PhD-FSTM-2025-003
Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 03/02/2025 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG EN INFORMATIQUE

by

Raphaël Charles-Victor Ollando
Born on 2nd of December 1996 in Nancy (France)

AI-ENABLED TEST DATA AND SCHEDULE GENERATION METHODS FOR COMPLEX NETWORK SYSTEMS

DISSERTATION DEFENCE COMMITTEE

Dr. Seung Yeob SHIN, Dissertation Supervisor
Research Scientist, University of Luxembourg, Luxembourg

Prof. Dr. Fabrizio PASTORE, Chairman
Professor, University of Luxembourg, Luxembourg

Dr. Thang Xuan VU, Vice Chairman
Research Scientist, University of Luxembourg, Luxembourg

Prof Dr. Lionel BRIAND, Member
Professor, University of Ottawa, Canada, University of Limerick, Ireland

Prof Dr. Roberto NATELLA, Member
Professor, Università Degli Studi di Napoli Federico II, Italy

Acknowledgement

First and foremost, I would like to thank Dr. Seung Yeob Shin, not only for his supervision and guidance, but also for his endless optimism, support, and encouragement. I don't have the words to tell you how grateful and happy I am to have been under your supervision. You've been there for me through all these stages of both my life and my doctorate. Thanks for always being kind to me and giving me the support and freedom that I needed.

I would like to thank Prof. Dr. Lionel Briand for his everlasting commitment to my academic development. You have been a great inspiration to me, and without your dedicated feedback, I might not have succeeded on this career path. I am truly grateful to have been one of your students.

My deep gratitude also goes to Dr. Joël Grotz, for being so committed as my industry advisor at SES. Your insights have been very valuable to me and have shaped my development as a researcher.

My deep gratitude also goes to Dr. Siridopoulos Nikolas and Dr. Minardi Mario, for allowing me to spend almost a year as a research intern at SES Techcom. Your feedback has been invaluable to me and I could not have succeeded without your unconditional support and friendship. Thanks to you I had some really great moments at SES.

I am also very grateful to the remaining members of my defence committee, Prof. Dr. Fabrizio Pastore, Dr. Thang Xuan Vu, and Prof. Dr. Roberto Natella. Thank you for the valuable time and effort that you have spent reviewing my dissertation.

Another person for whom I am special thanks is Dr. Frank Zimmer. Long before embarking on this path, I had the opportunity to be a master student on your team. You gave me this curiosity for research and encouraged me to pursue this path. My life would have been quite different if I had not met you. And I will forever be grateful for this.

This dissertation would not have been possible without the unconditional support of my dear friends. To Andres, Asia, Alexandre, Chloé, Clément, Jose Mario, Johanna, Jun, Mario, Prezmek, Romane, Robin, Shuto, Vincent, and Vini — you have stood by me through every challenge, every high and low, and every twist and turn of this journey. I know you will also be there for the ones to come. I cannot thank you enough for everything. Thank you for being such dear and important friends to me. To Ahmed, Ali, Alireza, Alejandro, Angelo, Carla, Cristina, Claudio, Enrico, Gianluca, Jose Andres, Leonardo, Marcele, Marcello, Nyyti, Tahmineh, Tobias, and Yoann — thank you for the incredible friendship, the unforgettable adventures, the fun times,

the nights and days spent rethinking the world, the fights, the cuddles, the laughter, and the cries. I am deeply grateful to have met you during my time at the university and to have formed lifelong bonds with you. Thank you so much!
To all my colleagues and superiors at SVV, Prof. Dr. Bianculli, Prof. Dr. Pastore, Fitash, Jahanzaib, Sallam, Nicolas, Orlando, and Oualid, to name a few. Thank you all for your invaluable advice, the fun moments, and the engaging discussions we had those last 4 years, and all the fun we had at conferences or summer schools. It was a great time with you all!
Finally, I thank the Luxembourg National Research Fund (FNR) and SES for their financial support.

Last but not least, I would like to thank my family, especially my mom, dad, little sister, and little brother, for their endless support, care, encouragement, help in fulfilling my dreams and invitation to stay strong in facing all adversities. Thank you for always being there for me, for loving me and for being so patient with me. However, the people I need to thank the most are my Grandma and Grandpa. You gave up so much, lost so many dreams and hopes and faced so many hardships to give us all, as a family, the opportunity to have a life we could be happy in. As the first grandson, I am immensely proud to be standing there and to be so close to getting the title of doctor. Just like you Grandpa. This is why I dedicate this dissertation — and, in my hope, achievement — to you two. Wasígyâ kutsibutsibu, Múkakâ , Sokulu.

And finally, to each and every one of you, thank you for the wonderful time! That was epic!

Raphaël Ollando
University of Luxembourg
February 2025

Abstract

In our interconnected world, complex network systems are foundational to critical applications like telecommunications, Software-Defined Networking (SDN), and satellite systems. These network systems, with their intricate interdependencies and dynamic interactions, require rigorous testing to ensure reliability, performance, and compliance with industry standards. However, testing complex network systems entails significant challenges arising from their large scale, heterogeneity, and dynamic nature.

Effective testing strategies for complex network systems must generate realistic test data, simulate traffic patterns, and introduce controlled faults to evaluate their fault tolerance and recovery mechanisms. Additionally, efficient test scheduling and resource management are crucial for comprehensive coverage and optimal resource utilization. Hence, in this dissertation, we introduce AI-enabled methods for generating test data and scheduling tests in complex network systems, specifically focusing on SDNs and satellite systems.

In Chapter 3, we present FuzzSDN, a machine learning-guided fuzzing method for testing SDN controllers. FuzzSDN efficiently explores the test input space, generating test data that leads to system failures and learning failure-inducing models. FuzzSDN has shown a significant increase in fault detection rates and improved the diagnosis of system failures by providing interpretable models of failure-inducing conditions.

In Chapter 4, we present SeqFuzzSDN, a learning-guided fuzzing method for testing stateful SDN controllers. SeqFuzzSDN leverages the architecture and protocols of SDNs to test controllers in realistic operational settings. It employs Extended Finite State Machines (EFSMs) to guide its fuzzing step, thus resulting in effective and diverse tests that discover failures. Our results demonstrate that SeqFuzzSDN generates more diverse message sequences leading to failures within the same time budget and produces more accurate failure-inducing models, significantly outperforming other methods in terms of sensitivity.

Finally, we present in Chapter 5 a multi-objective approach for scheduling acceptance tests for mission-critical satellite systems. Using the Non-dominated Sorting Genetic Algorithm III (NSGA-III), this approach finds near-optimal feasible schedules that balance operational cost,

fragmentation, and resource efficiency. This method has improved the overall efficiency of test campaigns, reducing costs and ensuring thorough testing of satellite systems, while allowing engineers to perform trade-off analyses.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

In today's interconnected world, complex network systems are ubiquitous, underpinning a vast array of critical applications across various domains [1, 2, 3, 4]. These systems encompass various networks, including telecommunications, Software-Defined Networking (SDN), and satellite systems. Each of these networks is characterized by intricate interdependencies and dynamic interactions among numerous components [5, 3, 1, 2, 4].

Consequently, testing complex network systems has become increasingly important. These networks form the backbone of critical infrastructure, including telecommunications, data centers, and satellite systems. Ensuring their reliability, correctness, and performance is essential to prevent disruptions that could have widespread and severe consequences. Effective testing helps identify vulnerabilities, optimize performance, meet requirements, and ensure industry standards and regulations compliance.

However, testing complex network systems presents significant challenges due to their inherent complexity and dynamic nature. Firstly, the scale and heterogeneity of these networks, which often consist of numerous interconnected components with diverse technologies and protocols, pose significant difficulties [6, 4]. Secondly, the dynamic behaviour of network systems can change rapidly due to varying traffic patterns, user demands, and environmental conditions, making it challenging to capture and replicate these behaviours in tests [4]. Additionally, the interdependencies within a network mean that a failure in one part of the network can cascade and affect other parts of the network, necessitating tests that accurately assess system resilience [7, 8]. Finally, addressing hidden bugs and unexpected behaviours in the network is

crucial for maintaining system reliability, necessitating comprehensive testing strategies that account for the complexities of network environments. [4, 9].

Given these challenges, generating realistic and comprehensive test data becomes essential for effective testing of complex network systems. Test data must accurately reflect the conditions and scenarios that the network will encounter in real-world operations. This includes simulating realistic traffic patterns to assess the network's performance under various load conditions, introducing faults and failures in a controlled manner to test the network's fault tolerance and recovery mechanisms, and generating test data that uncover faults and evaluate overall system reliability [10, 11, 12, 13, 9].

Finally, an effective schedule of tests to be conducted is critical to ensure comprehensive coverage and efficient use of resources. This involves identifying and prioritizing the most critical components and scenarios for testing based on their impact on overall network performance and reliability. Leveraging automation tools to execute tests systematically and consistently can reduce the potential for human error and increase efficiency. Moreover, implementing continuous testing practices allows for regular assessment of the network's performance and reliability, especially after updates or changes. Efficiently managing testing resources, including hardware, software, and personnel, ensures that testing is conducted without unnecessary delays or costs.

Finally, the efficient organization of test execution is critical to ensure comprehensive coverage and efficient use of resources. This involves identifying and prioritizing the most critical components and scenarios for testing based on their impact on overall network performance and reliability [12, 11, 14]. Leveraging automation tools to plan the execution tests systematically and consistently can reduce the potential for human error and increase efficiency in managing testing resources, including hardware, software, and personnel, and ensuring that testing is conducted effectively without unnecessary delays or costs [12, 11].

1.2 Research Contributions

In this dissertation, we present AI-enabled methods for generating test data and scheduling tests in complex network systems, specifically focusing on the cases of Software-Defined Networks (SDNs) and Satellite Systems.

We first propose FuzzSDN, a machine learning-guided Fuzzing method for testing SDN-systems. In particular, FuzzSDN targets software controllers deployed in SDN-systems. FuzzSDN relies on fuzzing guided by machine learning (ML) to both (1) efficiently explore the test input space of an SDN-system's controller (generate test data leading to system failures) and (2) learn failure-inducing models that characterize input conditions under which the system fails. This is done in a synergistic manner where models guide test generation and the latter also aims at improving the models. A failure-inducing model is practically useful [15] for the following reasons: (1) It

facilitates the diagnosis of system failures. FuzzSDN provides engineers with an interpretable model specifying how likely are failures to occur, e.g., the system fails when a control message is encoded using OpenFlow V1.0 and contains IP packets, thus providing concrete conditions under which a system will probably fail. Such conditions are much easier to analyze than a large set of individual failures. (2) A failure-inducing model enables engineers to validate their fixes. Engineers can fix and test their code against the generated test data set. A failure-inducing model can also be used as a test data generator to reproduce the system failures captured in the model. Hence, engineers can better validate their fixes using an extended test data set. This contribution is presented in Chapter 3

We then propose SeqFuzzSDN, a learning-guided fuzzing method for testing stateful SDN controllers. SeqFuzzSDN leverages the architecture and protocols of SDNs. SeqFuzzSDN tests SDN controllers in a realistic operational setting without requiring any compile-time instrumentation, manual annotation of source code, and replacing an SDN switch with a fuzzer. Instead, SeqFuzzSDN sniffs and fuzzes control messages exchanged between the SDN controller and switches by being aware of the stateful behaviours of the controller. SeqFuzzSDN employs a fuzzing strategy guided by Extended Finite State machines (EFSMs) in order to efficiently explore the space of states of the SDN controller under test; generate effective and diverse tests (i.e., message sequences) to uncover failures; and infer accurate EFSMs that characterize the sequences of control messages leading to failures. Note that since the SDN communication protocol specifies various message fields, their values, and relations, guard conditions on state transitions in EFSMs are well-suited to capture state changes associated with these message fields, values, and relations. This contribution is presented in Chapter 4

Finally, we propose a multi-objective approach for scheduling acceptance tests for mission-critical satellite systems. Our approach includes a precise definition of the problem of scheduling the Satellite In-Orbit Testing (IOT) campaign, which accounts for schedule constraints; an algorithm based on Non-dominated Sorting Genetic Algorithm III (NSGA-III [16]) for finding near-optimal feasible IOT schedules; and fitness functions that evaluate the performance of IOT schedules by assessing their operational cost, fragmentation (as fragmented IOT schedules incur overheads), and efficiency in the use of test resources. Furthermore, we applied our approach to a representative Global Navigation Satellite System (GNSS), for which SES, our industrial partner, provides operational services. Our results show that an IOT campaign scheduled using our search-based approach, compared to a random search approach, finds feasible schedules that achieve an average improvement of 49.4% in the cost fitness, 60.4% in the fragmentation fitness, and 30% in efficiency of the test resource usage fitness. In addition, our approach demonstrates that, compared to Ant-Colony Optimization (ACO) approaches, which have been extensively cited in the literature. Specifically, our method outperforms a tailored baseline approach by 53.1% in cost efficiency, 58.3% in fragmentation, and 26.1% in efficiency of resource usage over the same period. Moreover, it provides practitioners with several equally viable schedules, enabling comprehensive trade-off analyses. Finally, our approach yields schedules that improve the cost efficiency by 538%, and the efficiency of the test resource usage by 39.42% compared to schedules

manually constructed by practitioners, while maintaining comparable performance in terms of fragmentation and requiring only 12.5% of the time needed by practitioners to construct an IOT schedule. Finally, we interviewed practitioners at SES to collect feedback on our approach. They highlighted the efficiency of schedule generation, as our automated approach generates feasible schedules much faster than manual methods, enabling quick adaptation to changing conditions, and the ability to produce several equally viable schedules, facilitating trade-off analysis. This contribution is presented in Chapter 5

1.3 Dissertation Outline

Chapter 2 describes the study systems, including Software-Defined Networks and In-Orbit Testing, and the technical background necessary to understand the work carried out in this dissertation

Chapter 3 introduces the background and defines the specific problem of learning failure-inducing models for testing SDN-systems. It first describes FuzzSDN, its evaluation in a large empirical study, and compares FuzzSDN with related works.

Chapter 4 introduce the background and defines the specific problem of learning-guided fuzzing for testing stateful SDN controllers. It first describes SeqFuzzSDN, its empirical evaluation study, and compares SeqFuzzSDN with related works.

Chapter 5 introduces the background and defines the specific problem of scheduling acceptance tests for mission-critical satellite systems. It first describes our IOT scheduling method, its evaluation in an empirical study, and describes the feedback gathered from practitioners.

Chapter 6 summarizes the dissertation contributions and discusses perspectives on future works.

Chapter 2

Background

This chapter introduces the study systems and technical background necessary to understand the work carried out in this dissertation. Section 2.1 describes our study subjects: Software-Defined Networks and Satellite In-Orbit Testing. Section 2.2 describes the core techniques used throughout this dissertation: Software Testing, Fuzzing, Supervised Machine Learning and Evolutionary algorithms.

2.1 Study subjects

2.1.1 Software-Defined Networks

Software-defined networks (SDN) [17] have emerged to enable programmable networks that allow system operators to manage their systems in a flexible and efficient way. SDNs have been widely deployed in many application domains, such as data centers [18, 19], the Internet of Things [20, 21], and satellite communications [22, 23]. The main idea behind SDNs is to transfer the control of networks from localized, fixed-behavior controllers distributed over a set of network switches (in traditional networks) to a logically centralized and programmable software controller. With complex software being an integral part of SDNs, developing SDN-based systems (SDN-systems), e.g., data centers, entails interdisciplinary considerations, including software engineering.

Architecture. The SDN architecture [17] separates the network into the control plane and the data plane, which is a key distinction from traditional networks that do not possess this

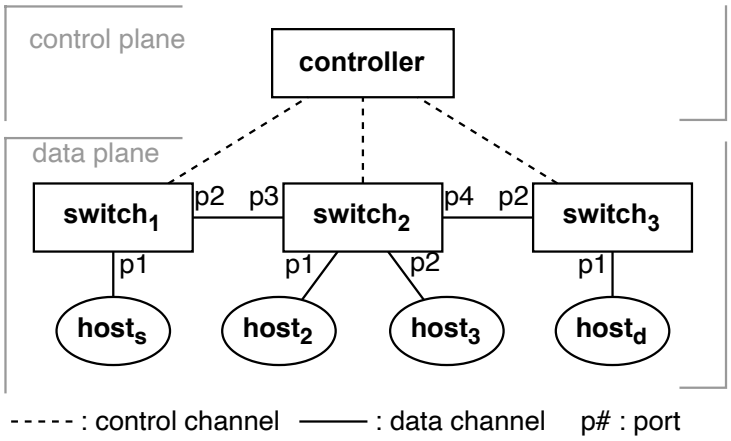


Figure 2.1: An example of an SDN topology.

separation. In the control plane, an SDN controller provides network administrators with a global view of the network, enabling centralised control over network operations. The centralised control allows administrators to optimally manage network resources and to effectively enforce network policies. Furthermore, an SDN controller provides engineers with APIs [17], enabling them to develop and install custom applications on the controller to address system-specific needs (e.g., dynamic adaptive traffic control application [24]). The data plane consists of SDN switches, which are responsible for forwarding data messages (i.e., data packets) based on the instructions provided by the controller. SDN switches are connected to hosts that generate and receive data messages. Such hosts can be servers, clients, and IoT devices in a networked system. In the SDN architecture, the SDN controller serves as the central software component that enables the provision of flexible and efficient network services.

Figure 2.1 depicts an SDN topology example that consists of a controller, three switches, and four hosts. The controller communicates with the three switches via control channels that carry control messages (e.g., OpenFlow messages [25]). The switches and hosts, on the other hand, are connected via data channels that carry data messages encapsulated by standard network protocols such as ARP [26, 27] and IP [28, 27].

Message. The SDN controller and switches in a network communicate by exchanging control messages. A control message is encoded as a sequence of values following a specific communication protocol. For instance, OpenFlow [25] is a de facto standard communications protocol used in many SDN-systems [29], enabling communication between the control and infrastructure layers. To exercise the behavior of the controller under test, therefore, testing explores the space of possible control messages.

In traditional networking systems, communication between devices usually employs well-established transport protocols, such as UDP [30] and TCP [31], for data transmission. For exchanging routing and state information, routers and switches use standard routing protocols, such as OSPF [32] and BGP [33]. Since these protocols were designed to serve specific purposes in the context of traditional static networks, the messages they encode are more similar and simpler

than those used in SDN. In contrast, SDN messages, such as those in OpenFlow, carry a variety of data including flow setups, modifications, and statistics, allowing for more flexibility in network services. However, this also entails a wider range of potential failures that should be accounted for during testing.

Failure. Like other software components, SDN controllers may have faults that can lead to service failures perceivable by users. These failures can manifest in various forms in the context of SDN-systems. Specifically, previous studies on SDN testing [34, 35, 36, 37, 38] have investigated the following failures relevant to SDN controllers: (1) Controller-switch disconnection. When communications between the controller and the switches are unexpectedly disconnected, the system obviously cannot operate as intended. For example, if a switch does not receive timely commands from the controller due to this disconnection, the switch relies on outdated forwarding rules installed earlier, possibly leading to dropping data flows entirely. (2) SDN operation stall. This failure refers to situations where the required execution of an SDN operation is unexpectedly prevented or delayed. For example, an SDN controller might stall the installation of forwarding rules, affecting the construction of the communication path in the SDN. (3) Incorrect understanding of network status. SDN controllers rely on a centralized view of the network to make control decisions. If this centralized view becomes inconsistent with the actual network, regardless of the reasons, e.g., receiving outdated or incorrect data from the network switches, the controller may fail to make decisions that reflect the network's actual conditions. For example, if a link between two switches is broken, but the controller's view still considers it operational, the controller fails to instruct the switches to reroute data flows around the broken link, resulting in dropped data flows. (4) Overutilization of resources. SDN controllers may overutilize their processing and memory units due to various reasons, such as handling an unexpectedly high volume of requests. For example, such a failure caused by a distributed denial-of-service (DDoS) attack [39] targeting the SDN controller can lead to slower response times and potential service outages. We note that, in addition to failures specific to SDN controllers, failures observed in traditional networks, such as communication breakdowns among hosts (e.g., data servers and clients) and network performance degradation, are also relevant in SDN-systems as its infrastructure layer usually employs traditional network protocols, such as TCP, UDP, and IP.

In traditional networking systems, failures tend to be localized. If a router or switch fails, only its directly connected neighbours (i.e., localized segments of the network) are affected. Similarly, if a server or client fails, only the applications and users directly reliant on that specific device are impacted. In contrast, SDN-systems have a centralized failure point in the controller. If the controller crashes or loses connection with the switches, the entire network can be affected. This central point of potential failure makes rigorous testing essential to ensure robustness and reliability.

2.1.2 In-Orbit Testing

We motivate our work using a case study from our industry partner, SES Techcom, which develops satellite-enabled solutions. Operators of satellites are tasked with ensuring optimal performance of their satellites' services once deployed in orbit. Given the critical role of satellite technology in supporting various services, such as broadcast television, global navigation and positioning systems, mobile communications, and other communication systems, operators must ensure that, over the lifespan of a satellite, the Quality-of-Service (QoS) remains within the standards defined by its application. Consequently, operators routinely conduct *In-Orbit Testing* (IOT) procedures to monitor the QoS of each satellite in the constellation they operate. These IOT procedures have four main objectives: (1) ensuring the behavior of the satellite remains consistent before and after launch; (2) verifying performance adherence to specifications, (3) forecasting end-of-life; and (4) investigating potential anomalies.

SES Techcom conducts routine monthly tests for the European GNSS constellation, Galileo. In this context, *IOT* procedures are divided into two categories: *Signal Quality Monitoring* (SQM) and *Routine In-Orbit Test* (RIOT). SQM procedures measure the satellite's signal quality and strength on each communication channel. Specifically, these procedures involve measuring the Modulated Effective Isotropic Irradiated Power (EIRP) [40, 41] approximately 15 times on each channel, with the overall testing duration lasting almost one hour per satellite. The SQM procedures are usually performed at the highest elevation available at any given pass of a satellite. RIOT procedures are performed sequentially throughout the full pass of the satellite, from the signal acquisition, typically around 3-5 degrees of elevation, until signal loss at a similar elevation. The duration of an RIOT phase ranges from 8 to 9 hours, depending on the satellite and the ground measuring station. During an RIOT phase, several IOT measurements are performed for every Galileo channel. Specifically, these IOT measurements include Modulated EIRP, IQ sample collection, out-of-band spurious measurement, navigation receiver data analysis, and Search-and-Rescue (SAR) check, if SAR is available [41].

Additionally, the antennas used to communicate with the satellite are large objects that require time to be precisely pointed toward the satellite under test. Due to the precise nature of satellite communication, test instruments and antenna alignment may need to be re-calibrated before conducting each test procedure. These factors introduce delays before conducting each test procedure in an IOT campaign, during which no tests can be performed, and must be taken into consideration in the scheduling process.

Currently, practitioners at SES Techcom manually schedule these *IOT* procedures, having determined that existing automated solutions are not practically applicable to their scheduling needs. However, this manual approach poses significant challenges and consumes valuable time for practitioners, particularly when the satellites' orbits have short revolution periods or substantial inclinations. Moreover, in the event of an emergency scenario, such as an unexpected degradation in QoS across the constellation, an IOT campaign must be scheduled and executed within a

condensed timeframe. With the Galileo constellation currently consisting of 23 satellites in orbit (soon to be 25), this presents a considerable challenge for the IOT operators. Hence, an algorithm that automatically solves the problem of scheduling IOT campaigns in practical time is highly desirable.

2.2 Technical Background

2.2.1 Fuzz Testing

Fuzzing is a technique that consists in providing invalid, unexpected, or random data as inputs to a computer program. The goal is to find security vulnerabilities and bugs by observing how the program behaves under these conditions. Since its early years, fuzzing has been extensively deployed to test software systems, such as network servers, shell applications, OS kernels, or libraries. It is particularly effective in uncovering memory corruption, crashes, and other critical issues that could be exploited by attackers.

Among the various research strands in fuzzing, some are particularly relevant to the work presented in this dissertation. Below, we discuss these studies and their contributions to improving fuzzing techniques.

The first research strand closely related to this dissertation's work is coverage-aware fuzzing techniques, which have been used in many studies across different domains [42, 43, 44]. For example, AFL [42] is a mutational, coverage-guided fuzzer that uses compile-time instrumentation and genetic algorithms to automatically generate test cases that uncover previously unexplored internal states in the program under test. AFL++ [43] is a fuzzing framework that expands on AFL, incorporating many state-of-the-art techniques that enhance fuzzing performance, thus enabling researchers to evaluate different combinations of such methods. However, applying such fuzzing tools, developed in other contexts, to test SDN controllers is far from being straightforward as there are differences in inputs, outputs, and states. For example, the notion of coverage, e.g., statements and branches, used in coverage-aware fuzzing tools is suitable for testing stateless programs, where outputs depend solely on inputs and do not depend on any memory of past interactions. However, an SDN controller is a stateful program. It takes as input sequences of control messages from the switches in the network, processes these messages, and outputs appropriate sequences of responses. Note that the controller's response is determined by the currently received message and its internal state, which is determined by previously processed messages. AFLNet [44], which extends AFL to test servers, is proposed to address this issue by utilizing state coverage rooted in finite state machines. In AFLNet, finite state machines are constructed using the response codes (also known as status codes) from network protocols, which indicate the result of a client's request to a server. However, AFLNet is not applicable when such response codes are not available, as in our context. Hence, we need a different notion of coverage

when testing SDN controllers. In addition to the coverage issue, existing techniques that enhance fuzzing performance for single programs are not easily applicable to testing SDN controllers. For example, the forkserver technique implemented in AFL++, which uses the fork mechanism to reduce the high cost of initialization, is not applicable to our context. Since an SDN controller interacts with multiple switches connected to various hosts, testing the controller requires the costly initialization of not only the controller but also the other components in the SDN-system. Further, testing the controller impacts the states of these other components, making it difficult to efficiently reset and maintain a consistent testing environment. While coverage-aware fuzzing tools, such as AFL++, provide significant advances in testing stateless programs, applying them to test SDN controllers therefore raises difficult challenges.

Another research strands that closely relate to the work present in this dissertation are stateful fuzzing techniques [45, 46, 47, 48] Numerous research studies have explored the use of Finite State Machines (FSMs) and Extended Finite State Machine (EFSMs) for testing complex systems. Gascon et al. [46] proposed Pulsar, a stateful black-box fuzzing technique aimed at discovering vulnerabilities in proprietary network protocols. Their proposed approach involves the inference of a Markov model (Deterministic Finite Automaton) from network traces, which are used to generate test cases using fuzzing primitives (i.e., paths in the automaton) defined by the model, and finally the selection of the test cases that maximise the coverage of the protocol stack. Pham et al. [47] proposed AFLNet, a grey-box fuzzer for network protocols implementation, based on AFL [42]. Their proposed technique takes a mutational approach and states feedback to guide the fuzzing of network-enabled servers. As explicated previously, AFLNet takes as input a corpus of server-client network and subsequently acts as a client. It replays modified versions of the initial message sequence sent to the server, preserving only the alterations that successfully expanded the coverage of the code or state space. From the newly discovered message sequences, AFLNet uses the server's response codes to build an FSM that describes the protocol states. From those inferred FSMs, their approach identifies regions in the state space that have been the least explored and systematically steers the fuzzing process towards the test of such regions. Natella [48] proposed StateAFL, a grey-box fuzzing technique that infers FSMs based on the in-memory states of a server, leveraging compile-time instrumentation and fuzzy hashing techniques; hence, it does not require response codes. During the fuzzing process, StateAFL guides the generation of new inputs to the server based on the inferred FSMs. It employs both byte-level and message-level fuzz operators, which do not rely on protocol specifications.

2.2.2 Software-Defined Network Testing

SDN (Software-Defined Networking) testingfocuses specifically on the unique challenges and requirements of SDN environments. As explicated in Section 2.1.1, SDN decouples the control plane from the data plane, allowing for more flexible and programmable network management. Testing SDN involves verifying the functionality, performance, and security of the SDN components, such as controllers, switches, and applications.

Testing SDNs has been primarily studied in the networking literature targeting various objectives, such as detecting security vulnerabilities and attacks [49, 50, 35, 34, 51, 36, 52], identifying inconsistencies among the SDN components (i.e., applications, controllers, and switches) [53, 37, 38], and analyzing SDN executions [54, 55, 56]. Among these various SDN testing techniques, fuzzing has emerged as a particularly effective method for uncovering vulnerabilities and ensuring robustness in SDN systems.

For example, Woo et al. [57] proposed RE-CHECKER to fuzz RESTful services provided by SDN controllers. RE-CHECKER fuzzes an input file, encoded in JSON format, that a network administrator uses to specify network policies (e.g., data forwarding rules). This results in generating numerous malformed REST messages for testing RESTful services in SDN. Dixit et al. [58] presented AIM-SDN to test the implementation of the network management datastore architecture (NMDA) [59] in SDN. AIM-SDN randomly fuzzes REST messages to test the NMDA implementation in SDN with regard to the availability, integrity, and confidentiality of datastores. Shukla et al. [38] developed PAZZ that aims at detecting faults in SDN switches by fuzzing data packet headers, e.g., IPv4 and IPv6 headers. Albab et al. [60] presented SwitchV to validate the behaviors of SDN switches. SwitchV uses fuzzing and symbolic execution to analyze the p4 [61] models that specify the behaviours of SDN switches. Lee et al. [37, 62] introduced AudiSDN that employs fuzzing to detect policy inconsistencies among SDN components (i.e., controllers and switches). AudiSDN fuzzes network policies submitted by administrators through the REST APIs. To increase the likelihood of discovering inconsistencies, AudiSDN employs rule dependency trees derived from the OpenFlow specification, which restrict valid relationships among rule elements.

2.2.3 Supervised Machine Learning

Machine learning (ML) is a subset of artificial intelligence (AI) that focuses on the development of algorithms and statistical models enabling computers to perform tasks without explicit instructions. Instead, ML systems learn from data, identifying patterns and making decisions based on that data. This approach has been widely adopted in various fields due to its ability to handle complex and large-scale data [63].

Supervised machine learning is one category of machine learning techniques that infers a model based on labelled datasets, most commonly used to solve *classification* or *regression* problems. We focus in this dissertation on *classification* techniques. A classification technique aims to predict the categorical label of new, unseen instances based on patterns learned from a training dataset. These techniques are essential in various applications, such as spam detection [64], image recognition [65], and medical diagnosis [66], where the goal is to assign predefined labels to input data [67].

Classification algorithms can be broadly categorized into several types, including Decision Trees [68], Support Vector Machines [69], K-Nearest Neighbors [67], Neural Networks [70], Rule-Based learners such as RIPPER (Repeated Incremental Pruning to Produce Error Reduction) [71]. Each of these algorithms has its strengths and weaknesses, making them suitable for different types of classification tasks.

Among these approaches, RIPPER is particularly valuable due to its adaptability to various classification problems and its ability to provide interpretable models. RIPPER explains the relationship between dependent and independent variables through a set of "*if-then*" rules, making it easier to understand and apply. This interpretability is especially beneficial in fields like software engineering, where understanding the decision-making process is crucial. Consequently, many studies in software engineering have utilized RIPPER for its effectiveness and clarity. In this section, we provide a brief explanation of RIPPER.

RIPPER. RIPPER is a type of rule-based learning algorithm used for classification and prediction. This model explains an outcome variable through a set of if-then rules, which can handle both categorical and continuous data. Variables in many real-world contexts, such as determining whether an email is spam or not, the presence or absence of a disease, and customer segmentation, can be effectively modeled using RIPPER. These variables often have a limited number of values, especially binary variables, which only have values of 0 or 1. However, RIPPER is also capable of handling continuous variables by discretizing them into intervals.

RIPPER begins by generating an initial set of rules from the training data. Each rule is created to cover a subset of the data, aiming to maximize the accuracy of classification for that subset. The algorithm then prunes the rules to remove any unnecessary conditions, reducing overfitting and improving generalization to new data. This pruning process involves evaluating the impact of removing each condition on the rule's accuracy and retaining only those conditions that contribute positively.

The resulting set of rules forms an interpretable model that can be easily understood and applied to new instances. Each rule in the model specifies a combination of conditions that, if met, lead to a specific classification outcome. This rule-based approach is particularly valuable in domains where interpretability and transparency are crucial, such as medical diagnosis and fraud detection.

RIPPER's effectiveness and simplicity have led to its widespread application in the field of software engineering and other areas. By providing clear and concise rules, RIPPER helps practitioners understand the underlying patterns in the data and make informed decisions based on those patterns.

2.2.4 Genetic Algorithms

Genetic algorithms (GAs), first introduced by Holland [72], are meta-heuristic algorithms widely employed to address complex optimization problems. Inspired by the principles of natural selection and genetics, GAs evolve an initial population of candidate solutions through iterative processes of selection, crossover, and mutation.

The computation of GAs may vary, but can generally be represented by the following algorithm:

Algorithm 1 Pseudo-Code of a Genetic Algorithm (GA)

Input:
s: desired population size
Output:
$Best$: Set of equally viable solutions

```
1:  // Initialization
2:  P ← ∅
3:  Best ← ∅
4:  for s times do
5:      P ← P ∪ {CreateIndividual()}
6:  end for
7:  // Evolution process
8:  repeat
9:      // Evaluate individuals
10:     for each P_i ∈ P do
11:         AssessFitness(P_i)
12:         if IsBest(P_i, Best) then
13:             Best ← P_i
14:         end if
15:     end for
16:     // Breed the new population
17:     Q ← ∅
18:     for s/2 times do
19:         P_a ← SelectParent(P)
20:         P_b ← SelectParent(P)
21:         {C_a, C_b} ← Crossover(P_a, P_b)
22:         Q ← Q ∪ {Mutation(P_a), Mutation(P_b)}
23:     end for
24:     // Replace the population
25:     P ← Replace(P, Q)
26: until the termination condition is reached
27: return Best
```

Algorithm 1 first randomly creates an initial population P with a size s (lines 1-6). This population is generated by randomly creating each individual, i.e., *candidate solution* or *chromosome* (lines 4-6).
An individual is represented by a fixed-length vector of parameters, i.e., *genes*, which represent any type of variable that should be optimized.

The algorithm then repeats the evolution process by evaluating the population P (lines 9-15), creating a new population Q through breeding (lines 16-23), and replacing the old population with a new population (lines 24-25). The algorithm stops when the termination condition is met. Specifically, for the evaluation of the given population P, the GA assesses the fitness of each individual P_i (lines 10-11) by leveraging one or several fitness functions. Fitness functions are key concepts of the GA that guide the definition of *optimal solutions*. These functions must be carefully crafted to ensure the success of the algorithm. Based on fitness values, the GA then selects the *Best* individuals in the population based on their fitness (lines 12-13).
The breeding process of the population P generates a new population Q using the selection, crossover, and mutation operators. Firstly, the GAs select two individuals P_a and P_b as parents from the population P (lines 19-20). The exact selection process is usually governed by a selection operator, with tournament selection being the most popular due to its simplicity and effectiveness [73, 74]. Secondly, the selected parents reproduce to create two children C_a and C_b through a crossover operator (line 21) This operator determines which traits the children will inherit from their parents, mimicking the biological crossover process. The *genes* (i.e., fixed-length vector) are merged to create the two children, ensuring each child inherits a proportional amount of genes from each parent, but so that each of them is unique. Typically, a crossover rate defines whether this operator should be applied; if not, the children become copies of their parents. Finally, a mutation changes some genes in the children individuals based on a mutation rate (line 22). The operator iterates over each gene and changes it when a random value exceeds the mutation rate. This operation allows the GA to explore other search areas, preventing it from getting stuck in local optima. It is crucial to adapt the mutation rate, as higher values may make the search process akin to a random search or prevent the algorithm from converging [74].
After breeding a new population Q, the replacement process selects individuals for the next generation (line 25). Several methods exist in the literature, such as generational replacement, steady-state replacement, and elitism [74]. The most commonly used methods generally incorporate some form of elitism, which ensures that the best individuals from the current generation are carried over to the next generation, preserving high-quality solutions and accelerating convergence.

One of the key advantages of GAs is their flexibility in parameter selection, allowing for a diverse range of solutions. Instead of converging on a single near-optimal solution, GAs often produce a set of Pareto-optimal solutions, providing multiple viable options for decision-makers. This characteristic is particularly beneficial in multi-objective optimization scenarios.

Chapter 3

Learning Failure-Inducing Inputs Models for Testing Software-Defined Networks

3.1 Introduction

In the context of developing SDN-systems, software testing becomes even more important and challenging when compared to what is required in traditional networks that provide static and predictable operations. In particular, even though the centralized controller in an SDN-system enables flexible and efficient services, it can undermine the entire communication network it manages. A software controller presents new attack surfaces that allow malicious users to manipulate the systems [75, 76]. For example, if malicious users intercept and poison communication in the system (using ARP spoofing [77]), such attacks broadly impact the system due to its centralized control. Furthermore, the centralized controller interacts with diverse kinds of components such as applications and network switches, which are typically developed by different vendors. Hence, the controller is prone to receiving unexpected inputs provided by applications, switches, or malicious users, which may cause system failures, e.g., communication breakdown.

To test an SDN controller, engineers need first to explore its possible input space, which is very large. A controller takes as input a stream of control messages which are encoded according to an SDN communication protocol (e.g., OpenFlow [25]). For example, if a control message is encoded with OpenFlow, it can have 2^{2040} distinct values [25]. Second, engineers need to

understand the characteristics of test data, i.e., control messages, that cause system failures. However, manually inspecting test data that cause failures is time-consuming and error-prone. Furthermore, misunderstanding such causes typically leads to unreliable fixes.

There are a number of prior research strands that aim at testing SDN-systems [49, 50, 35, 34, 51, 57, 36, 52, 60]. Most of them come from the network research field and focus on security testing relying on domain knowledge, e.g., known attack scenarios [34]. The most pertinent research works applied fuzzing techniques to different components of SDN-systems. For example, RE-CHECKER [57] fuzzes RESTful services provided by SDN controllers. SwitchV [60] relies on fuzzing and symbolic execution to test SDN switches. BEADS [35] tests SDN controllers by being aware of the OpenFlow specification. However, none of these fuzzing techniques employ interpretable machine learning techniques to guide their fuzzing process and to provide models that characterize failure-inducing conditions. Even though the software engineering community has introduced numerous testing methods, testing SDN-systems has gained little attention. The most pertinent research lines have proposed techniques for learning-based fuzzing [78, 79, 80] and abstracting failure-inducing inputs [15, 81] to efficiently explore the input space and characterize effective test data that cause system failures. Learn@Fuzz [78] employs neural-network-based learning methods for building a model of PDF objects for grammar-based fuzzing. A prior learning-guided fuzzing technique, named smart fuzzing [79], relies on deep learning techniques to test systems controlled by programmable logic controllers (PLC). SeqFuzzer [80] uses deep learning techniques to infer communication protocols underlying PLC systems and generate fuzzed messages. However, deep learning techniques are not suitable to characterize failure-inducing test data as they do not provide interpretable models. Furthermore, these techniques do not account for the specificities of SDNs, such as SDN architecture and communication protocol. Existing work on abstracting failure-inducing inputs [15, 81] targets software programs that take as input strings such as command-line utilities (e.g., find and grep), which are significantly different from SDN-systems. In summary, no existing work simultaneously tackles the problem of efficiently exploring the input space and accurately characterizing failure-inducing test data while accounting for the specificities of SDNs.

Contributions. In this chapter, we propose FuzzSDN, a machine learning-guided Fuzzing method for testing SDN-systems. In particular, FuzzSDN targets software controllers deployed in SDN-systems. FuzzSDN relies on fuzzing guided by machine learning (ML) to both (1) efficiently explore the test input space of an SDN-system's controller (generate test data leading to system failures) and (2) learn failure-inducing models that characterize input conditions under which the system fails. This is done in a synergistic manner where models guide test generation and the latter also aims at improving the models. A failure-inducing model is practically useful [15] for the following reasons: (1) It facilitates the diagnosis of system failures. FuzzSDN provides engineers with an interpretable model specifying how likely are failures to occur, e.g., the system fails when a control message is encoded using OpenFlow V1.0 and contains IP packets, thus providing concrete conditions under which a system will probably fail. Such conditions are much easier to analyze than a large set of individual failures. (2) A failure-inducing model enables engineers to validate

their fixes. Engineers can fix and test their code against the generated test data set. A failure-inducing model can also be used as a test data generator to reproduce the system failures captured in the model. Hence, engineers can better validate their fixes using an extended test data set.

We evaluated FuzzSDN by applying it to several systems controlled by well-known open-source SDN controllers: ONOS [82] and RYU [83]. In addition, we compared FuzzSDN with two state-of-the-art methods (i.e., DELTA [34] and BEADS [35]) that generate test data for SDN controllers and two baselines that learn failure-inducing models. As baselines, we extended DELTA and BEADS to produce failure-inducing models, since they were not originally designed for that purpose but were nevertheless our best options. Our experiment results show that, compared to state-of-the-art methods, FuzzSDN generates at least 12 times more failing control messages, within the same time budget, with a controller that is fairly robust to fuzzing. FuzzSDN also produces accurate failure-inducing models with, on average, a precision of 98% and a recall of 86%, which significantly outperforms models inferred by the two baselines. Furthermore, FuzzSDN produces failure-inducing conditions that are consistent with those reported in the literature [35], indicating that FuzzSDN is a promising solution for automatically characterizing failure-inducing conditions and thus reducing the effort needed for manually analyzing test results. Last, FuzzSDN is applicable to systems with large networks as its performance does not depend on network size. Our detailed evaluation results and the FuzzSDN tool are available online [84].

Organization. The rest of this chapter is organized as follows: Section 3.2 introduces the background and defines the specific problem of learning failure-inducing models for testing SDN-systems. Section 3.3 describes FuzzSDN. Section 3.4 evaluates FuzzSDN in a large empirical study. Section 3.5 compares FuzzSDN with related work. Section 3.6 concludes this chapter.

3.2 Background and Problem Description

When developing and operating an SDN-system, engineers must handle system failures that are triggered by unexpected control messages. In particular, engineers need to ensure that the system behaves in an acceptable way in the presence of failures. In an SDN-system, its controller is prone to receiving unexpected control messages from switches in the system [52]. For example, network switches, which are typically developed by different vendors, may send control messages that fall outside the scope of the controller's expectations. Such unexpected messages can also be sent by the switches due to various reasons such as malfunctions and bugs in the switches, as well as inconsistent implementations of a communication protocol between the controller and switches [37]. Furthermore, prior security assessments of SDNs have found several attack surfaces, leading to vulnerable applications and communications protocols that enable malicious actors to send manipulated messages over an SDN [34, 35].

When a failure occurs in an SDN-system, engineers need to determine the conditions under which such failures occur. These conditions define a set of control messages that cause the

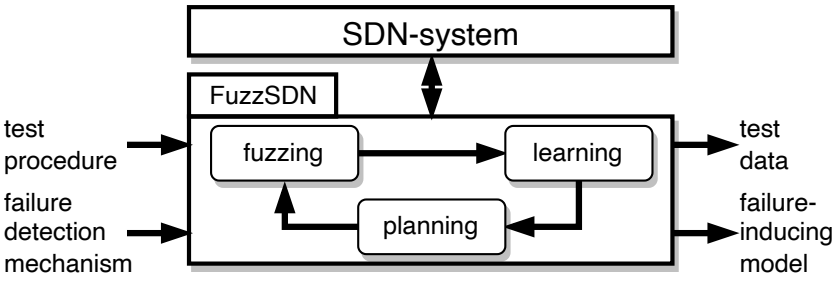


Figure 3.1: An overview of our ML-guided Fuzzing method for testing SDN-systems (FuzzSDN).

failure. Identifying such conditions in a precise and interpretable form is in practice useful, as it enables engineers to diagnose the failure with a clear understanding of the conditions that induce it. In addition, engineers can produce an extended set of control messages by utilizing the identified conditions to test the system after making changes to address the failure. In general, any fix should properly address other control messages that induce the same failure. Our work aims to both effectively test an SDN-system's controller by identifying control messages that lead to system failures, and then automatically identify an accurate failure-inducing model that characterizes conditions under which the SDN-system fails. Such conditions define a set of failure-inducing control messages.

3.3 Approach

Figure 3.1 shows an overview of our ML-guided Fuzzing method for testing SDN-systems (FuzzSDN). Specifically, FuzzSDN relies on fuzzing and ML techniques to effectively test an SDN-system's controller and generate a failure-inducing model that characterizes conditions under which the system fails. As shown in Figure 3.1, FuzzSDN takes as input a test procedure and a failure detection mechanism defined by engineers.

A test procedure consists of three steps: initializing the SDN-system, executing the test scenario, and tearing down the system. In the initialization step, the procedure configures the entities in the SDN-system, such as the controller, switches, and connections, bringing the system to a desired (ready) state for executing the test scenario. For example, during the initialization, the procedure might configure switches with empty or preinstalled forwarding tables, which define how to forward data packets through the SDN. This system configuration allows for testing how the controller instructs the switches in the given setting. The test scenario represents a specific operation (use case) of the SDN-system, involving the exchange of control messages between the controller and switches that FuzzSDN can fuzz. For example, if the test scenario specifies a data transmission from one host to another connected to the SDN, executing the test scenario leads to the exchange of control messages between the controller and switches. These messages are exchanged to discover the locations of the two hosts in the SDN and to instruct the relevant switches with the proper forwarding tables to enable data transmission. In the teardown step, the

procedure resets the SDN-system to its default (pristine) state after executing the test scenario, since FuzzSDN requires multiple independent executions of the test procedure.

A failure detection mechanism acts as a test oracle, determining whether the system fails or successfully completes the given test procedure. For example, depending on the test procedure, it detects instances of communication breakdown, controller crashes, or performance degradation. Since an SDN-system provides monitoring tools that enable engineers to oversee system behavior and performance, implementing such a failure detection mechanism is straightforward.

FuzzSDN then outputs test data and a failure-inducing model. The data includes the set of control messages that induced the failure detected by the failure detection mechanism. A failure-inducing model abstracts such test data in the form of conditions and probabilities pertaining to the failure.

As shown in Figure 3.1, FuzzSDN realizes an iterative process consisting of the following three steps: (1) The fuzzing step sniffs and modifies a control message passing through the control channel from the SDN switches to the controller. The fuzzing step repeats the execution of the input test procedure and modifies only one selected control message for each execution of the system. It then produces a labeled dataset that associates fuzzing outputs (i.e., modified control messages) and their consequences in the system (i.e., system failure or success). (2) The learning step takes as input the labeled dataset created by the fuzzing step and uses a supervised learning technique, e.g., RIPPER [71], to create a failure-inducing (classification) model. This model identifies a set of control messages that cause the failure defined in the failure detection mechanism. Over the iterations of FuzzSDN, such models are used to guide the behavior of the fuzzing step in the next iteration. (3) The planning step instructs the fuzzing step based on the failure-inducing model created by the learning step. Following such instructions, the fuzzing step then efficiently explores the space of control messages to be fuzzed and adds new data points (i.e., modified control messages and their consequences in the system) to the existing labeled dataset. The updated dataset is then used by the learning step to produce an improved failure-inducing model. FuzzSDN stops the iterations of the fuzzing, learning, and planning steps when the accuracy of the output failure-inducing model reaches an acceptable level or the execution time of FuzzSDN exceeds an allotted time budget. Below, we explain each step of FuzzSDN in detail.

We assume that engineers using FuzzSDN have expertise in SDN, thus equipping them with the capability to provide required inputs: test procedures and failure detection mechanisms. They should also possess knowledge of the SDN protocol to understand a failure-inducing model that characterizes a set of failure-inducing control messages. Such assumptions are reasonable since engineers need to devise test procedures and failure detection mechanisms when testing their SDN-systems, independently of FuzzSDN, and must know the SDN protocol as it defines the input space of the SDN controller under test.

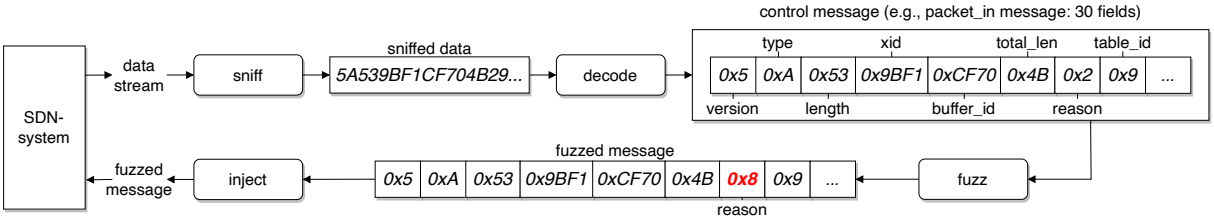


Figure 3.2: A data flow example of fuzzing a control message (e.g., packet_in message).

3.3.1 Fuzzing step: Initial fuzzing

During the fuzzing step, FuzzSDN manipulates control messages in an SDN-system to cause a system failure. To do so, FuzzSDN utilizes a man-in-the-middle attack, which is a well-known security attack technique in the network domain [77]. The attack technique enables FuzzSDN to intercept control messages transmitting through the control channel and inject modified messages. In addition, FuzzSDN pretends to be both legitimate participants (i.e., SDN switches and controllers) of the control channel. Hence, the system under test is not aware of FuzzSDN while it is running. We omit network-specific details of the attack technique, as they are not part of our contributions; instead, we refer interested readers to the relevant literature [77].

Figure 3.2 shows a data flow example that illustrates how our fuzzing technique manipulates a control message. As the control channel in the SDN-system transmits a data stream, the fuzzing step first sniffs it as a byte string. It then decodes the sniffed string according to the adopted SDN protocol (e.g., OpenFlow [25]) to identify a control message to be fuzzed. In Figure 3.2, the byte string 0x5A539BF1CF704B29 is sniffed and is then identified as a packet_in control message encoded in 30 fields according to OpenFlow. Note that a packet_in message is one of the control messages sent by a switch to a controller in order to notify that the switch receives a packet. For details of the packet_in message, we refer readers to the OpenFlow specification [25].

FuzzSDN fuzzes the control message by accounting for the syntax requirements (i.e., grammar) defined in the SDN protocol and then injects the fuzzed message into the control channel in the system. We note that FuzzSDN could apply a simple random fuzzing method that replaces the sniffed string with a random string. However, the majority of byte strings generated by random fuzzing would be invalid control messages that would be immediately rejected by the SDN message parser in the system [35]. FuzzSDN therefore accounts for the SDN protocol in order to generate valid control messages that test software components beyond the message parsing layer of the system, which is a desirable feature in practice [35].

Initial fuzzing. At the first iteration of FuzzSDN, since a failure-inducing model is not present, the fuzzing step behaves as described in Algorithm 2. Given a control message msg, the algorithm modifies it and returns a fuzzed message msg'. As shown on line 1, the algorithm first randomly selects a set F of fields in msg. For each field f in F, the algorithm replaces the original value of f with a new value randomly selected from its value range (lines 2-5). Hence, Algorithm 2

Algorithm 2 Initial fuzzing

Require:
msg: control message to be fuzzed
Ensure:
msg': control message after fuzzing

1: $F \leftarrow select\_rand\_fields(msg)$
2: $msg' \leftarrow msg$
3: **for all** $f \in F$ **do**
4: $\quad msg' \leftarrow replace(msg', f, rand\_valid(f))$
5: **end for**
6: **return** msg'

operates in linear time, relative to the number of message fields ($|F|$). Note that our ML-guided fuzzing method is described in Section 3.3.4. For example, in Figure 3.2, the algorithm modifies the sniffed packet_in message by replacing the reason field value 0x2 with 0x8, which is randomly chosen within its value range.

Data collection. To generate a failure-inducing model, FuzzSDN uses a supervised ML technique [85] that requires a labeled dataset the fuzzing step generates. Specifically, at each iteration of FuzzSDN, the fuzzing step is executed n times based on an allotted time budget. Each execution of the fuzzing step (re)runs the input test procedure (Figure 3.1) and modifies a control message. FuzzSDN then monitors the system response to the modified control message *msg* using the failure detection mechanism (Figure 3.1). We denote by *presence* (resp. *absence*) the label indicating that the failure is present (resp. absent) in the system response. For each iteration i of FuzzSDN, the fuzzing step creates a labeled dataset D_i by adding n tuples (msg_1, l_1), ..., (msg_n, l_n) to D_{i-1}, where a label l_j could be either *presence* or *absence* and $D_0 = \{\}$. We note that FuzzSDN uses an accumulated dataset $D = D_1 \cup \ldots \cup D_i$ to infer a failure-inducing model at each iteration i.

3.3.2 Learning step

We cast the problem of learning failure-inducing models into a binary classification problem in ML. Given a labeled dataset obtained from the fuzzing step, the learning step infers a prediction model, i.e., a failure-inducing model, that classifies a control message as either the *presence* or *absence* class. A classification result predicts whether or not a control message induces a system failure as captured by the detection mechanism.

FuzzSDN aims at providing engineers not only with failure-inducing control messages but also accurate conditions under which the system fails (as described in Section 3.1). Hence, we opt to use a learning technique that produces an interpretable model [86]. Engineers could use the prediction results to identify a set of control messages predicted to induce system failures as a test suite for testing the system. An interpretable model would help engineers figure out why the

failure occurs in the system. We encode fields (e.g., version, type, and length in Figure 3.2) of a control message as features in a labeled dataset so that an interpretable ML technique builds a failure-inducing model using fields as features. For example, such a model could explain that the system fails when receiving a control message with an incompatible version number, which is a higher version than the system supports.

FuzzSDN employs RIPPER (Repeated Incremental Pruning to Produce Error Reduction) [71], an interpretable rule-based classification algorithm, to learn a failure-inducing model. We opt to use RIPPER as it generates pruned decision rules that are more concise and thus interpretable than commonly used decision trees (e.g., C4.5 [87]), which are prone to the replicated subtree problem [85]. Further, RIPPER has been successfully applied to many software engineering problems involving classification and rule inference [88, 89, 90]. Given a labeled dataset D, RIPPER produces a set R of decision rules. A decision rule is a simple IF-condition-THEN-prediction statement, consisting of a condition on the fields of a control message (e.g., $version > 5 \wedge length \geq 10$) and a prediction indicating either the *presence* or *absence* class.

For a decision rule, the learning step measures a confidence score [85] to estimate the accuracy of the rule in predicting the actual class of control messages that satisfy the rule's condition. Specifically, given a labeled dataset D and a decision rule r, we denote by t the total number of control messages in D that satisfy the condition of r, i.e., the condition is evaluated to true. Among these t control messages, often, there are some control messages whose labels defined in D do not match r's prediction. We denote by f the number of such control messages. The learning step computes a confidence score $c(r)$ of r by $c(r) = (t - f)/t$. For example, given a rule r: IF $version > 5 \wedge length \geq 10$ THEN $class = presence$, suppose 88 control messages in a labeled dataset D satisfy the condition of r and 7 out of the 88 control messages are labeled with *absence*. Then, the confidence score $c(r)$ is $(88 - 7)/88 = 0.92$. FuzzSDN uses the confidence score $c(r)$ in the planning step to guide our fuzzing strategy.

3.3.3 Planning step

The planning step guides fuzzing based on a failure-inducing model, i.e., a set R of decision rules inferred from the learning step. Given an original control message to be fuzzed, the main idea is to generate a set of modified control messages using R. To this end, the planning step exploits these decision rules to generate effective control messages that induce failures.

Imbalance handling. Algorithm 3 describes how the planning step uses the set R of decision rules inferred in the current iteration of FuzzSDN to guide the fuzzing step for the next iteration. We note that the planning step accounts for the imbalance problem [85] that usually causes poor performance of ML algorithms. In a labeled dataset, when the number of data instances of one class is much higher than such number for another class, ML classification models have a

Algorithm 3 Planning

Require:
D: labeled dataset
R: decision rules
n: number of control messages to be fuzzed
Ensure:
B: budget distribution to the decision rules R

1: //count numbers of majorities and minorities in D
2: $minor \leftarrow get\_num\_minorities(D)$
3: $major \leftarrow get\_num\_majorities(D)$
4: //estimate numbers of majorities and minorities after the next iteration
5: $minor' \leftarrow \min((|D| + n)/2 - minor, n)$
6: $major' \leftarrow n - minor'$
7: //assign budgets to minority rules
8: $B \leftarrow \{\}$
9: $R^{minor} \leftarrow get\_minority\_rules(R)$
10: **for all** $r \in R^{minor}$ **do**
11: $b \leftarrow minor' \times c(r)/sum\_c(R^{minor})$
12: $B \leftarrow B \cup (r, b)$
13: **end for**
14: //assign budgets to majority rules
15: $R^{major} \leftarrow get\_majority\_rules(R)$
16: **for all** $r \in R^{major}$ **do**
17: $b \leftarrow major' \times c(r)/sum\_c(R^{major})$
18: $B \leftarrow B \cup (r, b)$
19: **end for**
20: **return** B

tendency to predict the majority class. In our study, such models are not practically useful, as engineers are more interested in control messages that cause system failures.

As shown on lines 1-3 of Algorithm 3, the planning step first counts the number *minor* (resp. *major*) of minority (resp. majority) control messages in the given labeled dataset D. Given the number n of control messages to be fuzzed in the next iteration, lines 4-6 of the algorithm then estimate the number $minor'$ (resp. $major'$) of control messages associated with the minority (resp. majority) class to be added to D to create a balanced dataset, containing $|D| + n$ control messages. Specifically, FuzzSDN needs $(|D| + n)/2 - minor$ control messages associated with the minority class to balance D in the next iteration. Table 3.1 shows examples of *minor*, *major*, $minor'$, and $major'$ at each iteration of FuzzSDN computed by Algorithm 3. For example, at the first iteration of FuzzSDN, when $minor = 10$, $major = 190$, and the number of control messages to be fuzzed $n = 200$, $minor'$ is calculated as $(200 + 200)/2 - 10 = 190$ and $major' = 10$.

Budget distribution. Lines 7-20 of Algorithm 3 describe how the planning step distributes the number n of control messages to be fuzzed in the next iteration to each decision rule $r \in R$. As shown on lines 9-13 of the algorithm, for each rule $r \in R^{minor}$ associated with the minority class, the planning step decides to use r for fuzzing based on its relative confidence score and the number

Table 3.1: Examples of *minor*, *major*, *minor′*, and *major′* computed by Algorithm 3, when the number n of control messages to be fuzzed is 200.

iteration	$\|D\|$	*minor*	*major*	*minor′*	*major′*
1	200	10	190	190	10
2	400	125	275	175	25
3	600	248	352	152	48
4	800	380	420	120	80
5	1000	495	505	105	95
6	1200	600	600	100	100

minor′ of control messages estimated on line 5. Specifically, the planning step associates r with the number of times r will be applied to fuzz control messages, i.e., $minor' \times c(r)/sum\_c(R^{minor})$, where $c(r)$ denotes the rule's confidence score (described in Section 3.3.2) and $sum\_c(R^{minor})$ is defined by $\sum_{r \in R^{minor}} c(r)$, the sum of confidence scores of the rules in R^{minor}. The algorithm therefore weighs the rules according to their confidence scores in order to maximize the chance of correct predictions. When fuzzing a control message guided by a rule r with a high confidence score (e.g., 0.99), the system response to the fuzzed control message would highly likely match the prediction of r. Lines 14-19 describe how the planning step handles rules associated with the majority class, which is the same as on lines 9-13. For example, at the first iteration of FuzzSDN shown in Table 3.1, let R be $\{r_1, r_2, r_3\}$ where r_1 and r_2 are associated with the minority class (e.g., *presence*) and r_3 with the majority class (e.g., *absence*). Given R, if $c(r_1) = 0.8$, $c(r_2) = 0.7$, and $c(r_3) = 0.8$, then Algorithm 3 distributes $minor' = 190$ (resp. $major' = 10$) to r_1 and r_2 (resp. r_3) as follows: $190 \times 0.8/(0.8 + 0.7) = 101$ and $190 \times 0.7/(0.8 + 0.7) = 89$ (resp. $10 \times 0.8/0.8 = 10$). For the second iteration, FuzzSDN then plans to apply r_1 101 times, r_2 89 times, and r_3 10 times to fuzz control messages. Algorithm 3 operates in linear time, relative to the number of rules ($|R|$), inferred by the learning step.

We note that during early iterations of FuzzSDN, the obtained datasets are likely imbalanced because control messages causing system failures are typically difficult to discover via purely random fuzzing, since most fuzzed messages are detected and addressed by the system under test to prevent such failures (see Table 3.1 and our experiment results in Section 3.4.6). In addition, due to the small sizes of training datasets in early iterations of FuzzSDN, RIPPER is often not able to produce accurate failure-inducing models. But as FuzzSDN continuously iterates the three steps within an allotted time budget, according to Algorithm 3, training datasets are becoming more balanced and larger (see Table 3.1), enabling RIPPER to produce increasingly accurate failure-inducing models. Furthermore, given S the space of all possible control messages, once a dataset is balanced, the algorithm enables the fuzz step to explore not only the space P of control messages that likely cause failures but also the remaining space $S \setminus P$ of control messages. Note that RIPPER infers a set of rules' conditions (which define P): $r_1, \ldots, r_k$, that are associated with the minority class and a single rule's condition (which define $S \setminus P$) in the form of $\neg r_1 \wedge \ldots \wedge \neg r_k$ for the majority class.

Algorithm 4 ML-guided Fuzzing

Require:
msg: control message to be fuzzed
B: budget distribution to the decision rules R
mu: mutation rate
Ensure:
msg': control message after fuzzing
B': budget distribution after fuzzing

1: //fuzz a control message based on a rule
2: $(r, b) \in B$
3: $F \leftarrow get\_fields(r, msg)$
4: $F' \leftarrow solve(r)$
5: $msg' \leftarrow replace(msg, F, F')$
6: //update the budget distribution
7: $B' \leftarrow B \setminus \{(r, b)\}$
8: **if** $b - 1 > 0$ **then**
9: $\quad B' \leftarrow B \cup \{(r, b - 1)\}$
10: **end if**
11: //mutate the fuzzed control message
12: **for all** $f \in all\_fields(msg) \setminus F$ **do**
13: $\quad$ **if** $rand(0, 1) \leq mu$ **then**
14: $\quad\quad msg' \leftarrow replace(msg', f, rand(f))$
15: $\quad$ **end if**
16: **end for**
17: **return** msg', B'

Progress monitoring. To monitor the progress of FuzzSDN, the planning step uses the standard precision and recall metrics [85] (described in Section 3.4.4). In our context, a high level of precision indicates that the inferred failure-inducing model is able to accurately predict the failure of interest. A failure-inducing model with high recall indicates that most of the control messages actually inducing the failure satisfy the failure-inducing conditions in the model. Hence, a failure-inducing model with a high level of precision and recall is desirable. To compute precision and recall values, FuzzSDN uses the 10-fold cross-validation technique [85]. In 10-fold cross-validation, a dataset D is split into 10 equal-size folds. Nine folds are used as a training dataset and the other one fold is retained as a test dataset. This process is thus repeated 10 times to compute precision and recall values.

3.3.4 Fuzzing Step: ML-guided Fuzzing

From subsequent iterations of FuzzSDN, the fuzzing step utilizes a set R of decision rules inferred by the learning step according to a budget distribution B computed by the planning step. Using a rule $r \in R$, the fuzzing step modifies a sniffed control message to satisfy the condition of r. Further, the fuzzing step employs a mutation operator to diversify fuzzed control messages beyond those restricted by R. Below we describe the fuzzing step in detail.

Algorithm 4 describes a fuzzing procedure that modifies a sniffed control message *msg* using a budget distribution B. As shown on lines 1-5 of the algorithm, the fuzzing step first chooses a budget assignment $(r, b) \in B$, where r denotes a rule to apply in fuzzing, and b denotes how many times the rule r will be exploited by the fuzzing step. Given the rule r, the algorithm selects a set F of message fields in *msg* that appear in the condition of r (line 3). Using an SMT solver [91], the algorithm solves the condition of r to find a set F' of message fields that satisfy the condition (line 4). Specifically, we use Z3 [92] – a well-known and widely used SMT solver – to solve such conditions. Line 5 of the algorithm then replaces the original fields F with the computed fields F'. For example, when FuzzSDN fuzzes a control message guided by the condition $version > 5 \wedge length \geq 10$, it assigns 6 to the *version* field and 20 to the *length* field of the control message as the assignments satisfy the condition.

Algorithm 4 modifies a single control message *msg* and outputs one fuzzed message msg'. Hence, the fuzzing step executes the algorithm n times to generate n number of fuzzed control messages. As shown on lines 6-10, the algorithm updates the budget distribution B with $(r, b-1)$ indicating that the rule r has been applied once. Note that the fuzzing step reruns the system under test for each execution of the algorithm.

As shown on lines 11-17 of Algorithm 4, the fuzzing step leverages a mutation technique to diversify fuzzed control messages. Recall that lines 1-5 of the algorithm modify only the message fields that appear in decision rules. Without mutation, decision rules inferred in the first iteration of FuzzSDN would determine the message fields being modified in all subsequent iterations, while other message fields would remain unchanged. Such a fuzzing method might miss important failure-inducing rules related to other unchanged fields.

The fuzzing step employs a uniform mutation operator [93] that randomly selects fields in a control message with a mutation rate *mu* and changes the fields' values to random values within their ranges. As shown on line 12 of Algorithm 4, the fuzzing step selects the fields in the sniffed control message *msg* that are not present in the exploited rule r. Hence, the return message msg' (line 14) also satisfies the condition of r, as mutated fields cannot affect it. For example, suppose a packet_in message encoded in 30 fields is sniffed to be fuzzed by FuzzSDN, and its version and length fields are included in the condition $version > 5 \wedge length \geq 10$ and hence fuzzed (lines 1-10). In this setting, FuzzSDN randomly selects fields (e.g., *reason* and *table_id*) that are not present in the condition, and then mutates the selected fields by assigning new random values within their ranges (lines 11-16).

We note that the computational complexity of Algorithm 4 is primarily determined by line 4, which uses Z3. The remaining computations scale linearly with the number of message fields. In our context, as described in Section 3.3.2, the rule r to be solved by Z3 is concise. Therefore, Algorithm 4 is expected to run in practical time, as empirically evaluated in our experiments (Section 3.4.6).

3.4 Evaluation

In this section, we present our empirical evaluation of FuzzSDN. Our full evaluation package is available online [84].

3.4.1 Research Questions

RQ1 (comparison): *How does FuzzSDN perform compared with state-of-the-art testing techniques for SDNs?* We investigate whether FuzzSDN can outperform existing techniques: DELTA [34] and BEADS [35] described in Section 3.4.4. We choose these techniques as they rely on fuzzing to test SDN-systems and their implementations are available online. Note that none of the prior methods that identify failure-inducing inputs [15, 81] account for the specificities of SDNs; hence, they are not applicable.

RQ2 (usefulness): *Can FuzzSDN learn failure-inducing models that accurately characterize conditions under which a system fails?* We investigate whether or not FuzzSDN can infer accurate failure-inducing models. In addition, we compare the failure-inducing conditions identified by FuzzSDN with those reported in the literature [35] to assess if these conditions are consistent with analyses from experts.

RQ3 (scalability): *Can FuzzSDN fuzz control messages and learn failure-inducing models in practical time?* We analyze the relationship between the execution time of FuzzSDN and network size. To do so, we conduct experiments with systems of various network sizes.

3.4.2 Simulation Platform

To evaluate FuzzSDN, we opt to use a simulation platform that emulates physical networks. Specifically, we use Mininet [94] to create virtual networks of different sizes. In addition, as Mininet employs real SDN switch programs, the emulated networks are very close to real-world SDNs. Hence, Mininet has been widely used in many SDN studies [35, 34, 21]. We note that FuzzSDN can be applied to test actual SDN-systems. However, using physical networks is prohibitively expensive for performing the types of large experiments involved in our systematic evaluations of FuzzSDN. We ran all our experiments on 10 virtual machines, each of which with 4 CPUs and 10GB of memory. These experiments took ≈45 days by concurrently running them on the 10 virtual machines.

3.4.3 Study subjects

We evaluate FuzzSDN by testing two actual SDN controllers, i.e., ONOS [82] and RYU [83], which are still maintained actively and have been widely used in both research and practice [34, 35, 38, 37, 53, 51]. Both controllers are implemented using the OpenFlow SDN protocol specification [25]. Since FuzzSDN fuzzes OpenFlow control messages, it can test any SDN controller that implements the OpenFlow specification. Regarding virtual networks, we synthesize five networks with 1, 3, 5, 7, and 9 switches controlled by either ONOS or RYU. In each network, all switches are interconnected with one another, i.e., fully connected topology. Each switch is connected to two hosts that can emulate any device that sends and receives data streams, e.g., video and sound streams. We note that our study subjects, i.e., 5×2 SDN-systems built on the five networks controlled by ONOS and RYU, are representative of existing SDN studies and real-world SDNs. For example, DELTA [34] (resp. BEADS [35]) was evaluated with an SDN-system including two (resp. three) switches controlled by ONOS and RYU, as running experiments with SDNs requires large computational resources [95]. Shin et al. [21] introduced an industrial SDN-system developed in collaboration with SES, a satellite operator, which contains seven switches controlled by ONOS.

3.4.4 Experimental setup

EXP1. To answer RQ1, we compare FuzzSDN with DELTA [34] and BEADS [35], which are applicable to our study subjects. DELTA is a security assessment framework for SDNs that enables engineers to automatically reproduce known SDN-related attack scenarios and discover new attack scenarios. For the latter, DELTA relies on random fuzzing that randomizes all fields of a control message without accounting for the specifics of the OpenFlow protocol. BEADS is an automated attack discovery technique based on fuzzing that assumes the OpenFlow protocol, aiming at generating fuzzed control messages that can pass beyond the message parsing layer of the system under test. We note that we reused the implementations available online. However, we had to adapt them in order to make them work in our experiments, though we minimized changes, since the original executables of DELTA and BEADS did not work even after discussions with the authors.

In EXP1, we use two synthetic systems with one switch controlled by either ONOS or RYU. For the test procedure (see Section 3.3) in EXP1, we use the pairwise ping test [96], applied in many SDN studies [34, 35, 75, 54], that allows us to detect whether or not hosts can communicate with one another. Regarding the failure detection mechanism (see Section 3.3) in EXP1, we consider switch disconnections as system failures since both DELTA and BEADS analyzed switch disconnections in their experiments. We further note that EXP1 identifies switch disconnections as failures only when they lead to a communication breakdown and the system fails to locate the causes of the failures, i.e., no relevant log messages related to the failures. EXP1 fuzzes the packet_in message [25], which has 57 bytes encoded in 30 fields, as SDN switches send this

message to the controller in the execution of the test procedure, and both DELTA and BEADS fuzz it. For details of OpenFlow messages, we refer readers to the OpenFlow specification [25]. We compare the number of fuzzed control messages that cause the switch disconnection failure across fuzzing approaches.

EXP2. To answer RQ2, we evaluate the accuracy of failure-inducing models inferred by FuzzSDN. To this end, we compare the models obtained by FuzzSDN with those produced by our baselines extending DELTA and BEADS. In addition, we examine our failure-inducing models in light of the literature [34, 35] discussing failure-inducing conditions.

EXP2.1. As baselines, we extend DELTA and BEADS, named DELTA^L and BEADS^L, to produce failure-inducing models. DELTA^L (resp. BEADS^L) encodes the fuzzing results obtained by DELTA (resp. BEADS^L) as a training dataset (see the dataset format described in Section 3.3.1) and uses RIPPER to learn a failure-inducing model from it. Unlike FuzzSDN, DELTA^L and BEADS^L do not leverage the inferred failure-inducing models to guide their fuzzing.

For ensuring fair comparisons of FuzzSDN, DELTA^L, and BEADS^L, EXP2.1 creates a test dataset containing 5000 fuzzing results for each method. EXP2.1 then measures the accuracy of the failure-inducing models obtained by the three methods using the standard precision and recall metrics [85]. Additionally, EXP2.1 measures imbalance ratios of the datasets obtained at each iteration of the three methods using the imbalance ratio metric [85].

We compute precision and recall values as follows: (1) precision $P = TP/(TP+FP)$ and (2) recall $R = TP/(TP+FN)$, where TP, FP, and FN denote the number of true positives, false positives, and false negatives, respectively. A true positive is a control message labeled with *presence* (see Section 3.3.1) and correctly classified as such. A false positive is a control message labeled with *absence* (see Section 3.3.1) but incorrectly classified as *presence*. A false negative is a control message labeled with *presence* but incorrectly classified as *absence*. We compute the imbalance ratio of a dataset as follows: imbalance ratio $I = 1 - (minor/major)$, where *minor* and *major* denote the number of control messages in the dataset D, labeled with the minority and majority class, respectively. In EXP2.1, we use two synthetic systems with one switch controlled by either ONOS or RYU. EXP2.1 applies the pairwise ping test and fuzzes 8000 packet_in messages with FuzzSDN, DELTA^L, and BEADS^L.

EXP2.2. Jero et al. [35] manually inspected their SDN testing results obtained with BEADS and identified some conditions on message fields that led to system failures. EXP2.2 examines our failure-inducing models to assess the extent to which our models are consistent with their manual analysis results.

For EXP2.2, we use two synthetic systems with one switch controlled by either ONOS or RYU. EXP2.2 fuzzes the following five types of control messages: packet_in (57 bytes, 30 fields), hello (8 bytes, 4 fields), barrier_reply (8 bytes, 4 fields), barrier_request (8 bytes, 4 fields), and

flow_removed (55 bytes, 22 fields), which are manipulated by BEADS. We randomly selected these five message types from the 16 types of control messages analyzed in the prior study of BEADS, to keep the expected cost of running our experiments manageable (see Section 3.4.5). In EXP2.2, FuzzSDN fuzzes 8000 control messages of each type. To fuzz the barrier_request and the barrier_reply control messages, we use a test procedure that connects switches to an SDN controller as it generates the control messages. For the remaining types of control messages, we use the pairwise ping test [96]. Regarding failure types, EXP2.2 detects unexpected broadcasts from switches and unexpected switch disconnections as failures. These are studied in existing work (BEADS). The broadcasting mechanism of ARP (Address Resolution Protocol) [97] is used to discover host locations in the SDN. If broadcasting occurs unexpectedly, it may lead to the installation of incorrect forwarding rules on the switches in the SDN, possibly resulting in information leakage (since data can be forwarded to unintended hosts) or connectivity losses. Regarding the other types of failure, if the communication between the controller and the switches is unexpectedly disconnected, the SDN-system obviously cannot operate as intended. This is because the controller monitors the network status based on the messages received from the switches and the behavior of the switches is directed by the controller.

EXP3. To answer RQ3, we study the correlation between the execution time of FuzzSDN and the 2×5 synthetic systems (described in Section 3.4.3) with 1, 3, 5, 7, and 9 switches controlled by either ONOS or RYU, respectively. We measure the execution time of each iteration of FuzzSDN and the execution time of configuring Mininet and the SDN controllers. Our conjecture is that the execution time of FuzzSDN does not depend on network sizes. However, we also conjecture that there is a correlation between the configuration time of Mininet and SDN controllers and network sizes. Such configuration time includes the time for initializing controllers and Mininet, creating virtual networks, and activating controllers. EXP3 uses the pairwise ping test.

3.4.5 Parameter Tuning

Recall from Section 3.3 that FuzzSDN must be configured with the following parameters: number of control messages to be fuzzed at each iteration, mutation rate, and RIPPER parameters. For tuning the parameters, we ran initial experiments relying on hyperparameter optimization [85] based on guidelines in the literature [85, 98]. In our initial experiments, we assessed 10 configurations of the parameters' values to select the best one to be used in further experiments. We selected these 10 configurations using a grid search [85]. To select the best configuration, for each configuration, we ran FuzzSDN for four days to ensure there were no notable changes in the results and measured the precision and recall values of the obtained failure-inducing model. For our experiments, we set the number of control messages to be fuzzed at each iteration to 200 and the mutation rate to $1/|F|$, where $|F|$ denotes the number of fields in a control message to be fuzzed. The parameter values of RIPPER used in our experiments can be found in our repository [84].

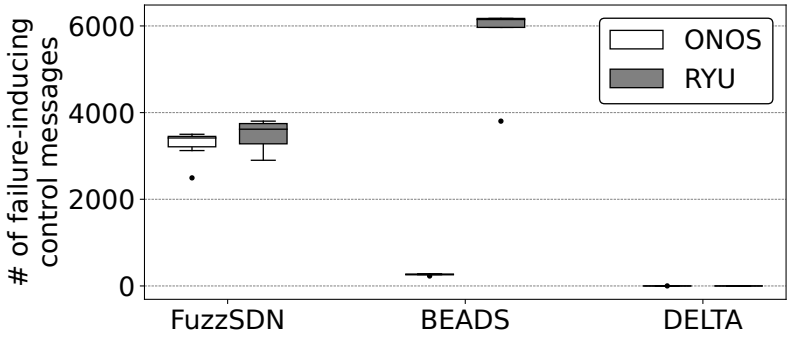


Figure 3.3: Comparing FuzzSDN, BEADS, and DELTA based on the number of fuzzed control messages that cause the switch disconnection failure. The boxplots (25%-50%-75%) show distributions of the numbers of failure-inducing control messages obtained from 10 runs of EXP1, testing either ONOS or RYU.

To fairly compare FuzzSDN and the other approaches (i.e., DELTA, BEADS, DELTA^L, BEADS^L), we assign to them the same computation budget: four days for ONOS and two days for RYU. Within this budget, FuzzSDN generates a balanced dataset (described in Section 3.3.2), and precision and recall values of the inferred failure-inducing models reach their plateaus. We note that the configuration time of RYU to run FuzzSDN is approximately half that of ONOS. Hence, we set different budgets so that FuzzSDN fuzzes similar numbers of control messages for ONOS and RYU. Since FuzzSDN is randomized, we repeat our experiments 10 times.

The parameters of FuzzSDN and our experiments can certainly be further tuned to improve the accuracy of FuzzSDN. However, we were able to convincingly and clearly support our conclusions with the selected configuration, using the study subjects (described in Section 3.4.3). Hence, this chapter does not report further experiments on optimizing those parameters.

3.4.6 Experiment Results

RQ1. Figure 3.3 compares FuzzSDN, BEADS and DELTA when testing the two study subjects controlled by either ONOS or RYU (EXP1). The boxplots depict distributions (25%-50%-75% quantiles) of the numbers of control messages that cause the switch disconnection failure. The results shown in the figure are obtained from 10 runs of EXP1. To fairly compare FuzzSDN, BEADS, and DELTA, they were assigned the same computation budget: four days for ONOS and two days for RYU (as described in Section 3.4.5).

As shown in Figure 3.3, for ONOS, FuzzSDN generates significantly more failure-inducing control messages than BEADS and DELTA. On average, 3425 failure-inducing control messages are generated by FuzzSDN, in contrast to 270 by BEADS and 3 by DELTA. Most of the control messages manipulated by DELTA are filtered out by the message parsing layers of the controllers, which is consistent with the finding reported in the BEADS study [35]. Regarding the application of BEADS to RYU, the situation is apparently more complicated. RYU is in fact much less

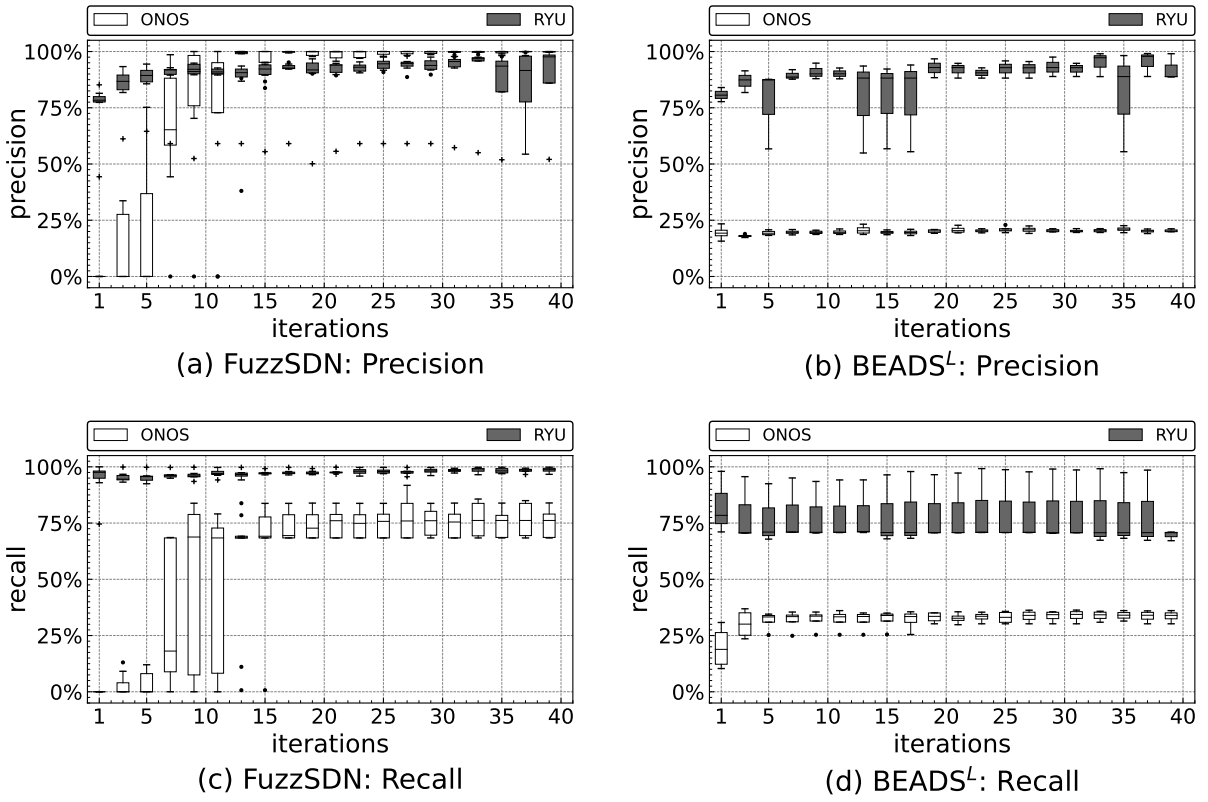


Figure 3.4: Comparing distributions of precision and recall values obtained from FuzzSDN and BEADS^L that test the systems controlled by either ONOS or RYU (see EXP2.1). The boxplots (25%-50%-75%) show distributions of precision (a, b) and recall (c, d) values obtained from 10 runs of EXP2.1.

robust than ONOS when handling fuzzed control messages. It is therefore easy to fuzz messages leading to failures with RYU. But recall that FuzzSDN aims at generating a balanced labeled dataset containing control messages that are associated with both the *presence* and *absence* of failures in similar proportions (Section 3.3.2). This is not the case of BEADS which then generates a very large proportion of failure-inducing control messages with RYU, more than that observed with FuzzSDN.

The answer to **RQ1** *is that* FuzzSDN significantly outperforms BEADS and DELTA. In particular, our experiments show that FuzzSDN is able to generate a much larger number of control messages that cause failures when the SDN controller is relatively robust to fuzzed messages (e.g., ONOS). Such robustness is a desirable and common feature in industrial SDN controllers.

RQ2. Figure 3.4 compares precision (a, b) and recall (c, d) values obtained from FuzzSDN and BEADS^L for the study subjects controlled by either ONOS or RYU and the test dataset (EXP2.1). The boxplots in Figures 3.4(a) and 3.4(b) (resp. 3.4(c) and 3.4(d)) show distributions (25%-50%-75% quantiles) of precision (resp. recall) values over 40 iterations of the methods obtained from 10 runs of EXP2.1. Note that each iteration of BEADS^L adds 200 fuzzed control messages (the same as FuzzSDN) to a dataset and learns a failure-inducing model. In contrast to FuzzSDN, BEADS^L does not use the failure-inducing model to guide fuzzing. We omit the results

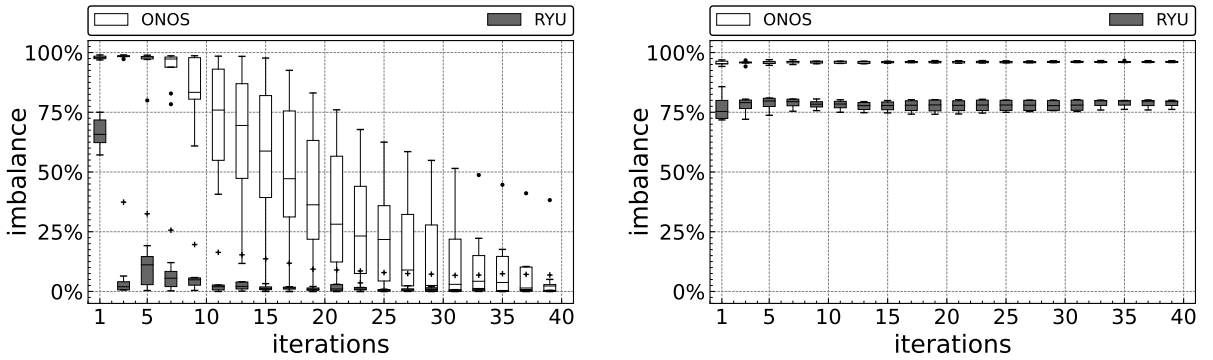


Figure 3.5: Comparing distributions of imbalance ratios obtained from FuzzSDN and BEADS^L that test the synthetic systems controlled by either ONOS or RYU (see EXP2.1). The boxplots (25%-50%-75%) show distributions of imbalance ratios obtained from 10 runs of EXP2.1.

obtained by DELTA^L because the labeled dataset it created contains only a few failure-inducing control messages, as reported in RQ1.

As shown in Figure 3.4, the failure-inducing models obtained by FuzzSDN yield higher precision and recall than those obtained by BEADS^L over 40 iterations. The results show that, after 20 iterations, there are no notable changes in precision and recall values. Specifically, for ONOS (resp. RYU), FuzzSDN achieves, on average, a precision of 99.8% (resp. 95.4%) and a recall of 75.5% (resp. 96.7%) after 20 iterations. In contrast, BEADS^L achieves, for ONOS (resp. RYU), on average, a precision of 20.9% (resp. 90.5%) and a recall of 29.9% (resp. 70.7%) after 20 iterations. The 20 iterations of FuzzSDN took, on average, 2.33 days for ONOS and 1.10 days for RYU.

Figure 3.5 shows the comparison of imbalance ratios for datasets obtained from FuzzSDN and BEADS^L that test the study subjects controlled by either ONOS or RYU (EXP2.1). The boxplots depict distributions of imbalance ratios over 40 iterations of the methods, compiled from 10 runs of EXP2.1. Note that the lower the imbalance ratio, the more balanced the dataset. In Figure 3.5, we omitted the results obtained by DELTA^L as they provide no additional findings, which are discussed below.

As shown in Figure 3.5, the datasets generated by FuzzSDN are significantly more balanced than those generated by BEADS^L over 40 iterations. Specifically, after 40 iterations, FuzzSDN achieves, on average, an imbalance ratio of 5.47% for ONOS and 1.38% for RYU. In contrast, BEADS^L achieves, on average, an imbalance ratio of 96.2% for ONOS and 78.9% for RYU. DELTA^L produces datasets that are even more imbalanced than the other two methods, with, on average, an imbalance ratio of 99.9% for both ONOS and RYU over 40 iterations. The results thus indicate that FuzzSDN effectively addresses the imbalance problem over iterations, leading to accurate characterization of failure-inducing models (see the precision and recall results in Figure 3.4). However, BEADS^L and DELTA^L, which do not tackle the imbalance problem,

Table 3.2: Summary of the EXP2.2 results. Five types of control messages are fuzzed for each experiment with the system controlled by ONOS.

message type	message size	# rules (FuzzSDN)	# fields (FuzzSDN)	# fields (manual)	all included?
packet_in	57b,30f	32	12	3	yes
hello	8b,4f	21	4	3	yes
flow_removed	55b,22f	12	4	3	yes
barrier_request	8b,4f	8	4	3	yes
barrier_reply	8b,4f	3	3	3	yes

b: bytes, f: fields

produce highly imbalanced datasets across all iterations, leading to lower precision and recall of failure-inducing models (see Figure 3.4).

Table 3.2 presents the summary of our experiment results obtained from EXP2.2. In the experiments, recall that FuzzSDN fuzzes the following five types of control messages: packet_in, hello, flow_removed, barrier_request, and barrier_reply. For each control message type, the table shows the message size, the number of rules generated by FuzzSDN, the number of fields in the inferred rules, and the number of fields that appear in failure-inducing strategies reported in a prior study [35]. Such strategies were manually defined by analysts, e.g., the switch disconnection failure can occur when changing the version, type, and length fields in a packet_in message. The "all included?" column in the table indicates whether or not the fields in the existing failure-inducing strategies appear in the failure-inducing model obtained by FuzzSDN.

From Table 3.2, we see that, for each failure case, all fields in the corresponding failure-inducing strategy described in existing work appear in the failure-inducing models obtained by FuzzSDN. Hence, the results show that FuzzSDN does not miss any important field related to system failures. However, FuzzSDN discovers more fields relevant to failures than the ones reported in prior work. For example, FuzzSDN found 32 rules with 12 fields for the packet_in experiment. Nevertheless, we believe that inspecting those 32 precise rules is considerably more efficient for understanding failure-inducing conditions than manually inspecting 8000 fuzzed control messages. Regarding our results for RYU, we refer the reader to our repository [84] since our findings are similar to those of ONOS.

The answer to **RQ2** *is that* FuzzSDN generates accurate failure-inducing models in practical time, thus faithfully characterizing failure conditions. In addition, the failure analysis results reported in the literature are consistent with the failure-inducing models produced by FuzzSDN.

RQ3. Figure 3.6 shows the mean execution times of the planning, fuzzing, and learning steps of FuzzSDN and the mean configuration times of Mininet and ONOS for each iteration of FuzzSDN. The figure presents the results obtained from EXP3 using five study subjects with 1, 3, 5, 7, and 9 switches controlled by ONOS. Note that the y-axis of the figure is a 10-base log scale in seconds.

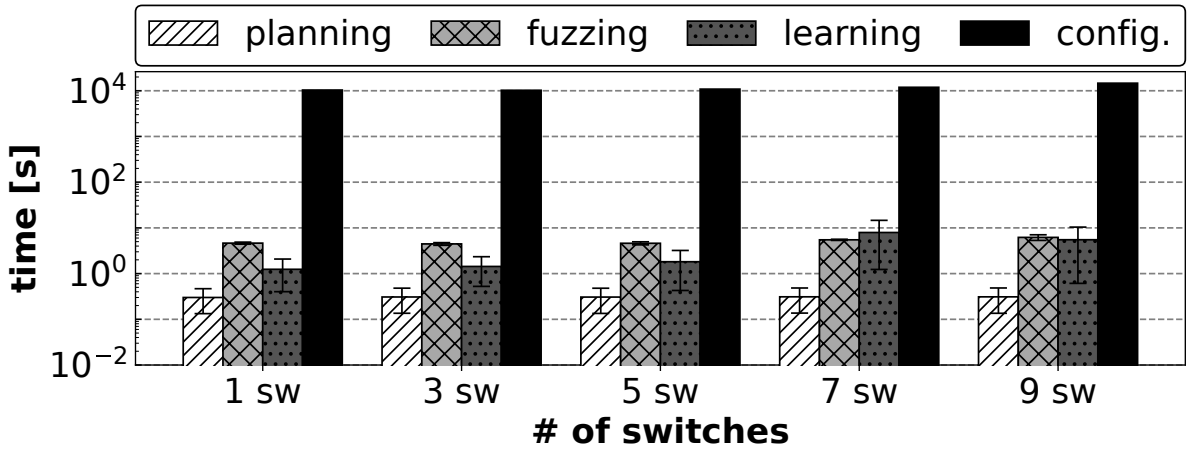


Figure 3.6: Comparing execution times of FuzzSDN when varying network sizes as follows: 1, 3, 5, 7, and 9 switches (see EXP3). The bars show the mean execution times of each step of FuzzSDN and the mean configuration times of ONOS and Mininet computed based on 40 iterations of FuzzSDN. The vertical lines on the bars show the standard errors of the mean values.

As shown in Figure 3.6, the execution times of the three FuzzSDN steps do not depend on the network sizes. For each FuzzSDN iteration that generates, on average, 200 fuzzed control messages, the planning step takes 300ms, the fuzzing step 5.1s, and the learning step 3.5s, which align with our expectations for executing them in practical time. However, 200 configurations of Mininet and ONOS at each iteration takes, on average, 2.85h with 1 switch, 3.00h with 3 switches, 3.02h with 5 switches, 3.31h with 7 switches, and 4.07h with 9 switches, which dominate the overall testing times. Recall that the configuration time of RYU takes approximately half that of ONOS. Even though the configuration times of an SDN controller and Mininet are not in the scope of our study, the results indicate a technical bottleneck to be further investigated to shorten testing time. Regarding the memory requirement, our experiments consumed, at most, 1.2GB of memory, including FuzzSDN, an SDN controller, and the simulation platform. In particular, ONOS used ≈1GB of memory and Mininet ≈15MB of memory per switch. Hence, FuzzSDN does not add significant overhead to the current simulation practice for SDNs.

The answer to **RQ3** *is that* the execution time of FuzzSDN has no correlation with the size of network. Hence, FuzzSDN is applicable to complex systems with large networks.

3.4.7 Threats to Validity

Internal validity. To mitigate potential internal threats to validity, our experiments compared FuzzSDN with two state-of-the-art solutions (DELTA and BEADS) that generate failure-inducing control messages for testing SDN controllers. Though DELTA and BEADS were our best options, they do not generate failure-inducing models. Therefore, we extended them to produce failure-inducing models and support the comparison of FuzzSDN with these baselines.

External validity. The main concern regarding external validity is the possibility that our results may not generalize to different contexts. In our experiments, we applied FuzzSDN to several SDNs and two actively maintained SDN controllers, i.e., ONOS and RYU. We ensured that our synthetic SDN-systems, consisting of five networks with 1, 3, 5, 7, and 9 switches, controlled by either ONOS or RYU, are representative of existing SDN studies and real-world SDN-systems (e.g., emergency management systems [21]). Further, the SDN-systems used in our experiments are more complex and larger than those previously used to assess DELTA and BEADS, which contain at most three switches. In general, the performance of SDN fuzzing techniques, such as FuzzSDN, DELTA, and BEADS, is not correlated with the size of SDN, as these techniques sniff and modify a control message passing through the control channel between the SDN switches and the controller. Specifically, sniffing a network interface and fuzzing a network packet that passes through the interface does not depend on the size of an SDN-system. To evaluate FuzzSDN in a realistic setting, we used actual SDN controllers and switch software while emulating only the physical networks, including links, host devices, and switch devices.

The prototype implementation of FuzzSDN supports OpenFlow, which is a de facto standard SDN protocol used in many SDN-systems [34, 35, 38, 54, 53, 37]. As a result, we were able to apply FuzzSDN to actual SDN controllers (i.e., ONOS and RYU) and compare it with existing tools (i.e., DELTA and BEADS), given their support for OpenFlow. To apply FuzzSDN to SDN-systems that employ other SDN protocols, such as Cisco OpFlex [99] and ForCES [100], one must adapt the sniffing and injecting steps of FuzzSDN (see Figure 3.2) to correctly decode and encode control messages, respectively. However, this adaptation does not affect the fuzzing, learning, and planning steps of FuzzSDN. We therefore expect that, while the adaptation requires engineering effort to update the sniffing and injecting steps, it does not impact FuzzSDN's performance.

FuzzSDN is developed to be generally applicable to any SDN-system. Our evaluation package and the FuzzSDN tool are available online [84] to (1) facilitate reproducibility of our experiments and (2) enable researchers and practitioners to use and adapt FuzzSDN. Nevertheless, further case studies in other contexts, including industry SDN-systems that employ different SDN protocols, as well as user studies involving practitioners, remain necessary to further investigate the generalizability of our results.

Limitations. FuzzSDN requires users to provide a test procedure (e.g., pairwise ping test) and select a control message (e.g., packet_in) to be fuzzed. These design choices enable FuzzSDN to generate failure-inducing messages and models within reasonable time budgets (e.g., 4 days for ONOS and 2 days for RYU). Automatically exploring possible use scenarios (i.e., test procedures) and sequences of messages, while accounting for state changes in the SDN controller, can help engineers reduce their manual efforts in testing (e.g., providing test procedures and selecting a message to be fuzzed). However, efficiently and effectively exploring such spaces is a hard problem that requires further research. Given its promising results, we believe that FuzzSDN is

nevertheless a practical solution and serves as a solid foundation for researchers and practitioners to further enhance automation in testing SDN controllers.

3.5 Related Work

In this section, we discuss related work in the areas of SDN testing, fuzzing, and characterizing failure-inducing inputs.

SDN testing. Testing SDNs has been primarily studied in the networking literature targeting various objectives, such as detecting security vulnerabilities and attacks [49, 50, 35, 34, 51, 36, 52], identifying inconsistencies among the SDN components (i.e., applications, controllers, and switches) [53, 37, 38], and analyzing SDN executions [54, 55, 56]. Among these, we discuss SDN testing techniques that rely on fuzzing, as they constitute the most closely related research. Woo et al. [57] proposed RE-CHECKER to fuzz RESTful services provided by SDN controllers. RE-CHECKER fuzzes an input file, encoded in JSON format, that a network administrator uses to specify network policies (e.g., data forwarding rules). This results in generating numerous malformed REST messages for testing RESTful services in SDN. Dixit et al. [58] presented AIM-SDN to test the implementation of the network management datastore architecture (NMDA) [59] in SDN. AIM-SDN randomly fuzzes REST messages to test the NMDA implementation in SDN with regard to the availability, integrity, and confidentiality of datastores. Shukla et al. [38] developed PAZZ that aims at detecting faults in SDN switches by fuzzing data packet headers, e.g., IPv4 and IPv6 headers. Albab et al. [60] presented SwitchV to validate the behaviors of SDN switches. SwitchV uses fuzzing and symbolic execution to analyze the p4 [61] models that specify the behaviors of SDN switches. Lee et al. [37, 62] introduced AudiSDN that employs fuzzing to detect policy inconsistencies among SDN components (i.e., controllers and switches). AudiSDN fuzzes network policies submitted by administrators through the REST APIs. To increase the likelihood of discovering inconsistencies, AudiSDN employs rule dependency trees derived from the OpenFlow specification, which restrict valid relationships among rule elements. In contrast to these existing methods, FuzzSDN fuzzes SDN control messages to test SDN controllers (as DELTA and BEADS do). Furthermore, FuzzSDN uses ML to guide fuzzing and learn failure-inducing models.

Fuzzing. To efficiently generate effective test data, fuzzing has been widely applied in many application domains [101]. The research strands that most closely relate to our work are fuzzing techniques based on ML for testing networked systems [79, 80]. Chen et al. [79] proposed a fuzzing technique for testing cyber-physical systems, which contain sensors and actuators distributed over a network. The proposed approach relies on a deep learning technique to fuzz actuators' commands that can drive the CPS under test into unsafe physical states. Zhao et al. [80] developed SeqFuzzer that enables engineers to test communication systems without prior knowledge of the systems' communication protocols. SeqFuzzer infers communication protocols using a deep learning technique and generates test data based on the inferred protocols.

Unlike these prior research threads, FuzzSDN applies ML and fuzzing in the context of testing SDN-systems. In addition, FuzzSDN employs an interpretable ML technique to provide engineers with comprehensible failure-inducing models.

Besides the fuzzing techniques mentioned above that leverage ML for testing networked systems, coverage-aware fuzzing techniques have been used in many studies across different domains [42, 43, 44]. For example, AFL [42] is a mutational, coverage-guided fuzzer that uses compile-time instrumentation and genetic algorithms to automatically generate test cases that uncover previously unexplored internal states in the program under test. AFL++ [43] is a fuzzing framework that expands on AFL, incorporating many state-of-the-art techniques that enhance fuzzing performance, thus enabling researchers to evaluate different combinations of such techniques. However, applying such fuzzing tools, developed in other contexts, to test SDN controllers is far from being straightforward as there are differences in inputs, outputs, and states. For example, the notion of coverage, e.g., statements and branches, used in coverage-aware fuzzing tools is suitable for testing stateless programs, where outputs depend solely on inputs and do not depend on any memory of past interactions. However, an SDN controller is a stateful program. It takes as input sequences of control messages from the switches in the network, processes these messages, and outputs appropriate sequences of responses. Note that the controller's response is determined by the currently received message and its internal state, which is determined by previously processed messages. AFLNet [44], which extends AFL to test servers, is proposed to address this issue by utilizing state coverage rooted in finite state machines. In AFLNet, finite state machines are constructed using the response codes (also known as status codes) from network protocols, which indicate the result of a client's request to a server. However, AFLNet is not applicable when such response codes are not available, as in our context. Hence, we need a different notion of coverage when testing SDN controllers. In addition to the coverage issue, existing techniques that enhance fuzzing performance for single programs are not easily applicable to testing SDN controllers. For example, the forkserver technique implemented in AFL++, which uses the fork mechanism to reduce the high cost of initialization, is not applicable to our context. Since an SDN controller interacts with multiple switches connected to various hosts, testing the controller requires the costly initialization of not only the controller but also the other components in the SDN-system. Further, testing the controller impacts the states of these other components, making it difficult to efficiently reset and maintain a consistent testing environment. While coverage-aware fuzzing tools, such as AFL++, provide significant advances in testing stateless programs, applying them to test SDN controllers therefore raises difficult challenges.

Characterizing failure-inducing inputs. Recently, a few research strands aimed at characterizing input conditions under which a system under test fails [15, 81]. Gopinath et al. [15] introduced DDTEST that abstracts failure-inducing inputs. DDTEST aims at testing software programs, e.g., JavaScript translators and command-line utilities, that take as input strings. For abstracting failure-inducing inputs, DDTEST uses a derivation tree that depicts how failure-inducing strings can be derived. Kampmann et al. [81] presented ALHAZEN that

learns circumstances under which the software program under test fails. ALHAZEN also targets software programs that process strings. ALHAZEN relies on ML to learn failure-inducing circumstances in the form of decision trees. Compared to our work, we note that the context of these research threads is significantly different from our application context: SDNs. Hence, they are not applicable to generate test data and learn failure-inducing models for SDN controllers. To our knowledge, FuzzSDN is the first attempt that applies ML for guiding fuzzing and learning failure-inducing models by accounting for the specificities of SDNs.

3.6 Conclusions

We developed FuzzSDN, an ML-guided fuzzing method for testing SDN-systems. FuzzSDN employs ML to guide fuzzing in order to (1) generate a set of test data, i.e., fuzzed control messages, leading to failures and (2) learn failure-inducing models that describe conditions when the system is likely to fail. FuzzSDN implements an iterative process that fuzzes control messages, learns failure-inducing models, and plans how to better guide fuzzing in the next iteration based on the learned models. We evaluated FuzzSDN on several synthetic SDN-systems controlled by either one of two SDN controllers. Our results indicate that FuzzSDN is able to generate effective test data that cause system failures and produce accurate failure-inducing models. Furthermore, FuzzSDN's performance does not depend on the network size and is hence applicable to systems with large networks.

In the future, we plan to extend FuzzSDN to account for sequences of control messages. Fuzzing multiple control messages at once poses new challenges due to message intervals, message dependencies, and state changes in the system under test. To learn failure-inducing models from control message sequences, we will further investigate ML techniques that process sequential data. In addition, we will extend FuzzSDN to generate diverse test cases, defined by message sequences, aiming at maximizing state coverage during testing of SDN controllers. To this end, we will conduct further research to define and quantify the diversity of test cases and the state coverage of SDN controllers, and incorporate these into our fuzzing framework. In the long term, we plan to further validate the generalizability and usefulness of FuzzSDN by applying it to additional SDN-systems and conducting user studies.

Data Availability

Our evaluation package and the FuzzSDN tool are available online [84] to (1) increase the reproducibility of our experiments and (2) enable researchers and practitioners to use and adapt FuzzSDN.

Chapter 4

Learning-Guided Fuzzing for Testing Stateful SDN Controllers

4.1 Introduction

An SDN controller is a stateful software component that maintains a holistic view of the SDN, capturing the state information of network devices, links, and the controller itself. This enables the controller to provide dynamic network operations in an efficient and effective manner. However, testing stateful SDN controllers is challenging. An SDN controller interacts with multiple network devices through sequences of inbound and outbound control messages defined in the underlying SDN communication protocol (e.g., OpenFlow [25]). If a failure can occur only in a certain state of an SDN controller, discovering such a stateful failure requires engineers to identify message sequences that bring the controller into that state. However, discovering such stateful failures is a hard problem due to the potentially infinite number of possible sequences of control messages. This is because the size of sequences is unbounded, and there are various types of control messages with different sizes. In addition, even if engineers obtain sequences of control messages that cause failures, manual inspection of these sequences is time-consuming and error-prone. This may result in misunderstandings of the causes of failures and hence the application of unreliable fixes.

Fuzzing techniques have been widely applied for testing various network systems [48, 47, 34, 35]. Among these, state-aware fuzzing techniques that do not depend on protocol specifications could be considered for testing SDN controllers, as, to our knowledge, no existing state-aware fuzzing techniques account for the specificities (e.g., architecture and protocol) of SDNs. For example, AFLNet [47] constructs finite state machines (FSMs) based on the response codes generated by

the server under test and uses these FSMs to guide the fuzzing process. AFLNet employs common byte-level fuzz operators, such as bit flipping as well as the insertion, deletion, and substitution of byte blocks. However, AFLNet operates under the working assumption that communication protocols embed special codes in response messages, which is not always the case, as in our SDN context. StateAFL [48] infers FSMs based on the in-memory states of the server, leveraging compile-time instrumentation and fuzzy hashing techniques; hence, it does not require response codes. During the fuzzing process, StateAFL guides the generation of new inputs to the server based on the inferred FSMs. It employs both byte-level and message-level fuzz operators, which do not rely on protocol specifications. NSFuzz [102] uses a combination of static analysis and manual annotation on the server's source code to identify states based on program variables and construct FSMs that capture the transitions between these states. It then performs FSM-guided fuzzing using fuzz operators similar to those in AFLNet. However, the state-aware fuzzing techniques introduced in this research strand are applicable to the server-client architecture by replacing a client with a fuzzer. The fuzzer replays captured message sequences and modifies them during the fuzzing process. In contrast, the SDN architecture differs significantly from the server-client architecture. For example, in the SDN architecture, communication is initiated between an SDN controller and switches, whereas in the server-client architecture, clients typically initiate requests to the server. Additionally, SDN switches also communicate with one another to enable network communication and services. Therefore, replacing an SDN switch with a fuzzer for testing an SDN controller is challenging. Furthermore, the working assumptions of these techniques, such as response codes, compile-time instrumentation, and source-code analysis and annotation, make them difficult to apply when testing an SDN controller. SDN operators are more concerned with potential failures that can occur in realistic scenarios, such as when a malicious user intercepts messages and disrupts the SDN during its operation [34, 35, 103, 54, 75].

There are some prior studies [34, 35, 103] that test SDN controllers by taking into account the architecture and protocols of SDNs. For example, DELTA [34] is a security assessment framework for SDNs. It reproduces existing SDN-related attack scenarios and uncovers new ones through fuzzing. Specifically, in fuzzing, DELTA modifies control messages by treating them as byte streams and randomising them. BEADS [35] is an automated attack discovery tool for SDNs. In contrast to DELTA, BEADS fuzzes control messages while adhering to the SDN protocol (i.e., OpenFlow), aiming to create test scenarios that can exercise components beyond the protocol parsers of SDN controllers. FuzzSDN [103] also adheres to the SDN protocol in its fuzzing process to test components beyond the protocol parsers of an SDN controller. In addition, FuzzSDN employs machine learning techniques to infer failure-inducing models that characterise the conditions under which failures occur, and uses them to guide the fuzzing. These techniques position their fuzzers between the SDN controller and the SDN switches to sniff and modify control messages, leveraging the man-in-the-middle attack strategy [104]. Hence, they do not require any modifications, replacements, annotations, or instrumentation of the components (i.e., switches and controllers) in SDNs, enabling the testing of SDN controllers in a realistic setting. However, these techniques, which account for the architecture and protocols of SDNs, do not consider the stateful nature of SDN controllers.

Contributions. In this chapter, we propose SeqFuzzSDN, a learning-guided fuzzing method for testing stateful SDN controllers. SeqFuzzSDN aligns with the aforementioned research strand that leverages the architecture and protocols of SDNs. Hence, SeqFuzzSDN test SDN controllers in a realistic operational setting without requiring any compile-time instrumentation, manual annotation of source code, and replacing an SDN switch with a fuzzer. Instead, SeqFuzzSDN sniffs and fuzzes control messages exchanged between the SDN controller and switches by being aware of the stateful behaviours of the controller. SeqFuzzSDN employs a fuzzing strategy guided by Extended Finite State machines (EFSMs) in order to (1) efficiently explore the space of states of the SDN controller under test, (2) generate effective and diverse tests (i.e., message sequences) to uncover failures, and (3) infer accurate EFSMs that characterise the sequences of control messages leading to failures. Note that since the SDN communication protocol specifies various message fields, their values, and relations, guard conditions on state transitions in EFSMs are well-suited to capture state changes associated with these message fields, values, and relations.

We evaluated SeqFuzzSDN by applying it to two well-known open-source SDN controllers: ONOS [82] and RYU [83]. Additionally, we compared SeqFuzzSDN against our extensions of three state-of-the-art (SOTA) methods—DELTA [34], BEADS [35], and FuzzSDN [103]—which were used as baselines for generating tests for SDN controllers. We extended DELTA, BEADS, and FuzzSDN to produce EFSM models, since these SOTA methods were not originally designed to generate such models. It is important to note that, these three baselines are the best available options for evaluating SeqFuzzSDN when testing SDN controllers by fuzzing control messages. Our experiment results show that SeqFuzzSDN significantly outperforms the three baselines. Specifically, compared to the baselines, SeqFuzzSDN generates more diverse and effective tests (i.e., message sequences) that lead to failures, as well as more accurate EFSMs that characterise failure-inducing message sequences. In addition, SeqFuzzSDN can be applied to large SDNs since its performance is independent of the network size. Our complete evaluation results and the SeqFuzzSDN tool can be accessed online [105].

Organisation. The remainder of this chapter is structured as follows: Section 4.2 provides the background and defines the specific problem of testing stateful SDN controllers. Section 4.3 details the steps of SeqFuzzSDN. Section 4.4 presents the empirical evaluation of SeqFuzzSDN. Section 4.5 compares SeqFuzzSDN with related work. Finally, Section 4.6 concludes the chapter.

4.2 Background and problem description

Message sequences. In the control plane of an SDN, an SDN controller exchanges sequences of control messages with SDN switches to establish and manage communication among hosts, monitor network status, and enforce network policies. In the data plane of an SDN, hosts exchange sequences of data messages through SDN switches to transmit and receive various types of data, such as audio and video streams. For example, Table 4.1 presents an example sequence

Table 4.1: An example sequence of messages for discovering host locations. The messages in this table are generated by the hosts, switches, and a controller depicted in Figure 2.1.

m_i	message	sender	receiver	channel
1	arp_req(host_d)	host_s	switch_1	data
2	pkt_in(arp_req(host_d))	switch_1	controller	control
3	pkt_out(arp_req(host_d),flood)	controller	switch_1	control
4	arp_req(host_d)	switch_1	switch_2	data
5	pkt_in(arp_req(host_d))	switch_2	controller	control
6	pkt_out(arp_req(host_d),flood)	controller	switch_2	control
7	arp_req(host_d)	switch_2	host_2	data
8	arp_req(host_d)	switch_2	host_3	data
9	arp_req(host_d)	switch_2	switch_3	data
10	pkt_in(arp_req(host_d))	switch_3	controller	control
11	pkt_out(arp_req(host_d),flood)	controller	switch_3	control
12	arp_req(host_d)	switch_3	host_d	data
13	arp_rep(host_d)	host_d	switch_3	data
14	pkt_in(arp_rep(host_d))	switch_3	controller	control
15	pkt_out(arp_rep(host_d),port2)	controller	switch_3	control
16	arp_rep(host_d)	switch_3	switch_2	data
17	pkt_in(arp_rep(host_d))	switch_2	controller	control
18	pkt_out(arp_rep(host_d),port3)	controller	switch_2	control
19	arp_rep(host_d)	switch_2	switch_1	data
20	pkt_in(arp_rep(host_d))	switch_1	controller	control
21	pkt_out(arp_rep(host_d),port1)	controller	switch_1	control
22	arp_rep(host_d)	switch_1	host_s	data

of messages aimed at discovering host locations (i.e., MAC addresses [27]) in the SDN network shown in Figure 2.1.

Regarding the example sequence listed in Table 4.1, we consider an SDN setup in which the controller in Figure 2.1 is unaware of a path across switches that enables the transmission of data messages from host_s to host_d. The address resolution protocol (ARP) is typically used to map an IP address of a host to its physical (MAC) address [26, 27]. The first ARP message m_1 generated by host_s is an ARP request aimed at obtaining the MAC address of host_d. The ARP request reaches to switch_1 that is connected to host_s. The switch then sends the ARP request to the controller by encapsulating it through the packet-in control message m_2. The controller is now aware of the information regarding the source of the ARP request, i.e, host_s. However, since the controller does not know the location of host_d, it instructs switch_1 to flood the ARP request to the connected switches using the packet-out message m_3. The ARP request is then flooded in the network (via m_4 to m_{11}) until it reaches the destination host_d (via m_{12}). The destination host_d then sends the ARP reply m_{13} to switch_3 in order to inform the source host_s of its location (MAC). Note that, at this stage, since the controller knows the location of host_s, it directly instructs the three switches with the exact directions (i.e., port numbers) to forward the ARP reply (see m_{14} to m_{22}). After this procedure, the controller usually installs forwarding rules for both ARP and IP messages to the switches, resulting in different sequences of messages compared to the example sequence mentioned above.

Failures. Like any software component, SDN systems are susceptible to failures that can affect their functionality. These failures may result in service disruptions noticeable to users. Numerous studies have examined these failures in the context of SDN testing [34, 35, 103, 37, 38] Furthermore, the centralisation of SDN system logic within its controller makes it a critical point of failure. A controller crash or loss of connection with the switches can disrupt the entire network. This vulnerability underscores the necessity for thorough testing to ensure the system's robustness and reliability. Such testing entails exploring the state space of the SDN system, including scenarios that are not easily reached. Unfortunately, no work has yet investigated how to automate such state space exploration in SDN systems. Recall from Section 4.1 that existing stateful fuzzing techniques [47, 48, 102] rely on certain working assumptions that are not valid in the context of SDN testing. In particular, the SDN architecture differs from the client-server architecture. Hence, there is no client in an SDN that can act as a fuzzer, which employs capturing, replaying, and fuzzing strategies to replace the client.

Problem. SDN controllers are inherently stateful. They manage complex states that encompass their internal states, the connected switches' states, and the overall network states. When developing and operating SDN systems, engineers must address system failures triggered by unexpected sequences of control messages. Specifically, they must ensure that the system behaves acceptably regardless of its current state. In SDN systems, a stateful controller is susceptible to entering incorrect states, sending unexpected messages, or triggering system failures. These failures may occur only when the controller, the connected switches, or the network reach specific states. For instance, an erroneous control message may be handled correctly under nominal conditions, but if the same message is transmitted during a state of 'recovery' of the system, a system failure may occur. When such a failure occurs, engineers must determine the state of the controller at the time of the failure and identify the sequence of messages that led to that state. Precisely identifying these conditions is crucial, as it enables engineers to diagnose the failure with a clear understanding of the conditions that caused it. Additionally, engineers can utilise this information to generate extended sets of control message sequences for testing the system after implementing fixes.

Our work aims to effectively test the state space of an SDN system's controller by identifying control messages that lead to system failures, and then automatically derive an accurate model that characterises the sequences of messages leading to a failure.

4.3 Approach

4.3.1 Overview

Figure 4.1 shows an overview of SeqFuzzSDN. SeqFuzzSDN takes as input a test procedure and a failure detection mechanism. The test procedure specifies the steps required to (1) initialise the

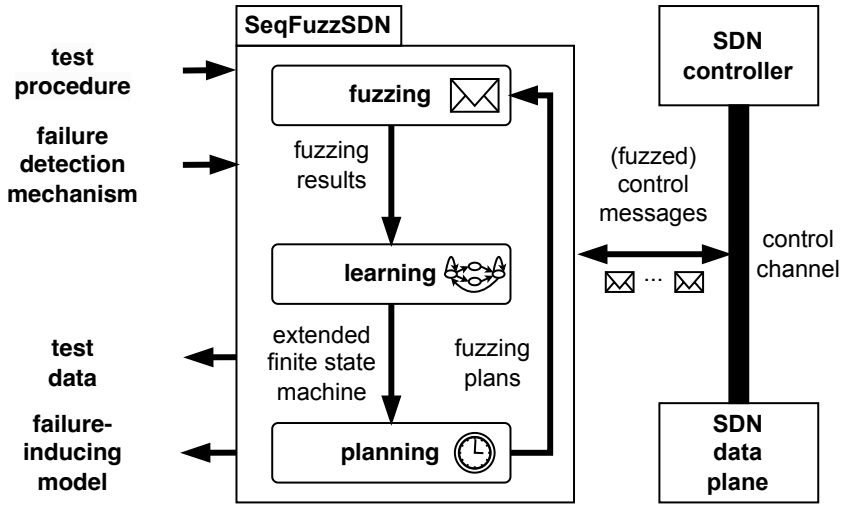


Figure 4.1: Approach overview.

controller under test, switches, and hosts in an SDN, (2) execute a use scenario, e.g., pair-wise ping test [96], to test the controller, and (3) properly tear down the SDN based on the given use scenario to test the controller again. Note that depending on the given use scenario, sequences of control messages exchanged between the controller and switches can vary. The failure detection mechanism, defined by engineers for the given test procedure, allows SeqFuzzSDN to determine whether the controller fails. For example, unexpected communication breakdowns and significant performance degradation can be considered as failures depending on the given test procedure. Regarding the outputs of SeqFuzzSDN, it produces a test data set and a failure-inducing model. The former contains sequences of control messages that are fuzzed by SeqFuzzSDN and lead to failures detected by the failure detection mechanism. The failure-inducing model characterises sequences of control messages leading to either successes or failures. When the failure detection mechanism does not detect any failures, SeqFuzzSDN considers the corresponding message sequences as successful. In summary, SeqFuzzSDN aims at generating a test data set that contains diverse failure-inducing sequences of control messages and a failure-inducing model that accurately characterises them.

SeqFuzzSDN is an iterative fuzzing method consisting of three steps (see Figure 4.1), as follows: (1) The fuzzing step involves sniffing and modifying control messages that pass through the control channel between the SDN controller and the SDN switches. Hence, it does not require any changes to the SDN controller and switches. (2) The learning step takes as inputs the control message sequences and failure detection results obtained from the fuzzing step. The learning step then builds a model to characterise the message sequences. Specifically, the learning step infers an extended finite state machine (EFSM) [106] that captures the controller's behaviour in terms of state transitions representing control messages received or sent by the controller. Unlike FSMs, EFSMs can capture state transitions associated with data variables, which are essential for modelling state changes caused by control messages. Control messages typically involve control operations that depend on data (e.g., flow tables and packet statistics). The inferred EFSM contains two types of final states representing success and failure, enabling SeqFuzzSDN

to classify and predict which sequences of state transitions (i.e., control messages) induce either success or failure. (3) The planning step takes as input the EFSM inferred by the learning step and generates fuzzing plans. These fuzzing plans aim to guide the fuzzing step in efficiently exploring the possible space of control message sequences and discovering diverse failure-inducing sequences of control messages. In the following subsections, we provide detailed descriptions of the three steps in SeqFuzzSDN.

4.3.2 Fuzzing

The fuzzing step of SeqFuzzSDN relies on the man-in-the-middle attack (MITM) technique [104], which is widely used and studied in the network security domain. This technique enables SeqFuzzSDN to position itself between the controller under test and the SDN switches that are communicating with the controller. Using MITM, SeqFuzzSDN can intercept control messages and potentially fuzz them while ensuring that the controller and switches remain unaware of the presence of SeqFuzzSDN. Furthermore, employing this attack technique allows SeqFuzzSDN to generate realistic potential threats (i.e., unexpected sequences of control messages) that the controller may face in practice.

When fuzzing control messages, SeqFuzzSDN accounts for the syntax requirements (i.e., grammar) defined in an SDN protocol (e.g., OpenFlow) to ensure fuzzed control messages are syntactically valid. An SDN controller typically rejects syntactically invalid messages at its message parsing layer [35]. Hence, producing valid control messages is desirable in practice to test the controller's behaviour beyond the parsing layer [35, 103].

SeqFuzzSDN employs a mutation-based fuzzing strategy [107] in which fuzz (i.e., mutation) operators introduce small changes to sniffed control messages while adhering to the syntax requirements of the messages. Below, we first describe five fuzz operators employed in SeqFuzzSDN that can modify control messages and their sequences. We then describe in detail how SeqFuzzSDN uses the fuzz operators.

Fuzz operators. When SeqFuzzSDN sniffs a control message, it can apply one of the following fuzz operators: deletion, insertion, duplication, delay, and modification. These operators are based on those used in BEADS [35], with modifications tailored for the learning-guided fuzzing of SeqFuzzSDN. We describe further details of the fuzz operators below.

Deletion. The deletion operator drops an intercepted message. For example, when SeqFuzzSDN intercepts a packet-in message from the control channel, it can omit retransmitting the message to the channel, thereby deleting the packet-in message from the control channel.

Insertion. The insertion operator inserts a new control message into the control channel. For example, SeqFuzzSDN can insert a new packet-in message to the control channel while it sniffs

Algorithm 5 Modification: Syntax-aware random

Input:
msg: control message to be fuzzed
pf: probability of fuzzing a field
Output:
msg′: control message after fuzzing

1: $F \leftarrow$ GET_FIELDS(msg)
2: $msg' \leftarrow msg$
3: **for all** $f \in F$ **do**
4: **if** RAND$(0,1) \leq pf$ **then**
5: $msg' \leftarrow$ REPLACE($msg', f,$ RAND_VALID(f))
6: **end if**
7: **end for**
8: **return** msg'

messages passing through the channel. Note that, such a new message is either predefined by engineers or randomly generated, as configured in SeqFuzzSDN.

Duplication. The duplication operator duplicates a sniffed message. For example, when SeqFuzzSDN intercepts a packet-in message, it can copy the same message and resend both the original and copied messages to the control channel. Hence, the channel carries the duplicated packet-in messages.

Delay. The delay operator holds a control message for a certain amount of time. For example, SeqFuzzSDN can hold an intercepted packet-in message for 200ms and resend it after the delay time. When SeqFuzzSDN holds a synchronous message (e.g., barrier-request), the sender will also wait for a response from the receiver. However, if SeqFuzzSDN delays an asynchronous message (e.g., packet-in), the sender continues its processing without waiting for the receiver to respond. Note that the delay time can be configured in SeqFuzzSDN.

Modification. The modification operator modifies the content (i.e., fields) of an intercepted control message. For example, when SeqFuzzSDN intercepts a packet-in message, it can change the version field of the message and inject the fuzzed message into the control channel.

We note that the modification operator behaves differently for the initial fuzzing phase and the subsequent learning-guided fuzzing phases. Algorithm 5 shows how the modification operator functions during the initial fuzzing phase when there is no guidance available for fuzzing. Given an intercepted message *msg*, the modification operator parses the content of *msg* in terms of its fields (line 1). For each field f of *msg* and a given probability of fuzzing a field pf, the operator replaces its value with a random value within its syntactically valid value range (lines 2-7). The operator then returns the fuzzed message msg' to transmit it through the control channel.

Algorithm 6 Initial fuzzing

Input:
pm: probability of fuzzing a message
Output:
seq': sequence of messages after fuzzing

```
1: seq' ← ⟨⟩
2: repeat
3:     msg ← RECEIVE()
4:     // no fuzzing
5:     if RAND(0, 1) > pm then
6:         SEND(msg)
7:         seq' ← APPEND(seq', msg)
8:         continue
9:     end if
10:    // fuzzing
11:    op ← RAND_SELECT_FUZZ_OPERATOR()
12:    if op is a deletion operator then
13:        // do nothing
14:    else if op is an insertion operator then
15:        msg' ← GET_MESSAGE(op)
16:        SEND(msg, msg')
17:        seq' ← APPEND(seq', msg, msg')
18:    else if op is a duplication operator then
19:        SEND(msg, msg)
20:        seq' ← APPEND(seq', msg, msg)
21:    else if op is a delay operator then
22:        t ← GET_DELAY(op)
23:        DELAY_SEND(msg, t)
24:        seq' ← DELAY_APPEND(seq', msg, t)
25:    else if op is a modification operator then
26:        msg' ← MODIFY(msg, op)
27:        SEND(msg')
28:        seq' ← APPEND(seq', msg')
29:    end if
30: until the test procedure has finished
31: return seq'
```

In the subsequent iterations of SeqFuzzSDN, the modification operator leverages planning outputs obtained from the learning and planning steps (see Figure 4.1). For readability, we describe the modification operator guided by learning in Section 4.3.5, after introducing the learning and planning steps.

Initial fuzzing. During the initial fuzzing phase of SeqFuzzSDN, since no failure-inducing model has been inferred, SeqFuzzSDN applies the fuzz operators randomly, as described in Algorithm 6. The algorithm takes as input a probability pm of fuzzing a message. It then returns a sequence seq' of messages after fuzzing. While executing a given test procedure (see the repeat block on lines 2-30 in Algorithm 6), SeqFuzzSDN intercepts each of the control messages

passing through the control channel between the SDN controller and switches (line 3). For each control message *msg* and the given probability *pm*, SeqFuzzSDN decides whether it fuzzes *msg* or not (line 5). When SeqFuzzSDN does not fuzz *msg*, it resends *msg* to the control channel and appends *msg* to seq' to record a processed message sequence (lines 6-7). For fuzzing the message *msg*, SeqFuzzSDN randomly selects one of the five fuzz operators (line 11). SeqFuzzSDN then applies the selected operator to *msg* and updates seq' accordingly (lines 12-29).

Data collection. To generate failure-inducing models, SeqFuzzSDN relies on an inference technique that takes as input event traces and produces an extended finite state machine (EFSM), such as Mint [108]. This EFSM captures the event traces as state transitions with guard conditions. In our context, an event trace corresponds to a message sequence seq' obtained from the fuzzing step. Each event e in the trace is associated with the corresponding message *msg* listed in seq'. Specifically, an event e is a tuple (l, m, v), where l denotes the type of *msg*, m denotes the fuzz operator applied to *msg*, v denotes the field values of *msg*. Note that m can be null if *msg* is not fuzzed in the given message sequence seq'. For example, consider a control message sequence in which a hello control message [25], used to discover and establish a connection between the controller and switches, was delayed by 200ms using the delay operator. SeqFuzzSDN encodes this hello message into an event e as follows: (`hello`, `delay` : 200, $<$ `0x5`, `0x0`, `0x10`, `0xA34BF` $>$), where the field values of the hello message are version = `0x5`, type = `0x0`, length = `0x10`, and xid = `0xA34BF`. The last event in the trace indicates either success or failure, determined by the failure detection mechanism for the given sequence seq' of messages. In addition, the event e is associated with both the sender and receiver of *msg*.

We note that, at each iteration i of SeqFuzzSDN, the fuzzing step executes the input test procedure (see Figure 4.1) n times, determined by a time budget. Hence, for each iteration i, the fuzzing step generates a dataset D_i that contains n event traces, i.e., $|D_i| = n$.

4.3.3 Learning

At each iteration i of SeqFuzzSDN, the learning step takes as input a dataset D of event traces obtained from the fuzzing step through the 1st to ith iterations, i.e., $D = D_1 \cup \ldots \cup D_i$. The learning step then outputs an EFSM inferred based on D. The inferred EFSM M is then used to guide the fuzzing process, which entails exploiting state transitions in M, exploring less-visited states in M, and discovering new states not captured in M. Furthermore, SeqFuzzSDN provides engineers with an accurate EFSM, achieved through iterative refinement of M. This EFSM serves as a failure-inducing model that characterises the generated failure-inducing message sequences (i.e., event traces), enabling engineers to gain a more comprehensive understanding of failure-inducing sequences rather than individually inspecting each of them.

We note that, to infer an EFSM, SeqFuzzSDN relies on Mint [108], a state-of-the-art model inference tool that takes as input a dataset containing event traces and produces an EFSM.

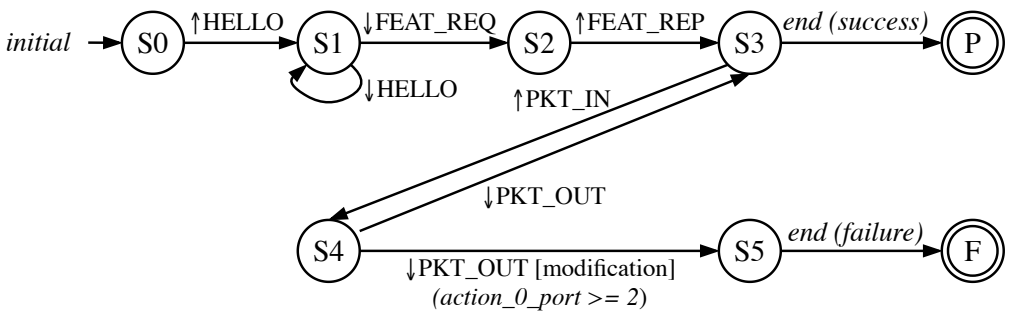


Figure 4.2: A simplified EFSM example produced by SeqFuzzSDN.

We opted to use Mint since it is one of the few tools available online and has been applied in many software engineering studies [109, 110]. In addition, the implementation of Mint is the most reliable among the tools available online, enabling us to focus on developing the main contributions of SeqFuzzSDN.

States and transitions. An SDN controller takes as input control messages and produces control messages in response, which are observable via MITM techniques [104]. In an EFSM inferred by SeqFuzzSDN, which captures sequences of observed control messages, a state is a placeholder for the transitions between different sequences of these messages, rather than representing a specific internal condition of the controller, which is not visible from SeqFuzzSDN. In this state, the controller is capable of processing a particular control message and generating a corresponding response message. A transition is defined as a tuple (s, l, c, m, d), where s denotes a source state, l denotes the type of a control message, c denotes a guard condition on the fields of the message, m denotes a fuzz operator applied to the message, and d denotes a destination state. In an EFSM produced by SeqFuzzSDN, using the dataset D containing event traces, each transition (s, l, c, m, d) corresponds to an event (l, m, v) in D (see the event definition in Section 4.3.2).

For example, Figure 4.2 shows an EFSM produced by SeqFuzzSDN, simplified for clarity. The EFSM contains eight states in total. Among these states, state `S0` represents the initial state of the EFSM, state `P` represents the success state, and state `F` represents the failure state. Additionally, the EFSM includes ten transitions. For instance, in the transition from state `S4` to state `S5`, `S4` serves as the source state (s), `PACKET_OUT` as the label (l), `action_0_port` ≥ 2 as the guard condition (c), `modification` as the mutation operator (m), and `S5` as the destination state (d). Note that the arrow ↑ (resp. ↓) annotated before each label (e.g., ↑`HELLO` and ↓`HELLO`) indicates that a message is received by the controller (resp. sent by the controller).

Guard condition inference. SeqFuzzSDN aims at efficiently producing an accurate EFSM that correctly captures the event traces D. Since the overall accuracy of an EFSM highly depends on the accuracy of transitions' guard conditions, in this section, we first explain how SeqFuzzSDN uses Mint to infer guard conditions from D. For further details, such as merging states and

removing non-determinism, we refer readers to the paper introducing MINT [108]. We then introduce how SeqFuzzSDN efficiently infers an EFSM from D in Section 4.3.3.

MINT employs a supervised machine learning algorithm [111] that requires labelled datasets to infer guard conditions of state transitions. To create labelled datasets, SeqFuzzSDN groups events in the event traces D based on the event type defined by (l, m), where the control message type l and the fuzz operator m are elements in an event $e = (l, m, v)$. This ensures that each group contains events with the same event type. For each event e in an event group, SeqFuzzSDN then labels e with the type of the next event following e in the corresponding event trace of e in the event traces D, as required by MINT. More precisely, given an event trace e_1, ..., e_i, e_{i+1}, ..., e_n, the event e_i in an event group is labelled with the type (l_{i+1}, m_{i+1}) of e_{i+1}. For each event group E, SeqFuzzSDN creates a dataset that contains pairs of the field values of an event $e \in E$ and the assigned label of e.

For example, Table 4.2 shows the creation of six datasets (see Table 4.2 (b)) from two event traces (see Table 4.2 (a)). The datasets could be used to infer the EFSM presented in Figure 4.2. As shown in Table 4.2 (a), Trace 1 and Trace 2 contain six distinct event types (l, m): ($HELLO$, `null`), ($FEAT\_REQ$, `null`), ($FEAT\_REP$, `null`), (PKT_IN, `null`), (PKT_OUT, `null`), and (PKT_OUT, `modification`). For each event type, SeqFuzzSDN creates its corresponding dataset as presented in Table 4.2 (b). Note that the third and last columns in the table indicate the content of a labelled dataset, including field values of a control message and associated labels (i.e., the next event type).

Regarding supervised machine learning, SeqFuzzSDN relies on RIPPER (Repeated Incremental Pruning to Produce Error Reduction) [71], an interpretable rule-based classification algorithm. RIPPER has shown successful applications in many software engineering problems involving classification and condition inference [88, 90, 112]. In particular, we select RIPPER because it generates pruned decision rules (i.e., if-conditions) that are more concise and, as a result, more interpretable than commonly used tree-based classification algorithms, such as C4.5 [113], which are susceptible to the replicated subtree problem [111].

Sampling event traces. Due to the computational complexity of the model inference problem, existing model inference techniques (including MINT) face scalability issues [109]. In the context of SeqFuzzSDN, the number of event traces can continuously grow as SeqFuzzSDN iterates through the fuzzing, learning, and planning steps multiple times and fuzzes message sequences corresponding to those event traces. Hence, when the number of event traces and their events become large (e.g., 5000 event traces, containing 15000 events), MINT either crashes due to running out of memory or takes a prohibitively long time to complete its execution. To address the scalability problem in our context, when learning an EFSM at each iteration i, SeqFuzzSDN uses a subset D^s of event traces instead of all event traces D generated up to the current iteration.

Table 4.2: An example illustrating the creation of datasets based on event traces: (a) Two event traces (i.e., Trace 1 and Trace 2). (b) Six datasets created based on Trace 1 and Trace 2.

(a) event traces

	event (e)	label (l)	mutation (m)	value (v)
Trace 1	e_{10}	HELLO	null	$v(e_{10})$
	e_{11}	HELLO	null	$v(e_{11})$
	e_{12}	FEAT_REQ	null	$v(e_{12})$
	e_{13}	FEAT_REP	null	$v(e_{13})$
	e_{14}	PKT_IN	null	$v(e_{14})$
	e_{15}	PKT_OUT	null	$v(e_{15})$
	e_{16}	end(success)	null	null
Trace 2	e_{21}	HELLO	null	$v(e_{21})$
	e_{22}	HELLO	null	$v(e_{22})$
	e_{23}	FEAT_REQ	null	$v(e_{23})$
	e_{24}	FEAT_REP	null	$v(e_{24})$
	e_{25}	PKT_IN	null	$v(e_{25})$
	e_{26}	PKT_OUT	modification	$v(e_{26})$
	e_{27}	end(failure)	null	null

(b) datasets

event type (l, m)	event (e)	value (v)	next event type (l', m')
(HELLO, null)	e_{10}	$v(e_{10})$	(HELLO, null)
	e_{11}	$v(e_{11})$	(FEAT_REQ, null)
	e_{21}	$v(e_{21})$	(HELLO, null)
	e_{22}	$v(e_{22})$	(FEAT_REQ, null)
(FEAT_REQ, null)	e_{12}	$v(e_{12})$	(FEAT_REP, null)
	e_{23}	$v(e_{23})$	(FEAT_REP, null)
(FEAT_REP, null)	e_{13}	$v(e_{13})$	(PKT_IN, null)
	e_{24}	$v(e_{24})$	(PKT_IN, null)
(PKT_IN, null)	e_{14}	$v(e_{14})$	(PKT_OUT, null)
	e_{25}	$v(e_{25})$	(PKT_OUT, modification)
(PKT_OUT, null)	e_{15}	$v(e_{15})$	(end(success), null)
(PKT_OUT, modification)	e_{26}	$v(e_{26})$	(end(failure), null)

Let M be an EFSM inferred at the $i-1$th iteration of SeqFuzzSDN. At iteration i, to sample event traces from the event traces D_i obtained at i and those used in learning M, SeqFuzzSDN first separate D_i into *accepted* and *rejected* traces. Given an EFSM M, accepted traces are traces that are already explained by M, i.e., traces that follow paths (i.e., transition sequences) in M. Note that when guard evaluations are needed while SeqFuzzSDN walks over M with traces, it uses Z3 [92], a well-known and widely applied SMT solver. In contrast, rejected traces refer to traces that do not follow any path in M. Hence, to create a set of event traces for learning a new EFSM M' at iteration i, SeqFuzzSDN includes the rejected traces in the set in order to ensure that they are explained by M'. However, SeqFuzzSDN does not include the accepted traces in the learning process because they do not contribute to refining M into M'.

Algorithm 7 Sampling event traces. Note that the sets of event traces used in this algorithm contain only success or failure event traces.

Input:
M: EFSM
$\mathcal{D}^M$: set of success (resp. failure) event traces used to generate M
$\mathcal{D}_i$: set of success (resp. failure) event traces obtained from the ith iteration of SeqFuzzSDN
$n_{\mathcal{D}}$: maximum size of an output set of event traces
Output:
$\mathcal{D}$: set of success (resp. failure) event traces for learning a new EFSM

```
1: // Case: include all traces
2: if |D^M ∪ D_i| ≤ n_D then
3:     D ← D^M ∪ D_i
4:     return D
5: end if
6:
7: // Case: replace traces
8: n_r ← |D^M ∪ D_i| − n_D // number of traces to replace
9: for n_r times do
10:    𝔾 ← GROUP_BY_PATH(D^M, M) // 𝔾: set of trace groups
11:    G ← SELECT_MAX_GROUP(𝔾)
12:    t ← RAND_SELECT_TRACE(G)
13:    D^M ← D^M \ {t}
14: end for
15: D ← D^M ∪ D_i // |D| = n_D
16: return D
```

SeqFuzzSDN then further separates the rejected event traces obtained from iteration i of SeqFuzzSDN into success event traces and failure event traces. Drawing inspiration from the observation that balanced datasets often yield higher accuracy in ML [111, 114], SeqFuzzSDN manages two distinct sets of event traces: one leading to success and the other to failure. These sets have the same maximum number of event traces and are used together to learn an EFSM.

Algorithm 7 presents our heuristic for sampling event traces. SeqFuzzSDN applies the algorithm separately to both success-rejected event traces and failure-rejected event traces. The algorithm takes as input an EFSM M inferred at iteration $i-1$, a set $\mathcal{D}^M$ of success (resp. failure) event traces used to learn M, a set $\mathcal{D}_i$ of success (resp. failure) rejected event traces obtained from iteration i, and the maximum size $n_{\mathcal{D}}$ of an output set $\mathcal{D}$. The algorithm then outputs a set $\mathcal{D}$ of success (resp. failure) event traces for learning a new EFSM. As shown on lines 1–5 of the algorithm, when the size of $\mathcal{D}^M \cup \mathcal{D}_i$ does not exceed the maximum size $n_{\mathcal{D}}$, the algorithm returns $\mathcal{D}^M \cup \mathcal{D}_i$. Otherwise, on line 8, the algorithm computes the number n_r of event traces to remove from $\mathcal{D}^M$ to ensure that the output set D contains $n_{\mathcal{D}}$ event traces (see line 15). On lines 9-14, the algorithm removes n_r event traces from $\mathcal{D}^M$ as follows: It first partitions $\mathcal{D}^M$ into groups, each containing event traces that follow the same path in M. It then selects a group G that contains the largest number of event traces compared to the other groups. On lines 12-13, it randomly selects an event trace t and removes it from $\mathcal{D}^M$. On lines 15-16, the algorithm returns $\mathcal{D}^M \cup \mathcal{D}_i$, where $|\mathcal{D}| = n_{\mathcal{D}}$. Note that the selection mechanism on lines 10-11 aims at minimising

information loss in $\mathcal{D}$ with regard to learning an EFSM. Since the selection mechanism (lines 10-11) selects an event trace from group G containing the largest number of event traces and removes the selected trace from $\mathcal{D}^M$ (lines 12-13), the remaining traces in G will still contribute to creating a new EFSM that contains the same path (i.e., no information loss after the removal) and accepts the remaining traces.

4.3.4 Planning

The planning step of SeqFuzzSDN takes as input an EFSM and outputs fuzzing plans to guide the subsequent fuzzing iteration. The fuzzing plans are defined as sequences of state transitions, i.e., paths in an EFSM, that guide the fuzzing step at the subsequent iteration. SeqFuzzSDN produces the fuzzing plans, aiming at (O1) exploring less-visited or new states of the controller under test, (O2) improving the accuracy of a failure-inducing model (i.e., EFSM) and (O3) increasing the diversity of message sequences (i.e., event traces) exercised for testing the controller. Hence, SeqFuzzSDN employs a multi-objective search algorithm [115] to address the planning problem. Below, we describe the multi-objective search-based planning approach in SeqFuzzSDN by defining the solution representation, the fitness functions, and the search algorithm.

Representation. Given an EFSM M obtained from the learning step, a candidate solution is a set C of sequences of state transitions (i.e., paths) in M where each transition sequence starts from the initial state s_1 of M and ends at a state s_o selected during search, representing a valid traversal of M. Depending on a fuzzing probability, each transition sequence in C can be associated with a fuzz operator m_o—deletion, insertion, duplication, delay, or modification described in Section 4.3.2—to be applied when the controller's state is s_o in the subsequent iteration of SeqFuzzSDN.

Fitness functions. SeqFuzzSDN aims at searching for candidate solutions with regard to the three objectives: (O1) coverage, (O2) accuracy, and (O3) diversity, described earlier. To quantify how a candidate solution fits these three objectives, below we define three fitness functions.

Coverage. SeqFuzzSDN relies on an EFSM M that models the state changes of the controller under test. To test various behaviours of the controller, SeqFuzzSDN aims at finding a candidate solution that ensures a similar (ideally equal) number of visits to each state in M. Hence, each state in M can be explored in different ways regarding how is reached and what happens after traversing it. Given an EFSM M at iteration i of SeqFuzzSDN and a set D of event traces obtained from the first to the ith iterations, SeqFuzzSDN counts the number of visits for each state in M by traversing M using each event trace in D.

To quantify the extent to which a candidate solution C satisfies the coverage objective regarding the state-visit numbers, SeqFuzzSDN leverages Shannon's Entropy [116]. In general, entropy characterises the average level of uncertainty inherent to the stochastic variable's possible

outcomes. In our context, the entropy defines the level of uncertainty associated with visits to a state in an EFSM M. Intuitively, the higher the entropy, the more evenly the states in M are visited.

Let S be a set of all states in an EFSM M obtained at iteration i and D be a set of event traces obtained from the first to the ith iterations. For each state $s \in S$, we denote by $nv(s, C)$ the sum of the following: (1) the number of visits to s by the event traces in D, and (2) the number of visits to s expected by a candidate solution C. We denote by $nv(S, C)$ the total number of state visits for S and define $nv(S) = \sum_{s \in S} nv(s, C)$. Based on Shannon's entropy equation, we formulate the following fitness function $\mathit{fitcov}(S, C)$ for the coverage objective as below. SeqFuzzSDN aims at maximising the fitness $\mathit{fitcov}(S, C)$.

$$\mathit{fitcov}(S, C) = -\sum_{s \in S} \frac{nv(s, C)}{nv(S, C)} \log_2 \frac{nv(s, C)}{nv(S, C)}$$

We note that, in practice, an EFSM inference technique is not always able to infer an EFSM M that allows the traversal of all event traces in D [108]. Hence, SeqFuzzSDN computes $nv(s, C)$ using those event traces in D that are traceable by M and a candidate solution C. To improve the accuracy of an inferred EFSM over iterations of SeqFuzzSDN, it accounts for an additional fitness function described below.

Accuracy. SeqFuzzSDN builds an EFSM M using MINT, which relies on supervised machine learning. Recall from Section 4.3.3 that MINT converts the event traces D into labelled training datasets (i.e., event groups) for building supervised classifiers. Hence, building accurate classifiers is beneficial to improve the overall accuracy of an EFSM M.

Note that the imbalance problem [111] is one of the main reasons that usually cause the low performance of supervised classification algorithms. In a labelled dataset, when the number of data instances of a class is significantly different from that of the other classes, classification algorithms tend to favour predicting the majority class, which is often not desirable in practice [111]. Hence, SeqFuzzSDN aims to address imbalance by planning to generate control message sequences that alleviate the problem.

To quantitatively assess the imbalance problem of each event group E (i.e., labelled training dataset) converted from the event traces D, SeqFuzzSDN uses the multi-class imbalance metric [117]. Given an event group E obtained from the event traces D, we denote by $nc(E)$ the number of classes in E, $ni(E)$ the number of data instances in E, and $ni(c)$ the number of data instances labelled with the class c. According to the multi-class imbalance metric, the imbalance ratio $ir(E)$ of E is computed as follows:

$$ir(E) = \frac{nc(E) - 1}{nc(E)} \sum_{c \text{ in } E} \frac{ni(c)}{ni(E) - ni(c)}$$

For example, consider an event group E that consists of three classes ($nc(E) = 3$)—namely c_1, c_2, c_3—along with a total of 1200 data instances ($ni(E) = 1200$). In the case where the class distribution is balanced (i.e., $ni(c_1) = ni(c_2) = ni(c_3) = 400$), the imbalance ratio is $ir(E) = 1$. However, in a situation where the class distribution is imbalanced, such as $ni(c_1) = 5$, $ni(c_2) = 200$, $ni(c_3) = 995$, the imbalance ratio increases to $ir(E) \approx 3.37$.

To estimate the degree to which a candidate solution C (i.e., fuzzing plan) impacts the imbalance problem, SeqFuzzSDN augments each event group E obtained from the event traces D using C. Recall from Section 4.3.3 that each event group E contains events that have the same event type. The class assigned to an event e in E is determined by the event following e in the corresponding event trace (i.e., message sequence) containing e. Hence, we can estimate how many new events will be added to each event group when SeqFuzzSDN generates message sequences guided by a candidate solution C. Precisely, given a sequence $(s_1, l_1, m_1, c_1, d_1)$, ..., $(s_i, l_i, m_i, c_i, d_i)$, $(s_{i+1}, l_{i+1}, m_{i+1}, c_{i+1}, d_{i+1})$, ..., $(s_o, l_o, m_o, c_o, d_o)$ of state transitions in C, SeqFuzzSDN can, for example, augment an event group E that corresponds to the event type (l_i, m_i) with a new event that is labelled with (l_{i+1}, m_{i+1}). We denote by $ir(E, C)$ the imbalance ratio of an event group that contains both the labelled events in the event group E and the augmented events from C. Below, we define the fitness function $(fitacc)(D, C)$ for the accuracy objective, where $ng(D)$ denotes the number of event groups in D. SeqFuzzSDN aims at maximising the fitness $\mathit{fitacc}(D, C)$.

$$\mathit{fitacc}(D, C) = \sum_{E \text{ in } D} ir(E, C)/ng(D)$$

Diversity. SeqFuzzSDN aims at testing the controller under test using diverse sequences of control messages. To this end, at each iteration i of SeqFuzzSDN, it plans to guide fuzzing in the i+1th iteration to generate sequences of control messages that are different from the sequences exercised from the first to the ith iterations, which are captured in the event traces D. Given a candidate solution C, SeqFuzzSDN quantifies the difference between D and event traces (i.e., message sequences) that can be produced by C using the normalised compression distance (NCD) [118]. NCD measures the difference between two objects X and Y based on their compression, the Kolmogorov complexity [119], and the information distance [120]. Precisely, $NCD(X, Y)$ is defined as follows:

$$NCD(X, Y) = \frac{Z(XY) - \min\{Z(X), Z(Y)\}}{\max\{Z(X), Z(Y)\}}$$

where $Z()$ is an actual compressor such as gzip [121], $Z(X)$ and $Z(Y)$ are the compressed sizes of the objects X and Y, and $Z(X, Y)$ is the compressed size of the concatenation of X and Y. Note that $NCD(X, Y) = 0$ indicates that the two objects are identical in terms of compressed information. In contrast, $NCD(X, Y) = 1 + \epsilon$ implies that they are distinct, where ϵ is a small positive value dependent on how closely the compressor Z approximates the Kolmogorov complexity. We opt to use NCD because it is applicable for comparing two sets of event traces, wherein individual events can have different message types, fuzz operators, and field values. Furthermore, the lengths of the event traces may differ from one another, and the two sets

contain different numbers of event traces. Hence, applying simple sequence comparison methods is not straightforward in our context.

Given the event traces D, in order to use NCD as the diversity fitness for a candidate solution C, SeqFuzzSDN predicts event traces T^C to be generated in the subsequent iteration. Specifically, for each transition sequence $p = (s_1, l_1, m_1, c_1, d_1)$, ..., $(s_i, l_i, m_i, c_i, d_i)$, $(s_{i+1}, l_{i+1}, m_{i+1}, c_{i+1}, d_{i+1})$, ..., $(s_o, l_o, m_o, c_o, d_o)$ in C, the sequence p is converted into an event trace $tr = (l_1, m_1, nil)$, ..., (l_i, m_i, nil), (l_{i+1}, m_{i+1}, nil), ..., (l_o, m_o, nil) by excluding the source and destination states s and d from the transitions while preserving their message type l and the fuzz operator m, along with their original order. Note that, in predicted event traces, field values are set to nil (i.e., $v_i = nil$) since transition sequences do not capture field values.

To quantify the degree to which a candidate solution C is different from the event traces D, we denote by T^C the predicted event traces when fuzzing is guided by C, and below, we define the fitness function $\mathit{fitdiv}(D, C)$ to address the diversity objective. SeqFuzzSDN aims at maximising the fitness $\mathit{fitdiv}(D, C)$.

$$\mathit{fitdiv}(D, C) = NCD(D, D \cup T^C)$$

Computational search. SeqFuzzSDN employs NSGA-II (Non-Dominated Sorting Genetic Algorithm II) [122], which has been applied in many software engineering studies [123, 124, 125, 126, 127], to search for a near-optimal fuzzing plan (i.e., solution C). Algorithm 8 describes the search process. Briefly, the algorithm first generates an initial population $\mathbf{P}$ (lines 1-6), containing n_p candidate solutions. Subsequently, the algorithm evolves the population iteratively until finding the ideal Pareto front or the allocated time budget is exhausted (line 9-24). At each iteration, the algorithm evaluates each candidate solution $C \in \mathbf{P}$ according to the fitness functions defined in section 4.3.4 (line 11-15). The algorithm then updates the archive $\mathbf{P}_\alpha$ (lines 16-17). It then computes the Pareto ranking of the solutions in the archive $\mathbf{P}_\alpha$, along with their associated sparsities, based on their fitness values (line 18-19). These ranks and sparsities are used to select the appropriate n_p solutions to be kept in the archive, as well as to identify the best Pareto Front (line 20-21). The algorithm then creates a new population $\mathbf{P}$ by breeding the solutions in the archive (lines 22-23). After the search process (lines 9-24), the algorithm returns a selected solution (lines 25-26). Below, we describe in detail the initial population generation, breeding, and solution selection mechanisms that are specific to SeqFuzzSDN.

Initial population. Given an EFSM M, SeqFuzzSDN generates an initial population for the search, containing n candidate solutions. Algorithm 9 describes how SeqFuzzSDN creates a candidate solution at the beginning of the search process (see line 4 of Algorithm 8). Algorithm 9 takes as input an EFSM M, the number n of transition sequences in a candidate solution C, a probability μ of fuzzing a message, and the number k of (different) shortest paths. At each iteration of the repeat block (lines 2-11), the algorithm finds a transition sequence p to be added into C, and this process is repeated n times. To find a transition sequence p, the algorithm first randomly selects a state s in M (line 3). It then finds the k shortest paths from the initial state of M to the selected

Algorithm 8 Searching best candidate traces to be used in the EFSM-guided fuzzing step, based on NSGA-II.

Input:
M: An EFSM
D: A set of generated event traces
n_p: size of the population and the archive
n_s: size of a candidate solution
n_k: number of shortest paths used during the generation of a candidate solution
μ_f: candidate solution fuzzing probability
μ_c: crossover probability
μ_m : mutation probability
Output:
C_b: Best solution

1: // generate the initial population
2: $\mathbf{P} \leftarrow \emptyset$
3: **repeat**
4: $C \leftarrow$ GenerateCandidate(M, n_s, μ_f, n_k)
5: $\mathbf{P} \leftarrow \mathbf{P} \cup C$
6: **until** $|\mathbf{P}| = n_p$
7: // create an empty archive
8: $\mathbf{P}_\alpha \leftarrow \emptyset$
9: **repeat**
10: // assess the fitness of each individual
11: **for each** $C \in \mathbf{P}$ **do**
12: $f_1(C) = \mathit{fitcov}(\mathit{states}(M), C)$
13: $f_2(C) = \mathit{fitacc}(D, C)$
14: $f_3(C) = \mathit{fitdiv}(D, C)$
15: **end for**
16: // update the archive
17: $\mathbf{P}_\alpha \leftarrow \mathbf{P}_\alpha \cup \mathbf{P}$
18: ComputeFrontRanks$(\mathbf{P}_\alpha)$
19: ComputeSparsities$(\mathbf{P}_\alpha)$
20: $\mathbf{P}_\alpha \leftarrow$ SelectArchive$(\mathbf{P}_\alpha, n_p)$
21: $\mathit{BestFront} \leftarrow$ ParetoFront$(\mathbf{P}_\alpha)$
22: // create a new population
23: $\mathbf{P} \leftarrow$ Breed$(\mathbf{P}_\alpha, n_p, \mu_c, \mu_m)$
24: **until** *BestFront* is the ideal Pareto front **or** the algorithm run out of time
25: $C_b \leftarrow$ SelectOne$(\mathit{BestFront})$
26: **return** C_b

state s using the k-shortest path algorithm [128] (line 4). SeqFuzzSDN uses the k-shortest path algorithm to obtain different transition sequences (paths) from the initial state to s. Given the fuzzing probability μ, the algorithm decides whether it applies a fuzz operator or not (line 6). If the algorithm decides to apply a fuzz operator, the algorithm randomly selects one of the five fuzz operators described in Section 4.3.2 (line 7). It then associates the transition sequence p with the selected fuzz operator m. This guides SeqFuzzSDN in the subsequent iteration to apply m when the controller reaches the selected state s following the transition sequence p. Since we do not know what will happen after applying m in the subsequent iteration of SeqFuzzSDN,

Algorithm 9 Creating a candidate solution

Input:

M: EFSM to generate a candidate solution (i.e., paths on M)
n: size of a candidate solution
μ : probability of fuzzing a message
k: number of shortest paths to generate

Output:

C : candidate solution

```
1:  C ← ∅
2:  repeat n times
3:      s ← RAND_SELECT_STATE(M)
4:      P ← FIND_K_SHORTEST_PATHS(M, s, k)
5:      p ← RAND_SELECT_PATH(M, P)
6:      if RAND(0, 1) ≤ μ then
7:          op ← RAND_SELECT_FUZZ_OPERATOR()
8:          ASSOCIATE_FUZZ_OPERATOR(p, s, op)
9:      end if
10:     C ← C ∪ {p}
11: end
12: return C
```

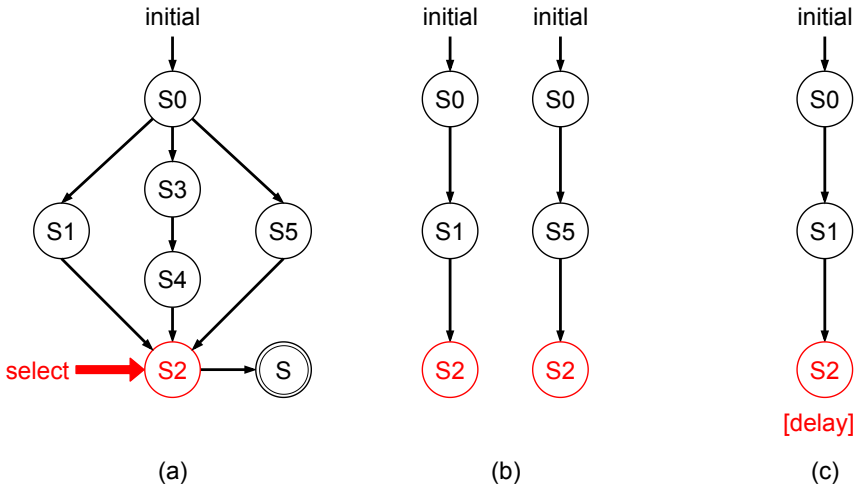


Figure 4.3: An example illustration of generating a candidate solution from a simple EFSM: (a) a simple EFSM for clarity, (b) two shortest paths from S0 to S2, and (c) a candidate solution and its associated fuzz operator, i.e., delay.

it allows SeqFuzzSDN to potentially discover new states that are not captured in the current EFSM M.

For example, Figure 4.3 illustrates how SeqFuzzSDN generates initial candidate solutions using a simple EFSM M (Figure 4.3 (a)) for brevity. Given M, when Algorithm 9 selects state S2, it then finds two shortest paths (Figure 4.3 (b)). After that, the algorithm randomly selects a fuzz operator (e.g., delay) to apply to the selected candidate solution.

Breeding. The breeding mechanism uses the following genetic operators [122]: *selection*, *crossover*, and *mutation* operators. SeqFuzzSDN employs the binary tournament selection and the one-point crossover [122]. Specifically, given two parent solutions C^l and C^r, each containing transition sequences (paths) $\{p_1^l, \ldots, p_i^l, \ldots, p_j^l\}$ and $\{p_1^r, \ldots, p_i^r, \ldots, p_k^r\}$, respectively, the crossover operator randomly selects a crossover point i. It then generates two offspring solutions by swapping transition sub-sequences separated by i between the parents, resulting in $\{p_1^r, \ldots, p_i^l, \ldots, p_j^l\}$ and $\{p_1^l, \ldots, p_i^r, \ldots, p_k^r\}$. Further, SeqFuzzSDN relies on the uniform mutation operator [93]. Specifically, SeqFuzzSDN first randomly selects a transition sequence p in a candidate solution C. It then replaces p with a new transition sequence obtained with Algorithm 9, setting the parameter n to 1 to create a single transition sequence.

Selecting a near-optimal solution. Algorithm 8, which is based on NSGA-II, outputs a set of Pareto-optimal solutions, which are equally viable with respect to the three objectives regarding coverage, accuracy, and diversity. However, SeqFuzzSDN requires selecting one of the solutions to guide fuzzing at the subsequent iteration. Various methods to select a near-optimal solution in a Pareto front have been proposed in the literature, such as selecting a *knee solution* [129], or selecting a *corner solution* [130] for a specific objective. In our context, SeqFuzzSDN uses a knee solution, which is often favoured in search-based software engineering studies [129, 131]. This preference is due to the observation that selecting other solutions on the front to achieve a slight improvement in one objective could result in a significant deterioration in at least one other objective [129]. Given the three objectives regarding coverage, accuracy, and diversity, SeqFuzzSDN favours a candidate solution that achieves a balanced optimisation across all these objectives.

Given a selected set of candidate solutions, containing planned paths (state transitions), SeqFuzzSDN finds transitions that are associated with the modification fuzz operator and a guard condition. It then solves the guard condition using Z3 in order to apply the modification fuzz operator, ensuring the guard condition is satisfied during our EFSM-guided fuzzing (described in Section 4.3.5). For example, given a state transition $(s_i, \uparrow PACKET\_IN, f_k < 20 \wedge f_k > 8, modification, s_j)$, SeqFuzzSDN solves the guard condition, such as $f_k = 10$. When the transition is exploited during fuzzing, SeqFuzzSDN modifies a PACKET_IN message by assigning 10 to the field f_k of the message. Note that SeqFuzzSDN solves guard conditions during the (offline) planning step, rather than the (online) fuzzing step, in order to improve efficiency during fuzzing.

4.3.5 EFSM-Guided Fuzzing

After the initial fuzzing step, SeqFuzzSDN uses the learning and planning outputs to guide fuzzing sequences of control messages to test the SDN controller. This section first describes an EFSM-guided fuzzing method in SeqFuzzSDN, and then illustrates the method through a running example.

EFSM-guided fuzzing algorithm. Algorithm 10 describes the fuzzing procedure in SeqFuz-

Algorithm 10 EFSM-Guided Fuzzing

Input:
M: EFSM generated from the learning step
C: set of planned paths on M
Output:
seq' : sequences of messages after fuzzing
C': set of planned paths after applying one of them

```
1:  seq' ← ⟨⟩
2:  repeat
3:      s ← CURRENT_STATE(M, seq')
4:      P ← FIND_APPLICABLE_PATHS(C, M, seq')
5:      msg ← RECEIVE()
6:      TN ← FIND_APPLICABLE_TRANSITIONS(s, msg, P)
7:      op ← ∅
8:      if TN ≠ ∅ then
9:          tn ← RAND_SELECT(TN)
10:         op ← GET_FUZZ_OPERATOR(tn)
11:     end if
12:     if op is a fuzz operator then
13:         msg ← FUZZ(msg, op)
14:         seq' ← APPEND(seq', msg, op)
15:     else
16:         seq' ← APPEND(seq', msg)
17:     end if
18:     SEND(msg)
19: until the test procedure has finished
20: p ← FIND_USED_PATH(M, C, seq')
21: C' ← C \ {p}
22: return seq', C'
```

zSDN once an EFSM M is available, after the initial fuzzing step. The algorithm takes as input an EFSM M and a set C of planning paths (i.e., sequences of state transitions) on M, and iterates lines 2-19 until the test procedure has finished executing. At the beginning of each iteration, on line 3, the algorithm first identifies the current state s in M according to the currently observed sequence seq' of control messages. On line 4, SeqFuzzSDN then finds a set P of applicable paths from the set C of planning paths to guide fuzzing. The applicable paths contain state transitions on M that start from the current state s. Below, Algorithm 11 further describes this procedure. On line 5, the algorithm receives a control message msg passing through the SDN control channel and then finds a set TN of applicable transitions from P. The applicable transitions start from the current state s, are triggered by an event type l corresponding to msg, and, if there are guards, the guards hold on the field values of msg. On lines 7-11, if some applicable transitions are found, the algorithm randomly selects a transition tn among the applicable transitions (line 9), and gets the fuzz operator op of tn, if tn has one (line 10). On lines 12-15, if the fuzz operator op is present, the algorithm applies it to msg (line 13) and appends it (line 14) to the output sequence seq' along with the applied fuzz operator (op). On lines 16-17, if no fuzz operator is present, the

Algorithm 11 Finding Applicable Paths

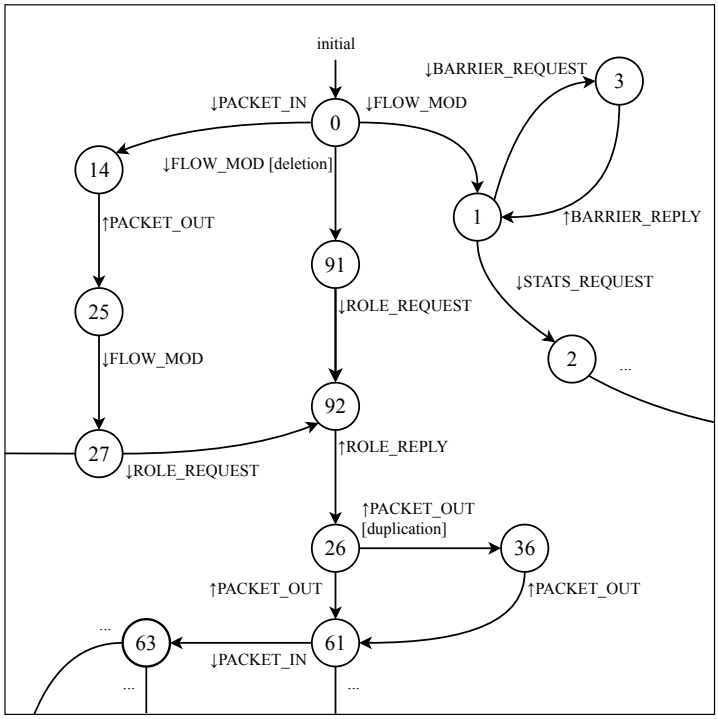
Input:
C: set of planned paths
M: EFSM
seq: sequence of messages
Output:
C': set of applicable paths

1: $C' \leftarrow \emptyset$
2: $p^s \leftarrow \text{WALK}(M, seq)$
3: $s \leftarrow \text{CURRENT\_STATE}(M, seq')$
4: **for each** $p \in C$, where s is on p **do**
5: $p' \leftarrow \text{SUBPATH}(p, s)$
6: **if** for all $s \in p'$, $s \in p^s$, and all s appear in the same order on both p' and p^s **then**
7: $C' \leftarrow C' \cup \{p\}$
8: **end if**
9: **end for**
10: **return** C'

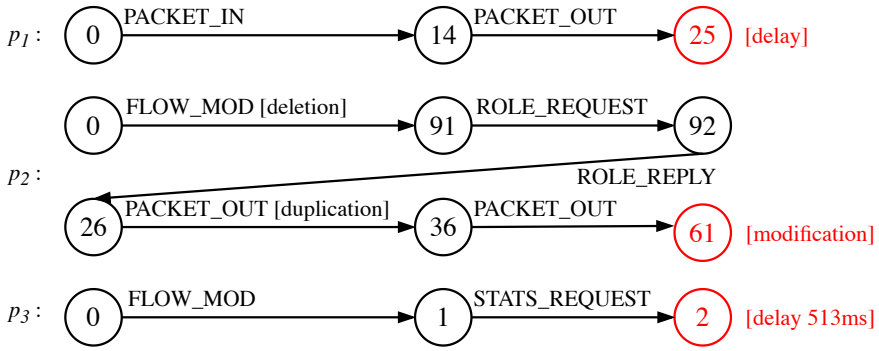
algorithm simply appends the originally received message *msg* to the output sequence *seq*′. On line 18, the algorithm sends back the *msg*, which could be fuzzed, into the control channel. Since, at each iteration of SeqFuzzSDN, the test procedure is run multiple times, on lines 20-21, the algorithm removes a planned path that has been applied, enabling the subsequent executions of the test procedure to be fuzzed, guided only by the remaining planned paths.

Algorithm 11 identifies a set C' of applicable paths on an EFSM M based on a given sequence *seq* of control messages and a set C of planned paths on M. On lines 1-3, the algorithm initialises a return set C' of applicable paths on M, converts *seq* into a path p^s on M, and identifies the current state s on M for the given *seq*. The algorithm then examines each path p in C to determine whether the current state s appears on p (line 4) and whether the sub-path p' of p from the start state to s in M is a derivative of p^s that corresponds to the sequence *seq* of control messages (lines 5-6). If p' is a derivative of p^s, p' can be derived from p^s by deleting some transitions without changing the order of the remaining transitions. Recall from Algorithm 9 that SeqFuzzSDN uses the k-shortest path algorithm to create planned paths. Hence, Algorithm 11 checks whether planed (sub-)paths on M are derivatives of the paths in M that correspond to sequences of control messages. The algorithm then returns a set C' of applicable paths that satisfy the conditions described above.

EFSM-guided fuzzing example. Figure 4.4 presents a part of an EFSM M (Figure 4.4a) inferred from the learning step and three planned paths C (Figure 4.4b) in M created by the planning step. Given the EFSM M and the planned paths C, when SeqFuzzSDN begins executing the test procedure (e.g., ping test), Algorithm 10 starts with the initial state 0 in M to perform EFSM-guided fuzzing. Then all the three planned paths, p_1, p_2, and p_3 shown in Figure 4.4b, are identified as applicable paths. The algorithm then receives the first control message (generated by the test procedure), which, in this example, we assume to be "↑HELLO". In this case, however,



(a) A partial EFSM example inferred from the learning step of SeqFuzzSDN.



(b) Three planned paths to guide fuzzing, created by the planning step of SeqFuzzSDN.

Figure 4.4: Output examples of the learning and planning steps: (a) a partial EFSM and (b) three planned paths on the EFSM.

there are no planned paths that contain transitions starting from state 0 and taking the event (message) "↑HELLO". Hence, the "↑HELLO" message is sent back into the control channel without any modification. The "↑HELLO" message is then appended to the output sequence seq', as follows:

$$seq' = \langle \uparrow \texttt{HELLO} \rangle$$

After receiving the "↑HELLO" message, in this example, the algorithm receives three more control messages, as follows: "↓HELLO", "↓FEATURES_REQUEST", and "↑FEATURES_REPLY". In these cases, the EFSM M remains in state 0 since there are no transitions from state 0 that can be taken by the three messages. In addition, there are no applicable paths. As a result, those

messages are sent back to the control channel, and the output sequence seq' is as follows:

$$seq' = \langle \uparrow\texttt{HELLO},\ \downarrow\texttt{HELLO},\ \downarrow\texttt{FEATURES\_REQUEST},\ \uparrow\texttt{FEATURES\_REPLY} \rangle$$

Next, the algorithm receives the "↓FLOW_MOD" message, while the EFSM M is in state 0. Then, the algorithm identifies paths p_2 and p_3 as applicable paths since they contain transitions starting from state 0 and taking the event "↓FLOW_MOD". Among the two transitions, i.e., $(0,\ \downarrow FLOW\_MOD,\ nil,\ deletion,\ 91)$ on p_2 and $(0, \downarrow FLOW\_MOD, nil, nil, 14)$ on p_3, the algorithm randomly selects the second one on p_3. Since no fuzz operator is associated to the transition, the algorithm simply sends the message back to the control channel, and appends the message to the output sequence seq' of control messages, as follows:

$$seq' = \langle \uparrow\texttt{HELLO},\ \downarrow\texttt{HELLO},\ \downarrow\texttt{FEATURES\_REQUEST},\ \uparrow\texttt{FEATURES\_REPLY},\ \downarrow\texttt{FLOW\_MOD} \rangle$$

In the subsequent iteration of the algorithm, the current state of the EFSM M changes to state 1, as there is a transition from state 0 to 1 that takes the "↓FLOW_MOD" event. The algorithm then finds only p_3 as an applicable path since it has a transition starting from state 1. If the algorithm receives the "↓BARRIER_REQUEST" message, the message is forwarded to the control channel without applying any fuzz operators, as there are no applicable transitions on p_3, and is appended to the output sequence seq'.

After this iteration, the algorithm changes the current state of the EFSM M to state 6, since there is a transition from state 1 to 6 taking the "↓BARRIER_REQUEST" event. In this case, there are no applicable paths. If the algorithm receives the "↓BARRIER_REPLY" message, it sends the message back to the control channel without any modification and updates the output sequence seq'.

Since there is a transition from state 6 to 1 in the EFSM M, in the next iteration of the algorithm, the current state is set to state 1. Path p_3 is applicable in this situation. If the algorithm receives the "↓STATS_REQUEST" message, the corresponding transition on p_3 is identified by the algorithm. However, since there is no fuzz operator associated to the transition, the algorithm sends the message back to the control channel and updates the output sequence seq'.

In the next iteration of the algorithm, the current state of the EFSM M is set to state 2 by taking the transition from state 1 to 2 due to the "↓STATS_REQUEST" event. In the state, path p_3 is applicable. Note that, however, state 2 is the end state of path p_3 and is associated with the delay fuzz operator, which holds a message for 513ms and then sends it back to the control channel. This indicates that for any receiving message, the algorithm applies the delay fuzz operator. If the algorithm receives the "↑STATS_REPLY", it applies the delay fuzz operator. From the subsequent iterations of the algorithm, no planned paths are applicable as p_3 has been exploited in its entirety. Hence, the algorithm simply forwards the receiving messages to the control channel and updates the output sequence seq' until the end of the test procedure

execution. After executing the test procedure, we can obtain the following sequence of control messages, which leads to the SDN controller failing:

$$seq' = \langle \uparrow\texttt{HELLO},\ \downarrow\texttt{HELLO},\ \downarrow\texttt{FEATURES\_REQUEST},\ \uparrow\texttt{FEATURES\_REPLY},\ \downarrow\texttt{FLOW\_MOD},\ \downarrow\texttt{BARRIER\_REQUEST},\ \uparrow\texttt{BARRIER\_REPLY},\ \downarrow\texttt{STATS\_REQUEST},\ \uparrow\texttt{STATS\_REPLY}[\texttt{delay 513ms}],\ \downarrow\texttt{ERROR},\ \texttt{FAILURE}\rangle$$

4.4 Evaluation

In this section, we empirically evaluate SeqFuzzSDN. Our complete evaluation package is available online [105].

4.4.1 Research Questions

RQ1 (comparison). *How does SeqFuzzSDN compare against other state-of-the-art fuzzing techniques for SDNs?* We investigate whether SeqFuzzSDN can outperform state-of-the-art testing techniques for SDNs, including DELTA [34], BEADS [35], and FuzzSDN [103]. We choose these techniques as they rely on fuzzing to test SDN controllers and their implementations are available online.

RQ2 (ablation study). *How does the sampling technique employed by SeqFuzzSDN influence its performance?* We assess the impact of the sampling technique (defined in Algorithm 7), which is our heuristic for sampling event traces to learn EFSMs. Specifically, we assess the impact of the technique in terms of execution time, the accuracy of EFSMs, and the diversity and coverage of the fuzzing results. To achieve this, we compare SeqFuzzSDN with its variant SeqFuzzSDN^{NS}, which does not sample event traces, and subsequently analyse the impact of the sampling algorithm.

RQ3 (scalability). *Can SeqFuzzSDN fuzz sequences of control messages and learn stateful failure-inducing models in practical time?* We investigate the correlation between SeqFuzzSDN's execution time and network size. To do so, we carry out experiments involving SDNs of different network sizes.

4.4.2 Simulation Platform

To conduct large-scale experiments, we employ a simulation platform that emulates the physical networks. Specifically, we utilise Mininet [94] to create virtual networks of various sizes. Mininet

leverages real-world SDN switch programs, resulting in emulated networks that closely match real-world SDNs. Hence, Mininet has been widely adopted in numerous SDN studies [35, 34, 24, 103].

We note that SeqFuzzSDN can also be applied to actual physical SDNs. However, assessing SeqFuzzSDN on actual physical networks through large-scale experiments, such as the ones reported in this chapter, is prohibitively expensive in terms of both cost and time.

Our experiments were conducted on 10 virtual machines, each equipped with 4 CPUs and 10GB of RAM. Each experiment was conducted with a time budget of 5 days for ONOS and 3 days for RYU.We note that, within this budget, the sensitivity values of the EFSMs generated by SeqFuzzSDN reach their plateaus. Due to the randomness of SeqFuzzSDN, we repeated our experiments 10 times. These experiments took approximately 60 days of concurrent execution on the 10 virtual machines.

4.4.3 Study Subject

We evaluate SeqFuzzSDN by testing two open-source and actively maintained SDN controllers, ONOS [82] and RYU [83], both of which are still widely used in SDN studies [34, 35, 103, 38, 37, 53, 51]. Both controllers' implementations are based on the OpenFlow SDN protocol specification. SeqFuzzSDN, which fuzzes OpenFlow control messages, is therefore capable of testing any SDN controller that adheres to the OpenFlow specification.

For our evaluation, we created five virtual networks with 1, 2, 4, 8, and 16 switches respectively. Each network is managed by either ONOS or RYU. In each network, the switches possess emulated physical connections with all the other switches, forming a fully connected topology. Each switch is connected to two hosts, simulating devices that transmit and receive data, such as video and audio streams.

We note that the study subjects, comprising of 5×2 synthetic systems built on the five networks managed by ONOS and RYU, are representative of both existing SDN studies and real-world SDNs. For instance, in prior SDN studies testing ONOS and RYU, DELTA was evaluated using an SDN with two switches, BEADS was evaluated using an SDN with three switches, and FuzzSDN was evaluated on SDNs with 1, 3, 5, 7, and 9 switches, due to the significant computational resources required for conducting experiments with SDNs.

4.4.4 Experimental setup

EXP1. To answer RQ1, we conduct a comparative analysis of SeqFuzzSDN with three other SDN testing tools: FuzzSDN [103], DELTA [34], and BEADS [35]. FuzzSDN is a testing framework that generates rule-based failure-inducing models and test cases. FuzzSDN employs a grammar-based

machine learning-guided fuzzing technique, which enables it to progressively refine the generated failure-inducing models, offering interpretable models that describe the conditions leading to a failure. DELTA is a security framework designed for SDNs that allows engineers to automatically replicate established attack scenarios associated with SDNs and uncover new attack scenarios through fuzzing. DELTA accomplishes this by changing control messages, employing a fuzzing technique that randomises the control message byte stream, regardless of the OpenFlow protocol specificities. Lastly, BEADS is an automated attack discovery technique that relies on a range of mutation (fuzz) operators, with the aim of discovering attack scenarios. BEADS also fuzzes control messages but employs strategies such as message dropping, duplication, delay, and modification while adhering to the OpenFlow specification. This allows BEADS to generate fuzzed control messages that can pass beyond the message parsing layer of the system under test.

To compare SeqFuzzSDN with these three SDN testing tools, we create three baselines: FUZZSDN^E, DELTA^E and BEADS^E. These baselines extend FuzzSDN, DELTA and BEADS respectively, to infer EFSMs, as the original testing tools do not produce EFSMs as part of their test outputs. FUZZSDN^E (resp. DELTA^E and BEADS^E) encodes the fuzzed control messages and the test output (i.e., success and failure) as a dataset to infer EFSMs. The baselines then use MINT to generate EFSMs. Unlike SeqFuzzSDN, FUZZSDN^E, DELTA^E, and BEADS^E do not leverage the generated EFSM to guide their fuzzing operations.

We use two synthetic systems, each with a single switch, controlled by either ONOS or RYU. We leverage a test procedure (see Section 4.3) that specifies a pairwise ping test [96], which has been used in many SDN studies [34, 35, 75, 54, 103]. This test procedure is important as it enables practitioners to verify communication between hosts, measure latency, detect packet loss, and identify routing issues. For the failure detection mechanism, we identify spurious switch disconnections. In our experiments, we identify switch disconnections that lead to communication breakdowns as failures. These failures cannot be localised using stack traces to pinpoint the causes of the failures.

In our comparison, we count the number of failures observed during the execution of SeqFuzzSDN and the baselines. In addition, from the final EFSMs inferred by the four tools, we measure the number of unique loop-free paths (corresponding to message sequences) that lead to failures. This allows us to assess how many distinct failure-inducing sequences of state changes are captured in the EFSMs. To further compare the four tools, we analyse the sensitivity of each EFSM, calculated using the formula: sensitivity = #accepted/(#accepted + #rejected), where #accepted and #rejected are the number of traces accepted and rejected by the EFSM, respectively. In our context, an EFSM with high sensitivity is desirable as it is less likely to miss possible failure-inducing sequences of control messages. To fairly calculate sensitivity, we elected to create a dataset that maintains a balanced representation of success and failure traces across all tools, thereby reducing potential biases toward a specific tool. To do so, we created a test dataset containing 800 fuzzing results, with an equal split of 400 success traces and 400 failure traces. These fuzzing results were randomly sampled from separate runs of SeqFuzzSDN,

FuzzSDN, DELTA, and BEADS, with each tool contributing 200 results, evenly divided into 100 success traces and 100 failure traces.

Additionally, we measure the diversity of fuzzed message sequences obtained from the four tools using the Normalised Compression Distance (NCD) for multisets [132]. Recall from Section 4.3 that the fuzzed message sequences vary in length, message types, and message values, making the application of simple sequence comparison metrics difficult. In our context, a high NCD value indicates that the fuzzed sequences of control messages (i.e., tests) are diverse, reducing the likelihood of redundancy or overly similar tests.

EXP2. To answer RQ2, we compare SeqFuzzSDN to its variant, named SeqFuzzSDN^{NS}. At each learning step, instead of using the sampling technique (see Algorithm 7), SeqFuzzSDN^{NS} uses all the collected event traces to infer an EFSM.

In this experiment, we use the same synthetic systems as those used in EXP1. Our test procedure specifies a pairwise ping test, and our failure detection mechanism identifies unexpected communication breakdowns. This experiment counts the number of iterations of the fuzzing, learning, and planning steps within the time budget and measures the execution time of each step. In addition, we compare SeqFuzzSDN and SeqFuzzSDN^{NS} by measuring the sensitivity of the final EFSMs obtained after the time budget expires. To ensure fair comparisons between SeqFuzzSDN and SeqFuzzSDN^{NS}, we created a test dataset comprising 1000 fuzzing results, evenly split into 500 success traces and 500 failure traces. These results were obtained from separate runs of SeqFuzzSDN, SeqFuzzSDN^{NS}, FUZZSDN^{E}, DELTA^{E}, and BEADS^{E}, with each method contributing 200 results, evenly split into 100 success traces and 100 failure traces. Therefore, this test dataset is not biased toward either SeqFuzzSDN or SeqFuzzSDN^{NS}. Furthermore, we measure the coverage and diversity degrees (defined in Section 4.3.4) of the planned paths (corresponding to message sequences) obtained at the last iteration, allowing us to assess the effectiveness of the EFSMs in generating message sequences that cover diverse states.

EXP3. To answer RQ3, we investigate the correlation between the resource consumption of SeqFuzzSDN and the size of the five synthetic systems described in Section 4.4.3, each with 1, 2, 4, 8, and 16 switches, controlled by either ONOS or RYU. For this experiment, we use a test procedure that implements the pairwise ping test, similar to EXP1 and EXP2. Compared to EXP1 and EXP2, the sequences of control messages produced by the test procedure in EXP3 differ significantly in terms of their lengths. This is due to the fully connected topology in EXP3, which includes multiple switches.Moreover, when there are more than two switches, the topology introduces switching loops [27], further increasing the number of events in a trace. We measure the time required to configure Mininet and the SDN controller, perform the test procedure, and execute each step of SeqFuzzSDN (i.e., fuzzing, learning, and planning).

4.4.5 Parameter Setting

As described in Section 4.3, SeqFuzzSDN takes as input parameters that can be tuned to improve its efficiency and effectiveness. For clarity and reproducibility, this section provides all the parameter values and describes how we set them. We note that, given the extremely long execution time required for applying automated hyperparameter optimisation techniques in our context, we manually set some of the parameters as described below.

In the learning step, the parameters to be tuned are those of the sampling technique (Algorithm 7) and MINT [108]. For the sampling technique, we set the number (n_{ts}) of event traces to 1000, limiting the maximum size of the dataset used by MINT. This configuration allowed SeqFuzzSDN to generate EFSMs in practical time (approximately 100 minutes). For MINT, we configured the parameter values of RIPPER [71] as follows: three folds, a minimal weight of 2.0, and two optimisation runs as specified by the default setting in WEKA [111].

In the planning step, we set the size of a candidate solution (n_s) to 200 in order to match the number of test procedure executions to be performed in each iteration of SeqFuzzSDN. This ensures that a candidate solution contains the 200 traces to be followed during the 200 executions of the test procedure. We set the candidate solution fuzzing probability (μ_f) to 0.5, as we want to strike a balance between exploitation and exploration of the generated EFSM. The crossover probability (μ_c) and the mutation probability (μ_m) in the planning step were set to 0.8 and 0.02, respectively, following published guidelines. The size of the population and archive, n_p, is set to 100 and the search generates 50 populations, allowing the planning step to be completed within a reasonable time (on average, 79 minutes for our ONOS study subject, and 44 minutes for our RYU study subject).

The remaining parameters were tuned using hyperparameter optimisation [111], following guidelines from the literature [111, 98]. We evaluated 10 different configurations of SeqFuzzSDN using grid search [111]. As a result of this optimisation process, we set the remaining parameters as follows: the initial fuzzing probability (μ) of a message is 0.3, the minimum merging score (k) of MINT is 1, and the number (n_k) of shortest paths used during the generation of a candidate solution is 15.

The parameters of SeqFuzzSDN used in our experiments could be further refined to improve efficiency and effectiveness. However, the configuration we chose produced results that are satisfactory to support our findings. As a result, we have not included additional experiments aimed at optimising these parameters in this chapter.

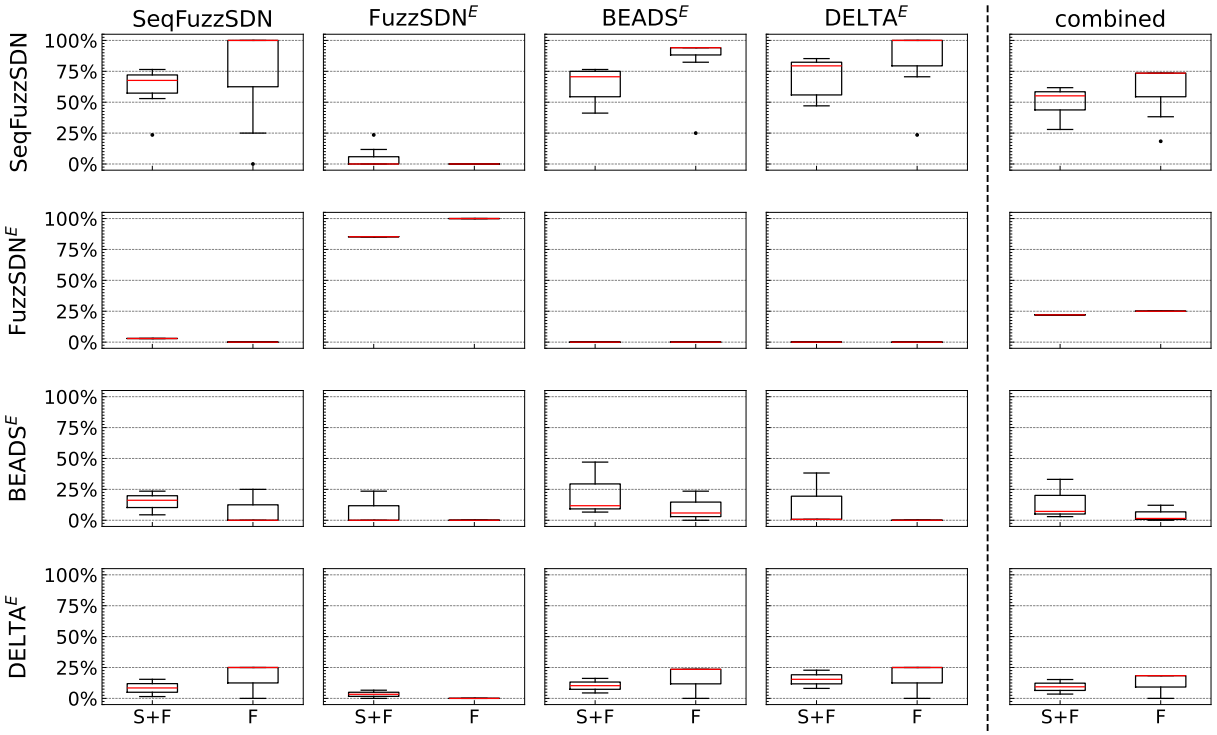


Figure 4.5: Comparing the sensitivity of the EFSMs generated by SeqFuzzSDN, FuzzSDN^E, Beads^E, and Delta^E, the five plots in each row display the sensitivity of the corresponding tool. The first four columns represent the sensitivity of the EFSMs assessed using the test dataset containing message sequences generated by each tool. Sensitivity is assessed using message sequences that lead to both success and failure, denoted by (S+F), and only failure, denoted by (F). The last column represents the sensitivity assessed using all datasets generated by the four tools. The boxplots (25%-50%-75%) show the distribution of sensitivity over 10 runs of each tool.

4.4.6 Experiment Results

To answer the research questions, we assessed the results obtained from both the ONOS and RYU subjects. Since the findings from the ONOS results are consistent with those from the RYU results, this section presents only the ONOS results for brevity. Note that the results for our RYU study subject are presented in Appendix A.1.

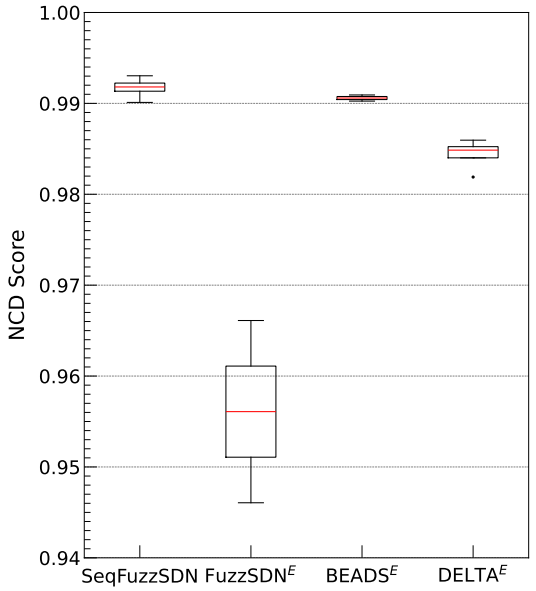
RQ1. Figure 4.5 compares the sensitivity of the EFSMs measured using the test dataset, which contains message sequences and their test results obtained from the four tools: SeqFuzzSDN, FuzzSDN^E, Beads^E, and Delta^E. The last column of the first row in the figure shows that, when evaluating all the message sequences produced by these tools, on average, SeqFuzzSDN achieves a sensitivity of 49.89% on the message sequences leading to both success and failure (referred to as the combined S+F dataset) and 60.19% on the message sequences leading only to failure (referred to as the combined F dataset). For brevity, we refer to datasets containing message sequences generated by each tool that result in both success and failure as the [tool] S+F dataset and those that result only in failure as the [tool] F dataset. Specifically, as shown in the first row of the figure, SeqFuzzSDN achieves, on average, a sensitivity of 67.1% on the SeqFuzzSDN S+F dataset and 92.3% on the SeqFuzzSDN F dataset, 0.23% on the FuzzSDN^E

S+F dataset and 0.00% on the FuzzSDN^E F dataset, 71.27% on the Beads^E S+F dataset and 86.88% on the Beads^E F dataset, and 73.30% on the Delta^E S+F dataset and 89.59% on the Delta^E F dataset.

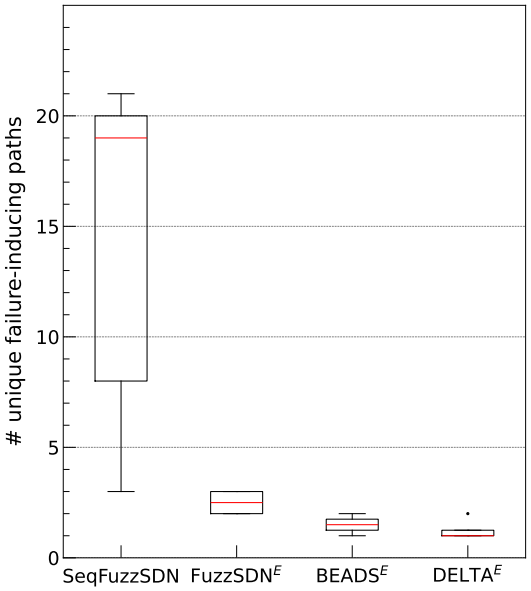
For FuzzSDN^E, Beads^E, and Delta^E, respectively, the figure (the last column of the 2nd, 3rd, and 4th rows) shows that their EFSMs' sensitivities are, on average, 22.06%, 14.40%, and 9.38% on the combined S+F dataset, and 25.00%, 4.53%, and 12.25% on the combined F dataset. Specifically, as shown in the first column of the figure, starting from the 2nd row, using the SeqFuzzSDN S+F dataset (and the SeqFuzzSDN F dataset), FuzzSDN^E, Beads^E, and Delta^E achieve, respectively, on average, sensitivities of 0.84%, 4.78%, and 3.68% (and 0.0%, 0.0%, and 4.55%). Regarding the FuzzSDN^E S+F dataset (and the FuzzSDN^E F dataset), as shown in the 2nd column of the figure, FuzzSDN^E, Beads^E, and Delta^E achieve, respectively, on average, sensitivities of 54.62%, 1.35%, and 0.00% (and 66.39%, 0.00%, and 0.00%). For the Beads^E S+F dataset (and the Beads^E F dataset), shown in the 3rd column, these three baselines achieve, respectively, on average, sensitivities of 0.00%, 6.37%, and 5.75% (and 0.00%, 0.00%, and 4.28%). Lastly, when using the Delta^E S+F dataset (and the Delta^E F dataset), these baselines achieve, respectively, on average, sensitivities of 0.00%, 5.64%, and 8.69% (and 0.00%, 0.00%, and 4.01%).

These results show that SeqFuzzSDN achieves, on average, a higher sensitivity compared to the baselines, and the differences are statistically significant. However, note that the EFSM produced by SeqFuzzSDN rejects most of the failure-inducing message sequences obtained from FuzzSDN^E, as SeqFuzzSDN and FuzzSDN^E use significantly different fuzzing methods. While FuzzSDN^E fuzzes a single message by modifying its fields' values, SeqFuzzSDN fuzzes a sequence of messages using multiple fuzz operators (i.e., delay, modification, duplication, deletion, and insertion). Consequently, the message sequences that lead to failure are significantly different between the two tools, resulting in producing very different EFSMs, which cannot accept the message sequences generated by the other tool. Even when the same failures are triggered, the generated traces differ due to these distinct paths. However, recall that the EFSM produced by FuzzSDN^E rejects most of the message sequences generated by SeqFuzzSDN, Beads^E, and Delta^E, indicating that the EFSMs are specific only to FuzzSDN^E.

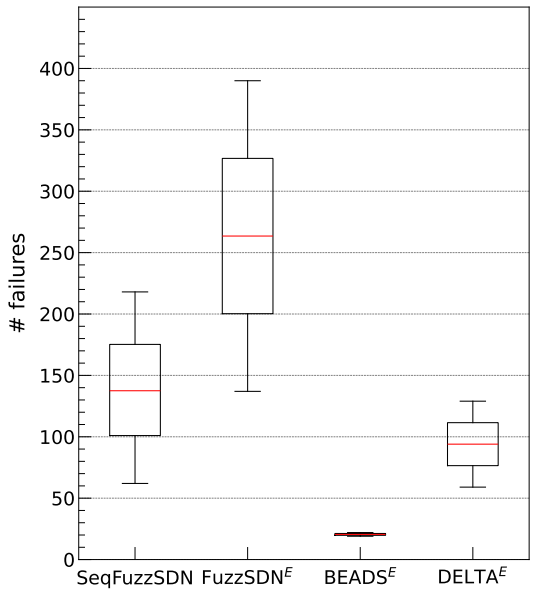
Figure 4.6 compares (a) the NCD scores of the message sequences, (b) the number of unique failure-inducing paths in the EFSMs, and (c) the number of message sequences leading to failure, which are obtained from 10 runs of SeqFuzzSDN, FuzzSDN^E, Beads^E, and Delta^E. Figure 4.6a shows that SeqFuzzSDN achieves a higher NCD score, with an average of 0.99, compared to those of the baselines. Figure 4.6b shows that, on average, SeqFuzzSDN was able to infer an EFSM containing 18 unique loop-free paths that lead to failure, which is significantly higher than the others. From these results, we found that SeqFuzzSDN generates more diverse sequences of control messages that exercise a larger number of state changes compared to the baselines.



(a) NCD scores



(b) Number of unique failure paths



(c) Number of failures

Figure 4.6: Comparing (a) the NCD scores of the message sequences, (b) the number of unique failure-inducing paths in the EFSMs, and (c) the number of message sequences leading to failure, all obtained from SeqFuzzSDN, FuzzSDN^E, Beads^E, and Delta^E. The boxplots (25%-50%-75%) show the distribution of each metric over 10 runs of each tool.

However, Figure 4.6c shows that FuzzSDN^E generates a larger number of message sequences (an average of 265) leading to failure compared to the other tools, while SeqFuzzSDN generates, on average, 140 message sequences leading to failure, thus outperforming Beads^E and Delta^E. Even though FuzzSDN^E outperforms SeqFuzzSDN in terms of number of failures, recall from Figure 4.6a and Figure 4.6b that FuzzSDN^E generates message sequences that are less diverse and exercise significantly fewer number of state changes compared to SeqFuzzSDN. Furthermore, as described in Section 4.3, SeqFuzzSDN aims to generate a balanced number of message sequences that lead to success and failure, rather than focusing solely on the latter.

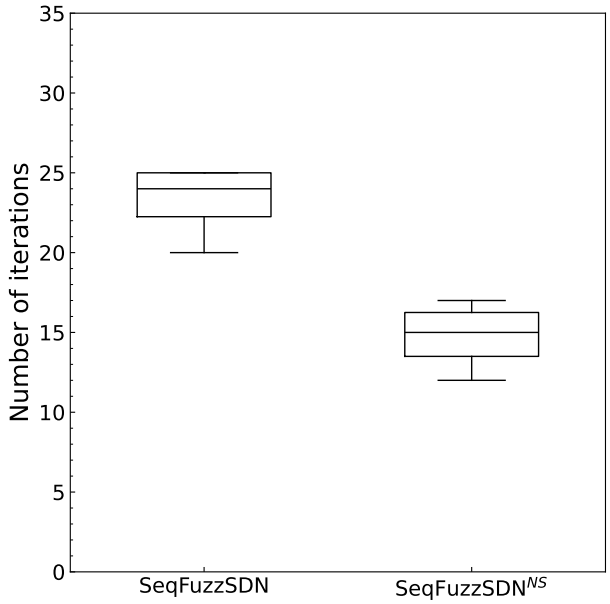


Figure 4.7: Comparing the number of iterations completed by SeqFuzzSDN and SeqFuzzSDN^{NS} within a 5-day time budget. The boxplots (25%-50%-75%) show the distribution of iteration counts over 10 runs of each tool.

The answer to **RQ1** *is that* SeqFuzzSDN significantly outperforms the baselines that extend FuzzSDN, Beads, and Delta. In particular, our experiment results indicate that SeqFuzzSDN can generate more diverse sequences of control messages leading to failure than those obtained from the baselines, while also providing EFSMs that accurately capture failure-inducing message sequences.

RQ2. Figure 4.7 presents a comparison of the number of iterations for the fuzzing, learning, and planning steps completed by SeqFuzzSDN and SeqFuzzSDN^{NS} within a time budget of 5 days. The boxplots show the distributions (25%-50%-75% quantiles) of the number iterations performed by SeqFuzzSDN and SeqFuzzSDN^{NS}, obtained from 10 runs of EXP2. As shown in the figure, SeqFuzzSDN can execute significantly more iterations than SeqFuzzSDN^{NS}. For a time budget of 5 days, SeqFuzzSDN completes, on average, 25 iterations, while SeqFuzzSDN^{NS} completes approximately 15 iterations. This result indicates that the sampling technique, which caps the maximum size of the dataset for Mint, allows SeqFuzzSDN to complete more iterations within the same time frame. In contrast, SeqFuzzSDN^{NS}, which permits the dataset to grow continuously over iterations, completes fewer iterations. Note that each iteration of SeqFuzzSDN (and SeqFuzzSDN^{NS}) tests the SDN controller 200 times; hence, the sampling technique enables SeqFuzzSDN to test the SDN controller, on average, 1000 times more than SeqFuzzSDN^{NS}.

In addition, Figure 4.8 compares SeqFuzzSDN and SeqFuzzSDN^{NS} with regard to the execution times per iteration for the fuzzing, learning, and planning steps over a time budget of 5 days. The bar graph shows the average execution times taken by SeqFuzzSDN and SeqFuzzSDN^{NS} for the fuzzing, learning, and planning steps at each iteration, based on 10 runs of EXP2.

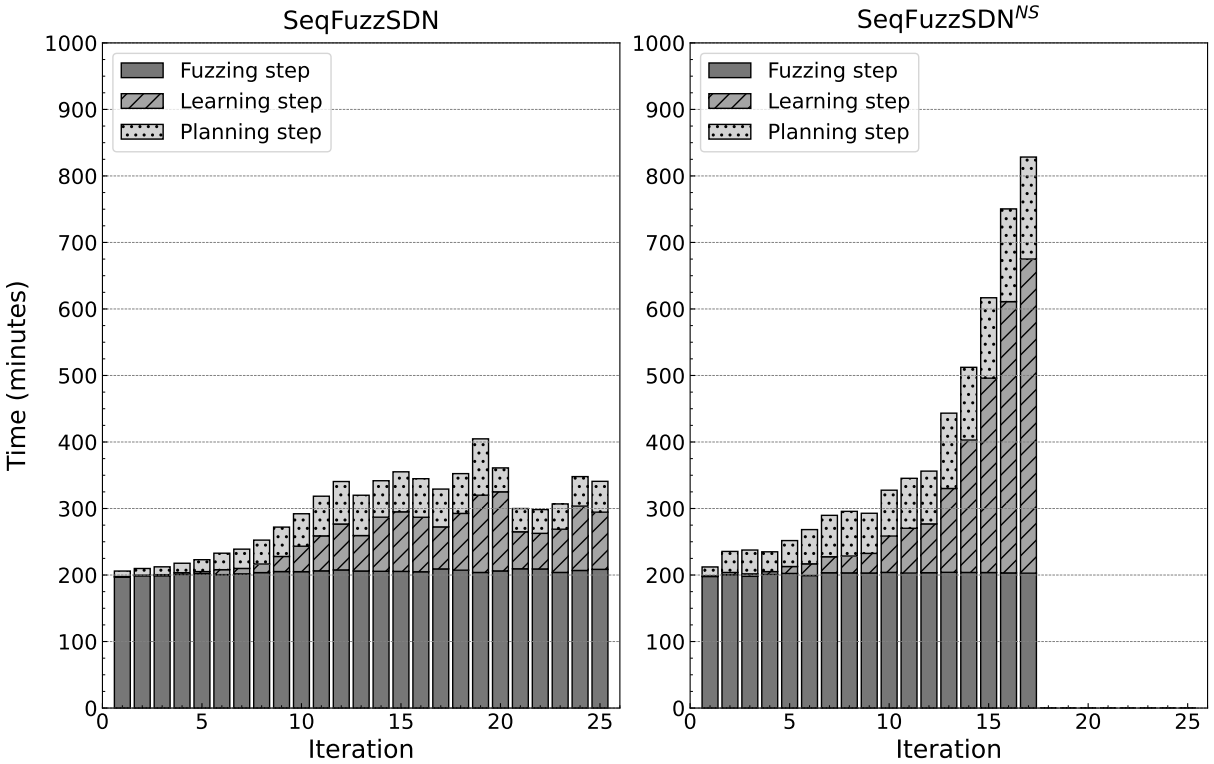


Figure 4.8: Comparing the execution time per iteration for the fuzzing, learning, and planning steps of SeqFuzzSDN and SeqFuzzSDN^{NS} within a 5-day time budget. The execution times shown in this figure are the average values observed over 10 runs of EXP2.

The results show that the fuzzing time per iteration remains constant at around 200 minutes for both SeqFuzzSDN and SeqFuzzSDN^{NS}, indicating that the fuzzing step is independent of the tool used. For the planning step, Figure 4.8 shows that the planning time does not exceed 150 minutes in both SeqFuzzSDN and SeqFuzzSDN^{NS}. Figure 4.8 also suggests that, for SeqFuzzSDN^{NS}, the time required to learn an EFSM increases exponentially with each iteration due to the growing size of the dataset fed to Mint. Furthermore, we observe that, on the 17th iteration of SeqFuzzSDN^{NS}, the learning time reaches the 12-hour timeout limit, thus preventing SeqFuzzSDN^{NS} from completing any further iterations. This finding aligns with the literature [110, 133, 109], as inferring EFSMs is a complex problem that scales poorly with larger input sizes. In contrast, the results for SeqFuzzSDN indicate that the time required for inferring an EFSM (i.e., the learning step) remains below 115 minutes due to the application of the sampling technique. Thus, based on the results shown in Figure 4.8, we can further conclude that applying the sampling technique enables SeqFuzzSDN to overcome the scalability issues associated with the complexity of learning EFSMs.

Furthermore, Table 4.3 presents the statistical test results for the distributions of sensitivity, diversity, and coverage (described in Section 4.3) achieved by SeqFuzzSDN and SeqFuzzSDN^{NS} after 10 runs of EXP2, using the Wilcoxon Rank-Sum test [134] with an α value of 0.05. On average, SeqFuzzSDN (resp. SeqFuzzSDN^{NS}) achieves a sensitivity of 54.2% (resp. 52.9%), a diversity of 0.9925 (resp. 0.9920), and a coverage of 0.5533 (resp. 0.6599). We observed that the differences in sensitivity ($p = 0.14$) and diversity ($p = 0.9$) are not significant, while the difference

Table 4.3: Statistical significance analysis using the Wilcoxon Rank-Sum test for sensitivity, diversity, and coverage results obtained from 10 runs of EXP2.

Metric	Average (SeqFuzzSDN)	Average (SeqFuzzSDN^{NS})	p-value	Statistical Significance ($\alpha = 0.05$)
Sensitivity	0.542	0.529	0.571	Not Significant
Diversity	0.9925	0.9920	0.297	Not Significant
Coverage	0.5533	0.6599	0.0124	Significant

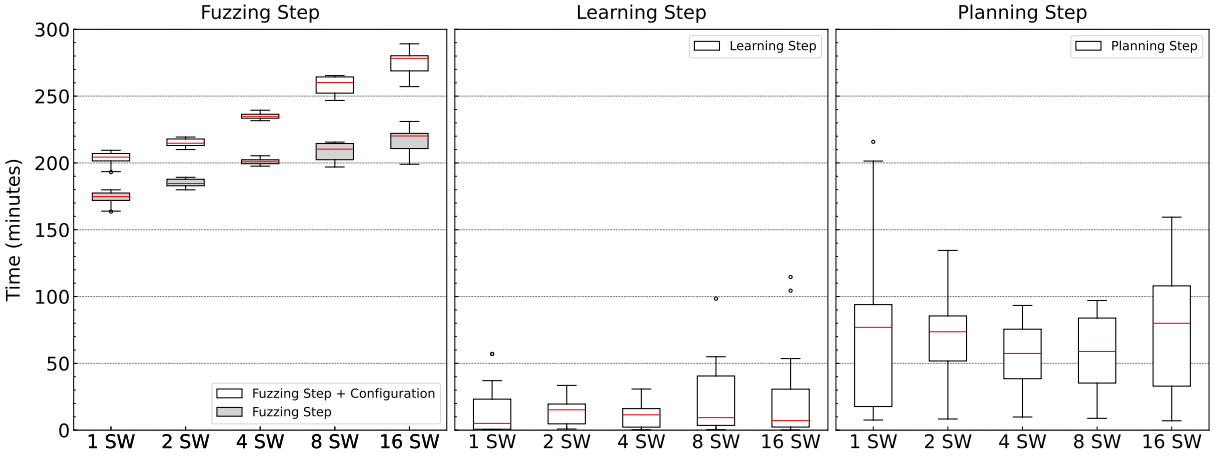


Figure 4.9: Boxplots (25%-50%-75%) representing the distributions of time taken in minutes for the fuzzing, learning, and planning steps of SeqFuzzSDN. This figure includes the times observed over 10 runs of SeqFuzzSDN with 1, 2, 4, 8, and 16 switch configurations.

in coverage ($p = 0.01$) is. The results indicate that the use of the sampling technique does not negatively impact the sensitivity of the generated EFSMs nor the diversity of the generated message sequences. However, the coverage achieved by SeqFuzzSDN has significantly improved, suggesting that the states in the EFSM are explored more thoroughly. One possible explanation for the improved coverage is that the increased number of iterations gives SeqFuzzSDN more opportunities to refine EFSMs with respect to the coverage objective targeted at the planning step.

The answer to **RQ2** *is that* the sampling technique introduced in SeqFuzzSDN reduces its computation cost, allowing for more iterations to be performed within a given time budget. This helps overcome scalability issues in inferring EFSMs without compromising the accuracy of the EFSMs and the diversity of the generated message sequences. Additionally, the sampling technique significantly improves SeqFuzzSDN's coverage, leading to a more thorough exploration of the search space.

RQ3. Figure 4.9 presents the distributions of execution times (25%-50%-75% boxplots) for the fuzzing, learning, and planning steps of SeqFuzzSDN. These execution times were measured using the five study subjects in EXP3, which consist of 1, 2, 4, 8, and 16 switches controlled by ONOS. As shown in Figure 4.9, the execution time taken for the fuzzing step is, on average, 203 minutes for the 1-switch configuration, 215 minutes for 2 switches, 235 minutes for 4 switches, 257 minutes for 8 switches, and 274 minutes for 16 switches. The learning step took, on average,

15 minutes for the 1-switch configuration, 14 minutes for 2 switches, 11 minutes for 4 switches, 25 minutes for 8 switches, and 26 minutes for 16 switches. The planning step took, on average, 79 minutes for the 1-switch configuration, 69 minutes for 2 switches, 56 minutes for 4 switches, 56 minutes for 8 switches, and 76 minutes for 16 switches.

The results show that there is no significant difference in the times required for the learning and planning steps across the five study subjects. However, the only time increase occurs during the fuzzing step, where test procedures are executed. This includes the time required to configure and teardown Mininet and the SDN controller. This increasing trend aligns with our expectations, as the execution time for a test procedure increases with its complexity. As described in Section 4.4.4, this is primarily due to the increasing number of messages exchanged between the switches and the controller as the number of switches and their connections grows [27] This increase in time is independent of SeqFuzzSDN, as it solely depends on the complexity of the test procedures executed. Note that when 16 switches are fully connected, the pairwise ping test procedure produces on average 30.73 control messages, whereas the same test procedure produces 10.46 control messages with only one switch.

The answer to **RQ3** *is that* the primary factor affecting the execution time of SeqFuzzSDN is its fuzzing time, which is influenced by the number of control messages generated by a test procedure. Consequently, SeqFuzzSDN is applicable to complex systems with large networks, provided that the execution time of a test procedure remains within an acceptable time budget.

4.4.7 Threats to Validity

Internal validity. To address potential threats to internal validity, we compared SeqFuzzSDN against three state-of-the-art tools (Delta, Beads, and FuzzSDN), which have been used to generate failure-inducing control messages for testing SDN controllers. However, Delta, Beads, and FuzzSDN do not generate failure-inducing models that consider the sequences of messages exchanged between the controller and switches. Consequently, we extended these tools as baselines to produce EFSMs, allowing for a comparative analysis between SeqFuzzSDN and these baselines.

External validity. The principal external validity threat to SeqFuzzSDN is the risk that it may not be adaptable to different contexts, such as other SDN systems with different switch configurations or controllers. To address this potential threat, we conducted experiments with SeqFuzzSDN against multiple SDNs and two popular SDN controllers found in the literature, namely ONOS and RYU. We varied our synthetic systems, which comprise five networks with 1, 2, 4, 8, and 16 switches, respectively, managed by either ONOS or RYU.

Additionally, the prototype implementation of SeqFuzzSDN is compatible with OpenFlow, a widely accepted standard protocol for SDNs, which has been used in numerous SDN studies and

practices [34, 35, 38, 54, 53, 37]. As a consequence, we were able to successfully apply SeqFuzzSDN to real-world SDN controllers (ONOS and RYU) and compare it to existing tools (i.e., DELTA, BEADS, and FuzzSDN), considering their support for OpenFlow. To utilise SeqFuzzSDN with systems that incorporate other SDN protocols, such as Cisco OpFlex [99] and ForCES [100], it is necessary to modify the sniffing and injection mechanisms of SeqFuzzSDN to decode and encode control messages. However, these modifications do not affect the fuzzing, learning, and planning steps. Therefore, we anticipate that, although such modification necessitates engineering effort to revise the sniffing and injection mechanisms, they do not impact SeqFuzzSDN's efficiency and effectiveness. However, to further explore the applicability of our findings, it is essential to conduct additional case studies in various settings. This includes industrial systems that use different SDN protocols and user studies that involve practitioners.

4.5 Related Works

In this section, we discuss related works in the areas of SDN testing, fuzzing, and characterising failure-inducing inputs. Readers familiar with our previous work may notice significant similarities. This is because this work builds upon, and extends our previous research. As such, much of the foundational literature and related work remain relevant and are thus referenced here. We believe this will provide a comprehensive context for both new readers and those familiar with our prior work.

SDN testing. The study of SDN testing in the networking literature focuses on various objectives, such as detecting security vulnerabilities and attacks [49, 50, 35, 34, 51, 36, 52], identifying inconsistencies among the SDN components (i.e., applications, controllers, and switches) [135, 37, 38], and analysing SDN executions [54, 55, 56]. In this discussion, we focus on SDN testing methods that utilise fuzzing, as they are the most relevant to our research. Lee et al. [37, 62] proposed AUDISDN, a framework that employs a fuzzing technique to detect policy inconsistencies among SDN components (i.e., controllers and switches). AUDISDN relies on the fuzzing of network policies configured by the administrators through the REST APIs of the SDN components. To increase the probability of uncovering inconsistencies, AUDISDN restricts valid relationship elements by building rule dependency trees from the specification of the OpenFlow protocol. RE-CHECKER, proposed by Woo et al. [57], is designed to fuzz the RESTful services offered by SDN controllers. It fuzzes an input file in JSON format, which is used by a network administrator to define network policies, such as data forwarding rules. This process generates a large number of malformed REST messages for testing RESTful services in SDN. Dixit et al. [58] introduced AIM-SDN to test the implementation of the Network Management Datastore Architecture (NMDA) in SDN. AIM-SDN uses random fuzzing of REST messages to test the NMDA implementation in SDN, focusing on the availability, integrity, and confidentiality of datastores. Shukla et al. [38] created PAZZ, which is designed to identify faults in SDN switches by fuzzing data packet headers, such as IPv4 and IPv6 headers. Finally, Albab et al. [60] introduced SWITCHV to verify the behaviours of SDN switches. SWITCHV employs

fuzzing and symbolic execution to analyse the p4 models that define the behaviours of SDN switches. In contrast to these methods, SeqFuzzSDN fuzzes SDN control messages to test SDN controllers, similar to DELTA, BEADS, and FuzzSDN. Furthermore, SeqFuzzSDN uses learned EFSMs to guide the fuzzing process and characterise the messages sequences that may lead to a system failure.

Fuzzing and Stateful Testing. To efficiently generate effective test data, fuzzing has been widely applied in many application domains [101]. The research strands that most closely relate to ours are stateful fuzzing techniques [45, 46, 47, 48] Numerous research studies have explored the use of FSMs and EFSMs for testing complex systems. Gascon et al. [46] proposed PULSAR, a stateful black-box fuzzing technique aimed at discovering vulnerabilities in proprietary network protocols. Their proposed approach involves the inference of a Markov model (Deterministic Finite Automaton) from network traces, which are used to generate test cases using fuzzing primitives (i.e., paths in the automaton) defined by the model, and finally the selection of the test cases that maximise the coverage of the protocol stack. Pham et al. [47] proposed AFLNET, a grey-box fuzzer for network protocols implementation, based on AFL [42]. Their proposed technique takes a mutational approach and states feedback to guide the fuzzing of network-enabled servers. AFLNET takes as input a corpus of server-client network and subsequently acts as a client. It replays modified versions of the initial message sequence sent to the server, preserving only the alterations that successfully expanded the coverage of the code or state space. From the newly discovered message sequences, AFLNET uses the server's response codes to build an FSM that describes the protocol states. From those inferred FSMs, their approach identifies regions in the state space that have been the least explored and systematically steers the fuzzing process towards the test of such regions. Natella [48] proposed STATEAFL, a grey-box fuzzing technique that infers FSMs based on the in-memory states of a server, leveraging compile-time instrumentation and fuzzy hashing techniques; hence, it does not require response codes. During the fuzzing process, STATEAFL guides the generation of new inputs to the server based on the inferred FSMs. It employs both byte-level and message-level fuzz operators, which do not rely on protocol specifications. Kim et al. [136] proposed AMBUSHER, a protocol-state-aware fuzzing technique for testing the *"East-West"* protocol of distributed SDN controllers. AMBUSHER takes as input a test configuration which includes the alphabet of the protocol used as well as the cluster information. In its first phase, AMBUSHER uses a dummy network node to generate queries between the controllers, and a dummy controller to log such queries generated in the network. In its second phase, the logged cluster queries are then used by a state machine learner to infer a state machine of the cluster's protocol. In its third phase, AMBUSHER explores the inferred state machine to extract message sequences. Those message sequences are then used as seeds for the fuzzing process, in which attack scenarios are generated by randomising the message sequences. In its final phase, AMBUSHER leverages the randomised sequences to test the cluster *"East-West"* interfaces. Among these, AMBUSHER is the most relevant to SeqFuzzSDN, as both take into account the SDN architecture, which differs from the server-client architecture. Compared to AMBUSHER, SeqFuzzSDN fuzzes and infers EFSMs based on sequences of control messages exchanged through the control channel of the SDN (i.e., *"South"* interface). To our

knowledge, SeqFuzzSDN is the first SDN testing method that focuses on the *"South"* interface of SDN controller while accounting for the statefulness of SDNs.

Characterising failure-inducing inputs. Recently, several research efforts have focused on identifying the input conditions that cause a system under test to fail [15, 81]. Gopinath et al. [15] introduced DDTEST, which abstracts inputs that lead to failures. DDTEST is designed to test software programs, such as JavaScript translators and command-line utilities, that accept string inputs. It uses a derivation tree to represent how failure-inducing strings are generated. Kampmann et al. [81] developed ALHAZEN, which identifies the conditions under which software programs fail. ALHAZEN also targets software that processes strings and uses machine learning to learn failure-inducing conditions in the form of decision trees. In the domain of SDN systems, Ollando et al. [103] introduced FuzzSDN, a machine learning-guided Fuzzing method for testing SDN controllers. FuzzSDN learns an interpretable classification model that characterises conditions on a control message's fields under which the controller fails. We note that these methods do not attempt to create a failure-inducing model for sequential data, which makes those methods not suitable for our objectives. To our knowledge, SeqFuzzSDN is the first approach that applies an EFSM-guided fuzzing approach to infer failure-inducing models, in the form of EFSMs, with a focus on SDNs. Specifically, SeqFuzzSDN tests SDN controllers by accounting for the architecture and protocols unique to SDNs, which differ from other systems (e.g., server-client systems). Further, SeqFuzzSDN tests SDN controllers without requiring any modifications or instrumentation of the controllers or their networks, enabling the SDN testing in realistic operational settings.

4.6 Conclusions

We developed SeqFuzzSDN, a learning-guided fuzzing method for testing stateful SDN controllers. SeqFuzzSDN uses a fuzzing strategy, guided by EFSMs, in order to (1) efficiently explore the space of states of the SDN controller under test and (2) infer EFSMs that characterise the sequence of messages that may make the system fail. SeqFuzzSDN implements an iterative process that fuzzes sequences of control messages, learns an EFSM, and plans how to guide the subsequent fuzzing steps by leveraging the learned EFSM. We evaluated SeqFuzzSDN on several synthetic systems controlled by two different SDN controllers. In addition, we compared SeqFuzzSDN against our extended versions of three SOTA methods for testing SDN controllers, which served as baselines in our evaluation. Our results show that SeqFuzzSDN significantly outperforms the baselines by generating effective and diverse tests (i.e., sequences of control messages), that cause the system to fail, and by producing accurate EFSMs.

In the future, we'll devise a learning technique that will allow SeqFuzzSDN to learn stateful models incrementally, addressing scalability issues in inferring EFSMs. This poses new challenges due to the complexity of continuously updating and maintaining complex dependencies between states and transitions, without losing any of the previously learned information. Further, we also

aim to confirm the applicability and effectiveness of SeqFuzzSDN by testing it on more SDN systems and performing user studies.

Data Availability

Our evaluation package and the SeqFuzzSDN tool can be accessed online [105] to allow researchers and practitioners to (1) reproduce our experiments and (2) utilize and modify SeqFuzzSDN.

Chapter 5

Test schedule generation for acceptance testing of mission-critical satellite systems

5.1 Introduction

Mission-Critical Systems (MCS), such as satellite systems, healthcare systems, or nuclear power plant control systems, are developed and rigorously tested to ensure they meet specific operational requirements before being put into operation. MCSs, require some additional testing phases during their life-cycle, which are referred to as *Operational Acceptance Testing* [11]. Operational acceptance testing is required for MCSs to ensure that they meet all specified operational requirements, function correctly under real-world conditions, and are ready for deployment and sustained operation.

In satellite development and operation, *In-Orbit Testing* (IOT) is an important operational acceptance testing activity. IOT is routinely performed following the successful deployment of a satellite, where various subsystems of the satellite are tested while in orbit. The aim of such tests is to compare the performance of the satellite with its pre-launch data and tests, ensuring that no degradations have occurred due to the stresses of launch, and the environmental conditions in space [137]. Additionally, IOT aims to confirm that the satellite's operations meet the specified requirements.

IOT involves scheduling the testing campaign, i.e., the test suite, which comprises various test procedures to be performed on the satellite under test, which is part of a constellation of satellites. This scheduling is inherently complex, as it must account for several factors, such as the frequency and duration of the satellite's visibility to a specific antenna, the cost associated with antenna usage, and the time required to configure and orient the antennas before each test.

This chapter addresses the gap in the literature regarding Operational Acceptance Testing (OAT) for mission-critical satellite systems. While most research focuses on User Acceptance Testing (UAT) and test case prioritization for software regression testing, our work accounts for the specificity of IOT in satellite systems, such as antenna-related, operational cost, and context-switching constraints. Previous methods, such as those proposed by Shin et al. [138], automate test case prioritization for Cyber-Physical Systems (CPS) by considering time budget constraints, uncertainty, and hardware damage risks. However, our approach extends this by specifically addressing the scheduling challenges and resource constraints unique to satellite systems. Unlike Shin et al. [138], we account for factors like antenna usage costs and the time required for antenna configuration and orientation. In the domain of test case prioritization, works by Arrieta et al. [139, 140] and Wang et al. [141] focus on optimizing test execution time and success rates within CPS configurations. These methods, however, do not consider the scheduling conflicts or the availability of operators, which are critical in our context. Our approach integrates these aspects, ensuring that resource availability and operator constraints are factored into the scheduling process. Furthermore, while the satellite control resource scheduling problem (SCRSP) and ground measurement and control resource allocation (GMCRA) have been explored by Marinelli et al. [142], Zhang et al. [143], Gao et al. [144], Wu et al. [145], Zhang et al. [146, 147] and others, their methods primarily address satellite communication requests and do not fully cater to the specific needs of IOT scheduling. Our work builds on these foundations but introduces additional considerations for the frequency and duration of satellite visibility, antenna usage costs, and the logistical challenges of the antenna configuration overhead for each test.

Contributions. This chapter addresses the problem of scheduling the IOT campaign in an efficient and effective manner. Specifically, our contributions are as follows: (1) *A multi-objective approach for scheduling acceptance tests for mission-critical satellite systems.* Our approach includes (a) a precise definition of the problem of scheduling the IOT campaign, which accounts for schedule objectives and constraints; (b) an algorithm based on Non-dominated Sorting Genetic Algorithm III (NSGA-III [16]) for finding near-optimal feasible IOT schedules; and (c) fitness functions that evaluate the performance of IOT schedules by assessing their operational cost, fragmentation (as fragmented IOT schedules incur overheads), and efficiency in the use of test resources. (2) *An industrial case study.* We applied our approach to a representative Global Navigation Satellite System (GNSS), for which SES Techcom, our industrial partner, provides operational services. Our results show that an IOT campaign scheduled using our search-based approach, compared to a random search approach, finds feasible schedules that achieve an average improvement of 49.4% in the cost fitness, 60.4% in the fragmentation fitness, and 30% in efficiency of the test resource usage fitness. In addition, our approach demonstrates that,

compared to Ant-Colony Optimization (ACO) approaches, which have been extensively cited in the literature. Specifically, our method outperforms a tailored baseline approach by 53.1% in cost efficiency, 58.3% in fragmentation, and 26.1% in efficiency of resource usage over the same period. Moreover, it provides practitioners with several equally viable schedules, enabling comprehensive trade-off analyses. In addition, our approach yields schedules that improve the cost efficiency by 538%, and the efficiency of the test resource usage by 39.42% compared to schedules manually constructed by practitioners, while maintaining comparable performance in terms of fragmentation and requiring only 12.5% of the time needed by practitioners to construct an IOT schedule. (3) *Practitioners' feedback on our IOT scheduling approach.* Finally, we interviewed practitioners at SES to collect feedback on our approach. They highlighted the following: (a) the efficiency of schedule generation, as our automated approach generates feasible schedules much faster than manual methods, enabling quick adaptation to changing conditions and needs, and (b) the ability to produce several equally viable schedules, facilitating trade-off analysis.

Organization. In this chapter, we first describe the background of this research work in Section 5.2. Next, Section 5.3 describes our approach to generate test schedules for acceptance testing of mission-critical satellite systems. In Section 5.4, we perform our empirical evaluation, discuss threats to validity, and present lessons learned from interviews with practitioners. Finally, Section 5.6 concludes the chapter.

5.2 Background

5.2.1 Motivating Case Study

We motivate our work using a case study from our industry partner, SES Techcom, which develops satellite-enabled solutions. Operators of satellites are tasked with ensuring optimal performance of their satellites' services once deployed in orbit. Given the critical role of satellite technology in supporting various services, such as broadcast television, global navigation and positioning systems, mobile communications, and other communication systems, operators must ensure that, over the lifespan of a satellite, the Quality-of-Service (QoS) remains within the standards defined by its application. Consequently, operators routinely conduct *In-Orbit Testing* (IOT) procedures to monitor the QoS of each satellite in the constellation they operate. These IOT procedures have four main objectives: (1) ensuring the behavior of the satellite remains consistent before and after launch; (2) verifying performance adherence to specifications, (3) forecasting end-of-life; and (4) investigating potential anomalies.

Our industry partner, SES Techcom conducts routine monthly tests for the European GNSS constellation, Galileo. In this context, *IOT* procedures are divided into two categories: *Signal Quality Monitoring* (SQM) and *Routine In-Orbit Test* (RIOT). SQM procedures measure the

satellite's signal quality and strength on each communication channel. Specifically, these procedures involve measuring the Modulated Effective Isotropic Irradiated Power (EIRP) [40, 41] approximately 15 times on each channel, with the overall testing duration lasting almost one hour per satellite. The SQM procedures are usually performed at the highest elevation available at any given pass of a satellite. RIOT procedures are performed sequentially throughout the full pass of the satellite, from the signal acquisition, typically around 3-5 degrees of elevation, until signal loss at a similar elevation. The duration of an RIOT phase ranges from 8 to 9 hours, depending on the satellite and the ground measuring station. During an RIOT phase, several IOT measurements are performed for every Galileo channel. Specifically, these IOT measurements include Modulated EIRP, IQ sample collection, out-of-band spurious measurement, navigation receiver data analysis, and Search-and-Rescue (SAR) check, if SAR is available [41].

Additionally, the antennas used to communicate with the satellite are large objects that require time to be precisely pointed toward the satellite under test. Due to the precise nature of satellite communication, test instruments and antenna alignment may need to be re-calibrated before conducting each test procedure. These factors introduce delays before conducting each test procedure in an IOT campaign, during which no tests can be performed, and must be taken into consideration in the scheduling process.

Currently, practitioners at SES Techcom manually schedule these *IOT* procedures, having determined that existing automated solutions are not practically applicable to their scheduling needs. However, this manual approach poses significant challenges and consumes valuable time for practitioners, particularly when the satellites' orbits have short revolution periods or substantial inclinations. Moreover, in the event of an emergency scenario, such as an unexpected degradation in QoS across the constellation, an IOT campaign must be scheduled and executed within a condensed timeframe. With the Galileo constellation currently consisting of 23 satellites in orbit (soon to be 25), this presents a considerable challenge for the IOT operators. Hence, an algorithm that automatically solves the problem of scheduling IOT campaigns in practical time is highly desirable.

5.2.2 IOT Requirements and Constraints:

To create a suitable schedule for an IOT campaign, several key factors specific to the problem must be considered.

Context switching. Minimizing context switching for satellite operators is essential for maintaining the efficiency and accuracy of the IOT campaign. Frequent transitions between IOT procedures and other satellite operation tasks can increase cognitive load for operators, raising the likelihood of errors. Additionally, context switching incurs time and resource costs, as operators must reorient themselves with each new task. Streamlining workflows and grouping similar tasks can reduce the need for context switching.

Utilization of test resources. Efficient use of IOT resources is crucial, ensuring that the equipment used for the IOT procedures (e.g., antennas, satellites, and test devices) is optimally utilized with minimal interruptions. Efficient IOT schedules ensure that these hardware resources are not overused, reducing not only operational costs but also the risk of hardware failures. Inefficient test schedules increase the likelihood of hardware malfunctions due to several factors, such as overheating and exposure to harsh environmental conditions. Additionally, efficient IOT schedules limit exposure to external disruptions, such as power outages, ensuring the integrity of the tests. Studies show that the probability of hardware failure rises with continuous operation [148]. By keeping the test campaign efficient, engineers can maintain optimal equipment performance and achieve more reliable results.

Operational costs. The operational costs associated with performing IOT campaigns are significant and multifaceted. These costs include the expenses related to the use of the antennas. Additionally, there are costs associated with allocating human resources, including the personnel required to operate the IOT campaign. Efficient management of these resources is essential for cost optimization.

5.3 Approach

This section describes our approach to addressing the following problem: For an IOT campaign to test satellites in a constellation, how can we create suitable IOT schedules that (1) enable efficient use of the antenna resources required for the test procedures in the IOT campaign, (2) reduce the frequency of context switching for operators conducting the IOT campaign, and (3) minimize the costs directly associated with executing the test procedures.

5.3.1 IOT Scheduling Concepts

Our approach utilizes four concepts to find the most suitable schedule for an IOT campaign, *satellite passes*, *test procedures*, *procedure schedules*, and *slot schedules*. Below, we precisely describe these concepts.

Satellite pass. Any satellite orbiting the Earth, except for those in geostationary orbit, can only be observed from a specific location on Earth during the period when the satellite passes in the visibility range of the ground station. We refer to this period as a *satellite pass*. A satellite pass begins when the satellite rises above the horizon, reaches its zenith (highest elevation in the sky), and ends when it descends below the horizon. During a pass, various activities such as communication, data collection, or observation activities between the satellite and ground stations can take place. Furthermore, we define a pass of a satellite s over a location r, denoted

α_r^s, as follows:

$$\alpha_r^s = \{t_{start}, t_{max}, t_{end}, \theta_{start}, \theta_{max}, \theta_{end}, \phi_{start}, \phi_{max}, \phi_{end}\}$$

where t_{start}, t_{max}, and t_{end} represent the time at which satellite s begins its pass, reaches its maximum elevation, and finishes its pass at location r, respectively; θ_{start}, θ_{max}, and θ_{end} represent the elevation angles at which s begins its pass, reaches its maximum elevation, and finishes its pass at r respectively; and ϕ_{start}, ϕ_{max}, and ϕ_{end} represent the azimuth angles at which s begins its pass, reaches its maximum elevation and finishes its pass at r, respectively. Similarly, we define $\Gamma_r^s(t_1, t_2) = \{\alpha_r^s \mid \alpha_r^s \text{ occurs between } t_1 \text{ and } t_2\}$ the set of satellite passes of satellite s over location r during a time period ranging from t_1 and t_2.

Test procedure. A *Test procedure* refers to a specific IOT procedure that is to be performed on a given satellite s. Much like a satellite pass, a test procedure is characterized by a period during which the IOT procedure is conducted. Formally, we define a test procedure associated with a satellite s, labeled τ_s, as follows:

$$\tau_s = \{t_{start}^s, t_{end}^s, \text{Type}, \delta_c, \alpha_r^s\}$$

where t_{start}^s and t_{end}^s represent the start and end times of the test procedure, respectively, Type represents the type of test procedure that is performed, δ_c represents the configuration time required before performing the test procedure (e.g., repositioning the antenna, booting the equipment, etc.), and α_r^s is the associated satellite pass. We note that $t_{start}^s < t_{end}^s$, $t_{start} \leq t_{start}^s$, and $t_{end}^s \leq t_{end}$, where $t_{start} \in \alpha_r^s \in \tau_s, t_{start}^s \in \tau_s, t_{end}^s \in \tau_s$, and $t_{end} \in \alpha_r^s \in \tau_s$.

Moreover, we can define the *span* between two individual test procedures τ_i and τ_j, where τ_i occurs before τ_j, denoted $\text{SPAN}(\tau_i, \tau_j)$ as the elapsed time between the beginning of τ_i and the end of τ_j, formally defined as follows:

$$\text{SPAN}(\tau_i, \tau_j) = \Delta t(t_{start}^i, t_{end}^j)$$

Procedure schedule. A *procedure schedule* consists of a collection of test procedures over a defined timeframe. Formally, we define a schedule, denoted as $\mathcal{S}$, as follows:

$$\mathcal{S} = \{\tau_1, \tau_2, \ldots, \tau_n\}$$

where each $\tau \in \mathcal{S}$ corresponds to an individual test procedure, as defined previously. Notably, we can define the *span* of a schedule $\mathcal{S}$ as the time elapsed between the beginning of the first test procedure, and the end of the last test procedure int $\mathcal{S}$, denoted $\text{SPAN}(\mathcal{S})$, and defined as follows:

$$\text{SPAN}(\mathcal{S}) = \Delta t \left(\min_{\tau_i \in \mathcal{S}} t_{start}^i, \max_{\tau_i \in \mathcal{S}} t_{end}^i \right)$$

Slot schedule. A *slot schedule* refers to the collection of time slots during which an operator's resources (e.g., engineers, antennas, and equipment) are allocated for performing the IOT

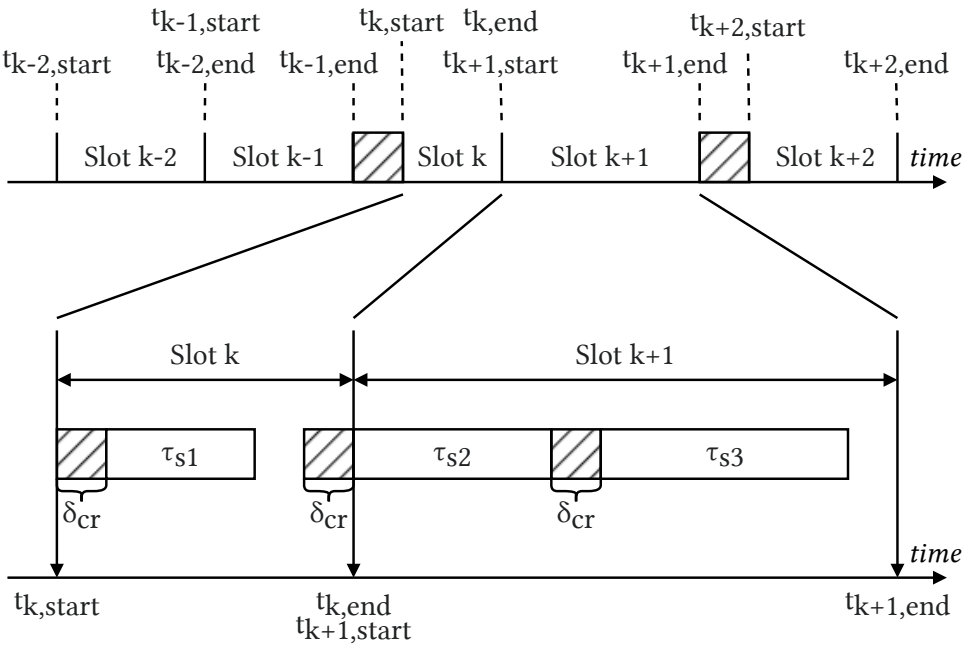


Figure 5.1: Example of a slot schedule and the corresponding slots.

procedures defined in a procedure schedule. Specifically, each procedure schedule $\mathcal{S}$ is associated with a unique slot schedule, denoted $\mathcal{Q}$, so that $\mathcal{Q} = \{(t_{j,start}, t_{j,end}) \mid j \in \{1, 2, \ldots, n\}\}$, where n is the number of time intervals, while $t_{j,start}$ and $t_{j,end}$ are the start and end time of the j-th interval, respectively. Figure 5.1 illustrates the relationship between a slot schedule and a procedure schedule. The top portion of the figure depicts a slot schedule consisting of five slots, defined as $\mathcal{Q} = \{(t_{k-2,\text{start}}, t_{k-2,\text{end}}), (t_{k-1,\text{start}}, t_{k-1,\text{end}}), (t_{k,\text{start}}, t_{k,\text{end}}), (t_{k+1,\text{start}}, t_{k+1,\text{end}}), (t_{k+2,\text{start}}, t_{k+2,\text{end}})\}$. The bottom portion of the figure represents three test procedures, τ_{s1}, τ_{s2}, and τ_{s3}, extracted from the procedure schedule $\mathcal{S}$. This figure demonstrates that the slots encompass the procedures within $\mathcal{S}$. Specifically, τ_{s1} is contained within the k-th slot of $\mathcal{O}$, while τ_{s2} and τ_{s3} are contained within the $(k+1)$-th slot. Additionally, the figure highlights that slots do not need to be contiguous or temporally aligned with test procedures.

IOT schedule. An *IOT schedule* is the outcome of the scheduling process, encompassing both the procedure schedule and the slot schedule. Formally, an IOT schedule, denoted as $\mathcal{P}$ is defined as:

$$\mathcal{P} = (\mathcal{S}, \mathcal{Q})$$

where $\mathcal{S}$ represents the procedure schedule and $\mathcal{Q}$ represents the slot schedule.

5.3.2 Identifying Conflicting Test Procedures

Conflict definition. As explained in Section 5.2, scheduling IOT procedures for a constellation of satellites is a challenging activity that involves several constraints. When a test procedure is

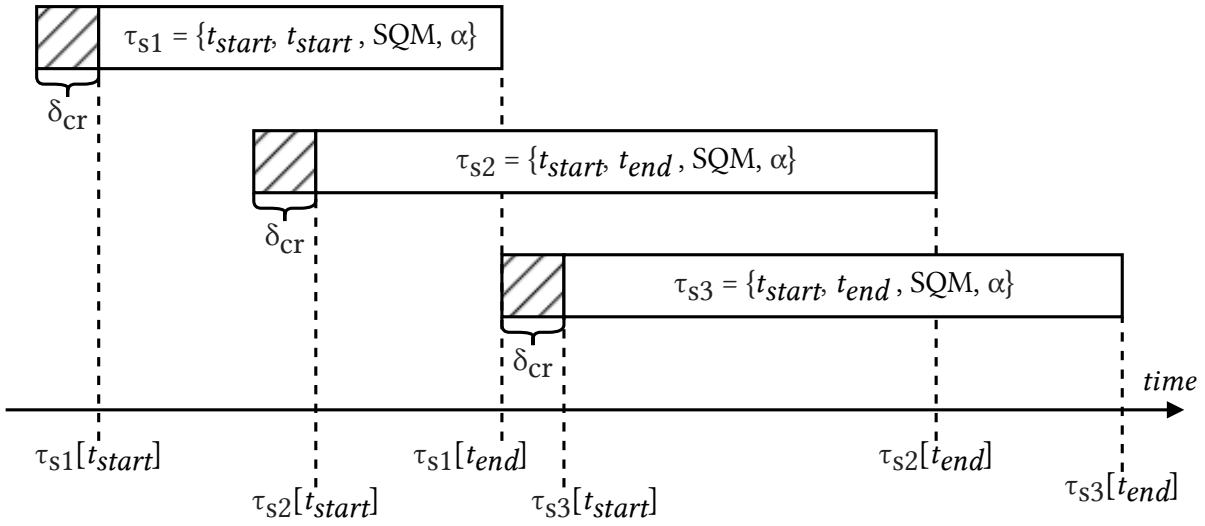


Figure 5.2: Three conflicting test procedures.

selected for scheduling, conflict may arise with other test procedures; they cannot be executed simultaneously due to resource contention, temporal constraints, or other dependencies.

As an example, consider an IOT campaign of the Galileo constellation, our test subject provided by SES Techcom, where a minimum of four satellites are constantly visible in the sky at all times. The IOT campaign is subject to the following constraints: Only a single antenna is available for use, meaning that only one test procedure can be conducted at any given time. The reconfiguration overhead, which includes the time required to program and orient the antenna before initiating a test procedure, is 15 minutes. SQM test procedures must be conducted for 45 minutes, centered around the satellite's highest elevation point. Recall from Section 5.2 that the purpose of an SQM is to detect potential hazardous deformations in the signal emitted by the satellite. Additionally, RIOT procedures are required to be performed for the entire duration of a satellite pass. This duration is defined as the period when the satellite's elevation is between 5° at the start and end of the pass. Recall from Section 5.2 that, the purpose of an RIOT is to test various capabilities of the satellite. These constraints imply that RIOT test procedures cannot be scheduled for concurrent testing with any other test procedures, and SQM test procedures may not be scheduled for concurrent testing either. This highlights why identifying conflicts is necessary for creating IOT schedules.

Let $\mathcal{T}$ be a set of test procedures involved in the creation of an IOT schedule. The set $\mathcal{T}$ is constructed by engineers who assign to each satellite pass $\alpha_s^r \in \Gamma_r^s$ an SQM or a RIOT test procedure, if applicable. We define, for each $\tau_i, \tau_j \in \mathcal{T}, \tau_i \neq \tau_j$, the "conflict" ξ_{τ_i,τ_j} between τ_i and τ_j such that $\xi_{\tau_i,\tau_j} = 1$ when there is a conflict and $\xi_{\tau_i,\tau_j} = 0$, otherwise. Subsequently, for a set of test procedures $\mathcal{T}$, we can define a set $\Xi(\mathcal{T})$ representing the tuples of test procedures that conflict with each other within $\mathcal{T}$, as follows:

$$\Xi(\mathcal{T}) = \{(\tau_i, \tau_j) \mid i, j \in \{1, 2, \ldots, |\mathcal{T}|\}, i \neq j, \xi_{\tau_i,\tau_j} = 1\}$$

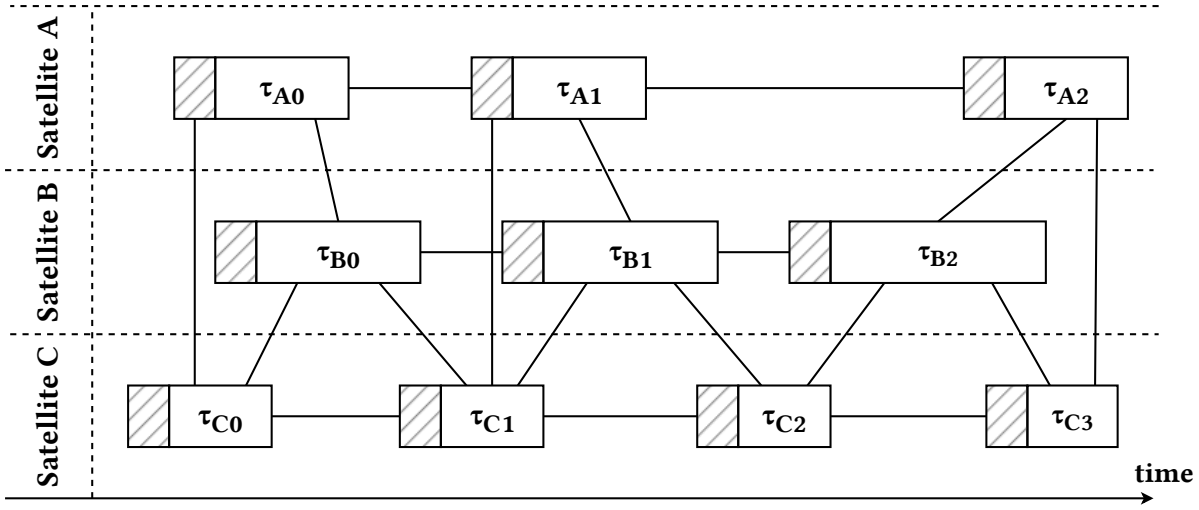


Figure 5.3: Example of a conflict graph created from the passes of three satellites (A, B, and C)

As an illustration, let us consider three test procedures τ_{s1}, τ_{s2}, and τ_{s3}, depicted in Figure 5.2. These test procedures have to be scheduled for three satellites s_1, s_2, and s_3, respectively, all from the same location r, and are of the identical type "SQM". We can see that the test procedure τ_{s2} starts after τ_{s1} begins but before τ_{s1} ends. Consequently, there is a conflict between τ_{s1} and τ_{s2}, resulting in $\xi_{\tau_{s1},\tau_{s2}} = 1$. Similarily, we observe a similar conflict between τ_{s2} and τ_{s3}, resulting in $\xi_{\tau_{s2},\tau_{s3}} = 1$. However, we can observe that no conflict occurs between τ_{s1} and τ_{s3}, resulting in $\xi_{\tau_{s2},\tau_{s3}} = 0$. Thus, we have, for this set of test procedures, the following conflict set $\Xi(\mathcal{T}) = \{(\tau_1, \tau_2), (\tau_2, \tau_3)\}$.

Conflict graph. Based on the previous definition of conflicting test procedures, we elect to represent the conflicts in a set of test procedures as a *conflict graph*. Conflict graphs are undirected graphs $G = (V, E)$, where each vertex corresponds to a unique test procedure and each edge $(i, j) \in E$ represents the presence or absence of a conflict between a test procedure i and a test procedure j [149]. Formally, for a set of test procedures $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$, we define a conflict graph as $G(\mathcal{T}) = (\mathcal{T}, \Xi(\mathcal{T}))$, where $\mathcal{T}$ is the set of vertices of G and $\Xi(\mathcal{T})$ is the set of edges. Figure 5.3 presents a conflict graph derived from the satellite passes of three satellites A, B, and C. Each row of Figure 5.3 corresponds to candidate test procedures to be conducted for a given satellite. For instance, in the first row, τ_{A0}, τ_{A1}, and τ_{A2} represent three separate candidate test procedures for satellite A. The test procedures are organized horizontally according to their time of occurrence. For instance, τ_{A0} occurs before τ_{A1}, but τ_{A0} overlaps with τ_{B0} and τ_{C0}. Each hatched square represents the required configuration time δ_{cr} for each task in the graph. Edges between nodes indicate conflicts between pairs of test procedures, meaning those procedures cannot be part of the same IOT schedule. For example, an edge exists between τ_{A0}, τ_{A1}, and τ_{A2} because they are different candidate test procedures of the same type for satellite A, and thus cannot be part of the same candidate IOT schedule. Similarly, an edge exists between τ_{A1} and τ_{B1} because these test procedures overlap in time, preventing their simultaneous execution. An edge also exists between τ_{B2} and τ_{C2} because, although they do not overlap in time, the configuration time required for τ_{B2} does not allow sufficient time for the antenna to be

repositioned after τ_{C2}. Using such a graph representation allows us to assess the feasibility of a procedure schedule efficiently. To know if a schedule is feasible, we simply use the relation

$$\mathit{feasible}(\mathcal{S}) = \begin{cases} 1 & \nexists \tau_i, \tau_j \in \mathcal{S}, (\tau_i, \tau_j) \in \Xi(\mathcal{T}) \\ 0 & \text{Otherwise} \end{cases}$$

For example, in the graph depicted in Figure 5.3, a feasible procedure schedule would be $\mathcal{S} = \{\tau_{A1}, \tau_{B0}, \tau_{C2}\}$ as none of those test procedures possesses an edge connecting it to another test procedure in the graph.

5.3.3 Schedule Optimization

In this section, we present our approach for optimizing the scheduling for an IOT campaign. Let t_{sc} and t_{se} be the start time and end time of the IOT campaign, r be the site on which the campaign is performed, $S_{sat} = \{s_1, s_2, \ldots, s_n\}$ be the set of the satellites to perform the IOT campaign on, $\Gamma = \{\Gamma_r^s(t_{sc}, t_{se}) \mid s \in S_{sat}\}$ be the set of all satellite passes that will occur over r for each satellite of S_{sat} during the IOT campaign, and $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ be the set of all possible test procedures that can be scheduled for the satellite passes in Γ.

We aim at finding a complete procedure schedule $\mathcal{S}$, i.e., a set of test procedures, that covers all the satellites and IOT procedures types to be performed, such that they are the (near-)optimal given the objectives described in the introduction of this section: the procedure schedules should (1) maximize the antenna resource usage, (2) reduce the number of context switching required from practitioners, and (3) minimize the monetary and organizational cost of executing such schedules. Once the engineers obtain a set of equally viable and near-optimal procedure schedules according to the objective described above,they can select a single schedule for the IOT campaign. This selection is made at the engineer's discretion, considering internal constraints, such as the availability of an IOT operator. We cast our solution for finding such procedure schedules into a *multi-objective search optimization problem* [74]. Following common practices for expressing multi-objective search problems, we define the representation of a solution, the fitness functions used for evaluation, and the computational search algorithm employed to design the solution.

Representation. Given a set $\mathcal{T}$ of possible test procedures, a solution of the optimization algorithm represents a subset $\mathcal{S} = \{\tau_1, \tau_2, \ldots, \tau_n\}$, where $\tau_i \in \mathcal{S}$ and $\tau_i \in \mathcal{T}$. To ensure that each procedure is unique and all combinations of pairs of type and satellite $\{\text{Type}, s\}$ specified by the problem are covered by a distinct test procedure in $\mathcal{S}$, the following conditions must be met for all distinct $\tau_i, \tau_j \in \mathcal{S}$: (1) the type, start time, and end time of τ_i are different from those of τ_j, and (2) for every combination $\{\text{Type}, s\}$ specified by the problem, there exists a unique $\tau_k \in \mathcal{S}$ such that $\text{Type}_k = \text{Type}$ and $\alpha_r^k = s$.

Fitness functions. Our method aims at searching for candidate procedure schedules with regards to three objectives: ($\mathcal{O}_1$) maximizing the resource usage efficiency, ($\mathcal{O}_2$) minimizing the context switching required from practitioners, and ($\mathcal{O}_3$) minimizing the monetary and organizational cost of executing such a schedule. To quantify how a candidate solution fits these three objectives, we define the following fitness functions:

Resource usage efficiency ($\mathcal{O}_1$). Recall from Section 5.2 that engineers aim at efficiently using IOT resources, particularly antennas. In most instances, engineers prioritize IOT schedules that are both short in duration and maximize the antenna usage over that time. Note that if test procedures in a schedule are conducted with minimal idle time, antenna usage during IOT time is maximized, and the schedule requires the minimum possible time. Thus, maximizing the usage of the antenna over that duration results in an efficient procedure schedule, minimizing the time required to complete an IOT campaign. We define an antenna efficiency metric. This metric estimates how much the antenna is used over the complete duration of a procedure schedule, and should be maximized.
Let $\mathcal{S}$ be a candidate schedule, δ_{rc} the reconfiguration time required between two consecutive test procedures, t_{start} and t_{end} be the start time and end time of the test procedure τ, and $\Delta t(t_{start}, t_{end})$ be the duration of a test procedure τ, Based on these definitions, we define the fitness function for objective $\mathcal{O}_1$, denoted $fituse(\mathcal{S})$, as follows:

$$\mathit{fituse}(\mathcal{S}) = \frac{1}{\textsc{span}(\mathcal{S})} \left((|\mathcal{S}| - 1)\, \delta_c + \sum_{\tau \in \mathcal{S}} \Delta t(\tau) \right)$$

$fituse(\mathcal{S})$ is calculated as the inverse of the total schedule span, $\textsc{span}(\mathcal{S})$, multiplied by the sum of reconfiguration times between consecutive test procedures and the total duration of all test procedures in the schedule. The maximum value of $fituse(\mathcal{S})$ $(= 1)$ is achieved when all test procedures $\tau \in \mathcal{S}$ are scheduled consecutively without any idle time. Inversely, the minimum value occurs if the idle time between test procedures becomes infinitely large.

For example, let us consider a procedure schedule $\mathcal{S} = \{\tau_1, \tau_2, \tau_3\}$. If $\textsc{Span}(\mathcal{S}) = 15h$, with each reconfiguration time $\delta_{rc} = 1h$, and the durations of the test procedures being $\Delta t(\tau_1) = 3h$, $\Delta t(\tau_1) = 4h$, and $\Delta t(\tau_1) = 2h$ respectively, then:

$$\mathit{fituse}(\mathcal{S}) = \frac{1}{15}\left((3-1) \times 1 + (3+4+2)\right) = \frac{1}{15} \times 12 = 0.8$$

Minimizing context switching ($\mathcal{O}_2$). Another important output of our search-based approach, as explained in Section 5.3.1, is the creation of a slot schedule to indicate when resources should be allocated to performing or monitoring an IOT's test procedure. However, as explained in Section 5.2, creating too many slots during an IOT campaign incurs various impacts due to overhead. On the contrary, adopting longer, consolidated periods in a slot schedule may enhance resource utilization by minimizing setup and teardown activities, reducing idle time, and maximizing equipment, facility, and operator availability. Hence, our second fitness function $\mathcal{O}_2$, denoted as $\mathit{fitfrag}(\mathcal{S})$, ensures that the IOT schedule possesses a slot schedule that involves as few context switching as possible.

We note that, in our approach, a slot schedule $\mathcal{Q}$ is determined based on a procedure schedule $\mathcal{S}$. However, creating a slot schedule depends on the operational context of each company conducting IOT. Hence, we present the slot scheduling algorithm specific to SES Techcom in Section 5.4.

To formally define *fitfrag*($\mathcal{S}$), given a candidate procedure schedule $\mathcal{S}$, we use the expression:

$$\mathit{fitfrag}(\mathcal{S}) = 1 - \frac{|\mathcal{Q}| - 1}{|\mathcal{S}| - 1}$$

where $|\mathcal{Q}|$ is the number of separate time slots in the slot schedule for $\mathcal{S}$. *fitfrag* reaches its maximum value (= 1) when there is no fragmentation in the schedule, i.e., all test procedures are scheduled consecutively under the same slot. Similarly, the function reaches its minimum value (= 0) when a unique slot is assigned to each test procedure individually.

For example, let us consider a procedure schedule $\mathcal{S}$ with four test procedures τ_1, τ_2, τ_3, and τ_4. If there are two slots in the slot schedule, covering τ_1, τ_2 and τ_3 then $|\mathcal{Q}| = 2$ and $|\mathcal{S}| = 4$. The fitness function would be calculated as follows:

$$\mathit{fitfrag}(\mathcal{S}) = 1 - \frac{2 - 1}{4 - 1} = 1 - \frac{1}{3} = 0.667$$

Minimizing the monetary and organizational cost ($\mathcal{O}_3$). Recall from Section 5.2 that, in the context of IOT, tests require the use of expensive and limited resources, that possess both monetary and organizational constraints. Thus, it is necessary when scheduling an IOT campaign to ensure that the generated procedure schedule encompasses both monetary and organizational implications. The third fitness function, denoted as *fitcost*($\mathcal{S}$), provides a means to evaluate such cost associated with a procedure schedule. When considering the allocation of resources, particularly antennas for an IOT campaign, there may exist critical thresholds below which the cost-effectiveness of dedicating the antenna exclusively to the campaign outweighs the benefits of allocating it for other concurrent test procedures. Such thresholds may be determined by various factors such as operational efficiency, resource availability, and opportunity costs. For instance, above a certain number of allotments per day, it may be more cost-effective to prioritize the IOT campaign, allocate resources for a slot that spans the entire day, and postpone the execution of other usage of the antenna to a later date, thereby minimizing overall costs. Similarly, there may exist thresholds or periods during which scheduling an IOT procedure may not be desirable. For instance, scheduling an IOT procedure outside of working hours may be inconvenient for practitioners and induce extra costs for them.

To formally define *fitcost*($\mathcal{S}$), given a candidate procedure schedule $\mathcal{S}$, we use the expression:

$$\mathit{fitcost}(\mathcal{S}) = \frac{\mathit{cost}(\mathcal{S}) - \mathit{cost}_{\min}}{\mathit{cost}_{\max} - \mathit{cost}_{\min}}$$

where $\mathit{cost}(\mathcal{S})$ is the cost of the candidate schedule $\mathcal{S}$, and $\mathit{cost}_{\min}$ (resp. $\mathit{cost}_{\max}$) is the minimal (resp. maximum) theoretical cost achievable. It is important to note that $\mathit{cost}(\mathcal{S})$ is a cost function defined internally by the IOT operators and is dependent on the specific operational

context. Additionally, $cost_{\min}$ and $cost_{\max}$ are inferred by IoT operators based on their domain knowledge of what would constitute the best-case and worst-case schedules in theory, for the current context. It should be noted that these theoretical schedules are not always guaranteed to be feasible.

Constraints. Considering constraints during the search process helps narrow down the search space, making the search process more efficient. The complexity of scheduling test procedures leads to a subset of solutions of $\mathcal{P}(\mathcal{T})$, that are infeasible, which makes part of the search space not efficient to explore. The definition of infeasible schedule solutions aligns with the definition described in Section 5.3.2, meaning that schedules are considered infeasible if they contain at least one conflict among the test procedures they include. By eliminating infeasible solutions, the search algorithm can focus on viable solutions, reducing computational time and resources.

Various techniques have been proposed in the literature to handle constraints and infeasible solutions, such as death penalties [150], static penalties [151, 152], repair algorithms [153, 152], or constraints as objectives [154]. In our algorithm, we use a Niched-Penalty approach proposed by Deb and Agrawal [155]. First, we define an inequality constraint that the algorithm uses to measure the degree of infeasibility of a solution:

$$g(\mathcal{S},\mathcal{T}) = \sum_{\substack{(\tau_i,\tau_j)\in\mathcal{S}^2 \\ \tau_i\neq\tau_j}} \xi_{\tau_1,\tau_2}$$

Subsequently, a penalty is applied to the fitness of $\mathcal{S}$ if $g(\mathcal{S},\mathcal{T}) > 0$, as follows:

$$F(\mathcal{S}) = \begin{cases} f(\mathcal{S}) & \text{if } g(\mathcal{S},\mathcal{T}) \leq 0 \\ f_{\max} + g(\mathcal{S},\mathcal{T}) & \text{otherwise} \end{cases}$$

where $F(\mathcal{S})$ represents the fitness vector of the candidate schedule $\mathcal{S}$, with each dimension corresponding to a fitness function. The term $f(\mathcal{S})$ denotes the objective function values for $\mathcal{S}$, and $f_{\max}$ is the maximum fitness value among all feasible solutions in the population.

Computational search. We use the Non-dominated Sorting Genetic Algorithm version 3 (NSGA-III) [16] to find (near-)optimal schedules of IOT test procedures, as shown in algorithm 12. The NSGA-III algorithm has been successfully applied to several software engineering problems [156, 157] involving optimization. The algorithm first generates a set candidate procedure schedules $\mathbf{P}$ (lines 2-6) and then evolves the population iteratively until finding the best non-dominated schedules (Pareto front) or exhausting the time budget (lines 8-33). In each iteration, the algorithm first assesses the fitness of the individuals $I \in \mathbf{P}$ using the fitness functions (lines 11-22), and applies a penalty if required. Calculating the fitness of the individual allows the algorithm to find which candidate schedule to keep in the archive and compute the Pareto front (lines 23-30). Subsequently, based on the archive and reference points, the algorithm breeds a new population $\mathbf{P}$ (line 32) using the following genetic operators: (1) *Selection* chooses

Algorithm 12 An algorithm for selecting the near-optimal procedure schedules, based on NSGA-III

Input:

$\mathcal{T}$: possible test procedures
n_p: size of the population and the archive
n_r: number of reference points
μ_c: crossover probability
μ_m: mutation probability

Output:

$\mathcal{NS}$: near-optimal procedure schedules

1: // generate the initial population
2: $\mathbf{P} \leftarrow \emptyset$
3: **repeat**
4: $I \leftarrow$ GenerateSchedule($\mathcal{T}$)
5: $\mathbf{P} \leftarrow \mathbf{P} \cup I$
6: **until** $|\mathbf{P}| = n_p$
7: // create an empty archive and reference points
8: $\mathbf{P}_\alpha \leftarrow \emptyset$
9: $\mathbf{R} \leftarrow$ GenerateReferencePoints(n_r)
10: **repeat**
11: // assess the fitness of each individual
12: **for each** $I \in \mathbf{P}$ **do**
13: **if** $g(I, \mathcal{T}) = 0$ **then**
14: $f_1(I) = \mathit{fituse}(I)$
15: $f_2(I) = \mathit{fitfrag}(I)$
16: $f_3(I) = \mathit{fitcost}(I)$
17: **else**
18: $f_1(I) = f_{1,\max} + g(I, \mathcal{T})$
19: $f_2(I) = f_{2,\max} + g(I, \mathcal{T})$
20: $f_3(I) = f_{3,\max} + g(I, \mathcal{T})$
21: **end if**
22: **end for**
23: $\mathbf{P}' \leftarrow$ AssociateWithReferencePoints($\mathbf{P}, \mathbf{R}$)
24: // update the archive
25: $\mathbf{P}_\alpha \leftarrow \mathbf{P}_\alpha \cup \mathbf{P}'$
26: ComputeFrontRanks($\mathbf{P}_\alpha$)
27: ComputeSparsities($\mathbf{P}_\alpha$)
28: $\mathbf{P}_\alpha \leftarrow$ SelectArchive($\mathbf{P}_\alpha, n_p$)
29: // update the Pareto front
30: $\mathcal{NS} \leftarrow$ ParetoFront($\mathbf{P}_\alpha$)
31: // create a new population
32: $\mathbf{P} \leftarrow$ Breed($\mathbf{P}_\alpha, n_p, \mu_c, \mu_m$)
33: **until** $\mathcal{NS}$ is the ideal Pareto front **or** the algorithm run out of time
34: **return** $\mathcal{NS}$

the candidates to be selected for reproduction by leveraging a *binary tournament* selection technique [74]; (2) *Crossover* generates offspring from two candidate schedules, using a one-point crossover technique [74]; (3) *Mutation* introduces diversity in the offspring by modifying some of the test procedures of the offspring according to a mutation rate, and a specific strategy. Below, we further describe our crossover and mutation operators.

Crossover. Our method employs a standard *one-point crossover* operator [74]. Specifically, given two parent candidate schedules $\mathcal{S}^l$ and $\mathcal{S}^r$, each containing IOT test procedures $\{\tau_1^l, \tau_2^l \ldots, \tau_n^l\}$ and $\{\tau_1^r, \tau_2^r \ldots, \tau_n^r\}$ respectively, the crossover operator randomly selects a crossover point i. It then generates two offspring by swapping some test procedures between the parents based on i, resulting in $\{\tau_1^r, \ldots, \tau_i^r, \tau_{i+1}^l, \ldots, \tau_n^l\}$ and $\{\tau_1^l, \ldots, \tau_i^l, \tau_{i+1}^r, \ldots, \tau_n^r\}$.

We note that the resulting child schedules might become infeasible after such a crossover operation. However, these infeasible schedules are managed through the constraint handling technique detailed previously (i.e., such schedules are inflicted with a penalty).

Mutation. The mutation operator is applied to the candidate procedure schedules generated by the crossover operation with a probability p_{mut}. The test procedures to be modified in the selected candidates are chosen using a uniform-mutation operator [74]. Next, our method takes the following steps to replace a test procedure, denoted τ, selected for mutation by the uniform-mutation operator: Algorithm 13 takes as input a set of possible test procedures for the mutation $\mathcal{T}_e$, the candidate schedule $\mathcal{S}$, the maximum probability of selecting a non-conflicting test procedure $p_{nc}^{\max}$, and the minimum probability of selecting a non-conflicting test procedure $p_{nc}^{\min}$.

First, Algorithm 13 initializes the probabilities of selecting a conflicting or non-conflicting task by calculating the ratio r (line 2), which represents the proportion of non-conflicting test procedures in the candidate schedule $\mathcal{S}$, as $r = 1 - \frac{\Xi(\mathcal{S})}{|\mathcal{S}|}$, where $\Xi(\mathcal{S})$ is the number of conflicts in $\mathcal{S}$ and $|\mathcal{S}|$ is the total number of test procedures in $\mathcal{S}$. The probability of selecting a non-conflicting test procedure, $P_{\text{non-conflicting}}$, is computed using linear interpolation between $p_{nc}^{\min}$ and $p_{nc}^{\max}$ as $P_{\text{non-conflicting}} = p_{nc}^{\min} + r \times (p_{nc}^{\max} - p_{nc}^{\min})$ (line 3). The probability of selecting a conflicting test procedure, $P_{\text{conflicting}}$, is then calculated as the complement of $P_{\text{non-conflicting}}$, such that $P_{\text{conflicting}} = 1 - P_{\text{non-conflicting}}$ (line 4)

Next, the algorithm assigns probabilities to each test procedure $\tau \in \mathcal{T}_e$ based on whether it conflicts with the candidate schedule $\mathcal{S}$ (lines 6-12). If τ conflicts with $\mathcal{S}$, it is assigned $P(\tau) = P_{\text{conflicting}}$; otherwise, it is assigned $P(\tau) = P_{\text{non-conflicting}}$. The assigned probabilities are then normalized to ensure they sum to 1 (lines 14-17). The total probability P_{total} is computed as the sum of all assigned probabilities, and each probability $P(\tau)$ is normalized by dividing it by P_{total}: $P(\tau) = \frac{P(\tau)}{P_{\text{total}}}$.

Algorithm 13 Mutation algorithm

Input:

$\mathcal{T}_e$: eligible test procedures for the mutation
$\mathcal{S}$: candidate solution
$p_{nc}^{\max}$: maximum probability of selecting a non-conflicting test procedure
$p_{nc}^{\min}$: minimum probability of selecting a non-conflicting test procedure

Output:

$\mathcal{S}_{mut}$: mutated candidate schedule

1: // initialize the probabilities of selecting a conflicting or non-conflicting test procedure
2: $r = 1 - \frac{\Xi(\mathcal{S})}{|\mathcal{S}|}$
3: $P_{\text{non-conflicting}} \leftarrow p_{nc}^{\min} + r \times (p_{nc}^{\max} - p_{nc}^{\min})$
4: $P_{\text{conflicting}} \leftarrow 1 - P_{\text{non-conflicting}}$
5: // assign probabilities to each $\tau \in \mathcal{T}_e$
6: **for** each $\tau \in \mathcal{T}_e$ **do**
7: **if** τ conflicts with $\mathcal{S}$ **then**
8: $P(\tau) \leftarrow P_{\text{conflicting}}$
9: **else**
10: $P(\tau) \leftarrow P_{\text{non-conflicting}}$
11: **end if**
12: **end for**
13: // normalize the probabilities
14: $P_{\text{total}} \leftarrow \sum_{\tau \in \mathcal{T}_e} P(\tau)$
15: **for** each $\tau \in \mathcal{T}_e$ **do**
16: $P(\tau) \leftarrow \frac{P(\tau)}{P_{\text{total}}}$
17: **end for**
18: // select a test procedure based on the probabilities
19: $\tau_{\text{selected}} \leftarrow$ select a test procedure from $\mathcal{T}_e$ using the probabilities $P(\tau)$
20: // mutate the candidate schedule
21: $\mathcal{S}_{mut} \leftarrow$ mutate $\mathcal{S}$ using τ_{selected}
22: **return** $\mathcal{S}_{mut}$

Algorithm 13 then selects a test procedure τ_{selected} from $\mathcal{T}_e$ using the probability $P(\tau)$ (line 19). Finally, the candidate schedule $\mathcal{S}$ is mutated using the selected test procedure τ_{selected} to obtain the mutated candidate schedule $\mathcal{S}_{mut}$ (line 21), which is then returned as the output (line 22).

5.4 Empirical Evaluation

This section empirically evaluates our approach through a real case study from SES Techcom. Our case study implementation, data, and results are available online [158].

5.4.1 Research Questions (RQs)

RQ1 (sanity check). *How does our search-based IOT scheduling approach perform compared to random search?* This research question is a crucial evaluation point to assess the effectiveness of any search-based approach [159, 160]. A search-based approach is expected to outperform a simple random-search significantly. If it does not, it would imply that the search process is unnecessary.

RQ2 (comparison to state of the art). *How does our approach compare to other search-based techniques?* We compare our approach to an Ant Colony Optimization (ACO) approach, which is a well-established technique for solving complex scheduling problems. By doing so, we aim to demonstrate the effectiveness and efficiency of our method, highlighting its potential advantages over existing state-of-the-art techniques. Thus RQ2 evaluates the quality of the generated schedules against those obtained from an ACO approach, focusing not only on the overall quality but also on the timing and computing resource usage.

RQ3 (usefulness). *How do the schedules generated by our approach compare with the ones generated by engineers?* To validate the usefulness of our approach, it is crucial to demonstrate that the schedules generated by our method offer a significant improvement over those manually constructed by experienced engineers. This research question is critical, as it allows us to provide empirical evidence on the advantages of employing an automated approach compared to traditional manual methods, thereby justifying the need for our approach. Thus, RQ3 evaluates the quality of the generated schedules over the schedules manually constructed by IOT engineers at SES Techcom.

5.4.2 Industrial Study Subject

We evaluate our approach on a representative case from SES Techcom, specifically, on an In-Orbit Testing (IOT) campaign for the European Global Navigation Satellite System (GNSS) Galileo. Such an IOT campaign represents, as discussed in Section 5.2, a type of operational acceptance testing.

Our evaluation relies on a practical configuration employed in an IOT campaign for the Galileo constellation, composed of twenty-three operational satellites (as of April 2024) and four spare satellites, typically not tested in such campaigns. The Galileo constellation orbits the Earth across three distinct Medium Earth Orbit (MEO) planes. These twenty-four Galileo satellites represent our System Under Test (SUT), where each satellite is denoted throughout the evaluation as $s_1, s_2, \ldots, s_{24}$.

During the IOT campaign, SES Techcom conducts SQM tests for each active satellite, complemented by RIOT tests on six of the twenty-four satellites. SQM test procedures are scheduled to

Algorithm 14 An algorithm for scheduling slots.

Input:
$\mathcal{S}$: procedure schedule to generate a slot schedule
Output:
$\mathcal{Q}$: slot schedule

```
1: // generate an initial set of slots
2: Q ← ∅
3: for each s ∈ S do
4:     Q ← Q ∪ {GenerateSlot(s)}
5: end for
6: // sanitize the slot schedule
7: CombineOverlappingSlots(Q)
8: ConsolidateSlots(Q)
9: return Q
```

coincide with the maximum elevation pass of each satellite. The tests can either start at, end at, or be centred around the maximum elevation point of the satellite, providing three distinct time slots for a single satellite pass. In contrast, the RIOT test procedures span the entirety of a satellite pass, starting and ending within a five-degree range of start and end elevation. Additionally, SES Techcom employs a single antenna for the execution of these test procedures throughout a standard IOT campaign.

In this IOT campaign, the duration is a maximum of two weeks, between, for example, the 1st of October 2024, and the 14th of October 2024 (denoted t_1 and t_2, respectively). To initialize our scheduling approach, we first generate a set of satellite passes at the location of the antenna between the 1st of October 2024 and the 14th of October 2024, $\Gamma_r(t_1, t_2) = \bigcup_{i=1}^{24} \Gamma_r^{s_i}(t_1, t_2)$. A set of test procedures $\mathcal{T}$ is then defined from $\Gamma_r(t1, t2)$, so that for each $\alpha \in \Gamma_r(t_1, t_2)$ an SQM test procedure is associated to it, and that for each $\alpha \in \Gamma_r((t_1, t_2))$ if α belongs to s_1, s_2, ..., s_6, and that $\theta_{start}, \theta_{end} \in \alpha, \theta_{start} \leq 5°, \theta_{end} \leq 5°$. We then generate the conflict graph of the test procedures, as per its definition (see Section 5.3.2). To run our search-based scheduling approach, as mentioned in Section 5.3.3, a slot scheduling algorithm is required. In our experiments, we employ Algorithm 14 to create a slot schedule for a given procedure schedule. From lines 1 to 5, an initial set of slots is generated by the algorithm for each test procedure present in the procedure schedule. Particularly, the time slots in this study start at the beginning of an hour, a quarter past the hour, half-past the hour, or three-quarters past the hour. Additionally, the duration of the time slot must be an integer multiple of one hour. From lines 6 to 7, several sanitization steps are conducted. First, slots that overlap with one another are merged so they form a single slot. Then, slots are consolidated according to the following internal policy at SES Techcom: if more than six hours in twenty-four hours are dedicated to performing IOT procedures, this entire period should dedicated solely to performing IOT procedures, i.e., all those slots are replaced by a single slot that spans a length of twenty-four hours. Finally, the resulting slot schedule is returned.

Moreover, we define the *cost* of a slot schedule $\mathcal{Q}$ as follows:

$$cost(\mathcal{Q}) = \sum_{b \in \mathcal{Q}} cost(b)$$

where q is a slot in the slot schedule $\mathcal{Q}$ and $cost(b)$ is defined as

$$cost(b) = \begin{cases} \frac{\text{SPAN}(b)}{60} \times 456 & \text{if SPAN}(b) < 24h \\ 3561 & \text{otherwise} \end{cases}$$

The cost function $cost(b)$ is determined by the span of the slot b. If the span is less than 24 hours, the cost is calculated as $\frac{\text{SPAN}(b)}{60} \times 456$, where the span of b in minutes is converted to hours ($\frac{\text{SPAN}(b)}{60}$) and then multiplied by 456. If the span is 24 hours or more, the cost is fixed at 3561.

5.4.3 Experimental Setup

For the evaluation of the research questions, we implemented the following three approaches: (1) GSC: Our approach using NSGA-III, described in Section 5.3, (2) ACO: An approach using an ant colony algorithm [161], without an explicit handling of conflicts, and (3) RS: A Random Search approach.

EXP1. To answer RQ1, we conduct a comparative analysis of our approach GSC against a random search approach RS. We implemented GSC, an IOT scheduling tool based on our approach, described in Section 5.3. Moreover, we implemented a random search approach as a baseline, RS, which, similarly to our approach, creates IOT schedules for a given set of satellite passes. The RS baseline creates IOT schedules by first randomly creating procedure schedules. The procedure schedules are created by randomly selecting n test procedures $\tau \in \mathcal{T}$ (see the definition of $\mathcal{T}$ in Section 5.3). To perform our comparison, we evaluate the results of both approaches by comparing the resulting fitness values of the solutions.

To further measure the effectiveness of our multi-objective search-based algorithm, we use the three quality indicators as described below, following established guidelines found in existing literature [162]. As the optimal solution for our study problem is unknown apriori, we construct a reference Pareto front by combining all non-dominated solutions obtained from each execution of the compared approaches. We then assess the *Generational Distance* (GD), a metric which measures the Euclidean distance between a specific solution and the nearest solution on a reference Pareto front [163]. The lower the GD metric, the closer a solution is to the optimal output. We then assess the *Spread* (SP), a measure of the distance between each point of the Pareto front [164]. The lower the SP metric, the more the non-dominated solutions are spread across the Pareto front, showing higher diversity. Finally, we assess the *Hypervolume* (HV), a quality indicator which represents the size of the space covered by a search algorithm [165]. The higher the HV metric, the more space the Pareto front covers, showing better performance.

EXP2. To answer RQ2, in this experiment, similarly to EXP1, we compare the implementation of our approach, GSC, against an approach implementation relying on an ant colony optimization algorithm, ACO. The ant colony algorithm is an optimization algorithm which takes inspiration from the foraging behaviour of ant colonies. It has been used in several combinatorial optimization problems, such as knapsack problems [166] and routing problems [167, 168]. Notably, the ant colony algorithm has been extensively used in the literature to solve the multi-satellite control resource scheduling problem (MSCRSP) [143, 144, 145, 146, 147], a combinatorial problem close to our IOT scheduling problem. Generally, those methods are based on the Max–min ant system (MMAS) proposed by Stützle and Hoos [169], we thus elect it to create a baseline approach to compare ACO performance with ours. We describe below our baseline implementation of MMAS.

Construction of the optimization problem: In MMAS [169], the optimization problem is assimilated to a graph $G = (A, E, T, H)$, where A is the set of test procedures, E is the set of edges that represent conflicts between each test procedure, and T and H are the vectors which represent the pheromone trail and heuristic information, respectively, both associated with the edges in E. The graph G is referred as the construction graph.

The solutions to the optimization problem are represented as paths on graph G. The ants create candidate solutions by taking randomized walks on the fully constructed graph G, guided by the pheromone trail intensity and current heuristic information on the edges. The conflicts definition Ξ is applied to ensure the ants do not form infeasible solutions while moving between vertices. After the ants complete their walk, the pheromone trails are updated.

Initializing the pheromone values: Firstly, the approach initializes all the pheromone values to the maximum value.

Constructing a solution: m ants are placed on randomly selected test procedures. Each ant uses a probabilistic decision-making rule, known as the random proportional rule, to determine which test procedure to select next at each step of the process. If ant k is at node i during iteration t, it will select the next node j based on a certain probability.

$$p_{ij}^{k}(t) = \begin{cases} \frac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{l \in N_i^k} [\tau_{il}]^{\alpha} \cdot [\eta_{il}]^{\beta}}, & \text{if } j \in N_i^k \\ 0, & \text{otherwise} \end{cases}$$

where η_{ij} is a heuristic value, τ_{ij} is the pheromone trail value, α and β are two parameters which determine the relative influence of the pheromone trail and the heuristic information, and N_i^k is the feasible procedures of the k-th ant when beginning at node i.

Updating the pheromone: After all the ants have completed an iteration, the pheromone trails are updated as follows:

$$\tau_{ij}(t+1) = \Big[(1-\rho)\tau_{ij}(t) + \Delta\tau_{ij}^{\text{best}}(t)\Big]_{\tau_{\min}}^{\tau_{\max}}$$

where ρ is the evaporation rate, comprised between 0 and 1, $\Delta\tau_{ij}^{\text{best}}$ is the quantity of pheromone laid by the k-th ant on the path visited. $\Delta\tau_{ij}^{\text{best}}$ is defined as follow:

$$\Delta\tau_{ij}^{\text{best}}(t) = \begin{cases} F(s^{\text{best}}(t)), & \text{if } (i,j) \in s^{\text{best}}(t) \\ 0, & otherwise \end{cases}$$

where s^{best} is the best-so-far solution, which is the best solution found during the current iteration or the global-best solution found by the k-th ant. F represent the fitness function that is used to assess the quality of the solution found by the ant. In our ACO baseline, we define it as

$$F(\mathcal{S}) = \frac{\mathit{fituse}(\mathcal{S}) + \mathit{fitfrag}(\mathcal{S}) + \mathit{fitcost}(\mathcal{S})}{3}$$

EXP3. To answer RQ3, we evaluate the usefulness of our approach, GSC, over IOT schedules generated by SES Techcom's IOT engineers. We obtained the test configurations and the IOT schedules created by the engineers for the most recent IOT campaign conducted by SES Techcom on November 7, 2023. Additionally, we acquired the original satellite passes used by the engineers to develop their schedules.

For the comparison, we executed our tool, GSC, 50 times using the same configurations and satellite passes as the engineers. In EXP3, we compared the cost, span, and number of slots required to execute the schedules generated by GSC and those created manually by the engineers. Furthermore, we assessed the average execution time for GSC to generate a set of schedules and compared it to the time reported by the engineers for creating their schedules.

5.4.4 Parameters Setting

We configured the hyperparameters for the search problem as follows: the population size is set to 200, the mutation rate to 0.2, the crossover rate to 0.8. Additionally, reference directions for half the population size are generated according to the *Riesz s-energy* principle [170]. These parameter values follow the guidelines present in the literature [171]. The termination condition of the approach is set according to the number of fitness function evaluations performed. To deduce the most efficient number, we conducted 25 initial experiments, each terminating at 150,000 fitness functions' evaluations, and monitored the evolution of the SP and HV metrics. We observed that, on average, after 50,000 fitness evaluations, the metrics did not show significant improvement. Therefore, we set the stopping conditions of our approach to 50,000 fitness function evaluations. Additionally, we constrain the approaches to a maximum time limit of one hour to ensure that no single run exceeds a reasonable duration.

Nonetheless, we note that those hyperparameters could be further tuned to increase the performance of our approach, however, the results obtained with the described values are sufficient and convincingly support our analysis. Therefore we do not report further optimizations of those parameters.

Table 5.1: Comparing GSC and RS Pareto front using the Hypervolume (HV), Spread (SP), and Generational Distance (GD) quality indicators.

Metric	p-value	$\hat{A}_{12}$	Mean GSC	Mean RS
SP	5.65e-3	1.0	**1.50**	0.0
HV	5.65e-3	1.0	**0.131**	0.0
GD	1.73e-17	1.0	**8.49e-3**	3.42e-1

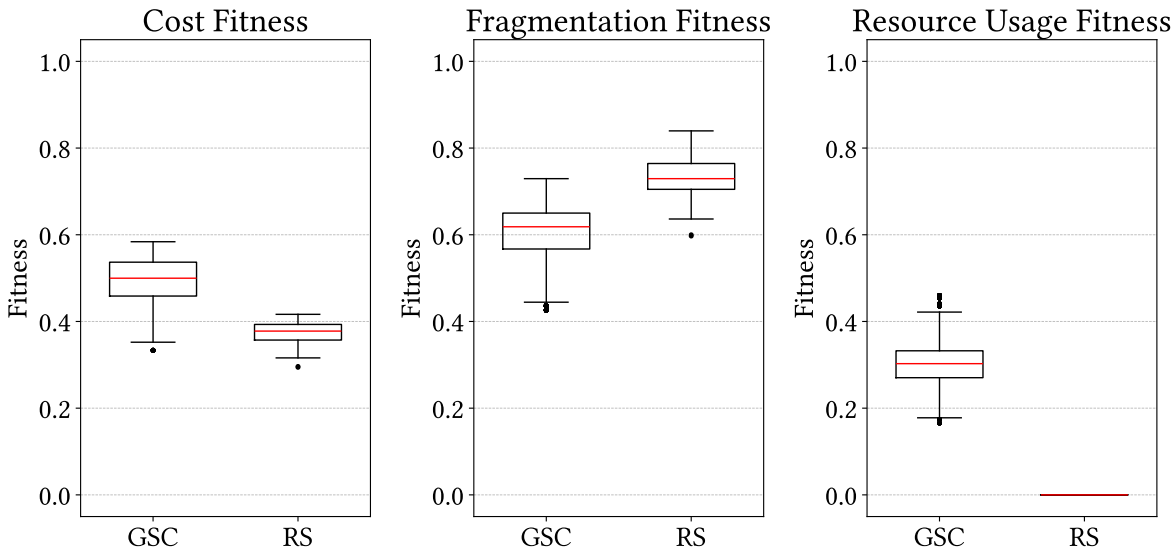


Figure 5.4: Comparing the GSC, GSR, and RS in terms of *fituse*, *fitfrag*, and *fitcost*

5.4.5 Experiment Results

RQ1. Table 5.1 compares the sets of schedules obtained by each method after 50 runs of each approach, in terms of their quality indicators. The table indicates that the solutions found by GSC (i.e., its Pareto front) are of significantly better quality than the solutions found from RS. Indeed, for each comparison, the p-values are lower than 0.05, and the $\hat{A}_{12}$ values indicate large effect size ($\hat{A}_{12} > 0.5$), which supports the hypothesis that the first distribution is significantly greater than the second distribution.

Figure 5.4 compares the distributions of the three fitness functions' values (see Section 5.3.3), for each set of solutions obtained after 50 runs of each approach. The results show that GSC (resp. RS) reaches a fitness of 49.5% (resp. 37.1%) for *fitcost*, 60.4% (resp. 73.1%) for *fitfrag*, and 30% (resp. 0%) for *fituse*. Those results indicate that in terms of fitness, GSC finds solutions that are more cost-effective (better *fitcost*) compared to RS, and finds schedules that improve on the test resource usage fitness function (better *fituse*). In terms of *fituse*, we note that the antenna resource usage cannot be calculated when schedules are not feasible which explains why the average fitness value *fituse* is 0 for RS. This also explains why the second fitness function (*fitfrag*) is higher for RS than for GSC, as when schedules are infeasible, there exists some overlap between

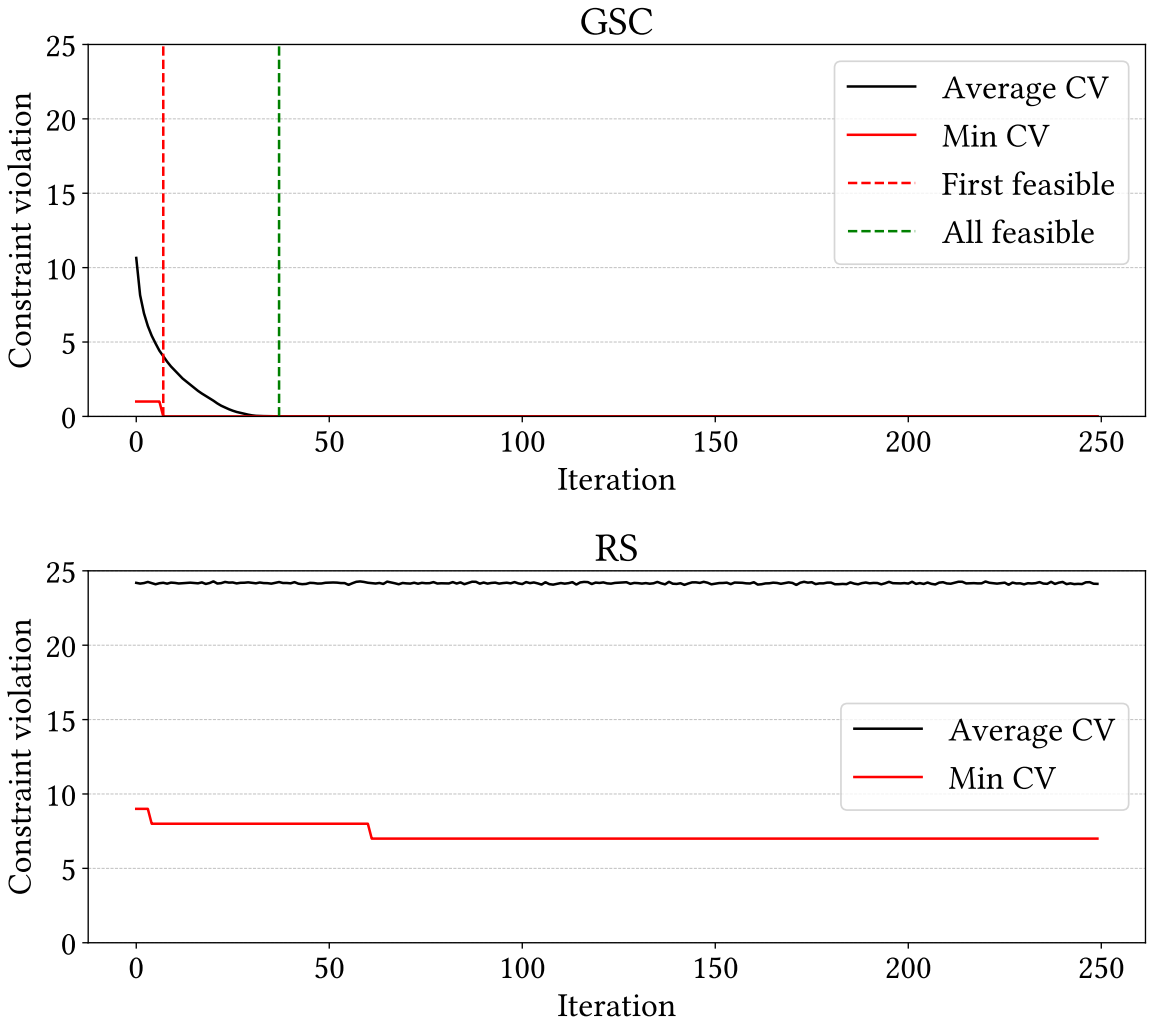


Figure 5.5: Comparing the progression of the constraint violation for GSC and RS.

the different test procedures, therefore this may tend to a better fragmentation as infeasible test procedure may be very close to each other or overlapping, thus presenting less gap between them.

To further analyse how GSC and RS handle infeasible schedules, Figure 5.5 compares the progression of average and minimum constraint violations over 50 runs of GSC and RS. The results show that on average, GSC can find a first feasible schedule candidate after seven iterations and, on average, all schedules become feasible after seven iterations. On the contrary, RS obtains schedules with, at best seven conflicts and the schedules display on average 24 conflicts at each iteration.

The answer to **RQ1** *is that* our approach, GSC, significantly outperforms RS in generating schedules for IOT campaigns. In particular, RS cannot generate schedules that do not violate the constraints imposed by the scheduling problem.

RQ2. Figure 5.6 compares the non-dominated solutions obtained after 50 runs of ACO described in EXP2, and our approach GSC. The figure shows that for the fitness functions *fitcost*, *fitfrag*,

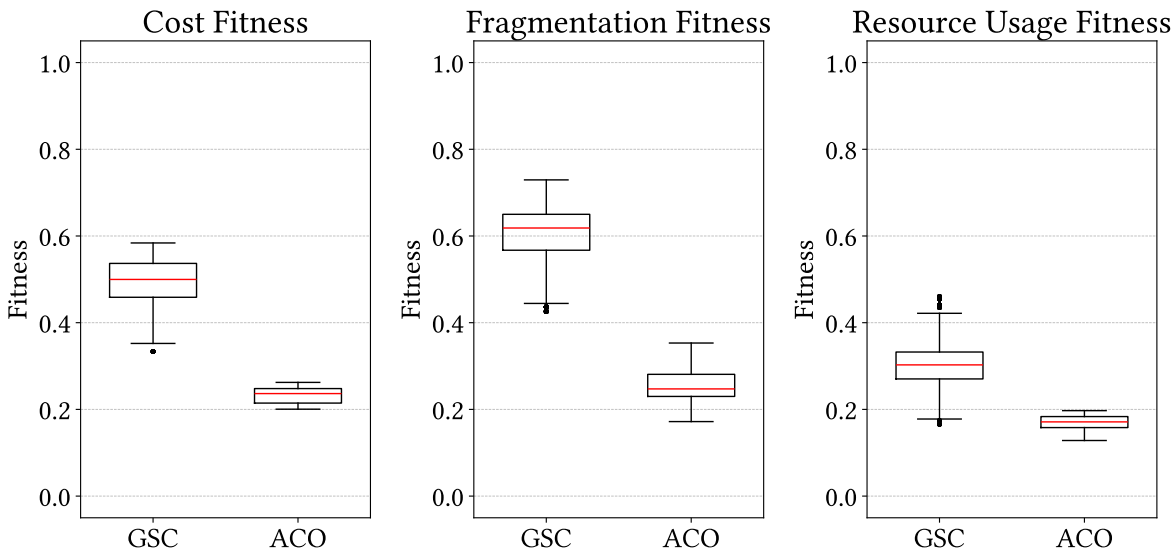


Figure 5.6: Comparing GSC and ACO in terms of (a) *fituse*, (a) *fitfrag*, and (a) *fitcost*

and *fituse*, GSC obtains, on average a score of 0.49, 0.60, and 0.30, respectively. In contrast, for the same fitness functions, ACO obtains, on average, a score of 0.23, 0.25, and 0.17, respectively.

Table 5.2: Comparison of the average number of iterations performed and feasible schedules obtained after 50 runs of the GSC and ACO.

Method	**Iterations (Average)**	**# Feasible Schedules (Average)**
GSC	250	38
ACO	13	1

Additionally, Table 5.2 compares the number of iteration performed, and number of solutions (i.e., feasible schedules) obtained on average for 50 runs of GSC and ACO. Each approach run was constrained to one hour to ensure a fair comparison. The table indicates that GSC, with a population of 200 candidate schedules, performs an average of 250 iterations, equating to 50,000 fitness evaluations. In contrast, ACO, with a population of 20 ants, performs an average of 13 iterations, resulting in 260 fitness evaluations. Furthermore, the figure shows that GSC yields an average of 38 equally viable schedules, whereas ACO produces a single feasible schedule.

Those results indicate that GSC's ability to perform more fitness evaluations allows it to explore the search space more thoroughly within a limited time frame. Additionally, by producing a greater number of equally viable solutions, GSC enables practitioners to conduct trade-off analyses.

The answer to **RQ2** *is that* our approach, being more time-efficient, allows for a more thorough exploration of the search space. This results in our approach outperforming ACO in terms of the fitness of the solutions found. Additionally, unlike ACO, our approach produces several equally viable schedules, enabling practitioners to perform trade-off analyses.

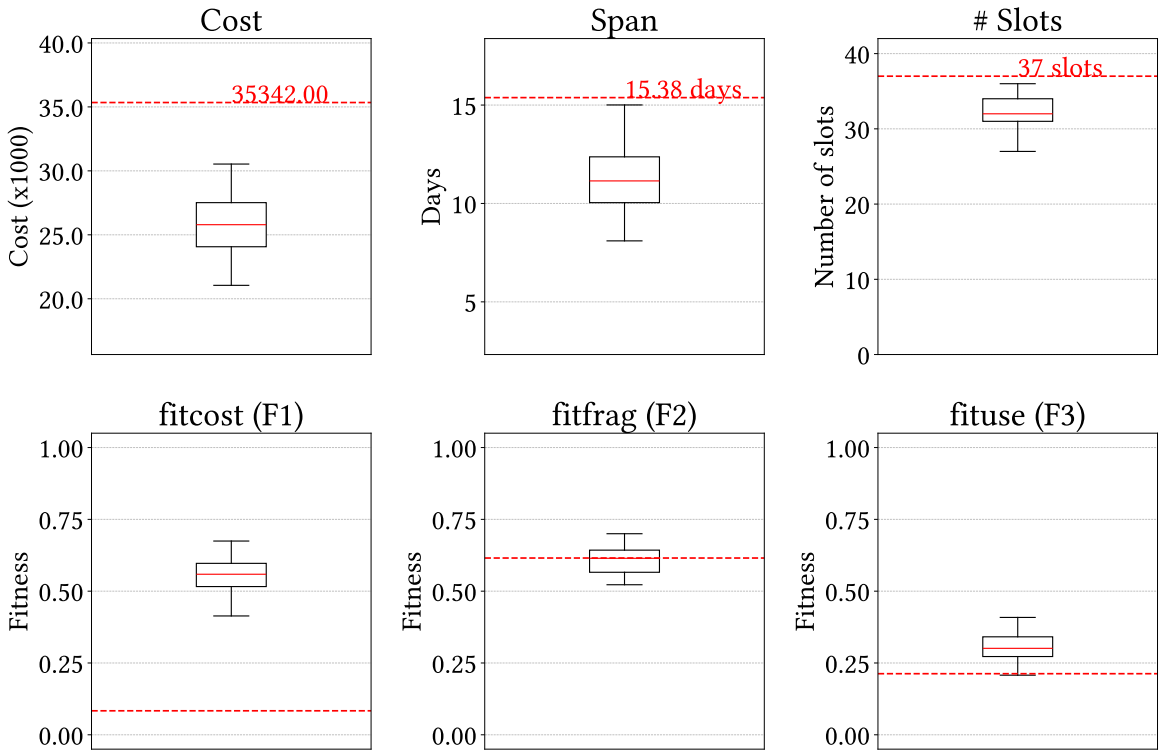


Figure 5.7: Comparing the span, cost, number of slots, *fitcost*, *fitfrag* and *fituse* of the our GSC against manually crafted schedules.

RQ3. We compare the schedules generated by our approach, GSC with the reference schedules provided by SES Techom (see Section 5.4.3). Figure 5.7 presents a comparative analysis of the distribution of schedule spans, costs, and the number of slots between the generated schedules and the reference schedule. Furthermore, Figure 5.7 illustrates the distribution of fitness values obtained from 10 runs of GSC, compared to the calculated fitness of the reference schedule. The figure indicates that the average cost for the generated schedules is 25,657, with an average span of 11 days and 32 slots. In comparison, the reference schedule has an average cost of 35,342, a span of 15.38 days, and 37 slots. This represents an average reduction of 27% in cost, 28% in span, and 13% in the number of slots compared to the manually crafted schedule.

Furthermore, the figure shows that, for *fitcost* function, generated schedules have an average fitness of 53%, compared to an average of 8.3% for the reference schedule. This represents an increase of approximately 538% in the average fitness, indicating that the generated schedules are significantly more cost-effective. Regarding *fitfrag*, the average fitness is approximately 60%, compared to 60.9% for the reference schedule. This indicates that, on average, our method is equally effective in minimizing the amount of context switching required from engineers. Finally, for *fituse*, the average fitness is approximately 29%, compared to 20.8% for the reference schedule. This represents a 39.42% increase in the efficiency of antenna resource usage for the generated schedules compared to the reference schedule.

The answer to **RQ3** *is that* the schedules generated by our approach are significantly more cost-effective, with shorter schedule spans and fewer slots. They also maintain comparable or better performance in terms of fragmentation and resource usage efficiency compared to those generated by expert engineers.

5.4.6 Lessons Learned

To further assess the practical usefulness of our approach, we reached out to three IOT engineers at SES Techcom to obtain feedback on our work and discuss possible improvements. The three engineers are currently working on IOT systems. One is a senior manager with several years of experience, while the other two are junior engineers currently in charge of IOT projects. All three engineers have previously handcrafted test schedules for IOT campaigns.

Following the evaluation of our tool conducted for RQ1 and RQ3, we provided it to the engineers. We began by presenting a detailed demonstration, explaining our approach's usage and the impact of its various parameters. Subsequently, the engineers were given a period of four hours to familiarize themselves with our implementation and utilize it to generate IOT schedules they had previously crafted manually. After this familiarization period, we conducted in-depth interviews to gather their feedback, focusing on the implementation's usability, the quality of the generated schedules, and their perspectives on integrating the implementation into their workflow. Overall, the experts' interviews highlighted three key takeaways from our work.

Efficiency of schedule generation. All three engineers acknowledged that the generated schedules are more conveniently arranged than those they thoroughly handcrafted. This observation aligns with our findings in RQ1 and RQ3, where we concluded that the procedure schedules and slot schedules generated by our approach outperform their handcrafted counterparts across all metrics, including fitness. Additionally, the engineers highlighted that the ability to generate schedules in an automated manner, is a significant advantage, as it requires a fraction of the time required for creating such schedules manually, and the automation of the scheduling process reduces the likelihood of human errors.

Multiple schedules enable tradeoff analysis. Engineers highlighted that receiving several schedules in a short time frame could significantly impact their operations. Specifically, they stated that this capability allows for quick adaptation to changing conditions (e.g., bad weather forecast, unavailability of equipment or personnel) during testing and better alignment with the needs of specific test campaigns. For example, some IOT campaigns require minimum fragmentation, even if it comes at the expense of cost. Having multiple equally viable schedules enables trade-off analysis for these varying needs and thus is beneficial. Additionally, one of the junior engineers mentioned that schedules with a shorter span could minimize the risk of issues arising during test campaigns, due to the test procedures being concentrated in fewer days, as described in Section 5.2.

Room for improvement. The senior engineer and one junior engineer noted that while the generated schedules are highly efficient, there is potential for further improvement. They mentioned that the reconfiguration time overhead between two test procedures is currently fixed but is largely dependent on the time required to reposition the antenna. This overhead can be reduced by selecting test procedures where the satellite positions at the end of one procedure and the start of the next are relatively close to each other. Furthermore, they suggested that the schedules could exhibit greater *"diversity"*, by increasing the spread of the final set of generated schedules. This could enhance the variety of schedules available to the engineers to choose from. Finally, they proposed adding additional constraints and requirements, such as formalizing the notion of risk associated with a schedule and incorporating it as one of the objective functions. According to their explanation, the risk of a schedule is mainly defined by the consecutive period during which the hardware is used. This means that executing many tests consecutively over a short period represents less risk than executing them separately over a longer time period. This notion is already partially covered by our second fitness function, *fitfrag*, but formally introducing it as an objective for our approach could be beneficial for practitioners. They also suggested introducing the concept of priority, suggesting that performing certain test procedures earlier than others may be beneficial. Therefore, future research could focus on improving our approach to account for those requirements.

5.4.7 Threats to Validity

Internal validity. The primary internal threat to validity is the potential presence of hidden variables that may weaken the relation between the results obtained for the different approaches. To minimize this impact, we evaluated each approach using the same parameter settings. Additionally, we disclose all the configurations and share our experimental data to ensure reproducibility.

External validity. The primary threat to external validity is the potential lack of generalizability of our results to other contexts. This threat can be further divided into two aspects: (1) the extent to which our approach can be applied to systems different from our case study, and (2) whether similar benefits observed in our case study can be replicated in different contexts. For the first aspect, we thoroughly described the IOT requirements in Section 5.2.2, which are necessary for our approach to be applicable. As long as any IOT context meets these requirements, our approach remains applicable. These requirements are based on generic IOT campaigns, which are relevant to many satellite systems. Regarding the second aspect, although our case study was conducted in a representative realistic setting, additional case studies are required to validate our approach. However, we note that our tool is currently being used by practitioners, providing further confirmation that our approach may be applicable in various IOT contexts.

5.5 Related Work

Operational acceptance testing. Most research on acceptance testing focuses on on acceptance testing for agile software development methodologies (e.g., SCRUM) and User Acceptance Testing (UAT) [172, 173, 174, 175, 176, 177]. However, few works in the literature focus on test case scheduling of Operational Acceptance Testing (OAT) [11]. A notable work in this field was proposed by Shin et al. [138], where they developed a methodology to automate test case prioritization for Cyber-Physical Systems (CPS) acceptance testing. Their method accounts for time budget constraints, uncertainty, and hardware damage risks posed by the sequential execution of test cases. Unlike their approach, our work studies OAT in the context of mission-critical satellite systems, which accounts for different factors, such as antenna utilization, operational cost and context switching.

Test case prioritization. A research strand closely related to our topic is test case prioritization, which has been widely studied in the literature, especially in the topic of software regression testing [178, 179, 180, 181]. Nevertheless, the research strand that most closely aligns with our work concerns the prioritization of test cases with respect to resource constraints. Many of those techniques consider test execution time as the primary resource to be used for the prioritization process, with other objectives being based on the source code of the software under test. For example, Arrieta et al. [139] proposed a weight-based multi-objective search algorithm that prioritizes test cases for configurable CPSs, within each system configuration in order to optimize the testing process, considering the test execution time and the success rate of the tests at hand. Wang et al. [141] proposed a multi-objective search-based approach that prioritizes test cases in CPSs considering the execution time of the test cases as well as the hardware resource requirements. Arrieta et al. [140] proposed a search-based approach that prioritizes test cases for CPS product lines, aiming to optimize the testing process cost-effectively. This approach focuses on reducing fault detection time, simulation time, and the time required to cover both functional and non-functional requirements. However, these works prioritize test cases without addressing scheduling challenges, as the tests are executed sequentially without considering resource availability. Additionally, they do not account for conflicts between candidate test cases, nor the availability of operators, making these approaches not directly applicable to our work.

Test case scheduling for satellite-systems The research strand that most closely aligns with our work is the satellite control resource scheduling problem (SCRSP), more specifically, the sub-problem of Ground Measurement and Control Resource Allocation (GMCRA). This research strand addresses the allocation of ground resources for satellite control and measurement activities. For instance, Marinelli et al. [142] introduced a Lagrangian heuristic algorithm for satellite range scheduling with resource constraints. By framing the problem at hand as a sequence of maximum weighted independent set problems on interval graphs, Marinelli et al. [142] applied a Lagrangian relaxation to schedule satellite communication requests from a ground station to the Galileo GNSS constellation. Zhang et al. [143], Gao et al. [144], Wu et al. [145], and Zhang et al. [143, 146, 147] proposed ant colony optimization-based algorithms which take into account

the task interval constraints, resource availability constraints, and satellite constraints to provide efficient scheduling solutions. However, none of these approaches are directly applicable to our problem because they do not fully address the specific constraints and requirements of scheduling IOT test procedures.

5.6 Conclusions

In this chapter, we presented an approach for generating IOT test schedules, including procedure schedules and slot schedules, for operational acceptance testing of mission-critical satellite systems. Our approach, based on a multi-objective search algorithm, allows practitioners to efficiently generate schedules while satisfying objectives related to test resource usage, operational costs, and context switching. We evaluated our approach using industry test cases from SES Techcom, specifically for the IOT campaigns of the Galileo GNSS constellation. Our results indicate that our approach effectively addresses the IOT scheduling problem and outperforms an approach based on ant colony optimization. The generated schedules show an average cost reduction of 538% compared to manually created schedules, while maintaining low context switching and improving test resource efficiency by 39.42%.

Interviews with engineers highlighted additional benefits of our approach, such as reducing human error in test schedule generation through automation, simplifying schedule creation (with implementation taking less than one hour compared to a full workday for manual schedules), and enabling trade-off analysis by providing multiple viable schedules.

Engineers also suggested enhancing our approach by increasing the diversity of generated schedules and incorporating additional constraints, such as risk and priority. Future work will focus on integrating these aspects and addressing the need for more diversity. Additionally, we plan to apply our approach to a broader range of study subjects, including multiple antennas and different constellations.

Chapter 6

Conclusion & Future Works

6.1 Conclusion

In this dissertation, we addressed the challenges in generating test data and schedules for complex network systems.

We first proposed the test data generation methods for Software-Defined Network (SDN) systems. We presented FuzzSDN, a Machine Learning-guided fuzzing method for testing SDN systems. Further, we presented SeqFuzzSDN, a learning-guided fuzzing method for testing stateful SDN controllers. Both methods were empirically evaluated on popular State-Of-The-Art (SOTA) SDN controllers, ONOS and RYU.

We evaluated FuzzSDN on several synthetic SDN systems controlled by either one of the two SOTA SDN controllers. Our results indicate that FuzzSDN is able to generate effective test data that cause system failures and produce accurate failure-inducing models. Additionally, we compared SeqFuzzSDN against our extended versions of three SOTA methods for testing SDN controllers, which served as baselines in our evaluation. Our results show that SeqFuzzSDN significantly outperforms the baselines by generating effective and diverse tests (i.e., sequences of control messages), that cause the system to fail, and by producing accurate EFSMs.

Finally, we addressed the problem of generating test schedules Specifically, we presented a method for generating test schedules, for acceptance testing of mission-critical satellite systems. Our method, based on a multi-objective search algorithm, allows practitioners to efficiently generate schedules that satisfy managing constraints related to test resource usage, while minimizing antenna reconfiguration overhead, operational costs, and context switching.

We evaluated our test schedule generation method using industry test cases from SES Techcom, specifically for the In-Orbit Test campaigns of the Galileo GNSS constellation. Our results indicate that our approach effectively addresses the problem while satisfying constraints and outperforms commonly used Ant-Colony algorithms. The generated schedules show an average cost reduction of 538% compared to manually created plans, while maintaining low context-switching and improving test resource efficiency by 39.42%.

Additionally, interviews conducted with practitioners highlighted the benefits of our method for generating test schedules, such as reducing human error in test schedule generation through automation, simplifying schedule creation (with implementation taking less than one hour compared to a full workday for manual schedules), and enabling trade-off analysis by providing multiple viable plans.

6.2 Future Works

In Chapter 3, we focused on employing Machine Learning (ML) algorithms to guide the fuzzing of Software-Defined Network (SDN) control messages. Therefore, we introduced FuzzSDN, an approach aimed at generating comprehensive sets of test data and learning failure-inducing models that describe the conditions under which the system is likely to fail. A natural extension of this work, as proposed in the final section of Chapter 3, is to extend our FuzzSDN framework by accounting for the stateful nature of SDN systems. In Chapter 4, we introduced SeqFuzzSDN, a method that employs a fuzzing strategy guided by Extended Finite State machines (EFSMs) to efficiently explore the space of states of the SDN controller under test, generate effective and diverse tests (i.e., message sequences) to uncover failures, and infer accurate EFSMs that characterize the sequences of control messages leading to failures.

Regarding our work described in Chapters 3 and 4, future work should focus on validating the applicability of both FuzzSDN and SeqFuzzSDN by testing these frameworks on a broader set of case studies involving various SDN controllers. Additionally, conducting more user studies will help confirm the results' generalizability.

Furthermore, the method proposed in Chapter 4 introduces new challenge related to scalability in inferring EFSMs. We addressed this challenge in SeqFuzzSDN by developing a sampling approach, aiming at minimizing information loss. However, a more robust solution would involve using the entire dataset at each iteration, rather than a sampled dataset for inferring EFSMs. Therefore, a viable future direction would be to devise a learning technique that will allow SeqFuzzSDN to learn stateful models incrementally, addressing scalability issues in inferring EFSMs. This poses new challenges due to the complexity of continuously updating and maintaining complex dependencies between states and transitions, without losing any of the previously learned information.

In addition to the works proposed in Chapter 3 and Chapter 4, we proposed in Chapter 5 a method for generating procedure and slot schedules for acceptance testing of mission-critical satellite systems. As noted by the practitioners, future prospects for this work include enhancing our method by increasing the diversity of generated schedules and incorporating additional constraints, such as risk and priority. Future work will focus on integrating these new demands. Additionally, we plan to apply our method to a broader range of study subjects, including multiple antennas and different constellations.

Appendix A

Appendix

A.1 Additional Results for RYU Study Subject

A.1.1 Results for RQ1

In EXP1, when ONOS is replaced with RYU, Figure A.1.1 corresponds to Figure 4.5. It shows that SeqFuzzSDN achieves a sensitivity of 49.18% on the message sequences leading to both success and failure (referred to as the combined S+F dataset) and 63.37% on the message sequences leading only to failure (referred to as the combined F dataset). Specifically, SeqFuzzSDN achieves, on average, a sensitivity of 60.92% on the SeqFuzzSDN S+F dataset and 90.0% on the SeqFuzzSDN F dataset, 5.04% on the FuzzSDN^E S+F dataset and 0.00% on the FuzzSDN^E F dataset, 63.87% on the Beads^E S+F dataset and 82.56% on the Beads^E F dataset, and 66.89% on the Delta^E S+F dataset and 80.91% on the Delta^E F dataset.

Figure A.1.1 also shows the average EFSM sensitivity for RYU, for FuzzSDN^E, Beads^E, and Delta^E, respectively, of 26.18%, 11.96%, and 5.43% on the combined S+F dataset, and 25.0%, 3.71%, and 1.67% on the combined F dataset. More specifically, using the SeqFuzzSDN S+F dataset (and the SeqFuzzSDN F dataset), these three baselines achieve, respectively, on average, sensitivities of 0%, 20.41%, and 4.87% (and 0%, 0.67%, and 3.20%). Regarding the FuzzSDN^E S+F dataset (and the FuzzSDN^E F dataset), these three baselines achieve, respectively, on average, sensitivities of 92.29%, 1.51%, and 2.26% (and 100%, 0%, and 0%). For the Beads^E S+F dataset (and the Beads^E F dataset), these three baselines achieve, respectively, on average, sensitivities of 0%, 18.05%, and 0% (and 0%, 11.50%, and 0%). Lastly, when using the Delta^E

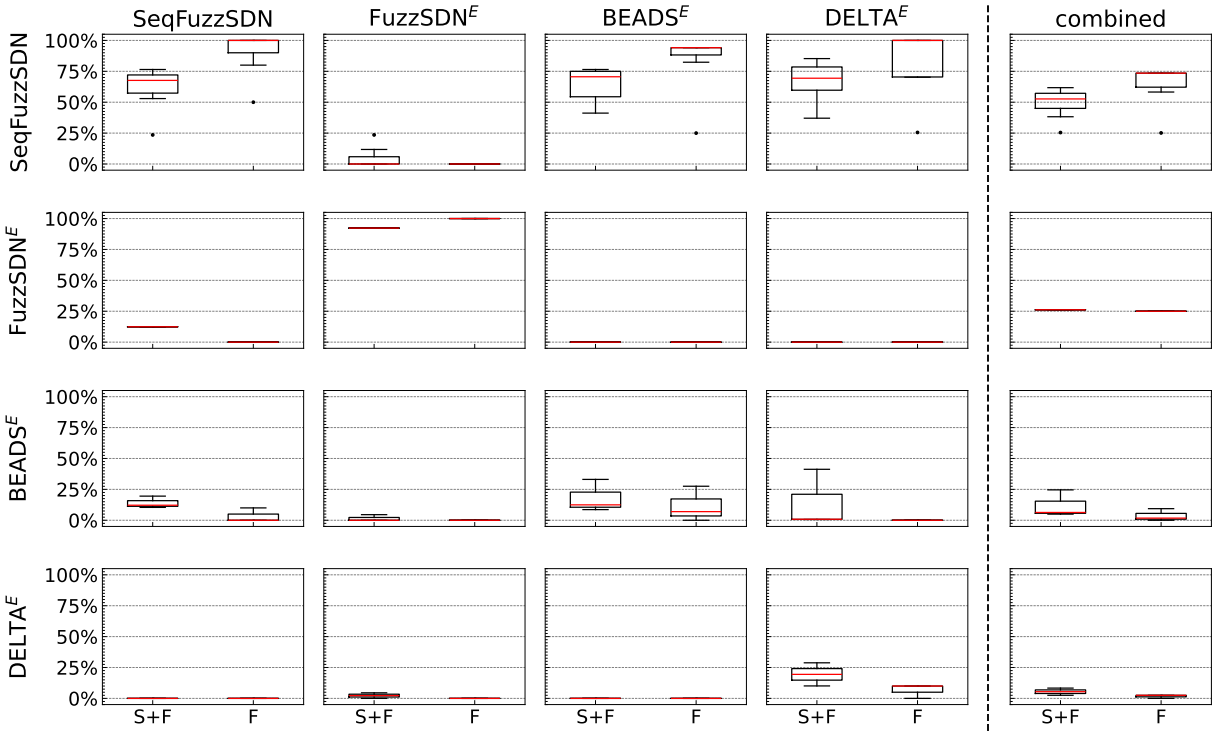
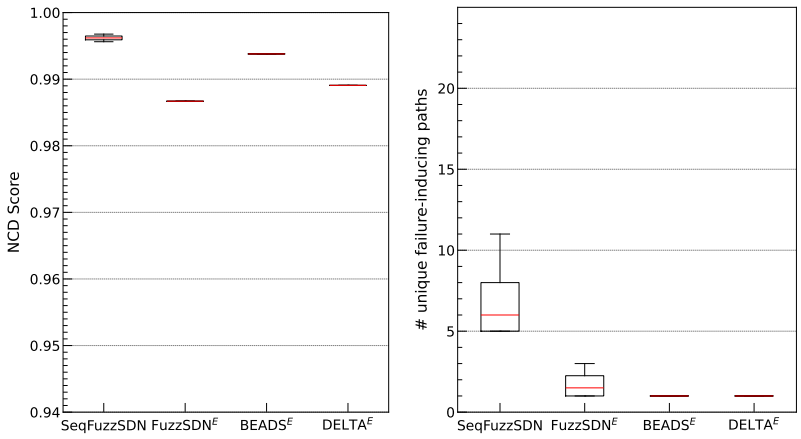


Figure A.1.1: Comparing the sensitivity of the EFSMs generated by SeqFuzzSDN, FuzzSDN^E, Beads^E, and Delta^E, the five plots in each row display the sensitivity of the corresponding tool. The first four columns represent the sensitivity of the EFSMs assessed using the test dataset containing message sequences generated by each tool. Sensitivity is assessed using message sequences that lead to both success and failure, denoted by (S+F), and only failure, denoted by (F). The last column represents the sensitivity assessed using all datasets generated by the four tools. The boxplots (25%-50%-75%) show the distribution of sensitivity over 10 runs of each tool in EXP1 (RYU).

S+F dataset (and the Delta^E F dataset), these baselines achieve, respectively, on average, sensitivities of 0%, 14.24%, and 19.44% (and 0%, 0%, and 6.67%).

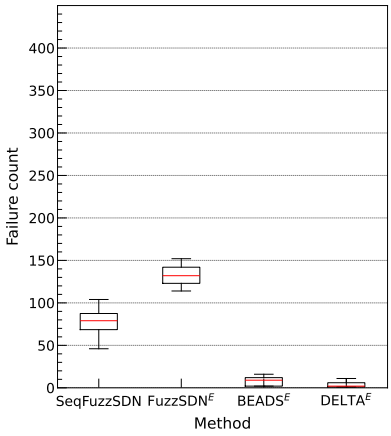
Figure A.1.2 compares (a) the NCD scores of the message sequences, (b) the number of unique failure-inducing paths in the EFSMs, and (c) the number of message sequences leading to failure, which are obtained from 10 runs of SeqFuzzSDN, FuzzSDN^E, Beads^E, and Delta^E for our RYU study subject. Figure A.1.2a shows that SeqFuzzSDN achieves a higher NCD score, with an average of 0.997, compared to those of the baselines. Figure A.1.2b shows that, on average, SeqFuzzSDN was able to infer an EFSM containing 6 unique loop-free paths that lead to failure, which is significantly higher than the others. From these results, similarly to our ONOS study subject, we found that SeqFuzzSDN generates more diverse sequences of control messages that exercise a larger number of state changes compared to the baselines.

However, Figure A.1.2c shows that FuzzSDN^E generates a larger number of message sequences (an average of 141) leading to failure compared to the other tools, while SeqFuzzSDN generates, on average, 76 message sequences leading to failure, thus outperforming Beads^E and Delta^E. As with our ONOS study subject, even though FuzzSDN^E outperforms SeqFuzzSDN in terms of number of failures, recall from Figure A.1.2a and Figure A.1.2b that FuzzSDN^E generates



(a) NCD scores

(b) Number of unique failure paths



(c) Number of failures

Figure A.1.2: Comparing (a) the NCD scores of the message sequences, (b) the number of unique failure-inducing paths in the EFSMs, and (c) the number of message sequences leading to failure, all obtained from SeqFuzzSDN, FuzzSDN^E, Beads^E, and Delta^E. The boxplots (25%-50%-75%) show the distribution of each metric over 10 runs of each tool in EXP1 (RYU).

message sequences that are less diverse and exercise significantly fewer number of state changes compared to SeqFuzzSDN. Furthermore, as described in Section 4.3, SeqFuzzSDN aims to generate a balanced number of message sequences that lead to success and failure, rather than focusing solely on the latter.

A.1.2 Results for RQ2

Figure A.1.3 compares SeqFuzzSDN and SeqFuzzSDN^{NS} with regard to the execution times per iteration for the fuzzing, learning, and planning steps over a time budget of 3 days, for our RYU study subject. Similarly to Figure 4.8, the bar graph shows the average execution times taken by SeqFuzzSDN and SeqFuzzSDN^{NS} for the fuzzing, learning, and planning steps at each iteration, based on 10 runs of EXP2 for RYU.

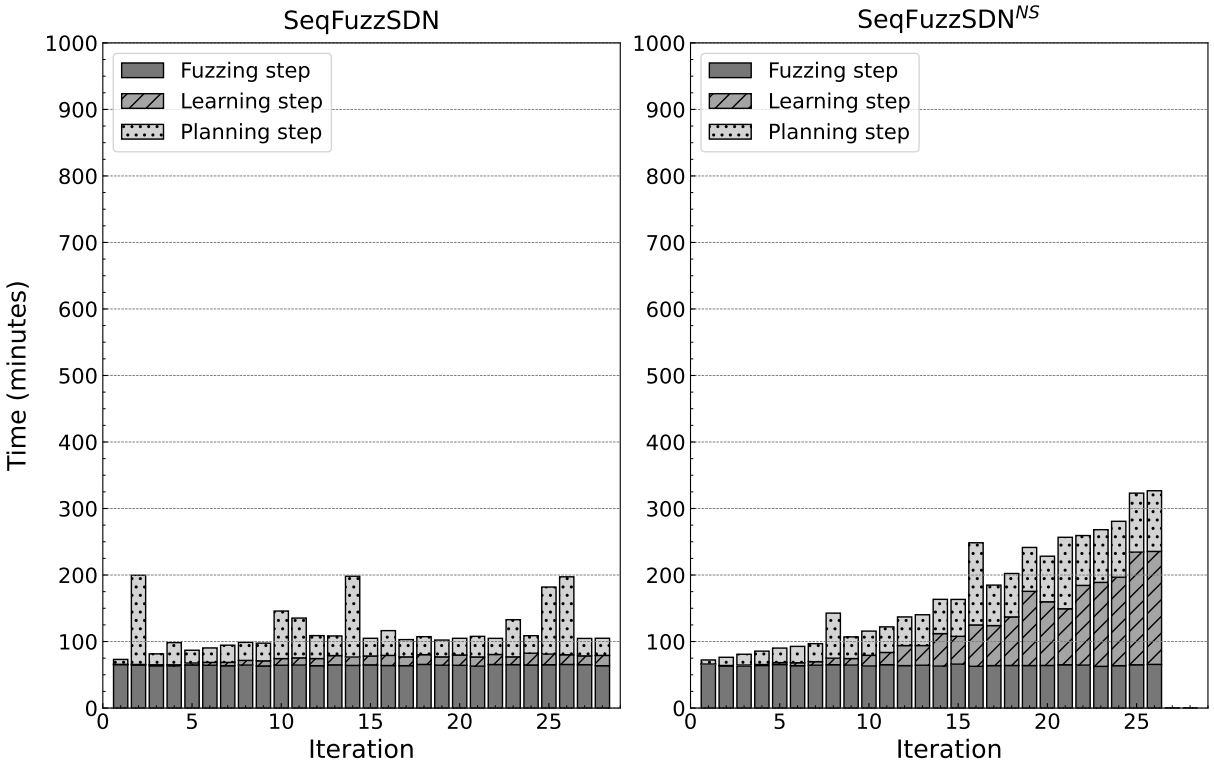


Figure A.1.3: Comparing the execution time per iteration for the fuzzing, learning, and planning steps of SeqFuzzSDN and SeqFuzzSDN^{NS} within a 3-day time budget. The execution times shown in this figure are the average values observed over 10 runs of EXP2 (RYU).

The results show that the fuzzing time per iteration remains constant at around 70 minutes for both SeqFuzzSDN and SeqFuzzSDN^{NS}, indicating that the fuzzing step is independent of the tool used. For the planning step, Figure A.1.3 shows that the planning time does not exceed 150 minutes in both SeqFuzzSDN and SeqFuzzSDN^{NS}. Figure A.1.3 also suggests that, for SeqFuzzSDN^{NS}, the time required to learn an EFSM increases exponentially with each iteration due to the growing size of the dataset fed to MINT. Furthermore, we observe that, as opposed to our ONOS study subject, the learning time for SeqFuzzSDN^{NS} does not reach the upper learning limit of 12h, but grows from under 1 minute to above 150 minutes. This finding aligns with the literature [110, 133, 109], as inferring EFSMs is a complex problem that scales poorly with larger input sizes. In contrast, the results for SeqFuzzSDN indicate that the time required for inferring an EFSM (i.e., the learning step) remains below 20 minutes due to the application of the sampling technique. Thus, based on the results shown in Figure A.1.3, we can further conclude that applying the sampling technique enables SeqFuzzSDN to overcome the scalability issues associated with the complexity of learning EFSMs.

Furthermore, Table A.1.1 presents the statistical test results for the distributions of sensitivity, diversity, and coverage (described in Section 4.3) achieved by SeqFuzzSDN and SeqFuzzSDN^{NS} after 10 runs of EXP2, using the Wilcoxon Rank-Sum test [134] with an α value of 0.05, for our RYU test subject. On average, SeqFuzzSDN (resp. SeqFuzzSDN^{NS}) achieves a sensitivity of 55.3% (resp. 53.4%), a diversity of 0.9976 (resp. 0.9975), and a coverage of 0.5866 (resp. 0.8248). We observed that the differences in sensitivity ($p = 0.18$) and diversity ($p = 0.7$) are

Table A.1.1: Statistical significance analysis using the Wilcoxon Rank-Sum test for sensitivity, diversity, and coverage results obtained from 10 runs of EXP2 (RYU).

Metric	Average (SeqFuzzSDN)	Average (SeqFuzzSDN^{NS})	p-value	Statistical Significance ($\alpha = 0.05$)
Sensitivity	0.553	0.534	0.385	Not Significant
Diversity	0.9976	0.9975	0.987	Not Significant
Coverage	0.5866	0.8248	0.0023	Significant

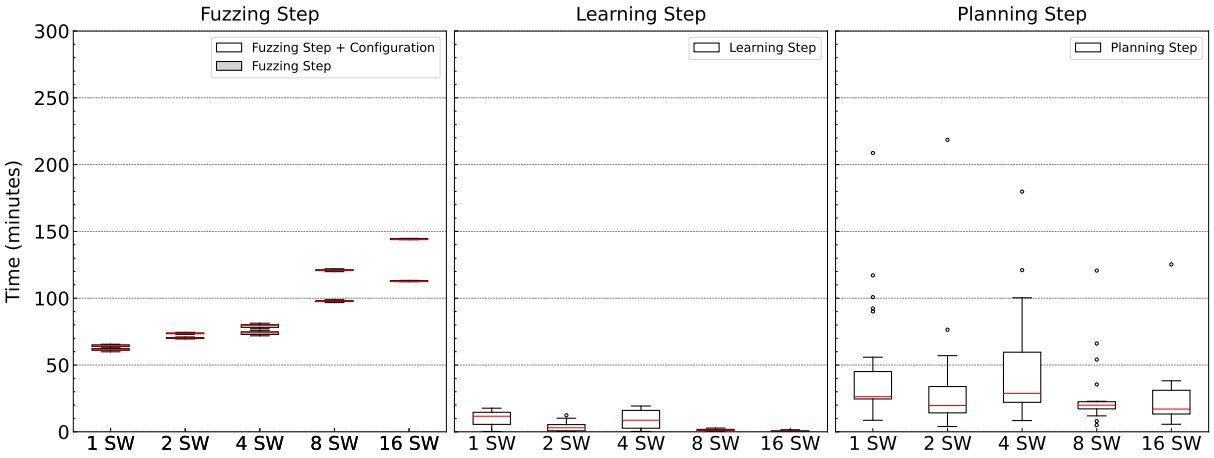


Figure A.1.4: Boxplots (25%-50%-75%) representing the distributions of time taken in minutes for the fuzzing, learning, and planning steps of SeqFuzzSDN. This figure includes the times observed over 10 runs of SeqFuzzSDN with 1, 2, 4, 8, and 16 switch configurations controlled by RYU.

not significant, while the difference in coverage ($p = 0.002$) is. The results indicate that the use of the sampling technique does not negatively impact the sensitivity of the generated EFSMs nor the diversity of the generated message sequences, on our RYU test subject. However, the coverage achieved by SeqFuzzSDN has significantly improved, suggesting that, similarly to our ONOS test subject, the states in the EFSM are explored more thoroughly.

A.1.3 Results for RQ3

Figure A.1.4 presents the distributions of execution times (25%-50%-75% boxplots) for the fuzzing, learning, and planning steps of SeqFuzzSDN, obtained from EXP3. These execution times were measured using the five study subjects in EXP3, which consist of 1, 2, 4, 8, and 16 switches controlled by RYU. As shown in Figure 4.9, the execution time taken for the fuzzing step is, on average, 203 minutes for the 1-switch configuration, 60 minutes for 2 switches, 70 minutes for 4 switches, 95 minutes for 8 switches, and 109 minutes for 16 switches. The learning step took, on average, 10 minutes for the 1-switch configuration, 3 minutes for 2 switches, 9 minutes for 4 switches, 2 minutes for 8 switches, and 1 minute for 16 switches. The planning step took, on average, 44 minutes for the 1-switch configuration, 31 minutes for 2 switches, 47 minutes for 4 switches, 28 minutes for 8 switches, and 33 minutes for 16 switches.

Bibliography

[1] Steven H Strogatz. Exploring complex networks. *nature*, 410(6825):268–276, 2001.

[2] Stefano Boccaletti, Vito Latora, Yamir Moreno, Martin Chavez, and D-U Hwang. Complex networks: Structure and dynamics. *Physics reports*, 424(4-5):175–308, 2006.

[3] Ernesto Estrada. *The structure of complex networks: theory and applications*. American Chemical Society, 2012.

[4] Reuven Cohen and Shlomo Havlin. *Complex networks: structure, robustness and function*. Cambridge university press, 2010.

[5] Dror Y. Kenett, Jianxi Gao, Xuqing Huang, Shuai Shao, Irena Vodenska, Sergey V. Buldyrev, Gerald Paul, H. Eugene Stanley, and Shlomo Havlin. *Network of Interdependent Networks: Overview of Theory and Applications*, pages 3–36. Springer, 2014.

[6] Waseem Ahmed and Yong Wei Wu. A survey on reliability in distributed systems. *Journal of Computer and System Sciences*, 79(8):1243–1255, 2013.

[7] Paolo Crucitti, Vito Latora, and Massimo Marchiori. Model for cascading failures in complex networks. *Physical Review E*, 69(4):045104, 2004.

[8] Sergey V Buldyrev, Roni Parshani, Gerald Paul, H Eugene Stanley, and Shlomo Havlin. Catastrophic cascade of failures in interdependent networks. *Nature*, 464(7291):1025–1028, 2010.

[9] John Wack, Miles Tracy, and Murugiah Souppaya. Guideline on network security testing. Technical Report SP 800-42, U.S. National Institute of Standards and Technology, Gaithersburg, MD, USA, October 2003.

[10] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[11] Paul Ammann and Jeff Offutt. *Introduction to software testing.* Cambridge University Press, second edition, 2016.

[12] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing.* John Wiley & Sons, 3rd edition, 2011.

[13] Klaus Wehrle, Mesut Günes, and James Gross. *Modeling and tools for network simulation.* Springer, 2010.

[14] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[15] Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O. Soremekun, and Andreas Zeller. Abstracting failure-inducing inputs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 237–248. ACM, 2020.

[16] Kalyanmoy Deb and Himanshu Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, 2014.

[17] Evangelos Haleplidis, Kostas Pentikousis, Spyros G. Denazis, Jamal Hadi Salim, David Meyer, and Odysseas G. Koufopavlou. Software-defined networking (SDN): Layers and architecture terminology. Information RFC 7426, Internet Research Task Force (IRTF), 2015.

[18] Dmitry Drutskoy, Eric Keller, and Jennifer Rexford. Scalable network virtualization in software-defined networks. *IEEE Internet Computing*, 17:20–27, 2013.

[19] Tao Wang, Fangming Liu, and Hong Xu. An efficient online algorithm for dynamic SDN controller assignment in data center networks. *IEEE/ACM Transactions on Networking*, 25:2788–2801, 2017.

[20] Wajid Rafique, Lianyong Qi, Ibrar Yaqoob, Muhammad Imran, Raihan Ur Rasool, and Wanchun Dou. Complementing IoT services through software defined networking and edge computing: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 22(3): 1761–1804, 2020.

[21] Seung Yeob Shin, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C. Briand, Chetan Arora, and Frank Zimmer. Dynamic adaptation of software-defined networks for IoT systems: A search-based approach. In *Proceedings of the 15th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 137–148, 2020.

[22] Ramon Ferrús, Harilaos Koumaras, Oriol Sallent, George Agapiou, Tinku Rasheed, M-A Kourtis, C Boustie, Patrick Gélard, and Toufik Ahmed. SDN/NFV-enabled satellite

communications networks: Opportunities, scenarios and challenges. *Journal of Physical Communication*, 18:95–112, 2016.

[23] Jiajia Liu, Yongpeng Shi, Lei Zhao, Yurui Cao, Wen Sun, and Nei Kato. Joint placement of controllers and gateways in SDN-enabled 5G-satellite integrated network. *IEEE Journal on Selected Areas in Communications*, 36(2):221–232, 2018.

[24] Seung Yeob Shin, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C. Briand, Chetan Arora, and Frank Zimmer. Dynamic adaptation of software-defined networks for iot systems: a search-based approach. In *Proceedings of the 15th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 137–148, 2020.

[25] Open Networking Foundation. OpenFlow switch specification, version 1.5.1. Specification ONF TS-025, Open Networking Foundation, 2015.

[26] David C. Plummer. An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. Information, Internet Engineering Task Force (IETF), 1982.

[27] Larry L Peterson and Bruce S Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, 2007.

[28] Jon Postel. Internet protocol. Information RFC 791, USC/Information Sciences Institute, 1981.

[29] Adrian Lara, Anisha Kolasani, and Byrav Ramamurthy. Network innovation using OpenFlow: A survey. *IEEE Communications Surveys & Tutorials*, 16:493–512, 2014.

[30] Jon Postel. User datagram protocol. Information RFC 768, USC/Information Sciences Institute, 1980.

[31] Jon Postel. Transmission control protocol. Information RFC 793, USC/Information Sciences Institute, 1981.

[32] John Moy. OSPF Version 2. Information RFC 2328, Ascend Communications, Inc., 1998.

[33] Yakov Rekhter, Tony Li, and Susan Hares. A Border Gateway Protocol 4 (BGP-4). Information RFC 4271, Internet Engineering Task Force (IETF), 2006.

[34] Seungsoo Lee, Changhoon Yoon, Chanhee Lee, Seungwon Shin, Vinod Yegneswaran, and Phillip Porras. DELTA: A security assessment framework for software-defined networks. In *Proceedings of the 24th Network and Distributed System Security Symposium*, pages 1–15, 2017.

[35] Samuel Jero, Xiangyu Bu, Cristina Nita-Rotaru, Hamed Okhravi, Richard Skowyra, and Sonia Fahmy. BEADS: Automated attack discovery in OpenFlow-based SDN systems. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 311–333, 2017.

[36] Abdullah M. Alshanqiti, Safi Faizullah, Sarwan Ali, Maria Khalid Alvi, Muhammad Asad Khan, and Imdadullah Khan. Detecting DDoS attack on SDN due to vulnerabilities in OpenFlow. In *Proceedings of the 2019 International Conference on Advances in the Emerging Computing Technologies*, pages 1–6, 2019.

[37] Seungsoo Lee, Seungwon Woo, Jinwoo Kim, Vinod Yegneswaran, Phillip A. Porras, and Seungwon Shin. AudiSDN: Automated detection of network policy inconsistencies in software-defined networks. In *Proceedings of the 39th IEEE Conference on Computer Communications*, pages 1788–1797, 2020.

[38] Apoorv Shukla, Said Jawad Saidi, Stefan Schmid, Marco Canini, Thomas Zinner, and Anja Feldmann. Toward consistent SDNs: A case for network state fuzzing. *IEEE Transactions on Network and Service Management*, 17(2):668–681, 2020.

[39] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *SIGCOMM Computer Communication Review*, 34(2):39—-53, 2004.

[40] Constantine A Balanis. *Antenna theory: analysis and design*. John wiley & sons, 2016.

[41] Gerard Maral, Michel Bousquet, and Zhili Sun. *Satellite communications systems: systems, techniques and technology*. John Wiley & Sons, 2020.

[42] Michał Zalewski. American Fuzzy Lop — whitepaper, 2016. URL `https://lcamtuf.coredump.cx/afl/technical_details.txt`.

[43] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies*, 2020.

[44] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNET: A greybox fuzzer for network protocols. In *Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification*, pages 460–465, 2020.

[45] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. Snooze: toward a stateful network protocol fuzzer. In *Proceedings of the 9th International Conference on Information Security*, pages 343–358, 2006.

[46] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *Security and Privacy in Communication Networks*, pages 330–347, 2015.

[47] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNET: A greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465, 2020.

[48] Roberto Natella. StateAFL: Greybox fuzzing for stateful network servers. *Empirical Software Engineering*, 27(7):191, 2022.

[49] Saurav Nanda, Faheem Zafari, Casimer DeCusatis, Eric Wedaa, and Baijian Yang. Predicting network attack patterns in SDN using machine learning approach. In *Proceedings of the 2016 IEEE Conference on Network Function Virtualization and Software Defined Networks*, pages 167–172, 2016.

[50] Suman Sankar Bhunia and Mohan Gurusamy. Dynamic attack detection and mitigation in iot using SDN. In *Proceedings of the 27th International Telecommunication Networks and Applications Conference*, pages 1–6, 2017.

[51] Peng Zhang. Towards rule enforcement verification for software defined networks. In *Proceedings of the 2017 IEEE Conference on Computer Communications*, pages 1–9, 2017.

[52] Juan Camilo Correa Chica, Jenny Cuatindioy Imbachi, and Juan Felipe Botero. Security in SDN: a comprehensive survey. *Journal of Network and Computer Applications*, 159:1–23, 2020.

[53] Yahui Li, Zhiliang Wang, Jiangyuan Yao, Xia Yin, Xingang Shi, Jianping Wu, and Han Zhang. MSAID: automated detection of interference in multiple SDN applications. *Computer Networks*, 153:49–62, 2019.

[54] Canini Marco, Venzano Daniele, Perešíni Peter, Kostić Dejan, and Rexford Jennifer. A NICE way to test OpenFlow applications. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, pages 127–140, 2012.

[55] Ramakrishnan Durairajan, Joel Sommers, and Paul Barford. Controller-agnostic SDN debugging. In Aruna Seneviratne, Christophe Diot, Jim Kurose, Augustin Chaintreau, and Luigi Rizzo, editors, *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 227–234, 2014.

[56] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 518–532, 2018.

[57] Seungwon Woo, Seungsoo Lee, Jinwoo Kim, and Seungwon Shin. RE-CHECKER: towards secure restful service in software-defined networking. In *Proceedings of the 2018 IEEE Conference on Network Function Virtualization and Software Defined Networks*, pages 1–5, 2018.

[58] Vaibhav Hemant Dixit, Adam Doupé, Yan Shoshitaishvili, Ziming Zhao, and Gail-Joon Ahn. AIM-SDN: attacking information mismanagement in sdn-datastores. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 664–676, 2018.

[59] Martin Björklund, Jürgen Schönwälder, Philip A. Shafer, Kent Watsen, and Robert Wilton. Network Management Datastore Architecture (NMDA). RFC 8342, 2018.

[60] Kinan Dak Albab, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, and Minlan Yu. Switchv: automated

SDN switch validation with P4 models. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 365–379, 2022.

[61] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *Computer Communication Review*, 44(3):87–95, 2014.

[62] Seungsoo Lee, Seungwon Woo, Jinwoo Kim, Jaehyun Nam, Vinod Yegneswaran, Phillip A. Porras, and Seungwon Shin. A framework for policy inconsistency detection in software-defined networks. *IEEE/ACM Transactions on Networking*, 30(3):1410–1423, 2022.

[63] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.

[64] Michael Crawford, Taghi M. Khoshgoftaar, Joseph D. Prusa, Aaron N. Richter, and Hamzah Al Najada. Survey of review spam detection using machine learning techniques. *Journal of Big Data*, 2(1):23, 2015.

[65] Zhong-Qiu Zhao, Peng Zheng, Shou-Tao Xu, and Xindong Wu. Object detection with deep learning: A review. *IEEE Transactions on Neural Networks and Learning Systems*, 30(11): 3212–3232, 2019.

[66] Igor Kononenko. Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in Medicine*, 23(1):89–109, 2001.

[67] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[68] Wei-Yin Loh. Classification and regression trees. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 1(1):14–23, 2011.

[69] Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. Support vector machines. *IEEE Intelligent Systems and their applications*, 13(4):18–28, 1998.

[70] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553): 436–444, 2015.

[71] William W. Cohen. Fast effective rule induction. In *Proceedings of the 12th International Conference on Machine Learning*, pages 115–123, 1995.

[72] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Chicago, 1975.

[73] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, volume 1, pages 69–93. Elsevier, 1991.

[74] Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[75] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. SPHINX: Detecting security attacks in software-defined networks. In *Proceedings of the 22nd Network and Distributed System Security Symposium*, pages 1–16, 2015.

[76] Christian Röpke and Thorsten Holz. SDN Rootkits: Subverting network operating systems of software-defined networks. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 339–356, 2015.

[77] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. A survey of man in the middle attacks. *IEEE Communications Surveys & Tutorials*, 18(3):2027–2051, 2016.

[78] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59, 2017.

[79] Yuqi Chen, Christopher M. Poskitt, Jun Sun, Sridhar Adepu, and Fan Zhang. Learning-guided network fuzzing for testing cyber-physical system defences. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, pages 962–973, 2019.

[80] Hui Zhao, Zhihui Li, Hansheng Wei, Jianqi Shi, and Yanhong Huang. SeqFuzzer: An industrial protocol fuzzing framework from a deep learning perspective. In *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification*, pages 59–67, 2019.

[81] Alexander Kampmann, Nikolas Havrikov, Ezekiel O. Soremekun, and Andreas Zeller. When does my program do this? learning circumstances of software behavior. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1228–1239, 2020.

[82] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an open, distributed SDN OS. In *Proceedings of the 3rd Workshop on Hot topics in Software Defined Networking*, pages 1–6, 2014.

[83] RYU Project Team. *RYU SDN Framework*. RYU Project Team, 1 edition, 2014.

[84] Raphael Ollando, Seung Yeob Shin, and Lionel C. Briand. [artifact repository] learning failure-inducing models for testing software-defined networks. `https://doi.org/10.6084/m9.figshare.20701354.v1`, 2023.

[85] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Data mining: practical machine learning tools and techniques*. Elsevier, 4 edition, 2016.

[86] Christoph Molnar. *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable*. Leanpub, 2 edition, 2022. URL `https://christophm.github.io/interpretable-ml-book`.

[87] Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

[88] Fitash Ul Haq, Donghwan Shin, Shiva Nejati, and Lionel C. Briand. Can offline testing of deep neural networks replace their online testing? *Empirical Software Engineering*, 26(90): 1–30, 2021.

[89] Baljinder Ghotra, Shane McIntosh, and Ahmed E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering*, pages 789–800, 2015.

[90] Caius Brindescu, Iftekhar Ahmed, Rafael Leano, and Anita Sarma. Planning for untangling: Predicting the difficulty of merge conflicts. In *Proceedings of the 42nd International Conference on Software Engineering*, pages 801–811, 2020.

[91] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[92] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceeding of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[93] El-Ghazali Talbi. *Metaheuristics: From design to implementation*. John Wiley & Sons, 1 edition, 2009.

[94] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, 2010.

[95] Fetia Bannour, Sami Souihi, and Abdelhamid Mellouk. Distributed SDN control: Survey, taxonomy, and challenges. *IEEE Communications Surveys & Tutorials*, 20:333–354, 2018.

[96] Robert T. Braden. Requirements for Internet Hosts - Communication Layers. Information RFC 1122, Internet Engineering Task Force (IETF), 1989.

[97] David C. Plummer. An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. Information, Internet Engineering Task Force (IETF), 1982.

[98] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 1 edition, 2019.

[99] Michael Smith, Robert Adams Edward, Mike Dvorkin, Youcef Laribi, Vijoy Pandey, Pankaj Garg, and Nik Weidenbacher. OpFlex Control Protocol. Internet Draft draft-smith-opflex-03, Internet Engineering Task Force, 2016.

[100] Joel M. Halpern, Robert Haas, Doria Avri, Ligang Dong, Weiming Wang, Hormuzd M. Khosravi, Jamal Hadi Salim, and Ram Gopal. Forwarding and Control Element Separation (ForCES) Protocol Specification. Information RFC 5810, Internet Engineering Task Force (IETF), 2010.

[101] Valentin J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47:2312–2331, 2021.

[102] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. Nsfuzz: Towards efficient and state-aware network service fuzzing. *ACM Transaction on Software Engineering and Methodolgies*, 32(6):160:1–160:26, 2023.

[103] Raphaël Ollando, Seung Yeob Shin, and Lionel C. Briand. Learning failure-inducing models for testing software-defined networks. *ACM Transaction on Software Engineering and Methodolgies*, 33(5):113:1–113:25, 2024.

[104] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. A survey of man in the middle attacks. *IEEE Communications Surveys & Tutorials*, 18(3):2027–2051, 2016.

[105] Raphael Ollando, Seung Yeob Shin, and Lionel C. Briand. [artifact repository] learning-guided fuzzing for testing stateful sdn controllers. `https://figshare.com/s/bb068dc048bc74d2d294`, 2024.

[106] Vangalur S. Alagar and Kasilingam Periyasamy. *Specification of Software Systems*, chapter Extended Finite State Machine, pages 105–128. Springer London, 2011.

[107] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. URL `https://www.fuzzingbook.org/`. Retrieved 2024-07-01 16:50:18+02:00.

[108] Neil Walkinshaw, Ramsay Taylor, and John Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, 2016.

[109] Donghwan Shin, Domenico Bianculli, and Lionel C. Briand. PRINS: scalable model inference for component-based system logs. *Empirical Software Engineering*, 27(4):1–32, 2022.

[110] S. S. Emam and J. Miller. Inferring extended probabilistic finite-state automaton models from software executions. *ACM Transactions on Software Engineering and Methodology*, 27(1):1–39, 2018.

[111] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Data mining: practical machine learning tools and techniques*. Elsevier, 4 edition, 2016.

[112] Qi Luo, Aswathy Nair, Mark Grechanik, and Denys Poshyvanyk. FOREPOST: finding performance problems automatically with feedback-directed learning software testing. *Empirical Software Engineering*, 22(1):6–56, 2017.

[113] John Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., 1993.

[114] Seifeddine Bettaieb, Seung Yeob Shin, Mehrdad Sabetzadeh, Lionel C. Briand, Michael Garceau, and Antoine Meyers. Using machine learning to assist with the selection of security controls during security assessment. *Empirical Software Engineering*, 25(4):2550–2582, 2020.

[115] Kalyanmoy Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley & Sons, 2001.

[116] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[117] Ana C. Lorena, Luís P. F. Garcia, Jens Lehmann, Marcilio C. P. Souto, and Tin Kam Ho. How complex is your classification problem? a survey on measuring classification complexity. *ACM Computing Surveys*, 52(5):107:0–107:34, 2019.

[118] R. Cilibrasi and P.M.B. Vitanyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[119] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *International Journal of Computer Mathematics*, 2(1-4):157–168, 1968.

[120] C.H. Bennett, P. Gacs, Ming Li, P.M.B. Vitanyi, and W.H. Zurek. Information distance. *IEEE Transactions on Information Theory*, 44(4):1407–1423, 1998.

[121] L. Peter Deutsch. Gzip file format specification version 4.3. Information RFC 1952, Aladdin Enterprises, 1996.

[122] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[123] Jaekwon Lee, Seung Yeob Shin, Lionel C. Briand, and Shiva Nejati. Probabilistic safe WCET estimation for weakly hard real-time systems at design stages. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–34, 2024.

[124] Jaekwon Lee, Seung Yeob Shin, Shiva Nejati, Lionel C. Briand, and Yago Isasi Parache. Estimating probabilistic safe WCET ranges of real-time systems at design stages. *ACM Transactions on Software Engineering and Methodology*, 32(2):37:1–37:33, 2023.

[125] Alessandro Calò, Paolo Arcaini, Shaukat Ali, Florian Hauer, and Fuyuki Ishikawa. Generating avoidable collision scenarios for testing autonomous driving systems. In *Proceeding of the 13th IEEE International Conference on Software Testing, Validation and Verification*, pages 375–386, 2020.

[126] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 533—-544, 2019.

[127] Seung Yeob Shin, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C. Briand, and Frank Zimmer. Test case prioritization for acceptance testing of cyber physical systems: a multi-objective search-based approach. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 49–60, 2018.

[128] David Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998.

[129] Jürgen Branke, Kalyanmoy Deb, Henning Dierolf, and Matthias Osswald. Finding knees in multi-objective optimization. In *Proceedings of the 8th International Conference on Parallel Problem Solving from Nature (PPSN'04)*, pages 722–731, 2004.

[130] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *Proceeding the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015.

[131] Tao Chen, Ke Li, Rami Bahsoon, and Xin Yao. FEMOSAA: Feature-guided and knee-driven multi-objective optimization for self-adaptive software. *ACM Transactions on Software Engineering and Methodology*, 27(2):1–50, 2018.

[132] Andrew R. Cohen and Paul M.B. Vitányi. Normalized compression distance of multisets with applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37 (8):1602–1614, 2015.

[133] Shaowei Wang, David Lo, Lingxiao Jiang, Shahar Maoz, and Aditya Budi. Chapter 21 - scalable parallelization of specification mining using distributed computing. In *The Art and Science of Analyzing Software Data*, pages 623–648. Morgan Kaufmann, 2015.

[134] Myles Hollander, Douglas A. Wolfe, and Eric Chicken. *Nonparametric Statistical Methods*. John Wiley & Sons, 2015.

[135] Yahui Li, Zhiliang Wang, Jiangyuan Yao, Xia Yin, Xingang Shi, Jianping Wu, and Han Zhang. MSAID: automated detection of interference in multiple SDN applications. *Computer Networks*, 153:49–62, 2019.

[136] Jinwoo Kim, Minjae Seo, Eduard Marin, Seungsoo Lee, Jaehyun Nam, and Seungwon Shin. Ambusher: Exploring the security of distributed sdn controllers through protocol state fuzzing. *IEEE Transactions on Information Forensics and Security*, 19:6264–6279, 2024.

[137] Peter Fortescue, John Stark, and Graham Swinerd. *Spacecraft Systems Engineering*. John Wiley & Sons, Ltd, 2011.

[138] Seung Yeob Shin, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C. Briand, and Frank Zimmer. Test case prioritization for acceptance testing of cyber physical systems: a multi-objective search-based approach. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18)*, pages 49–60, 2018.

[139] Aitor Arrieta, Shuai Wang, Goiuria Sagardui, and Leire Etxeberria. Test case prioritization of configurable cyber-physical systems with weight-based search algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 1053—1060, New York, NY, USA, 2016. Association for Computing Machinery.

[140] Aitor Arrieta, Shuai Wang, Goiuria Sagardui, and Leire Etxeberria. Search-based test case prioritization for simulation-based testing of cyber-physical system product lines. *Journal of Systems and Software*, 149:1–34, 2019.

[141] Shuai Wang, Shaukat Ali, Tao Yue, Øyvind Bakkeli, and Marius Liaaen. Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search. In *38th IEEE/ACM International Conference on Software Engineering Companion*, pages 182–191, 2016.

[142] Fabrizio Marinelli, Salvatore Nocella, Fabrizio Rossi, and Stefano Smriglio. A lagrangian heuristic for satellite range scheduling with resource constraints. *Computers & Operations Research*, 38(11):1572–1583, 2011.

[143] Na Zhang, Zu-ren Feng, and Liang-jun Ke. Guidance-solution based ant colony optimization for satellite control resource scheduling problem. *Applied Intelligence*, 35(3):436–444, 2011.

[144] Kebin Gao, Guohua Wu, and Jianghan Zhu. Multi-satellite observation scheduling based on a hybrid ant colony optimization. In *Proceedings of the 2nd International Symposium on Computer, Communication, Control and Automation (ISCCCA 2013)*, pages 675–678, 2013.

[145] Guohua Wu, Jin Liu, Manhao Ma, and Dishan Qiu. A two-phase scheduling method with the consideration of task clustering for earth observing satellites. *Computers & Operations Research*, 40(7):1884–1894, 2013.

[146] Zhaojun Zhang, Na Zhang, and Zuren Feng. Multi-satellite control resource scheduling based on ant colony optimization. *Expert Systems with Applications*, 41(6):2816–2823, 2014.

[147] Zhaojun Zhang, Funian Hu, and Na Zhang. Ant colony algorithm for satellite control resource scheduling problem. *Applied Intelligence*, 48(10):3295–3305, 2018.

[148] Patrick O'Connor and Andre Kleyner. *Practical Reliability Engineering*. John Wiley & Sons, 2012.

[149] Douglas Brent West. *Introduction to graph theory*. Upper Saddle River: Prentice hall, second edition, 2001.

[150] Hans-Paul Schwefel. *Numerical Optimization of Computer Models*. John Wiley & Sons, USA, 1981.

[151] Abdollah Homaifar, Charlene X Qi, and Steven H Lai. Constrained optimization via genetic algorithms. *Simulation*, 62(4):242–253, 1994.

[152] Mitsuo Gen and Runwei Cheng. *Genetic algorithms and engineering optimization*, volume 7. John Wiley & Sons, USA, 1999.

[153] Gunar E Liepins and Michael D Vose. Representational issues in genetic optimization. *Journal of Experimental & Theoretical Artificial Intelligence*, 2(2):101–115, 1990.

[154] Carlos A Coello Coello. Treating constraints as objectives for single-objective evolutionary optimization. *Engineering Optimization+ A35*, 32(3):275–308, 2000.

[155] Kalyanmoy Deb and Samir Agrawal. A niched-penalty approach for constraint handling in genetic algorithms. In *Artificial Neural Nets and Genetic Algorithms: Proceedings of the International Conference in Portorož*, pages 235–243, 1999.

[156] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using nsga-iii. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1263—-1270, New York, NY, USA, 2014. Association for Computing Machinery.

[157] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Ainhoa Arruabarrena, Leire Etxeberria, and Goiuria Sagardui. Pareto efficient multi-objective black-box test case selection for simulation-based testing. *Information and Software Technology*, 114:137–154, 2019.

[158] Raphael Ollando, Seung Yeob Shin, Mario Minardi, and Nikolas Sidiropoulos. [artifact repository] est schedule generation for acceptance testing of mission-critical satellite systems. `https://figshare.com/s/58931c1953bd4f9a0136`, 2024.

[159] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[160] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):1–61, 2012.

[161] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.

[162] Shuai Wang, Shaukat Ali, Tao Yue, Yan Li, and Marius Liaaen. A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 631–642, 2016.

[163] David A. Van Veldhuizen and Gary B. Lamont. Multiobjective evolutionary algorithm research: A history and analysis. Technical Report TR-98-03, Air Force Institute of Technology, 1998.

[164] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[165] Eckart Zitzler and Lothar Thiele. Multiobjective optimization using evolutionary algorithms — a comparative case study. In *Proceedings of the 5th International Conference on Parallel Problem Solving from Nature*, PPSN 'V, pages 292–301, 1998.

[166] Min Kong, Peng Tian, and Yucheng Kao. A new ant colony optimization algorithm for the multidimensional knapsack problem. *Computers & Operations Research*, 35(8):2672–2683, 2008.

[167] R. Montemanni, L. M. Gambardella, A. E. Rizzoli, and A. V. Donati. Ant colony system for a dynamic vehicle routing problem. *Journal of Combinatorial Optimization*, 10(4): 327–343, 2005.

[168] John E. Bell and Patrick R. McMullen. Ant colony optimization techniques for the vehicle routing problem. *Advanced Engineering Informatics*, 18(1):41–48, 2004.

[169] Thomas Stützle and Holger H. Hoos. MAX–MIN ant system. *Future Generation Computer Systems*, 16(8):889–914, 2000.

[170] Julian Blank, Kalyanmoy Deb, Yashesh Dhebar, Sunith Bandaru, and Haitham Seada. Generating well-spaced points on a unit simplex for evolutionary many-objective optimization. *IEEE Transactions on Evolutionary Computation*, 25(1):48–60, 2021.

[171] Andrea Arcuri and Gordon Fraser. On parameter tuning in search based software engineering. In *Proceedings of the 3rd International Symposium on Search Based Software Engineering (SSBSE'11)*, pages 33–47, 2011.

[172] Hareton KN Leung and Peter WL Wong. A study of user acceptance tests. *Software quality journal*, 6:137–149, 1997.

[173] Malte Finsterwalder. Automating acceptance tests for gui applications in an extreme programming environment. In *2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP'01)*, pages 114—-117, 2001.

[174] Fred D Davis and Viswanath Venkatesh. Toward preprototype user acceptance testing of new information systems: implications for software project management. *IEEE Transactions on Engineering management*, 51(1):31–46, 2004.

[175] Renate Löffler, Baris Güldali, and Silke Geisen. Towards model-based acceptance testing for scrum. *Softwaretechnik-Trends Band 30, Heft 3*, 2010.

[176] Olga Liskin, Christoph Herrmann, Eric Knauss, Thomas Kurpick, Bernhard Rumpe, and Kurt Schneider. Supporting acceptance testing in distributed software projects with integrated feedback systems: Experiences and requirements. In *IEEE 7th International Conference on Global Software Engineering*, pages 84–93. IEEE, 2012.

[177] Grischa Liebel, Emil Alégroth, and Robert Feldt. State-of-practice in gui-based system and acceptance testing: An industrial multiple-case study. In *Euromicro 39th Conference on Software Engineering and Advanced Applications*, pages 17–24. IEEE, 2013.

[178] Zheng Li, Mark Harman, and Robert M Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering*, 33(4):225–237, 2007.

[179] Benjamin Busjaeger and Tao Xie. Learning for test prioritization: an industrial case study. In *24th ACM SIGSOFT International symposium on foundations of software engineering*, pages 975–980, 2016.

[180] Cagatay Catal and Deepti Mishra. Test case prioritization: a systematic mapping study. *Software Quality Journal*, 21:445–478, 2013.

[181] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang NA Jawawi, and Rooster Tumeng. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 93:74–93, 2018.