN misconfigurations in satellite and terrestrial communication systems by significantly extending the capabilities of our approach named FIeld-based Security Testing of SDN Configurations Updates (FISTS). FISTS can be used to either select a subset of hosts with state changes to inspect (hosts selection) or to prioritize the inspection of the whole set of hosts (hosts prioritization). Our enhancements include the integration of 33 additional anomaly detection algorithms, an active anomaly discovery procedure (pFISTS-H), and a novel weakly supervised



(a) sCOF-W. (b) sKNN-W. (c) sOCSVM-W. (d) sCOF-f-W.

(e) sHBOS-f-W. (f) sKNN-f-W. (g) sOCSVM-f-W. (h) sPCA-f-W.

(i) sOH-W. (j) sOHP-W. (k) sOHPK-W.

Figure 5.7: Execution time of sorting procedures based on pFISTS-W

anomaly detection approach incorporating human-in-the-loop (pFISTS-W). Our extensive empirical evaluation demonstrates that integrating a broader spectrum of anomaly detection algorithms provides nuanced insights such as revealing scenarios where pruning does not universally enhance detection effectiveness (e.g., FISTS pipelines with HBOS, OCSVM, and PCA). The best FISTS pipelines for hosts selection and prioritization are the ones leveraging HBOS and COF, with HBOS being more scalable and thus preferable. For much larger datasets, the use of pFISTS-H with OCSVM and HITL using features (i.e., sOCSVM-f-H) will likely lead to same recall of HBOS but lower execution time. Last, approaches based on pFISTS-W perform well when valid and invalid hosts present a limited range of port numbers. Our findings not only validate the improved capabilities of our extended FISTS framework but also offer practical insights for industry practitioners. Our replication package is available online [240].

Chapter 6

Conclusion and Future Prospects

6.1 Conclusion

This dissertation confronts the security-assurance gap that has emerged alongside the rapid adoption of Edge and software-defined systems, concentrating on the reconfiguration of satellite-sector SDN services as an exemplar of a wider industrial challenge. The work begins by establishing a solid empirical foundation: an in-depth analysis of 147 documented vulnerabilities in four widely deployed edge-computing frameworks. That study demonstrates that exhaustive pre-deployment testing is infeasible because the edge's combinatorial space of interfaces, inputs, and outputs simply grows too quickly. Crucially, the analysis reveals that most flaws require only a single, easily triggered action—such as an unauthenticated HTTP request—to compromise confidentiality or authorisation, hence escaping traditional integrity-focused test oracles. These observations justify a strategic shift toward testing that occurs in the operational environment where misconfigurations manifest.

Building on this insight, the dissertation introduces FIeld-based Security Testing of SDN Configuration Updates (FISTS), an automated workflow that probes a network immediately before and after each configuration change, aligns matching nodes across the two snapshots, and then ranks anomalous behaviours with unsupervised machine-learning techniques. Evaluations performed on both simulated topologies and production data from a world-leading satellite operator show that FISTS, when paired with k-Nearest Neighbour–based ranking, attains a precision of 0.95 while preserving perfect recall, all without sacrificing scalability. These results confirm that learning-driven, in-the-field verification can expose misrouted or improperly filtered traffic—the kinds of defects that break confidentiality even when connectivity appears normal—far more effectively than off-line analysis alone.

The research then extends FISTS into a richer human-in-the-loop framework, systematically

benchmarking 37 anomaly-sorting procedures across 814 configuration variants on 300 heterogeneous datasets. The study finds that Histogram-Based Outlier Score (HBOS) and One-Class SVM, when augmented by expert feedback loops, offer the best overall balance between detection power and processing latency. Beyond its immediate contribution to satellite-network validation, this large-scale benchmark supplies a reusable performance reference for the broader anomaly-detection community, illustrating how domain experts can complement unsupervised models to reduce false positives while maintaining high recall.

Taken together, the three strands of the work (empirical vulnerability mapping, automated in the field testing, and human in loop for anomaly analysis) form a coherent pathway toward continuous, verifiable SDN reconfiguration. The study therefore demonstrates that it is both practical and advantageous for critical infrastructure operators to embed machine-learning-driven security checks directly into their deployment pipelines.

6.2 Future Prospects

We have developed automated techniques for identifying misconfiguration scenarios during the reconfiguration of software defined networks. We demonstrated the usability of proposed approach FISTS and also evaluated the scalability. However, there are opportunities for further work to extend our work. The sole purpose of the proposed approach is to passively assist security analyst during the reconfiguration update to determine the presence of vulnerable situations. Although the proposed approach is not an active intrusion detection system to detect and respond to threats in the real time, it could be leveraged to execute along side SDN to continuously verify the security properties of the system. This could be specifically beneficial in dynamic network (networks where flow tables are in reactive state and may change in real-time).

One possible future direction could be the development of proposed approach for other network communication protocols that includes (proprietary comm protocols) but are not limited to Bluetooth, Zigbee, LoRaWAN (Long Range Wide Area Network), CAN (Controller Area Network), Modbus, Profibus (Process Field Bus), BACnet (Building Automation and Control Networks), DNP3 (Distributed Network Protocol), M-Bus (Meter-Bus), GSM and KNX. Other possibilities could be 5G communicaiton protocols i.e., NR (New Radio), NG-RAN (Next Generation Radio Access Network), SDAP (Service Data Adaptation Protocol), PDCP (Packet Data Convergence Protocol), RLC (Radio Link Control), MAC (Medium Access Control), RRC (Radio Resource Control), and NAS (Non-

Access Stratum). Rather than developing the same approach for each protocol individually, instead; a generalized/wrapper method could be developed to facilitate the application of FISTS as is on the target communication protocol.

Further, the proposed FISTS has not been assessed in scenarios where the misconfiguration concerns the deployment of multiple applications such as Network Function Chaining (NFC).

Moreover, FISTS could be extended in the following ways: (1) to identify other security properties in the network such as Access Control and Non-repudiation (2) to automate the recommendation for the resolution of vulnerable state of the network (3) Iteratively self-learn over the period of execution with previous network reconfiguration activities.

Bibliography

[1] Kodheli, O., Lagunas, E., Maturo, N., *et al.*, "Satellite communications in the new space era: A survey and future challenges", *IEEE Communications Surveys & Tutorials*, vol. 23, no. 1, pp. 70–109, 2021. DOI: 10.1109/COMST.2020.3028247.

[2] SES Luxembourg, *SAT-based business-aviation internet services*, https://www.ses.com/find-service/commercial-aviation/business-aviation, Last Accessed: 2023.

[3] SES Luxembourg, *SAT-based maritime internet services*, https://www.ses.com/find-service/commercial-maritime, Last Accessed: 2023.

[4] VIASAT, *SAT-based fast internet*, https://www.viasat.com/satellite-internet/, Last Accessed: 2023.

[5] SES Luxembourg, *SAT-based disaster recovery*, https://www.ses.com/find-service/government/hadr, Last Accessed: 2023.

[6] SES Luxembourg, *SAT-based connection of remote communities*, https://www.ses.com/find-service/telco-mno, Last Accessed: 2023.

[7] Eutelsat, *SAT-based dth broadcasting*, https://www.eutelsat.com/en/satellite-communication-services/broadcasting-solutions.html, Last Accessed: 2023.

[8] Eutelsat, *SAT-based iot connectivity*, https://www.eutelsat.com/en/satellite-communications-services.html, Last Accessed: 2023.

[9] Al-Turjman, F. and Al-Turjman, F., *Edge computing*. Springer, 2019.

[10] Nadeau, T. D. and Gray, K., *SDN: Software Defined Networks: An authoritative review of network programmability technologies*. " O'Reilly Media, Inc.", 2013.

[11] Michel, O. and Keller, E., "Sdn in wide-area networks: A survey", in *2017 Fourth International Conference on Software Defined Systems (SDS)*, Valencia, Spain, 2017, pp. 37–42.

[12] Fu, C., Wang, B., and Wang, W., "Software-defined wide area networks (sd-wans): A survey", *Electronics*, vol. 13, no. 15, 2024, ISSN: 2079-9292. DOI: 10.3390/electronics13153011.

[13] Al Mtawa, Y., Haque, A., and Lutfiyya, H., "Migrating from legacy to software defined networks: A network reliability perspective", *IEEE Transactions on Reliability*, vol. 70, no. 4, pp. 1525–1541, 2021.

[14] Yungaicela-Naula, N. M., Sharma, V., and Scott-Hayward, S., "Misconfiguration in o-ran: Analysis of the impact of ai/ml", *Computer Networks*, p. 110 455, 2024.

[15] Blanco, D. F., Le Mouël, F., Lin, T., and Escudié, M.-P., "A comprehensive survey on software as a service (saas) transformation for the automotive systems", *IEEE Access*, vol. 11, pp. 73 688–73 753, 2023. DOI: 10.1109/ACCESS.2023.3294256.

[16] SAT, *SAT, real name hidden for double blind*, Last Accessed: 2023.

[17] 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.

[18] 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.

[19] SES Luxembourg, *SES broadcasting services*, https://www.ses.com/find-service/broadcasters, Last Accessed: 2022.

[20] 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.

[21] SES Luxembourg, *SES connectivity for commercial maritime*, https://www.ses.com/find-service/commercial-maritime, Last Accessed: 2022.

[22] SES Luxembourg, *SES connectivity for commercial aviation*, https://www.ses.com/find-service/commercial-aviation, Last Accessed: 2022.

[23] KubeEdge, *KubeEdge Edge Computing Framework*, https://kubeedge.io/en/, Last Accessed: 2022.

[24] Yomo Framework, *Yomo*, https://yomo.run/, Last Accessed: 2022.

[25] K3OS, *K3OS Edge Computing Framework*, https://k3os.io/, Last Accessed: 2022.

[26] Mainflux Framework, *Mainflux*, https://mainflux.com/, Last Accessed: 2022.

[27] 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.

[28] Fayad, M. and Schmidt, D. C., "Object-oriented application frameworks", *Commun. ACM*, vol. 40, no. 10, pp. 32–38, Oct. 1997, ISSN: 0001-0782. DOI: 10.1145/262793.262798. [Online]. Available: https://doi.org/10.1145/262793.262798.

[29] Bai, T., Pan, C., Deng, Y., Elkashlan, M., Nallanathan, A., and Hanzo, L., "Latency minimization for intelligent reflecting surface aided mobile edge computing", *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 11, pp. 2666–2682, 2020.

[30] MQTT, https://mqtt.org/, Last Accessed: 2022.

[31] Kubernetes, *Kubernetes pods*, https://kubernetes.io/docs/concepts/workloads/pods/, Last Accessed: 2022.

[32] Hagar, J. D., *IoT System Testing: An IoT Journey from Devices to Analytics and the Edge*. Apress, 2002.

[33] 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.

[34] ARM, *Microcontrollers and infrastructure manufacturer*, https://www.arm.com/, Last Accessed: 2022.

[35] Huawei, http://www.huawei.com, Last Accessed: 2022.

[36] ci4rail, *Computing Intelligence for Rail and Public Transport*, http://www.ci4rail.com, Last Accessed: 2022.

[37] Suse, *Suse software*, https://www.suse.com, Last Accessed: 2022.

[38] Gopalakrishna, N., Anandayuvaraj, D., Detti, A., Bland, F., Rahaman, S., and 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)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 1–8. DOI: 10.1145/3528227.3528565. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3528227.3528565.

[39] KubeEdge, *KubeEdge Development Process*, https://kubeedge-docs.readthedocs.io/en/latest/getting-started/contribute.html, Last Accessed: 2022.

[40] KubeEdge, *KubeEdge Integration Test Suite*, https://github.com/kubeedge/kubeedge/tree/master/tests/integration, Last Accessed: 2022.

[41] KubeEdge, *KubeEdge End-To-End Test Suite*, https://github.com/kubeedge/kubeedge/tree/master/tests/e2e, Last Accessed: 2022.

[42] KubeEdge, *KubeEdge Security Team*, https://github.com/kubeedge/community/tree/master/security-team, Last Accessed: 2022.

[43] KubeEdge, *KubeEdge Sig-Security Team*, https://github.com/kubeedge/community/tree/master/sig-security, Last Accessed: 2022.

[44] Kubernetes, *Kubernetes Security Special Interest Group*, https://github.com/kubernetes/community/tree/master/sig-security, Last Accessed: 2022.

[45] MainFlux, *Mainflux Benchmark*, https://github.com/mainflux/benchmark, Last Accessed: 2022.

[46] MainFlux, *Consulting and Security Audits*, https://mainflux.com/consulting.html, Last Accessed: 2022.

[47] K3OS, *K3OS Automated Test Suite*, https://github.com/rancher/k3os/blob/master/scripts/test, Last Accessed: 2022.

[48] Zetta, *Zetta Automated Test Suite*, https://github.com/zettajs/zetta/tree/master/test, Last Accessed: 2022.

[49] Jin, X., Katsis, C., Sang, F., Sun, J., Kundu, A., and Kompella, R., *Edge security: Challenges and issues*, 2022. DOI: 10.48550/ARXIV.2206.07164. [Online]. Available: https://arxiv.org/abs/2206.07164.

[50] Xiao, Y., Jia, Y., Liu, C., Cheng, X., Yu, J., and Lv, W., "Edge computing security: State of the art and challenges", *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1608–1631, 2019. DOI: 10.1109/JPROC.2019.2918437.

[51] Chen, E. Y., Pei, Y., Chen, S., Tian, Y., Kotcher, R., and Tague, P., "Oauth demystified for mobile application developers", in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 892–903, ISBN: 9781450329576. DOI: 10.1145/2660267.2660323. [Online]. Available: https://doi.org/10.1145/2660267.2660323.

[52] Alwarafy, A., Al-Thelaya, K. A., Abdallah, M., Schneider, J., and Hamdi, M., "A survey on security and privacy issues in edge-computing-assisted internet of things", *IEEE Internet of Things Journal*, vol. 8, no. 6, pp. 4004–4022, 2021. DOI: 10.1109/JIOT.2020.3015432.

[53] Mosenia, A. and Jha, N. K., "A comprehensive study of security of internet-of-things", *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 4, pp. 586–602, 2017. DOI: 10.1109/TETC.2016.2606384.

[54] Gazzola, L., Mariani, L., Pastore, F., and Pezze, M., "An exploratory study of field failures", in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2017, pp. 67–77.

[55] Bondavalli, A. and Simoncini, L., "Failure classification with respect to detection", in *[1990] Proceedings. Second IEEE Workshop on Future Trends of Distributed Computing Systems*, IEEE, 1990, pp. 47–53.

[56] Aysan, H., Punnekkat, S., and Dobrin, R., "Error modeling in dependable component-based systems", in *2008 32nd Annual IEEE International Computer Software and Applications Conference*, IEEE, 2008, pp. 1309–1314.

[57] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C., "Basic concepts and taxonomy of dependable and secure computing", *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[58] Chillarege, R., Bhandari, I. S., Chaar, J. K., *et al.*, "Orthogonal defect classification-a concept for in-process measurements", *IEEE Transactions on software Engineering*, vol. 18, no. 11, pp. 943–956, 1992.

[59] Cinque, M., Cotroneo, D., Kalbarczyk, Z., and 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)*, IEEE, 2007, pp. 585–594.

[60] Dempsey, K., Shah, N., Arnold, C., *et al.*, *NIST Special Publication 800-137 Information Security*, https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-137.pdf, Last Accessed: 2022.

[61] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S., "The oracle problem in software testing: A survey", *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[62] MITRE, *Common Vulnerabilities and Exposures project*, https://cve.mitre.org/cve/, Last Accessed: 2022.

[63] MITRE Corporation, https://www.mitre.org, Last Accessed: 2022.

[64] CVE Numbering Authorities (CNA), https://www.cve.org/ProgramOrganization/CNAs, Last Accessed: 2022.

[65] National Vulnerability Database, https://nvd.nist.gov, Last Accessed: 2022.

[66] Common Vulnerability Scoring System, https://www.first.org/cvss/, Last Accessed: 2022.

[67] Zaman, S., Adams, B., and Hassan, A. E., "Security versus performance bugs: A case study on firefox", in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11, Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 93–102, ISBN: 9781450305747. DOI: 10.1145/1985441.1985457. [Online]. Available: https://doi.org/10.1145/1985441.1985457.

[68] Linares-Vásquez, M., Bavota, G., and Escobar-Velásquez, C., "An empirical study on android-related vulnerabilities", in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 2–13. DOI: 10.1109/MSR.2017.60.

[69] Mazuera-Rozo, A., Bautista-Mora, J., Linares-Vásquez, M., Rueda, S., and Bavota, G., "The android os stack and its vulnerabilities: An empirical study", *Empirical Software Engineering*, vol. 24, no. 4, pp. 2056–2101, 2019.

[70] Cottrell, K., Bose, D. B., Shahriar, H., and Rahman, A., "An empirical study of vulnerabilities in robotics", in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2021, pp. 735–744. DOI: 10.1109/COMPSAC51774.2021.00105.

[71] Blessing, J., Specter, M. A., and 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.

[72] Catolino, G., Palomba, F., Zaidman, A., and Ferrucci, F., "Not all bugs are the same: Understanding, characterizing, and classifying bug types", *Journal of Systems and Software*, vol. 152, pp. 165–181, 2019, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.03.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219300536.

[73] Mozilla foundation, https://www.mozilla.org, Last Accessed: 2022.

[74] Apache foundation, https://www.apache.org/, Last Accessed: 2022.

[75] Linux foundation, https://www.kernel.org/, Last Accessed: 2022.

[76] Austin, A., Holmgreen, C., and Williams, L., "A comparison of the efficiency and effectiveness of vulnerability discovery techniques", *Information and Software Technology*, vol. 55, no. 7, pp. 1279–1288, 2013, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2012.11.007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584912002339.

[77] Zahid, M., Inayat, I., Daneva, M., and Mehmood, Z., "Security risks in cyber physical systems—a systematic mapping study", *Journal of Software: Evolution and Process*, vol. 33, no. 9, e2346, 2021. DOI: https://doi.org/10.1002/smr.2346. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2346. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2346.

[78] Tabrizchi, H. and Kuchaki Rafsanjani, M., "A survey on security challenges in cloud computing: Issues, threats, and solutions", *The journal of supercomputing*, vol. 76, no. 12, pp. 9493–9532, 2020.

[79] Ardagna, C. A., Asal, R., Damiani, E., and Vu, Q. H., "From security to assurance in the cloud: A survey", *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, pp. 1–50, 2015.

[80] Rojas, E., Doriguzzi-Corin, R., Tamurejo, S., *et al.*, "Are we ready to drive software-defined networks? A comprehensive survey on management tools and techniques", *ACM Computing Surveys*, vol. 51, no. 2, 2018, ISSN: 15577341. DOI: 10.1145/3165290.

[81] Saied, W. and Bouhoula, A., "A formal approach for automatic detection and correction of sdn switch misconfigurations", in *2020 16th International Conference on Network and Service Management (CNSM)*, IEEE, Niagara Falls, Canada, 2020, pp. 1–5.

[82] Saâdaoui, A., Ben Youssef Ben Souayeh, N., and Bouhoula, A., "Automated and optimized formal approach to verify sdn access-control misconfigurations", in *Testbeds and Research Infrastructures for the Development of Networks and Communities: 13th EAI International Conference, TridentCom 2018, Shanghai, China, December 1-3, 2018, Proceedings 13*, Springer, 2019, pp. 96–112.

[83] Al-Haj, S. and Tolone, W. J., "Flowtable pipeline misconfigurations in software defined networks", in *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Atlanta, GA, USA, 2017, pp. 247–252. DOI: 10.1109/INFCOMW.2017.8116384.

[84] Pan, H., Li, Z., Zhang, P., Cui, P., Salamatian, K., and Xie, G., "Misconfiguration-free compositional sdn for cloud networks", *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 3, pp. 2484–2499, 2022.

[85] Lebrun, D., Vissicchio, S., and Bonaventure, O., "Towards test-driven software defined networking", in *2014 IEEE Network Operations and Management Symposium (NOMS)*, 2014, pp. 1–9. DOI: 10.1109/NOMS.2014.6838225.

[86] *Versa Newtworks*. Versa Networks, 2024. [Online]. Available: https://versa-networks.com/.

[87] Manès, V. J., Han, H., Han, C., *et al.*, "The art, science, and engineering of fuzzing: A survey", *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[88] Lee, S., Kim, J., Woo, S., *et al.*, "A comprehensive security assessment framework for software-defined networks", *Computers & Security*, vol. 91, p. 101 720, 2020, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2020.101720. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404820300079.

[89] Jero, S., Bu, X., Nita-Rotaru, C., Okhravi, H., Skowyra, R., and Fahmy, S., "Beads: Automated attack discovery in openflow-based sdn systems", *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10453 LNCS, pp. 311–333, 2017. DOI: 10.1007/978-3-319-66332-6_14.

[90] "Fragscapy". (), [Online]. Available: https://github.com/AMOSSYS/Fragscapy.

[91] Pham, V. T., Bohme, M., and Roychoudhury, A., "Aflnet: A greybox fuzzer for network protocols", pp. 460–465, 2020. DOI: 10.1109/ICST46399.2020.00062.

[92] Black, C. and Scott-Hayward, S., *A Survey on the Verification of Adversarial Data Planes in Software-Defined Networks*. Association for Computing Machinery, 2021, vol. 1, pp. 3–10, ISBN: 9781450383189. DOI: 10.1145/3445968.3452092.

[93] "Scapy". (), [Online]. Available: https://scapy.net/.

[94] Zeng, H., Kazemian, P., Varghese, G., and McKeown, N., "Automatic test packet generation", in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, 2012.

[95] Fayaz, S. K., Yu, T., Tobioka, Y., Chaki, S., and Sekar, V., "Buzz: Testing context-dependent policies in stateful networks", in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, 2016, pp. 275–289.

[96] Perešíni, P., Kuźniar, M., and Kostić, D., "Monocle: Dynamic, fine-grained data plane monitoring", in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, 2015, pp. 1–13.

[97] Tseng, F.-H., Chang, K.-D., Liao, S.-C., Chao, H.-C., and Leung, V. C., "Sping: A user-centred debugging mechanism for software defined networks", *IET Networks*, vol. 6, no. 2, pp. 39–46, 2017.

[98] Bu, K., Wen, X., Yang, B., Chen, Y., Li, L. E., and Chen, X., "Is every flow on the right track?: Inspect sdn forwarding with rulescope", in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, IEEE, 2016, pp. 1–9.

[99] Agarwal, K., Rozner, E., Dixon, C., and Carter, J., "Sdn traceroute: Tracing sdn forwarding without changing network behavior", in *Proceedings of the third workshop on Hot topics in software defined networking*, 2014, pp. 145–150.

[100] Wang, Y., Bi, J., and Zhang, K., "A tool for tracing network data plane via sdn/openflow", *Science China Information Sciences*, vol. 60, no. 2, 2016.

[101] Hu, H., Han, W., Ahn, G.-J., and Zhao, Z., "Flowguard: Building robust firewalls for software-defined networks", in *Proceedings of the third workshop on Hot topics in software defined networking*, 2014, pp. 97–102.

[102] Zeng, H., Zhang, S., Ye, F., *et al.*, "Libra: Divide and conquer to verify forwarding tables in huge networks", in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 87–99.

[103] Kazemian, P., Varghese, G., and McKeown, N., "Header space analysis: Static checking for networks", in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, San Jose, CA: USENIX Association, Apr. 2012, pp. 113–126, ISBN: 978-931971-92-8. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian.

[104] Fayaz, S. K. and Sekar, V., "Testing stateful and dynamic data planes with flowtest", in *Proceedings of the third workshop on Hot topics in software defined networking*, 2014, pp. 79–84.

[105] Berkhin, P., "A survey of clustering data mining techniques", in *Grouping Multidimensional Data: Recent Advances in Clustering*, J. Kogan, C. Nicholas, and M. Teboulle, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 25–71, ISBN: 978-3-540-28349-2. DOI: 10.1007/3-540-28349-8_2.

[106] Xu, R. and Wunsch, D., "Survey of clustering algorithms", *IEEE Transactions on Neural Networks*, vol. 16, no. 3, pp. 645–678, 2005. DOI: 10.1109/TNN.2005.845141.

[107] Aggarwal, C. C. and Reddy, C. K., *Data Clustering: Algorithms and Applications*, 1st. Chapman & Hall/CRC, 2013, ISBN: 1466558210.

[108] Xu, D. and Tian, Y., "A comprehensive survey of clustering algorithms", *Annals of Data Science*, vol. 2, no. 2, pp. 165–193, 2015. DOI: 10.1007/s40745-015-0040-1. [Online]. Available: https://doi.org/10.1007/s40745-015-0040-1.

[109] King, R. S., *Cluster Analysis and Data Mining: An Introduction*. USA: Mercury Learning & Information, 2014, ISBN: 1938549384, 9781938549380.

[110] Zhang, T., Ramakrishnan, R., and Livny, M., "Birch: An efficient data clustering method for very large databases", in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '96, Montreal, Quebec, Canada: Association for Computing Machinery, 1996, pp. 103–114, ISBN: 0897917944. DOI: 10.1145/233269.233324. [Online]. Available: https://doi.org/10.1145/233269.233324.

[111] MacQueen, J., "Some methods for classification and analysis of multivariate observations", in *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability - Vol. 1*, L. M. Le Cam and J. Neyman, Eds., University of California Press, Berkeley, CA, USA, 1967, pp. 281–297.

[112] "Partitioning around medoids (program pam)", in *Finding Groups in Data*. John Wiley & Sons, Ltd, 1990, ch. 2, pp. 68–125, ISBN: 9780470316801. DOI: https://doi.org/10.1002/9780470316801.ch2. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470316801.ch2. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470316801.ch2.

[113] Ester, M., Kriegel, H.-P., Sander, J., and Xu, X., "A density-based algorithm for discovering clusters in large spatial databases with noise", in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, ser. KDD'96, Portland, Oregon: AAAI Press, 1996, pp. 226–231.

[114] Ankerst, M., Breunig, M. M., Kriegel, H.-P., and Sander, J., "Optics: Ordering points to identify the clustering structure", in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '99, Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1999, pp. 49–60, ISBN: 1581130848. DOI: 10.1145/304182.304187. [Online]. Available: https://doi.org/10.1145/304182.304187.

[115] Comaniciu, D. and Meer, P., "Mean shift: A robust approach toward feature space analysis", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 5, pp. 603–619, 2002. DOI: 10.1109/34.1000236.

[116] Müllner, D., "Modern hierarchical, agglomerative clustering algorithms", *arXiv preprint arXiv:1109.2378*, 2011.

[117] Rousseeuw, P. J., "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis", *Journal of Computational and Applied Mathematics*, vol. 20, pp. 53–65, 1987, ISSN: 0377-0427. DOI: https://doi.org/10.1016/0377-0427(87)90125-7.

[Online]. Available: https://www.sciencedirect.com/science/article/pii/0377042787901257.

[118] Chandola, V., Banerjee, A., and Kumar, V., "Anomaly detection: A survey", *ACM Comput. Surv.*, vol. 41, no. 3, Jul. 2009, ISSN: 0360-0300. DOI: 10.1145/1541880.1541882. [Online]. Available: https://doi.org/10.1145/1541880.1541882.

[119] Boukerche, A., Zheng, L., and Alfandi, O., "Outlier detection: Methods, models, and classification", *ACM Comput. Surv.*, vol. 53, no. 3, Jun. 2020, ISSN: 0360-0300.

[120] Breunig, M. M., Kriegel, H.-P., Ng, R. T., and Sander, J., "Lof: Identifying density-based local outliers", *SIGMOD Rec.*, vol. 29, no. 2, pp. 93–104, May 2000, ISSN: 0163-5808. DOI: 10.1145/335191.335388. [Online]. Available: https://doi.org/10.1145/335191.335388.

[121] Liu, F. T., Ting, K. M., and Zhou, Z.-H., "Isolation forest", pp. 413–422, 2008. DOI: 10.1109/ICDM.2008.17.

[122] Knorr, E. M. and Ng, R. T., "Algorithms for mining distance-based outliers in large datasets", in *Proceedings of the 24rd International Conference on Very Large Data Bases*, ser. VLDB '98, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 392–403, ISBN: 1558605665.

[123] Ramaswamy, S., Rastogi, R., and Shim, K., "Efficient algorithms for mining outliers from large data sets", *SIGMOD Rec.*, vol. 29, no. 2, pp. 427–438, May 2000, ISSN: 0163-5808. DOI: 10.1145/335191.335437. [Online]. Available: https://doi.org/10.1145/335191.335437.

[124] Goldstein, M. and Dengel, A., "Histogram-based outlier score (hbos): A fast unsupervised anomaly detection algorithm", *KI-2012: poster and demo track*, vol. 1, pp. 59–63, 2012.

[125] Schölkopf, B., Platt, J. C., Shawe-Taylor, J., Smola, A. J., and Williamson, R. C., "Estimating the support of a high-dimensional distribution", *Neural computation*, vol. 13, no. 7, pp. 1443–1471, 2001.

[126] Duan, L., Xu, L., Liu, Y., and Lee, J., "Cluster-based outlier detection", *Annals of Operations Research*, vol. 168, pp. 151–168, 2009.

[127] Tang, J., Chen, Z., Fu, A. W.-C., and Cheung, D. W., "Enhancing effectiveness of outlier detections for low density patterns", in *Advances in Knowledge Discovery and Data Mining: 6th Pacific-Asia Conference, PAKDD 2002 Taipei, Taiwan, May 6–8, 2002 Proceedings 6*, Springer, 2002, pp. 535–548.

[128] Goodge, A., Hooi, B., Ng, S.-K., and Ng, W. S., "Lunar: Unifying local outlier detection methods via graph neural networks", in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, 2022, pp. 6737–6745.

[129] Maćkiewicz, A. and Ratajczak, W., "Principal components analysis (pca)", *Computers & Geosciences*, vol. 19, no. 3, pp. 303–342, 1993.

[130] Hoffmann, H., "Kernel pca for novelty detection", *Pattern recognition*, vol. 40, no. 3, pp. 863–874, 2007.

[131] Wu, X., Xiao, L., Sun, Y., Zhang, J., Ma, T., and He, L., "A survey of human-in-the-loop for machine learning", *Future Generation Computer Systems*, vol. 135, pp. 364–381, 2022.

[132] Chai, C. and Li, G., "Human-in-the-loop techniques in machine learning.", *IEEE Data Eng. Bull.*, vol. 43, no. 3, pp. 37–52, 2020.

[133] Mosqueira-Rey, E., Hernández-Pereira, E., Alonso-Ríos, D., Bobes-Bascarán, J., and Fernández-Leal, Á., "Human-in-the-loop machine learning: A state of the art", *Artificial Intelligence Review*, vol. 56, no. 4, pp. 3005–3054, 2023. DOI: 10.1007/s10462-022-10246-w. [Online]. Available: https://doi.org/10.1007/s10462-022-10246-w.

[134] Tharwat, A. and Schenck, W., "A survey on active learning: State-of-the-art, practical challenges and research directions", *Mathematics*, vol. 11, no. 4820, 2023. DOI: 10.3390/math11040820.

[135] Li, Y. and Guo, L., "An active learning based tcm-knn algorithm for supervised network intrusion detection", *Computers Security*, vol. 26, no. 7, pp. 459–467, 2007, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2007.10.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404807001101.

[136] Malik, J. and Pastore, F., "Field-based security testing of sdn configuration updates", *IEEE Transactions on Reliability*, 2025.

[137] Kim, Y., Dán, G., and Zhu, Q., "Human-in-the-loop cyber intrusion detection using active learning", *IEEE Transactions on Information Forensics and Security*, 2024.

[138] Das, S., Wong, W.-K., Dietterich, T., Fern, A., and Emmott, A., "Incorporating expert feedback into active anomaly discovery", in *2016 IEEE 16th International Conference on Data Mining (ICDM)*, IEEE, 2016, pp. 853–858.

[139] Das, S., Wong, W.-K., Fern, A., Dietterich, T. G., and Siddiqui, M. A., "Incorporating feedback into tree-based anomaly detection", *arXiv preprint arXiv:1708.09441*, 2017.

[140] Lesouple, J. and Tourneret, J.-Y., "Incorporating user feedback into one-class support vector machines for anomaly detection", in *2020 28th European Signal Processing Conference (EUSIPCO)*, IEEE, 2021, pp. 1608–1612.

[141] Görnitz, N., Kloft, M., Rieck, K., and Brefeld, U., "Active learning for network intrusion detection", in *Proceedings of the 2nd ACM workshop on Security and artificial intelligence*, 2009, pp. 47–54.

[142] Yang, K., Ren, J., Zhu, Y., and Zhang, W., "Active learning for wireless iot intrusion detection", *IEEE Wireless Communications*, vol. 25, no. 6, pp. 19–25, 2018. DOI: 10.1109/MWC.2017.1800079.

[143] Lu, Y., Wang, J., Liu, M., *et al.*, "Semi-supervised machine learning aided anomaly detection method in cellular networks", *IEEE Transactions on Vehicular Technology*, vol. 69, no. 8, pp. 8459–8467, 2020. DOI: 10.1109/TVT.2020.2995160.

[144] Jiang, M., Hou, C., Zheng, A., *et al.*, "Weakly supervised anomaly detection: A survey", *arXiv preprint arXiv:2302.04549*, 2023.

[145] Engelen, J. E. van and Hoos, H. H., "A survey on semi-supervised learning", *Machine Learning*, vol. 109, no. 2, pp. 373–440, 2020. DOI: 10.1007/s10994-019-05855-6. [Online]. Available: https://doi.org/10.1007/s10994-019-05855-6.

[146] Triguero, I., García, S., and Herrera, F., "Self-labeled techniques for semi-supervised learning: Taxonomy, software and empirical study", *Knowledge and Information Systems*, vol. 42, no. 2, pp. 245–284, 2015. DOI: 10.1007/s10115-013-0706-y. [Online]. Available: https://doi.org/10.1007/s10115-013-0706-y.

[147] Tanha, J., "A multiclass boosting algorithm to labeled and unlabeled data", *International Journal of Machine Learning and Cybernetics*, vol. 10, no. 12, pp. 3647–3665, 2019. DOI: 10.1007/s13042-019-00951-4. [Online]. Available: https://doi.org/10.1007/s13042-019-00951-4.

[148] Borisov, V., Leemann, T., Seßler, K., Haug, J., Pawelczyk, M., and Kasneci, G., "Deep neural networks and tabular data: A survey", *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 6, pp. 7499–7519, 2024. DOI: 10.1109/TNNLS.2022.3229161.

[149] Jadidi, Z., Muthukkumarasamy, V., Sithirasenan, E., and Singh, K., "Flow-based anomaly detection using semisupervised learning", in *2015 9th International Conference on Signal Processing and Communication Systems (ICSPCS)*, 2015, pp. 1–5. DOI: 10.1109/ICSPCS.2015.7391760.

[150] Frandina, S., Lippi, M., Maggini, M., and Melacci, S., "On – Line Laplacian One – Class Support Vector Machines", pp. 186–193, 2013.

[151] Stankovic, J. A., "Research directions for the internet of things", *IEEE internet of things journal*, vol. 1, no. 1, pp. 3–9, 2014.

[152] MITRE, *Common Weaknesses Enumeration project*, https://cwe.mitre.org, Last Accessed: 2022.

[153] KubeEdge, *KubeEdge GitHub issue tracker*, https://github.com/kubeedge/kubeedge/issues, Last Accessed: 2022.

[154] Zetta., *Zetta Edge Computing Framework*, https://github.com/zettajs/zetta/wiki/Overview, Last Accessed: 2022.

[155] Bertolino, A., Braione, P., De Angelis, G., *et al.*, "A Survey of Field-based Testing Techniques", *ACM Computing Surveys*, vol. 54, no. 5, 2021, ISSN: 15577341. DOI: 10.1145/3447240.

[156] Malik, J., *Replication package*, https://zenodo.org/records/7826981, Last Accessed: 2024. DOI: 10.5281/zenodo.....

[157] Felderer, M., Büchler, M., Johns, M., Brucker, A. D., Breu, R., and Pretschner, A., "Security testing: A survey", in *Advances in Computers*, vol. 101, Elsevier, 2016, pp. 1–51.

[158] Mai, P. X., Goknil, A., Shar, L. K., Pastore, F., Briand, L. C., and Shaame, S., "Modeling security and privacy requirements: A use case-driven approach", *Information and Software Technology*, vol. 100, pp. 165–182, 2018, Available at https://orbilu.uni.lu/handle/10993/35498.

[159] KubeEdge, *KubeEdge Deployment using Keadm*, https://kubeedge.io/en/docs/setup/keadm/, Last Accessed: 2022.

[160] Clound Native Computing Foundation, https://www.cncf.io/, Last Accessed: 2022.

[161] Kubernetes, *Open-source system for automating deployment, scaling, and management of containerized applications*, https://kubernetes.io, Last Accessed: 2022.

[162] Mainflux, *Mainflux*, https://github.com/mainflux/mainflux/issues, Last Accessed: 2022.

[163] Zetta, *Zetta GitHub bug reports*, https://github.com/zettajs/zetta/issues, Last Accessed: 2022.

[164] Rancher, *Rancher container management*, https://rancher.com/, Last Accessed: 2022.

[165] Mosquitto, https://mosquitto.org, Last Accessed: 2022.

[166] VerneMQ Broker, *Vernemq*, https://vernemq.com/, Last Accessed: 2022.

[167] MITRE: CVE-2021-31938, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31938, Last Accessed: 2022.

[168] Microsoft, *Visual Studio Code Kubernetes Tools*, https://marketplace.visualstudio.com/items?itemName=ms-kubernetes-tools.vscode-kubernetes-tools, Last Accessed: 2022.

[169] MITRE: CVE-2021-3499, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3499, Last Accessed: 2022.

[170] MITRE: CVE-2020-8565, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8565, Last Accessed: 2022.

[171] Kubernetes, *Logging in Kubernetes*, https://github.com/kubernetes/community/blob/master/contributors/devel/sig-instrumentation/logging.md, Last Accessed: 2022.

[172] MITRE: CVE-2020-8559, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8559, Last Accessed: 2022.

[173] Ammann, P. and Offutt, J., *Introduction to software testing - 2nd Edition*. Cambridge University Press, 2016.

[174] MITRE: CVE-2021-25737, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-25737, Last Accessed: 2022.

[175] MITRE: CVE-2020-28914, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-28914, Last Accessed: 2022.

[176] MITRE: VE-2021-39159, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-39159, Last Accessed: 2022.

[177] MITRE: CVE-2021-34431, *CVE-2021-34431*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-34431, Last Accessed: 2022.

[178] MITRE: CVE-2019-11252, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11252, Last Accessed: 2022.

[179] MITRE: CVE-2021-28448, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-28448, Last Accessed: 2022.

[180] MITRE: CVE-2020-15157, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15157, Last Accessed: 2022.

[181] MITRE: CVE-2021-32783, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-32783, Last Accessed: 2022.

[182] MITRE: CVE-2021-38545, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-38545, Last Accessed: 2022.

[183] Nassi, B., Pirutin, Y., Galor, T., Elovici, Y., and 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*, ser. CCS '21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 1900–1914, ISBN: 9781450384544. DOI: 10.1145/3460120.3484775. [Online]. Available: https://doi.org/10.1145/3460120.3484775.

[184] Ben Nassi, Yaron Pirutin, Tomer Cohen Galor, Yuval Elovici, and Boris Zadov, https://www.nassiben.com/glowworm-attack, Last Accessed: 2022.

[185] KubeEdge, *KubeEdge Issue 1736*, https://github.com/kubeedge/kubeedge/issues/1736, Last Accessed: 2022.

[186] MITRE: CVE-2021-28166, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-28166, Last Accessed: 2022.

[187] MITRE: CVE-2020-35514, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-35514, Last Accessed: 2022.

[188] MITRE: CVE-2020-8558, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8558, Last Accessed: 2022.

[189] Zetta, *Zetta Issue 335*, https://github.com/zettajs/zetta/issues/335, Last Accessed: 2022.

[190] Mai, P. X., Pastore, F., Goknil, A., and Briand, L. C., "MCP: A security testing tool driven by requirements", in *ICSE'19*, 2019, pp. 55–58. DOI: 10.1109/ICSE-Companion.2019.00037.

[191] MITRE: CVE-2020-8563, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8563, Last Accessed: 2022.

[192] MITRE, https://github.com/kubeedge/kubeedge/issues/2362, Last Accessed: 2022.

[193] KubeEdge, *KubeEdge Issue 1017*, https://github.com/kubeedge/kubeedge/issues/1017, Last Accessed: 2022.

[194] MITRE: CVE-2020-8557, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8557, Last Accessed: 2022.

[195] MITRE: CVE-2020-13597, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-13597, Last Accessed: 2022.

[196] MITRE: CVE-2021-20218, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-20218, Last Accessed: 2022.

[197] Fabric8 Maven Plugin, https://maven.fabric8.io, Last Accessed: 2022.

[198] MITRE: CVE-2014-5278, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-5278, Last Accessed: 2022.

[199] MITRE: CVE-2021-21334, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-21334, Last Accessed: 2022.

[200] MITRE: CVE-2021-21251, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-21251, Last Accessed: 2022.

[201] MITRE: CVE-2020-2211, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-2211, Last Accessed: 2022.

[202] MITRE: CVE-2020-8566, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8566, Last Accessed: 2022.

[203] Kube-score, *Static code analysis for kubernetes object definitions.* https://kube-score.com/, Last Accessed: 2022.

[204] 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.

[205] Metasploit, *Metasploit edge computing framework*, https://www.metasploit.com", Last Accessed: 2022.

[206] 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=EAIaIQobChMIjbm99ZTBwE, Last Accessed: 2022.

[207] Shabtai, A., Elovici, Y., and Rokach, L., *A Survey of Data Leakage Detection and Prevention Solutions*. Springer Publishing Company, Incorporated, 2012, ISBN: 1461420520.

[208] SonarQube, https://www.sonarqube.org/, Last Accessed: 2022.

[209] Google, *Go lang*, https://go.dev, Last Accessed: 2022.

[210] Sonarsource, *Sonarsource tools for GO*, "https://rules.sonarsource.com/go", Last Accessed: 2022.

[211] Analysis Tools team, *Static analysis tools for GO*, https://analysis-tools.dev/tag/go, Last Accessed: 2022.

[212] Honnef, D., *Staticcheck: Static analysis tool for the go programming language*, "https://staticcheck.io/", Last Accessed: 2022.

[213] Cloud Native Computing Foundation, https://github.com/containerd/containerd, Last Accessed: 2023.

[214] Fisher, R. A., "On the interpretation of x2 from contingency tables, and the calculation of p", *Journal of the Royal Statistical Society*, vol. 85, no. 1, pp. 87–94, 1922, ISSN: 09528385. [Online]. Available: http://www.jstor.org/stable/2340521 (visited on 11/14/2022).

[215] Kubernetes, *Test Report on KubeEdge's Support for 100,000 Edge Nodes*, https://kubeedge.io/en/blog/scalability-test-report/, Last Accessed: 2022.

[216] Zetta Edge framework examples, https://www.zettajs.org/projects/, Last Accessed: 2022.

[217] KubeEdge Edge framework examples, *KubeEdge*, https://kubeedge.io/en/docs/developer/device_crd/, Last Accessed: 2022.

[218] Dai, H., Murphy, C., and Kaiser, G., "Configuration fuzzing for software vulnerability detection", in *2010 International Conference on Availability, Reliability and Security*, IEEE, 2010, pp. 525–530.

[219] Dai, H., Murphy, C., and Kaiser, G. E., "Confu: Configuration fuzzing testing framework for software vulnerability detection", in *Security-Aware Systems Applications and Software Development Methods*, IGI Global, 2012, pp. 152–167.

[220] Bertolino, A., Angelis, G. D., Frantzen, L., and Polini, A., "The plastic framework and tools for testing service-oriented applications", in *Software Engineering*, Springer, 2007, pp. 106–139.

[221] Bertolino, A., De Angelis, G., Kellomaki, S., and Polini, A., "Enhancing service federation trustworthiness through online testing", *Computer*, vol. 45, no. 1, pp. 66–72, 2011.

[222] De Angelis, G., Bertolino, A., and Polini, A., "(role) cast: A framework for on-line service testing", in *International Conference on Web Information Systems and Technologies*, SCITEPRESS, vol. 2, 2011, pp. 13–18.

[223] Hui, Z.-W., Huang, S., and Ji, M.-Y., "A runtime-testing method for integer overflow detection based on metamorphic relations", *Journal of Intelligent & Fuzzy Systems*, vol. 31, no. 4, pp. 2349–2361, 2016.

[224] Zhang, J., "An approach to facilitate reliability testing of web services components", in *15th International Symposium on Software Reliability Engineering*, IEEE, 2004, pp. 210–218.

[225] Manes, V. J., Han, H., Han, C., *et al.*, "The Art, Science, and Engineering of Fuzzing: A Survey", *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021, ISSN: 19393520. DOI: 10.1109/TSE.2019.2946563. arXiv: 1812.00140.

[226] Zalewski, M., *American Fuzzy Lop: a security- oriented fuzzer*, 2020. [Online]. Available: http://lcamtuf.coredump.cx/afl/.

[227] Koziol, J., *Charlie Miller Reveals His Process for Security Research*, 2010. [Online]. Available: https://resources.infosecinstitute.com/topic/how-charlie-miller-does-research/.

[228] Fink, G. and Bishop, M., "Property-based testing: A new approach to testing for assurance", *SIGSOFT Softw. Eng. Notes*, vol. 22, no. 4, pp. 74–80, Jul. 1997, ISSN: 0163-5948. DOI: 10.1145/263244.263267. [Online]. Available: https://doi.org/10.1145/263244.263267.

[229] Chaleshtari, N. B., Pastore, F., Goknil, A., and Briand, L. C., "Metamorphic testing for web system security", *IEEE Transactions on Software Engineering*, 2023, Accepted, available at https://arxiv.org/abs/2208.09505.

[230] Elder, S., Zahan, N., Shu, R., *et al.*, "Do I really need all this work to find vulnerabilities?", *Empirical Software Engineering*, vol. 27, no. 6, p. 154, 2022, ISSN: 1573-7616. DOI: 10.1007/s10664-022-10179-6. [Online]. Available: https://doi.org/10.1007/s10664-022-10179-6.

[231] OWASP, *OWASp Top Ten*, https://owasp.org/www-project-top-ten/, Last Accessed: 2022.

[232] Lyon, G. F., *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Sunnyvale, CA, USA: Insecure, 2009, Available at http://nmap.org, ISBN: 0979958717.

[233] Liu Fei Tony Ting, K. M. and Zhi-Hua, Z., "Isolation-based anomaly detection", *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 2012.

[234] Salvatore Sanfilippo, *hping*, https://github.com/antirez/hping, Last Accessed: 2024.

[235] NMAP team, *NMAP: Bypassing Firewall Rules*, https://nmap.org/book/firewall-subversion.html, Last Accessed: 2024.

[236] Burkov, A., *Machine Learning Engineering*. True Positive Incorporated, 2020, ISBN: 9781999579579. [Online]. Available: https://books.google.lu/books?id=HeXizQEACAAJ.

[237] Mann, H. B. and Whitney, D. R., "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other", *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947. DOI: 10.1214/aoms/1177730491. [Online]. Available: https://doi.org/10.1214/aoms/1177730491.

[238] Malik, J., *Replication package*, https://zenodo.org/records/10535145, Last Accessed: 2024. DOI: 10.5281/zenodo....

[239] Shodan. "Shodan: Search engine for internet-connected devices". Accessed: 2025-03-17. (2025), [Online]. Available: https://www.shodan.io.

[240] Malik, J., *Replication package*, https://zenodo.org/records/15188593 (full link shortened), Last Accessed: 2025. DOI: 10.5281/zenodo.15188593.