Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi

Deep Reinforcement Learning Driven Applications Testing

by

Andrea Romdhana

Theses Series **DIBRIS-TH-2023-XXXV**

DIBRIS, Università di Genova
Via Opera Pia, 13 16145 Genova, Italy http://www.dibris.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica, Bioingegneria,

Robotica ed Ingegneria dei Sistemi

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum

Deep Reinforcement Learning Driven Applications Testing

by

Andrea Romdhana

January, 2023

Dottorato di Ricerca in Informatica ed Ingegneria dei Sistemi
Indirizzo Informatica
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università degli Studi di Genova

DIBRIS, Univ. di Genova
Via Opera Pia, 13
I-16145 Genova, Italy
http://www.dibris.unige.it/

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum
(S.S.D. INF/01)

Submitted by Andrea Romdhana
DIBRIS, Univ. di Genova

. . . .

Date of submission: October 2022

Title: Deep Reinforcement Learning Driven Applications Testing

Advisors: **Mariano Ceccato, Paolo Tonella, Alessio Merlo**
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università di Genova

. . .

Ext. Reviewers:
ABHIK ROYCHOUDHURY - National University of Singapore
LEONARDO MARIANI - University of Milano Bicocca

Abstract

Applications have become indispensable in our lives, and ensuring their correctness is now a critical issue. Automatic system test case generation can significantly improve the testing process for these applications, which has recently motivated researchers to work on this problem, defining various approaches. However, most state-of-the-art approaches automatically generate test cases leveraging symbolic execution or random exploration techniques. This led to techniques that lose efficiency when dealing with an increasing number of program constraints and become inapplicable when conditions are too challenging to solve or even to formulate. This Ph.D. thesis proposes addressing current techniques' limitations by exploiting Deep Reinforcement Learning. Deep Reinforcement Learning (Deep RL) is a machine learning technique that does not require a labeled training set as input since the learning process is guided by the positive or negative reward experienced during the tentative execution of a task. Hence, it can be used to dynamically learn how to build a test suite based on the feedback obtained during past successful or unsuccessful attempts. This dissertation presents three novel techniques that exploit this intuition: ARES, RONIN, and IFRIT.

Since functional testing and security testing are complementary, this Ph.D. thesis explores both testing techniques using the same approach for test cases generation. ARES is a Deep RL approach for functional testing of Android apps. RONIN addresses the issue of generating exploits for a subset of Android ICC vulnerabilities. Subsequently, to better expose the bugs discovered by previous techniques, this thesis presents IFRIT, a focused testing approach capable of increasing the number of test cases that can reach a specific target (i.e., a precise section or statement of an application) and their diversity. IFRIT has the ultimate goal of exposing faults affecting the given program point.

Acknowledgements

I would like to thank Professors *Mariano Ceccato*, *Paolo Tonella*, and *Alessio Merlo*. Over those years, they went over their role, taking care of me and demonstrating to be great people other than good Professors. I am also grateful that *Fondazione Bruno Kessler* and the *University of Genova* allowed me to take on a position as a Ph.D. student.

Naturally, I am also indebted to the people at *Università della Svizzera Italiana* for allowing me to work at the *Software Institute* while doing my research.

I would also like to thank all my *friends* inside and outside the university. You made this journey less difficult!

Many thanks go to my *family* for always believing in me, supporting my choices, and counseling me. I could not ask for more.

Last but not least, I want to thank my sweetheart *Hanna* for standing by me no matter what.

Table of Contents

Chapter 1

Introduction

Software verification and validation are essential components of modern software development processes that aim to improve the relationship between expected and actual software behaviors. Testing is the most common activity in software verification and validation. Testing assesses the quality of a software system by determining whether the results obtained by running it with a limited set of inputs correspond to its expected behavior. Depending on which portion of the application under test (AUT) is targeted, the testing phase can be carried out at different granularity levels: unit testing targets single components, whereas system testing targets sets of integrated classes or components and the whole system [PY08]. Unit testing aims at exhibiting faults in single units and may miss faults that occur during the integration of the numerous AUT software layers or during system execution. As a result, system testing, or testing the fully integrated application from the end user's perspective, is a fundamental type of testing. System testing entails running applications through their interfaces and stimulating all the layers and components involved in the execution. Because the number and complexity of the entities typically involved in system-level execution can be significant, defining test cases that thoroughly sample and verify an application's behavior is difficult and costly. The automated generation and execution of system test cases can significantly improve software testing activities while lowering software development costs. As a result, the research community is becoming increasingly interested in automated system test case generation. This Ph.D. thesis studies the problem of automatically generating system test cases for a popular type of software system, *applications*. Applications are software systems designed to perform a specific task. An application could showcase a Graphical User Interface or not. Applications that receive input as a unique stimulus from the outside are known as *Non-interactive applications*. These applications receive input, process it, and return an output. Applications relying on a Graphical User Interface (GUI) as the primary point of user interaction are known as *Interactive applications*. These applications are used daily to perform various tasks ranging from travel and social networking to banking and shopping. For instance, the Google Play Store contains more than 2.5 million interactive applications [Sta22]. As op-

posed to receiving input, processing it, and producing an output, interactive applications interact with the users through stimuli on the Graphical User Interface (GUI). The application reacts to such stimuli by producing new information and allowing users to interact further with new GUI elements not available before. As a result, interactive applications aim at engaging users in a user-friendly way. This thesis studies the problem of testing non-interactive and interactive applications and investigates solutions ideally applicable to those types of applications. The typologies of testing that I will focus on are *GUI Testing*, *Security Testing*, and *Focused Testing*. Since functional testing and security testing are complementary, this Ph.D. thesis explores both testing techniques using the same approach for test cases generation. Subsequently, to better exhibit the bugs exposed by previous testing techniques, this thesis presents a focused testing approach capable of increasing the number of test cases that can reach a specific target (i.e., a precise section or statement of an application) and their diversity. Figure 1.1 shows the logical flow that connects the three testing techniques.

GUI Testing. GUI testing produces test cases capable of exposing functional bugs as depicted in Figure 1.1. We define the term GUI testing to mean that a GUI-based application, i.e., one that exposes a graphical user interface (GUI) to users, is tested solely by performing sequences of events (e.g., "click on button", "enter text", "open menu") on GUI widgets (e.g., "button", "text-field", "pull-down menu"). In GUI testing, the level of code coverage is regarded as an adequacy criterion, and bug detection is a goal to ensure adequate testing of the software. However, in all but the most trivial GUI-based systems, the space of all potential event sequences that may be executed is vast. Increasing the number of possible operations increases the sequencing problem exponentially [HCM10]. Moreover, the complexity of current applications makes their exploration trickier than in the past, as they can contain states that are difficult to reach and events that are hard to trigger. Due to these challenges, manual and automated testing can become a severe issue since the chances of achieving high code coverage diminish.

Security Testing. Security testing produces test cases capable of exposing vulnerability bugs as depicted in Figure 1.1. Security testing verifies and validates software system requirements related to security properties, e.g., confidentiality, integrity, availability, authentication, authorization, and non-repudiation. Sometimes security properties come as classical functional requirements, e.g., "user accounts are disabled after three unsuccessful login attempts", defining security as a functional quality characteristic. For example, web application security vulnerabilities such as Cross-Site Scripting or SQL Injection, which security testing techniques aim to address, are acknowledged problems [BBGM10] with thousands of vulnerabilities reported every year [Det22]. Furthermore, surveys published by Synopsys [Syn22] show the high cost of insecure software due to inadequate testing, even on an economic level. Therefore, support for security testing is essential to increase its effectiveness and efficiency in practice.

Focused Testing. Focused testing takes the bugs exposed by GUI testing and security testing generating additional test cases (Figure 1.1). Many software projects are constantly developing, and new versions are released continuously. These modifications may introduce new code or

alter the existing code's execution flow. Therefore, existing test suites may not adequately cover the new/modified code. As a result, it is crucial to automate the creation of test suites focusing on specific application sections, to ensure that such new/modified code sections do not introduce bugs into the system. Although monitoring the level of code coverage is a highly recommended practice, coverage strategies alone have limited capabilities in detecting real faults. Covering a faulty statement once does not guarantee that the fault will be activated, i.e., the application state will be infected and will propagate to an observable output [IH14, AGB+16, Voa92]. In addition to coverage, another key goal of test suite creation should be diversity because diversity can increase the chances of fault exposure [SYB18]. To address this problem, Menéndez et al.[MJS+21] proposed to focus the testing process on the code sections added/modified by the developers and to generate a diversified set of uniformly distributed test cases that exercise such sections. This methodology, known as *focused testing*, ensures that multiple, diverse tests (i.e., a *focused test suite*) exercise a few specific application elements.

There exist several approaches that automate the generation of test cases belonging to the aforementioned testing typologies. *Random testing* strategies [Goo20c, MTN13, MAZ+15, PLEB07] stimulate the AUT by producing pseudo-random test cases. However, random generation is neither effective nor efficient on complex applications. Model-Based strategies [AFT+12, SMC+17a, GSM+19] extract test cases from navigation models built employing static or dynamic analysis. If the model accurately reflects the AUT, a deep exploration can be achieved. Nonetheless, automatically constructed models tend to be incomplete and inaccurate. Structural strategies [ANHY12, GTDR18, MMM14, MJS+21, GHGM17] generate inputs using symbolic execution or evolutionary algorithms. These strategies are more powerful since a specific coverage target guides them. However, they do not take advantage of past exploration successes to dynamically learn the most compelling exploration strategy. Reinforcement Learning (RL) is a machine learning approach that does not require a labeled training set as input since the learning process is guided by the positive or negative reward experienced during the tentative execution of the task. Hence, it represents a way to dynamically build an optimal exploration strategy by taking advantage of past successful or unsuccessful moves. RL has been extensively applied to the problem of GUI and Android testing [MPRS12, PHW+20]. However, only the most basic RL (i.e., Tabular RL) has been applied to testing problems so far. In Tabular RL, the value of the state-action associations is stored in a fixed table. The advent of Deep Neural Networks (DNN) replaced Tabular approaches with Deep Learning ones, in which the action-value function is learned from the past positive and negative experiences made by one or more neural networks. When the state space to explore is extremely large (e.g., when an app has a significant amount of widgets), Deep RL has proved to be substantially superior to Tabular RL [BM95] [Rie05] [Li17]. This thesis studies the problem of defining cost-effective Deep RL-based testing approaches for interactive and non-interactive applications. I investigate ways to automatically generate system test cases that can satisfy the requirements expressed by testing typology without requiring the presence of artifacts expensive to produce.

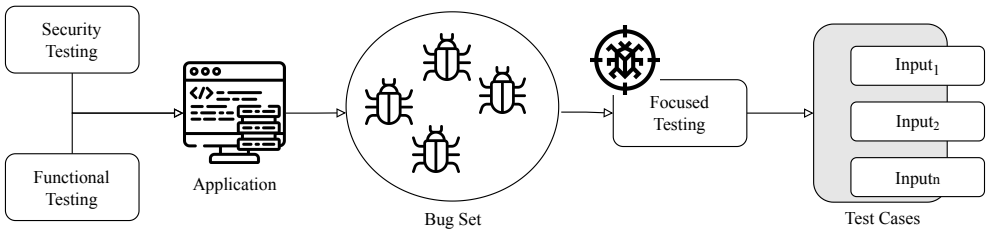

Figure 1.1: Logical flow between testing techniques

1.1 Motivation of Reinforcement Learning

Given the complexity of contemporary applications, the number of possible inputs or interaction sequences might be huge. Thus it is crucial to identify a set of meaningful inputs or interactions.

Depending on the testing typology, the set of meaningful inputs or interactions assumes a different meaning:

- In GUI testing, the objective is to generate test cases that maximize code coverage and bug detection.
- In focused testing, we aim to forge multiple, diverse test cases that exercise specific application elements.
- In security testing, the primary purpose is to detect and validate the most significant number of software vulnerabilities.

Reinforcement Learning is a machine learning approach that represents a viable option to produce meaningful inputs or interactions. In RL, the learning process is guided by the positive or negative reward experienced during the tentative testing of an AUT. Hence, it represents a way to dynamically build optimal test cases by taking advantage of past successful or unsuccessful moves. For example, a GUI interaction that only navigates through menus and windows is seldom meaningful because it does not increase the explored AUT surface, and the algorithm reward is negative. In contrast, a GUI interaction that goes in the "register user" window, fills out the form with valid data, clicks "register", and continues the exploration is meaningful because it increases the AUT's explored surface, experiencing positive reward.

1.2 Research Contribution

In this dissertation, I advance state of the art by defining techniques that exploit Deep Reinforcement Learning in different ways:

GUI Testing of Mobile Apps through Reinforcement Learning. This thesis presents the first Deep RL approach, ARES, for automated black-box testing of Android apps. ARES uses a DNN to learn the best exploration strategy from previous attempts. Thanks to DNN, it achieves high scalability, general applicability, and the capability to handle complex app behaviors. ARES implements multiple Deep RL algorithms that come with a set of configurable, often critical, hyperparameters. To speed up selecting the most appropriate algorithm for the AUT and fine-tuning its hyperparameters, I have developed another tool, FATE, which integrates with ARES. FATE is a simulation environment that supports rapid assessment of Android testing algorithms by running *synthetic* Android apps (i.e., abstract navigational models of real Android apps). On average, the execution of a testing session on a FATE synthetic app is 10 to 100 times faster than the execution of the same session on the corresponding real Android app. I applied ARES to two benchmarks made of 41 and 68 Android apps, respectively. The first benchmark compares the performance of the ARES algorithms, while the latter evaluates ARES w.r.t. the state-of-the-art testing tools for Android. Experimental results confirmed the hypothesis that Deep RL outperforms Tabular RL in exploring the state space of Android apps, as ARES exposed the highest number of faults and obtained the highest code coverage. Furthermore, I carried out a qualitative analysis showing that the features of Android apps that make Deep RL particularly adequate include, among others, the presence of concatenated activities and blocking activities protected by authentication.

The results of this research activity have been published as:

ANDREA ROMDHANA, ALESSIO MERLO, MARIANO CECCATO, PAOLO TONELLA, DEEP REINFORCEMENT LEARNING FOR BLACK-BOX TESTING OF ANDROID APPS, ACM TRANSACTIONS ON SOFTWARE ENGINEERING AND METHODOLOGY

Security Testing of Mobile Apps through Reinforcement Learning. This thesis proposes an approach to exploit generation, called RONIN, based on Deep Reinforcement Learning. Deep RL can be used to dynamically learn how to build an Intent that exposes a specific vulnerability based on the feedback obtained during past successful or unsuccessful attempts. More specifically, RONIN manipulates the parameters of the Intents by applying a sequence of actions to them. Each action receives positive feedback if we move closer to the target statement (i.e., the vulnerable statement) upon execution of the Intent; neutral (zero) feedback if the minimum distance between the statements that we reached and the target statement does not change; negative feedback if we increase the distance from the target w.r.t. the last Intent execution.

The results of this research activity are under journal review as:

ANDREA ROMDHANA, ALESSIO MERLO, MARIANO CECCATO, PAOLO TONELLA, ASSESSING THE SECURITY OF INTER-APP COMMUNICATIONS IN ANDROID THROUGH REINFORCEMENT LEARNING, COMPUTERS & SECURITY

Focused Testing through Reinforcement Learning. This thesis proposes IFRIT, a novel approach to focused testing based on Deep Reinforcement Learning. Deep RL can be used to

dynamically learn how to build a focused test suite based on the feedback obtained during past successful or unsuccessful attempts. IFRIT manipulates a test input by applying a sequence of modifiers (actions) to it. Each action receives positive feedback if the target statements are reached upon execution of the test input and such input was never generated before (to promote diversity); neutral (zero) feedback if the target statements get executed, but the input is not new; negative feedback if the input does not reach the target.

The results of this research activity have been published as:

ANDREA ROMDHANA, MARIANO CECCATO, ALESSIO MERLO, PAOLO TONELLA, IFRIT: FOCUSED TESTING THROUGH DEEP REINFORCEMENT LEARNING, 2022 IEEE CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION (ICST)

1.3 Problem Definition

Software testing is carried out to assess software qualities or identify flaws so developers can fix them. A good test has a high probability of finding an as-yet-undiscovered error, and a successful test uncovers one or more of such as-yet-undiscovered errors. Thus research and development on testing aim at efficiently performing effective testing – to find more errors in the requirement, design, and implementation and to increase confidence that the software has various qualities. Errors can be functional faults or security faults. Once a fault has been found, it is helpful to have several diversified scenarios that expose such faults. In this way, debugging and error fixing are simplified. To this aim, I decided to approach the problem of testing an application by investigating several approaches: GUI testing, security testing, and focused testing.

1.3.1 GUI Testing

In GUI testing, automating the generation of test cases for interactive applications faces one main objective: *detecting bugs*.

Bug detection is the metric we rely on to evaluate the quality of a test cases generator [HN11]. Bug detection is essential to improve software reliability. The evaluation of code coverage is the problem of identifying the parts of a program that did not execute in one or more runs of a program [TH02]. Covering more code we are also more likely to discover more faults. The purpose of a GUI test case generator is to identify the highest possible number of bugs.

1.3.2 Security Testing

Software security testing is the process of identifying whether the security features of an application's implementation are consistent with the design. We can divide software security testing into *security functional testing* and *security vulnerability testing*. *Security functional testing* ensures that software security functions are implemented correctly and consistently based on software security requirements. The application is tested as it is meant to be used, to assure that specific functions and activities of the code are working. Software security requirements mainly include data confidentiality, integrity, availability, authentication, authorization, access control, etc. *Security vulnerability testing* is to discover security vulnerabilities as an attacker. The term vulnerability refers to system design, implementation, operation, and management flaws. A vulnerability may be used to attack, resulting in a state of insecurity. Security vulnerability testing is to identify software security vulnerabilities and analyze how an application manages unexpected inputs. Because of this reason, Reinforcement Learning characteristics better apply to security vulnerability testing. In this thesis, I focus on security vulnerability testing, i.e., the generation of a test case generator whose purpose is to identify and exploit security vulnerabilities.

1.3.3 Focused Testing

The application components addressed by focused testing are said to be *program points (pp)* (i.e., specific nodes in the control flow graph). Given the space of the program inputs X, we denote by X_{pp} the sub-space of inputs whose execution traverses pp. A focused test input generator aims at producing inputs x that belong to X_{pp}. In addition to this, the generator should produce a *diverse* set of inputs that belongs to X_{pp}.

Following the work by Chakraborty et al. [CMV13], we define diversity using entropy. A diverse set is a set with high entropy. We define a generator G as an algorithm that creates inputs for a program P. We define a focused generator G_{pp} as a generator that generates inputs that traverse a specific program point pp. Considering the generator as a random variable whose values are inputs traversing pp, our goal is to make them as much diverse as possible, i.e., we aim at creating a focused generator G_{pp} whose entropy is maximized.

Since the entropy of a random variable is maximum when its probability distribution is uniform [Cov99], G_{pp} should be a uniform random variable, and the generator should be a *uniform focused generator*, i.e., a generator that gives the same generation probability to every input $x \in X_{pp}$.

We need to quantify how close a focused test input generator is to generating samples from a uniform distribution to measure its uniformity. The work in [MMP05] reports different statistical tests to measure it. These tests are divided into several categories: order statistics, spacing, order spacing, and collisions. Some of them do not apply to discrete distributions [MMP05], while

others do not manage gaps in the domain [Pla83]. Tests based on the collisions can deal with gaps, and discrete distributions [GR11a]. Consistently with the recommendation made by the authors of DFT [MJS+21], as a practical way to measure the degree of uniformity of a sample of values generated for a given random variable G_{pp}, we use the $L2$ test [GR11b]. The idea behind this test is that collisions (i.e., identical variable values) observed in the sample should be less than those that an ϵ-far uniform distribution defined on the same domain would generate, where ϵ is a user-defined tolerance. When this happens, the $L2$ test is passed; otherwise, it fails. The $L2$ test relies on the following formulas:

$$c = \sum_{x \neq y, (x,y) \in S \times S} \delta_{x,y}/2 \tag{1.1}$$

$$\theta = \binom{|S|}{2} \frac{1 + 3\epsilon^2/4}{n} \tag{1.2}$$

where S is a sample generated from G_{pp}; c counts the number of collisions in S using Kronecker delta $\delta_{x,y}$; ϵ defines the acceptable tolerance, i.e., the maximum allowed distance from the uniform probability distribution; n is the domain size. When $c < \theta$, the test is passed; otherwise it fails. In this thesis, I focus on the problem of developing a test generator that focuses the testing process on specific pp and generates a diversified set of uniformly distributed test cases that exercise such pp.

1.4 Thesis Organization

The thesis is organized as follows:

- Chapter 2 formalizes the problem of GUI testing, security testing, and focused testing.
- Chapter 3 describes the state of the art of automatic test case generation for interactive and non-interactive applications, discussing strengths and limitations.
- Chapter 4 describes the key concepts of Reinforcement Learning.
- Chapter 5 introduces ARES. It describes how ARES tests Android apps through the GUI and presents the results of an empirical evaluation of ARES.
- Chapter 6 presents RONIN. It describes how RONIN assesses the Security of Inter-Component Communications (ICC) in Android through Reinforcement Learning. RONIN detects and exploits vulnerabilities in ICC and presents the results of an empirical evaluation of RONIN.

- Chapter 7 presents IFRIT, an approach that uses Deep Reinforcement Learning to generate diverse inputs while maintaining a high reachability of the desired application component. RONIN achieves better results than state-of-the-art and baseline tools, improving reachability, diversity, and fault detection.
- Chapter 8 summarizes the contributions of the thesis and discusses open research directions.

Chapter 2

State of the Art

2.1 Automated Test Input Generation

The two main families of test input generators make use respectively of static/dynamic symbolic execution [CDE08, SMA05, GKS05] and search-based algorithms [McM04, FA13, PKT18]. Techniques based on symbolic execution encode the path constraints that must be satisfied to traverse a given path in a formula that can be passed to an SMT solver. Search-based algorithms rely instead on a fitness function that heuristically measures the distance between the path traversed when executing a candidate input and the coverage target. Inputs that get closer to the target are selected and iteratively improved until the target is covered. Both approaches succeed when the target is covered, but none attempt to generate multiple, uniformly distributed inputs that reach the target.

Differently from symbolic execution and search-based test generation, random test input generation [MAZ+15, PLEB07] ensures uniformity of the generated inputs by construction. However, when the target to be covered requires that very specific path conditions are satisfied, this approach has a very low probability of generating inputs whose execution can actually traverse the target. So, despite the high intrinsic uniformity of the generated inputs, this approach is often ineffective because of the low reachability of targets that are difficult to cover randomly.

Only a few works [AH12, FPCY16, BSRT19] include diversity among the test generator's goals, but none in the context of focused testing. Alshahwan et al. [AH12] maximize the diversity of the distribution of the outputs produced upon the execution of the automatically generated test inputs. So, they consider output, instead of input diversity. Feldt et al. [FPCY16] propose a new diversity metric, called test suite diameter, to quantify the degree of diversity in a test suite, but they do not use it directly for (focused) test generation. Biagiola et al. [BSRT19] use diversity as a criterion to select the most promising candidates because in-browser web test execution is computationally expensive, and diversity ensures wider exploration of behaviors. However, their

goal is different from the generation of uniformly distributed inputs that reach a target of interest.

2.2 GUI Testing

Recently, many researchers have investigated the challenging problem of automatically generating test cases for interactive applications. So far, the research community efforts have mostly focused on the problem of automatically generating GUI interactions. The techniques proposed for automating the generation of GUI interactions can be divided into four classes: *random approaches* [Goo20c, VKCF+15], *model-based approaches* [MPRS12, MBN03, AFT+14], *structural approaches* [GTDR18, DBCR20b, DBCR20b, MHJ16], and *machine learning-based approaches* [BGZ18, KSM+18, LYGC19].

2.2.1 Random Approaches

Random approaches are arguably the most simple type of test case generation technique. They produce test cases as sequences of events identified randomly or using simple heuristics. Monkey [Goo20c] is one of the most popular black-box GUI testing tools available in the official Android toolkit. It triggers events by interacting randomly with screen coordinates. This simple random approach works relatively well on some benchmark applications [CGO15a]. Nonetheless, Monkey tests involve many ineffective or repeated events, as there is no guidance to make the exploration efficient. Other random testing techniques use an observe-select-execute cycle. The application GUI is monitored to detect possible events before selecting the next one to execute [MTN13, VKCF+15]. This approach improves over Monkey. These techniques use different strategies to detect enabled events. Testar-Random targets desktop applications and uses the widget types to detect which events can be performed (e.g., a button widget accepts click events). Dynodroid targets Android apps and analyzes the source code to detect the event listeners registered for each widget (e.g., if a widget registers an onClick listener, it accepts clicks events). Because of their simplicity, these techniques can quickly generate and execute test cases; therefore, they are often used to perform stress testing. Generally, random testing techniques can be pretty effective as they can quickly cover significant portions of the AUT interaction space with their high-speed [BP14, CGO15b]. However, because of their lack of guidance, they struggle to cover those complex interactions that require a long and precise sequence of events. Thus, these approaches often fail to cover entire parts of the application execution space and might leave relevant AUT functionalities completely untested.

2.2.2 Model-based Approaches

Model-based approaches [AFT+12] [AFT+14] [GSS+20] first build navigation models of the Android application employing static or dynamic analysis, used to explore the application states efficiently. Then they extract test cases from such models to eventually expose bugs. Model-based approaches propose different definitions and encodings of GUI models. In general, GUI models are directed graphs where the *GUI windows* represent the set of *nodes*, and the *edges* are the set of *transitions between windows*. Windows are defined in terms of the widgets they contain, and edges are defined as

$$Egde : \langle W_{source}, W_{target}, Trigger \rangle \tag{2.1}$$

where W_{source} and W_{target} are the windows from which the transition originates and to which it arrives, while $Trigger$ is an event that operates on a widget in W_{source} that causes the current window to pass to W_{target}. Model-based approaches can be distinguished between approaches that first build the GUI model and then generate test cases and approaches that iteratively combine the building of the GUI model with test case generation. The approach of first building the GUI model and then using it for deriving test cases was firstly introduced in the paper by Memon et al. that defined GUI Ripper [MBN03]. GUI Ripper builds the AUT GUI model by performing a depth-first traversal of the AUT GUI and then uses the GUI model to derive a model, called EFG, of the partial execution order of the events in the GUI. GUI Ripper gave rise to a series of techniques that produce test cases by iteratively navigating the extracted GUI model utilizing various model abstractions, GUI coverage metrics based on combinatorial interaction testing, and heuristics [MX05, MBNR13, AFT+12, AFT+14]. AndroidRipper [AFT+12], and MobiGUITAR [AFT+14] try to maximize the exploration by using the ripping technique. Guo et al. [GSS+20] use static analysis to improve GUI exploration. Stoat [SMC+17a] uses a stochastic FSM to model the application behavior. The application model is built using dynamic analysis, enhanced with a weighted UI exploration strategy, and with the help of static analysis. Model-based approaches are limited by the completeness of the derived GUI model. Some relevant GUI elements might not be accessible without performing complex sequences of actions that the GUI exploration strategies used by these techniques might overlook. Model-based methodologies omit to test potentially essential areas of the GUI interaction spaces in these situations.

2.2.3 Structural Approaches

Structural approaches [ANHY12, GTDR18, MMM14, DBCR20b, MHJ16] leverage the AUT source code to drive test case generation and maximize AUT source code coverage, ideally, executing each AUT source code statement. Several structural approaches steer test case generation leveraging genetic algorithms, a classical search-based approach [Bac96] inspired by natural selection that has been used in many software engineering contexts [Har07, HJ01, HMZ12]. In

a nutshell, a genetic algorithm starts with a random population of solutions for its search problem. It iteratively evolves them (using crossover and mutations), selecting the fittest individuals (according to a fitness metric) for reproduction to produce a better population for the next iteration. In interactive application testing, the search problem is finding the set of test cases with the highest AUT statement coverage. Then, using statement coverage as the fitness metric, genetic algorithms evolve an initial set of random test cases combining/mutating them to retain those that achieve the highest coverage [MHJ16]. For instance, Sapienz [MHJ16] maximizes code coverage and bug revelation using a Pareto-optimal multi-objective search-based approach, which applies genetic operators such as mutation and crossover to produce new test cases. It can generate specific input for text fields by reverse-engineering the APK. This process occasionally results in invalid sequences discarded by the fitness functions that reward test cases with high coverage. TimeMachine [DBCR20b] improves Sapienz by identifying interesting states in the past and restarting the search process from them when the search stagnates. Other techniques instead use symbolic/concolic execution [CS13] to generate test cases that aim to achieve 100% code coverage [ANHY12, CCYW16, GKKP09]. Symbolic execution is notoriously costly as it requires analyzing the AUT source code, and it is typically used on rather small command-line software or at the unit level [11]. Ganov et al. proposed a technique that symbolically executes only GUI event handlers [GKKP09], while Cheng et al. use a cloud-based architecture that parallelizes concolic-execution on multiple machines to foster scalability [CCYW16]. Coverage-based approaches aim to maximize code coverage. Thus, they can often cover significant portions of the code and discard non-meaningful interaction sequences that cover little code. However, coverage-based approaches are limited by the cost of analyzing the source code and often do not scale well to complex applications.

2.2.4 Machine Learning Approaches

Some *Machine Learning-based approaches* [BGZ18] [KSM$^+$18] [LYGC19] use an explicit, supervised training process to learn from previous test executions. They can reuse previous knowledge acquired on different apps or past versions of the application under test. Approaches such as QBE [KSM$^+$18] make the transfer of knowledge to new apps possible by abstracting the application state in a form that is supposed to hold across different domains and implementations. However, the effectiveness of such a transfer learning process depends on the similarity between new and old apps. One of the first works proposing RL for GUI testing is AutoBlackTest (ABT) [MPRS12]. ABT uses Q-Learning to learn how to generate test cases that traverse the GUI of a desktop application triggering the highest number of changes in said GUI. This approach is based on the simplest form of RL, Tabular Q-Learning, whose effectiveness strongly depends on the Q-Table's initial values. ABT generates test cases alternating between events selected at random (80% of the time) with events that maximize the GUI changes (20% of the time) to balance between exploration of the GUI and exploitation of the Q-Learning. One of the most recent approaches to testing based on Deep Learning is Q-Testing [PHW$^+$20]. However, it also

uses Tabular Q-Learning as a backbone. At the same time, learning is limited to the computation of the similarity between Android application states, which determines the reward of the Q-Learning algorithm. This thesis aims at overcoming the limitations related to machine learning techniques.

2.3 Security Testing

The automatic detection of vulnerabilities in mobile apps is still an open problem despite decades of research efforts in the field. For example, a recent study [atlay] reported that mobile apps released in Q1 2021 across 18 of the most popular categories have - on average - 39 vulnerabilities per Android application that can affect the security and privacy of the users. To cope with such issues, the industry and the research community released several security assessment methodologies and tools to detect vulnerabilities by exploiting Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) techniques or a combination of both techniques, resulting in a classification into the following subcategories: *Static Analysis Security Testing (SAST)*, *Dynamic Analysis Security Testing (DAST)*, and *Static and Dynamic Analysis Security Testing*.

2.3.1 Static Application Security Testing

SAST approaches detect security hazards inside the applications without the burden of executing and testing them in a controlled environment. To this aim, they are easy to integrate into a CI/CD pipeline since the scanning process can be launched as soon as a team member commits code to a source code repository. Moreover, SAST tools can examine the entire surface of an application in opposition to DAST techniques that can explore only parts of the application reached by proper inputs. Notable examples include [LL05], MobSF [Mob22], and DroidPatrol [TSQ+19]. Livshits et al. [LL05] propose an approach based on a scalable and precise points-to analysis to find security vulnerabilities in Java applications. IccTA [LBB+15] is a methodology that exploits data-flow analysis techniques to find data leaks related to inter-component communication of Android applications. Amandroid [WRO18] leverages static analysis techniques that perform inter-component and intra-component data-flow point-to-point analysis in Android apps. HybriDroid [LDR16] searches for WebView bugs in web-based applications by analyzing the call graphs of Java and Javascript code to detect errors related to the usage of the JavaScript-Interface. However, static analysis techniques can not verify that a specific vulnerability is a true positive. Thus a security expert must conduct a further manual investigation to determine whether a malicious user can exploit a particular vulnerability.

2.3.2 Dynamic Application Security Testing

Dynamic application security testing is a method in which testers examine an application while running but have no knowledge of its internal interactions or designs at the system level and no access or visibility into the source program. DAST looks at an application from the outside in, examines its running state, and observes its responses to simulated attacks made by a testing tool. An application's responses to these simulations help determine whether the application is vulnerable and could be susceptible to a real malicious attack. An example of DAST tool is AppScan [HCL22], which targets desktop and web applications. It automatically crawls the target application and tests for vulnerabilities. Test results are prioritized and presented to allow the operator to triage issues quickly and hone in on the most critical vulnerabilities. Buzzer [CGLX15] targets the Android system, fuzzing its system services by sending requests with malformed arguments to them. Stowaway [FCH+11] detects permission overprivileged dynamically in Android apps. Mutchler et al. [MDM+15] look for vulnerabilities in Android web apps. IntentDroid [HTP15] dynamically stimulates an Android app's Intent interface to find flaws. However, DAST techniques can explore only parts of the application reached by proper inputs because they do not leverage information that a previous static analysis can gather.

2.3.3 Static and Dynamic Security Testing

A variety of approaches rely upon the conjunction of static and dynamic analysis to detect vulnerabilities. Saner [BCF+08] combines static and dynamic analysis techniques to identify faulty sanitization procedures in web applications that attackers can bypass. AFL [Zal22] is a security-oriented tool that employs compile-time instrumentation and genetic algorithms to automatically discover test cases that trigger new internal states in C programs, improving the functional coverage for the fuzzed code. Kelinci [KLP17] applies AFL-style fuzzing to Java applications. ContentScope [Jia13] examines Android application Content Providers to identify instances when data from those components leaked or was contaminated. This happens when one application manipulates the Content Provider of another application without the necessary permissions or authorization. AppAudit [XGL+15] is primarily concerned with discovering privacy leakage vulnerabilities in Android apps. AppCaulk [STDF14] detects and stops data breaches through static and dynamic analysis and the ability to establish data leak policies. The DynaLog [AYS16] framework leverages existing open-source tools to extract high-level behaviors, API calls, and critical events that can be used to examine an application. He et al. [HYHW19] developed a tool that can first identify the third-party libraries inside apps, then extracts call chains of the privacy source and sink functions during its execution, and finally evaluates the risks of privacy leaks of the third-party libraries according to the privacy leakage paths. However, these techniques can ideally find vulnerabilities in the applications under test automatically but can not determine whether such vulnerabilities are exploitable.

2.4 Focused Testing

Focused testing aims at testing specific, individual components of a program. In the work by Alipour et al. [AGGC16], the authors define focused testing as a black-box method aiming to reach a specific target. A specific API can be an example of a target. Alipour et al. use a general test generator to reach that target by activating or deactivating different generator options.

Menéndez et al.[MJS+21] adopt a white-box approach and increase the granularity. The components addressed by focused testing are said to be *program points (pp)* (i.e., specific nodes in the control flow graph). Instead of using general-purpose generators, Menéndez et al. use a generator based on SMT solvers. Their tool, i.e., DFT, does not generate inputs for the real program but its symbolic path abstraction. This introduces some assumptions on the possibility of deriving and solving precise path conditions for the components targeted by focused testing. Moreover, the speed of test suite generation might be affected negatively when the path constraints grow in size and complexity. However, random generators struggle to cover specific program points because they lack guidance. Moreover, an SMT solver loses efficiency when dealing with an increasing number of program constraints, and it becomes inapplicable when constraints are too challenging to solve or formulate.

Chapter 3

Background on Reinforcement Learning

After a general overview on RL, this section presents Tabular RL and Deep RL in more detail.

3.1 Overview on Reinforcement Learning

The objective of Reinforcement Learning [SB18] is to train an *agent* that interacts with some environment to achieve a given goal. The agent is assumed to be capable of sensing the *current state* of the *environment*, and to receive a feedback signal, named *reward*, each time the agent takes an *action*.

At each time step *t*, the agent receives an observation x_t, takes an action a_t that causes the transition of the environment from state s_t to state s_{t+1}. A state s_t is a complete description of the state of the environment. An observation x_t partially represents the state, which may omit information. The agent also receives a scalar reward $R(x_t, a_t, x_{t+1})$, that quantifies the goodness of the last transition.

For simplicity, let us assume $x_t = s_t$ [Ach18]. The behavior of an agent is represented by a *policy* π, i.e., a rule for deciding on what action to take, based on the perceived state s_t. A policy can be:

- Deterministic: $a_t = \pi(s_t)$, i.e. a direct mapping between states and actions;
- Stochastic: $\pi(a_t|s_t)$, a probability distribution over actions, given their state.

DDPG [LHP+15] and TD3 [FvHM18] are examples of RL algorithms that learn a deterministic policy, while SAC [HZAL18] is a RL algorithm that learns a stochastic policy.

The standard mathematical formalism used to describe the agent environment is a *Markov Decision Process (MDP)*. An MDP is a 5-tuple, $\langle S, A, R, P, \rho_0 \rangle$, where :

- S is the set of all valid states,
- A is the set of all valid actions,
- $R : S \times A \to \mathbb{R}$ is the reward function, with $r_t = R(s_t, a_t, s_{t+1})$,
- $P : S \times A \to P(s)$ is the transition probability function, with $P(s_{t+1}|s_t, a_t)$ being the probability of transitioning into state s_{t+1} starting from state s_t and taking action a_t,
- $\rho_0(s)$ is the starting state distribution.

Markov Decision Processes obey the *Markov property*: a transition only depends on the most recent state and action (and not on states/actions that precede the most recent ones).

The goal in RL is to learn a policy π which maximizes the so-called *expected return*, which can be computed as:

$$G(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$$

A discount factor $\gamma \in (0, 1)$ is needed for convergence. When $t \to \infty$, if $\gamma = 1$, the expected return and the RL algorithm would not converge. The discount factor determines how much the agent cares about rewards in the distant future relative to those in the immediate future: if γ is small, the agent will tend to be myopic and only learn about actions that produce an immediate reward. If γ is close to 1, the agent will evaluate each of its actions based on its estimated future rewards. τ is a sequence of states and actions in the environment $\tau = (a_0, s_0, a_1, s_1...)$, named *trajectory* or *episode*. Testing an Android app can be seen as a task divided into finite-length episodes. Almost all reinforcement learning algorithms involve estimating *value functions*—functions of states (or *action-value functions*—functions of state-action pairs) [SB18] that estimate the goodness to perform a given action in a given state. The notion of goodness is defined in terms of expected future return. The future rewards the agent can receive depend on what actions it will take. The value function $V^\pi(s)$ is defined as the expected return starting in a state s and then acting according to a given policy π:

$$V^\pi(s) = E[G_t|s_0 = s]$$

The action-value function $Q^\pi(s, a)$ can be used to describe the expected return after taking an action a in state s and thereafter following the policy π:

$$Q^{\pi}(s,a) = E[G_t|s,a]$$

Correspondingly, we can define the *optimal value function*, $V^*(s)$, as the $V^{\pi}(s)$ that gives the highest expected return when starting in state s and acting according to the optimal policy in the environment. The *optimal action-value function*, $Q^*(s,a)$, gives the highest achievable expected return under the constraints that the process starts at state s, takes action a, and then acts according to the optimal policy in the environment.

A policy that chooses greedy actions only concerning Q^* is optimal, i.e., knowledge of Q^* alone is sufficient for finding the optimal policy. As a result, if we have Q^*, we can directly obtain the optimal action, $a^*(s)$, via $a^*(s) = \text{argmax}_a\, Q^*(s,a)$. The optimal value function $V^*(s)$ and action-value function $Q^*(s,a)$ can be computed by means of a recursive relationship known as the *optimal Bellman equations*:

$$\begin{aligned} V^*(s_t) &= \max_a E[r(s_t,a_t) + \gamma V^*(s_{t+1})] \\ Q^*(s_t,a_t) &= E[r(s_t,a_t) + \gamma \max_{a_{t+1}}[Q^*(s_{t+1},a_{t+1})]] \end{aligned}$$

The optimal Bellman equation constrains the value of a state under an optimal policy to be equal to the expected return for the best action from that state. The optimal Bellman equations are a system of equations for each state. If the MDP of the environment is known, then the system of equations can be solved analytically, finding the optimal value function (or the optimal action-value function) and, at last, the optimal policy. Otherwise, approximate solutions are found iteratively.

3.2 Tabular RL

Tabular techniques refer to RL algorithms where the state and action spaces are approximated using value functions defined using tables. In particular, *Q-Learning* [WD92] is one of the most adopted algorithms of this family. Q-Learning aims to learn a so-called *Q-Table*, i.e., a table containing the value of each state-action pair. A Q-Table represents the current estimate of the action-value function $Q(s,a)$. Every time an action a_t is taken and a state s_t is reached, the associated state-action value in the Q-Table is updated as follows:

$$Q(s_t,a_t) := Q(s_t,a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t))$$

where α is the learning rate (between 0 and 1). If the learning rate is close to 0, the agent exploits only prior knowledge, while when close to 1, the agent considers only the most up-to-date information. γ is the discount factor applied to the future reward. Typically, γ ranges from

0.8 to 0.99 [MKS+13] [LHP+15] [FvHM18], while $\max_a Q(s_{t+1}, a)$ gives the maximum value for future rewards across all actions. It is used to update the reward for the current state-action pair.

RL algorithms based on the Tabular approach do not scale to high-dimensional problems because it is difficult to manually define a good initial Q-Table in such cases. In case a good initial Q-Table is unavailable, convergence to the optimal table through the update rule described above is too slow [LHP+15].

3.3 Deep Reinforcement Learning

In large or unbounded discrete spaces, where representing all states and actions in a Q-Table is impractical, Tabular methods become highly unstable and incapable of learning a successful policy[MKS+13]. The rise of Deep Learning, relying on the powerful function approximation properties of Deep Neural Networks, has provided new tools to overcome these limitations. One of the first Deep Reinforcement Learning algorithms is DQN (Deep Q-Networks) [MKS+13].

DQN uses convolutional neural networks to approximate the computation of the action-value function Q^π. Training of such neural networks is achieved by using the so-called *experience replay* [Lin93]. With experience replay, the agent's experience e_t is stored within a buffer D named *replay buffer*. The experience e_t is defined as the tuple:

$$e_t = (s_t, a_t, r_t, s_{t+1}, d).$$

The tuple contains the state s_t of the environment, the action a_t taken in s_t, the reward r_t, and the next state of the environment s_{t+1}. The parameter d represents a binary value that indicates whether the transition has led the agent to a terminal state. The algorithm occasionally retrieves random experience samples from the replay buffer to train the neural network. If the network learns only from consecutive samples of experience as they occurred sequentially in the environment, the samples would be highly correlated and lead to inefficient learning [MKS+13]. Instead, taking random samples from the memory replay breaks this correlation.

While DQN can indeed solve problems with high-dimensional observation spaces, it can only handle discrete and low-dimensional action spaces. The recent advancements over DQN described in the following paragraphs (namely, DDPG, TD3, and SAC) overcome such limitations and allow dealing with high-dimensional action spaces.

Deep Deterministic Policy Gradient (DDPG)

DDPG [LHP+15] is an *Actor-Critic* algorithm, i.e., it includes two roles: the *Critic*, which estimates the value function, and the *Actor*, which updates the policy π in the direction suggested

by the Critic. It is based on a deterministic policy gradient [SLH+14] that can operate over continuous action spaces.

More specifically, the Critic aims to learn an approximation of the function $Q^*(s)$.

Suppose that the approximator is a neural network $Q_\phi(s, a)$, and that we have a set ***D*** of past experiences e_t $(s_t, a_t, r_t, s_{t+1}, d)$. The parameter list ϕ represents the coefficients of the neural network model, and the objective of the training phase is to optimize them, while set ***D*** is the previously seen *replay buffer*. The replay buffer should be large enough to avoid overfitting. The loss function of the neural approximator is the so-called *Mean-Squared Bellman Error (MSBE)*, which measures how close $Q_\theta(s, a)$ is to satisfying the Bellman equation:

$$L(\phi, D) = E[(Q_\phi(s_t, a_t) - (r_t + \gamma(1 - d) \max_{a_{t+1}} Q_\phi(s_{t+1}, a_{t+1})))^2]$$

The term subtracted from $Q_\phi(s_t, a_t)$ is named the *target*, because minimization of MSBE makes the Q-function as close as possible to this value. Since the target depends recursively on the same parameter ϕ to train, MSBE minimization can become unstable. The solution is to use a second neural network, called *target network*, whose parameters are updated to be close to ϕ, but with some time delay that gives stability to the process.

The Actor's goal is to learn a deterministic policy $\pi_\theta(s)$ which maximizes $Q_\phi(s, a)$. Because the action space is continuous and we assume the Q-function is differentiable concerning the action, we can use gradient ascent for the policy parameter. In case the action space is non-differentiable, DDPG degenerates, performing like DQN[LHP+15].

Twin Delayed DDPG (TD3)

Although DDPG can often achieve good performance, it tends to be susceptible to the critical tuning of its hyperparameters. In fact, Fujimoto et al.[FvHM18] demonstrated that in Actor-Critic algorithms, such as DDPG, the policy update introduces accumulation errors and introduces an overestimation bias of the values of the Q-function. The researchers state that the introduced errors may be minimal but raise two concerns: (1) the error may develop into a more significant bias over many updates if left unchecked, and (2) an inaccurate value estimate may lead to poor policy updates. TD3 [FvHM18] is an algorithm that addresses this issue by introducing three major changes, mostly on the Critic side: (1) *Clipped Double Q-Learning*: TD3 learns two Q-functions instead of one and uses the smaller of the two Q-values as the target in the MSBE function. Using the smaller Q-value for the target, and regressing towards that, helps mitigate overestimation in the Q-function. (2) *Delayed Policy Update*: TD3 updates the policy and the target networks less frequently than the Q-function. Delaying policy updates reduces per-update errors and further improves performance. (3) *Target Policy Smoothing*: TD3 adds noise to the

target action to make it harder for the policy to exploit Q-function errors by smoothing out Q across changes in the action.

Soft Actor Critic (SAC)

The central feature of SAC [HZAL18] is entropy regularization. Using the entropy-regularized method, an agent gets a bonus reward at each time step which is proportional to the entropy of the policy at that time step. In fact, differently from TD3, the policy of SAC is non-deterministic, and the inclusion of entropy in the reward aims to promote policies with a wider spread of alternatives from which to choose stochastically. The RL problem becomes the problem of finding the optimal policy π^* according to the following equation:

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, E[\sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t, s_{t+1}) + \alpha H(\pi(\cdot|s_t)))].$$

The first term of the equation $E[\sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t, s_{t+1})]$ comes from standard RL algorithms, in which the objective is to maximize the expected sum of rewards. The second term $\alpha H(\pi(\cdot|s_t))$ contains the entropy H, that is directly controlled by the entropy regularization coefficient $\alpha > 0$. This parameter explicitly controls the explore-exploit trade-off. With higher α the exploration is encouraged, while lower α corresponds to more exploitation.

Chapter 4

GUI Testing of Mobile Apps through Reinforcement Learning

In this chapter, I present ARES, a Deep RL approach for black-box testing of Android apps. Experimental results show that it achieves higher coverage and fault revelation than the baselines, including state-of-the-art tools. I also investigated the reasons behind such performance qualitatively, and I have identified the key features of Android apps that make Deep RL particularly effective on them to be the presence of chained and blocking activities. Moreover, I have developed FATE to fine-tune the hyperparameters of Deep RL algorithms on simulated apps since it is computationally expensive to fine-tune them on real apps. To sum up, this chapter provides the following advancements to state of art:

- ARES, the first publicly available testing approach based on Deep Reinforcement Learning, released as open-source;
- FATE, a simulation environment for fast experimentation of Android testing algorithms, also available as open-source;
- A thorough empirical evaluation of the proposed approach, whose replication package is publicly available to the research community.

4.1 ARES: Approach

This section describes ARES (Application of REinforcement learning to android Software testing), our approach to black-box Android GUI testing based on Deep RL. Figure 6.1 shows an overview of the approach. The RL *environment* is represented by the Android application under

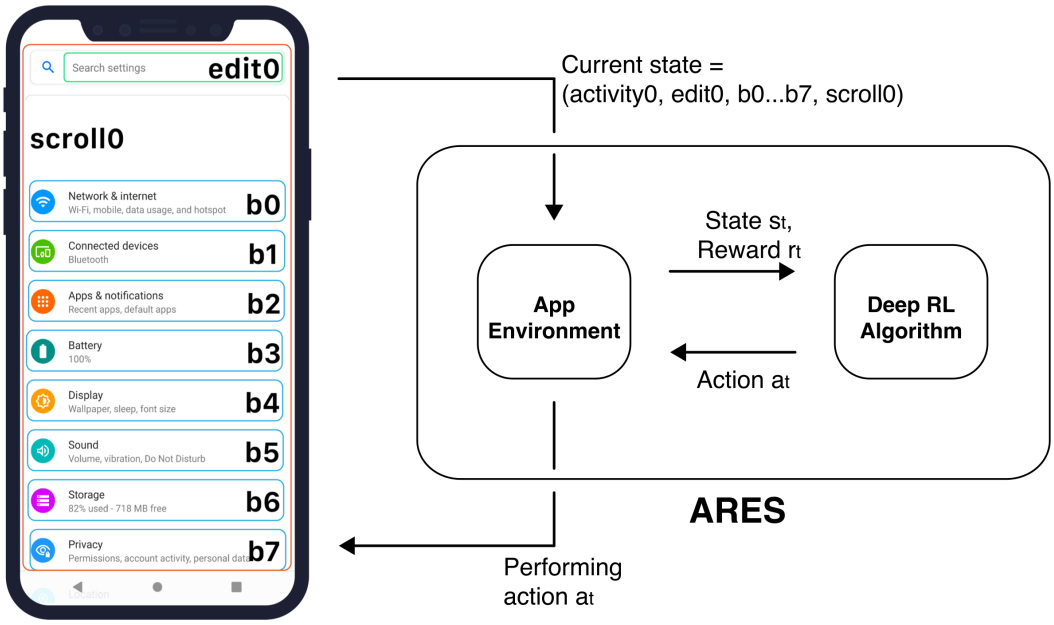

Figure 4.1: The ARES testing workflow. Information about widgets is extracted from the GUI and used to compute the state s_t and the reward r_t. b_i indicates the clickable buttons, $scroll_j$ indicates a portion of the GUI that we can scroll, and $edit_k$ an element that can be filled with text.

test (AUT), which is subject to several interaction steps. At each time step, assuming the GUI state is s_t, and the reward is r_t, ARES first takes action a_t. Then, it receives the new GUI state s_{t+1} of the AUT, and a reward r_{t+1} (not shown in Figure 6.1). Intuitively, if the new state s_{t+1} is similar to the prior state s_t, the reward is negative. Otherwise, the reward is a large positive value. In this way, ARES promotes the exploration of new states in the AUT, under the assumption that this is useful to test the application more thoroughly.

The reward is used to update the neural network, which learns how to guide the Deep RL algorithm to explore the AUT in depth. The update strategy depends on the Deep RL algorithm (DDPG, TD3, or SAC).

4.1.1 Problem Formulation

To apply RL, I have to map the problem of Android black-box testing to the standard mathematical formalization of RL: an MDP, defined by the 5-tuple, $\langle S, A, R, P, \rho_0 \rangle$. Moreover, I have to map the testing problem onto an RL task divided into several finite-length episodes.

State Representation

Our approach is black-box because it does not access the source code of the AUT. It only relies on the GUI of the AUT. The state $s_t \in S$ is defined as a combined state $(a_0, ...a_n, w_0, ...w_m)$. The first part of the state $a_0, ...a_n$ is a one-hot encoding of the current activity, i.e., a_i is equal to 1 only if the currently displayed activity is the i-th activity; it is equal to 0 for all the other activities. In the second part of the state vector, w_j is equal to 1 if the j-th widget is available in the current activity; otherwise, it is equal to 0.

Action Representation

User interaction events in the AUT are mapped to the action set A of the MDP. ARES infers executable events in the current state by analyzing the dumped widgets and their attributes (i.e., *clickable*, *long-clickable*, and *scrollable*). In addition to the widget-level actions I also use two system-level actions, namely *toggle internet connection* and *rotate screen*. These system-level actions are the only system actions that can be easily tested. In fact, since Android version 7.0, testing other system-level actions (like those implemented in Stoat [SMC$^+$17a]) would depend on the Android version used [SMC$^+$17b], [Goo20b], and would require a rooted device [Goo19a].

Moreover, certain apps, such as apps in the Finance and Entertainment categories (e.g., Netflix, SkyGo, Amazon Prime, and Lloyds), do not start on a rooted device due to the root checks that are on the apps [TM17] [Goo19b] [Mic20], compromising their testing.

Each action a is 3-dimensional: the first dimension represents the widget ARES is going to interact with or the identifier of system action. The second dimension specifies a string to be used as text input if needed. Actually, an index pointing to an entry in a dictionary of predefined strings is used for this dimension. The third dimension depends on the context: when the selected widget is both *clickable* and *long-clickable*, the third action determines which of the two actions to take. When ARES interacts with a *scrollable* object, the third dimension determines the scrolling direction.

Transition Probability Function

The transition function P determines which state the application can transit to after ARES has taken action. In our case, this is decided solely by the execution of the AUT: ARES observes the process passively, collecting the new state after the transition has occurred.

Reward Function

The RL algorithm used by ARES receives a reward $r_t \in R$ every time it executes an action a_t. I define the following reward function:

$$r_t = \begin{cases} \Gamma_1 & \text{if } act(s_{t+1}) \notin act(E_t) \text{ or } \textit{crash} \\ -\Gamma_2 & \text{if } pack(act(s_{t+1})) \neq pack(AUT) \\ -\Gamma_3 & \text{otherwise.} \end{cases} \tag{4.1}$$

with $\Gamma_1 \gg \Gamma_2 \gg \Gamma_3$ (in our implementation $\Gamma_1 = 1000$, $\Gamma_2 = 100$, $\Gamma_3 = 1$). To select the values of the reward, I conducted a preliminary experiment on a sub-sample of five apps (i.e., Antennapod, RedReader, Opentasks, Simple-Solitaire, and YalpStore). On these apps, I tested several types of rewards that are commonly used in literature: combination I ($\Gamma_1 = 1000$, $\Gamma_2 = 100$, $\Gamma_3 = 1$), combination II ($\Gamma_1 = 1$, $\Gamma_2 = 1$, $\Gamma_3 = 0$), and combination III ($\Gamma_1 = 100$, $\Gamma_2 = 10$, $\Gamma_3 = 1$). I found that combination I gives the best results.

The exploration of ARES is divided into *episodes*. At time t, the reward r_t is high (Γ_1) if ARES was able to transition to an activity never observed during the current episode E_t (i.e., the next activity $act(s_{t+1})$ does not belong to the set of activities encountered so far in E_t): if a new episode is started at $t + 1$, its set of activities is reset: $act(E_{t+1}) = \emptyset$.

Resetting the set of encountered activities at the beginning of each new episode is a technique that encourages ARES to visit and explore the highest number of activities in each episode to reinforce its explorative behaviors continuously. In contrast, if I provide the algorithm a significant reward only a few times (i.e., "sparsely"), the information to learn the optimal state-action combinations might be insufficient. The algorithm might fail to reproduce the sequence of actions leading to a high reward in the future. In that case, the performance of the algorithm results is poor. On the contrary, rewarding any activity change, regardless of its novelty, would encourage cycling behaviors [CA16].

The reward is high (Γ_1) also when a faulty behavior (*crash*) occurs. It is very low ($-\Gamma_2$) when the displayed activity does not belong to the AUT (i.e., the package of the current activity, $pack(act(s_{t+1}))$, is not the package of the AUT), as I aim to explore the current AUT only. In all other cases, the reward is moderately negative ($-\Gamma_3$), as the exploration remains inside the AUT, even if nothing new has been discovered.

4.2 ARES: Implementation

ARES features a custom environment based on the OpenAI Gym [BCP+16] interface, which is a de-facto standard in the RL field. OpenAI Gym is a toolkit for designing and comparing

RL algorithms and includes several built-in environments. It also includes guidelines for the definition of custom environments. Our custom environment interacts with the Android emulator [Goo20a] using the Appium Test Automator Framework [App20b].

4.2.1 Tool Overview

As soon as it is launched, ARES leverages Appium to dump the widget hierarchy of the GUI in the starting activity of the AUT. The widget hierarchy is analyzed by searching for clickable, long-clickable, and scrollable widgets. Afterward, these widgets are stored in a dictionary containing several associated attributes (e.g., resource-id, clickable, long-clickable, scrollable, etc.) and compose the *action vector*, i.e., the vector of executable actions in the current state. At each time step, ARES takes action according to the behavior of the exploration algorithm. Once the action has been fully processed, ARES extracts the new widget hierarchy from the current GUI and calculates its MD5 hash value. If it has the same MD5 value of the previous state, ARES leaves the action vector unchanged. If the MD5 value does not match, ARES updates the action vector. ARES performs the observation of the AUT state and returns the combined vector of activities and widgets. ARES organizes the testing of each app as a task divided into finite-length episodes. ARES aims to maximize the total reward received during each episode. Every episode lasts at least 250 time steps. Its duration is shorter only if the app crashes. I conducted a preliminary experiment on a sample app to select the ideal episode boundaries. I trained the same algorithm on this app by varying the episode length. Training characterized by short episodes results in poor performance due to the impossibility of exploring the application. Similarly, long episodes suffered from poor performance due to a low number of episodes completed. Once an episode comes to an end, the app is restarted, and ARES uses the acquired knowledge to explore the app more thoroughly in the next episode.

4.2.2 Application Environment

The application environment is responsible for handling the actions to interact with the AUT. Since the environment follows the guidelines of the Gym interface, it is structured as a class with two key functions. The first function `init(desired_capabilities)` is the initialization of the class. The additional parameter `desired_capabilities` consists of a dictionary containing the emulator setup and the application to be tested. The second function is the `step(a)` function, that takes an action `a` as command and returns a list of objects, including `observation` (current AUT state) and `reward`.

4.2.3 Algorithm Implementation

ARES is a modular framework that adopts a plugin architecture to integrate the RL algorithm to use. Hence, extension with a new exploration algorithm can be easily achieved. In the current implementation, ARES provides five different exploration strategies: (1) random, (2) Q-Learning, (3) DDPG, (4) SAC, (5) TD3. The random algorithm interacts with the AUT by randomly selecting an action from those in the action vector. Compared to Monkey [Goo20c], our random approach performs better since it selects only actions from the action vector. In fact, Monkey generates random, low-level events on the whole GUI, which could target no actual widget and then be discarded.

Our Q-Learning strategy implements the algorithm proposed by Watkins and Dayan [WD92]. The Deep RL algorithms available in ARES are DDPG, SAC, and TD3. Their implementation comes from the Python library *Stable Baselines* [HRE+18] and allows ARES to save the status of the neural network as a policy file at the end of the exploration. In this way, the policy can be loaded and reused on a new version of the AUT at a later stage, rather than restarting ARES from scratch each time a new AUT version is released. ARES is publicly available as open-source software at `https://github.com/H2SO4T/ARES`.

4.2.4 Compatibility

ARES has been successfully tested on Windows 10, macOS 11.1 (and older), Ubuntu 20 (and older), and Scientific Linux 7.5. ARES is fully compliant with parallel execution and enables parallel experiments to be performed on emulators or real devices, handling each instance completely separately. ARES is also compatible with several Android versions (i.e., it has been successfully tested on Android 6.0, 7.0, 7.1, 8.0, 8.1, 9.0, and 10.0). Moreover, since ARES is based on the standard OpenAIGym, new algorithms and exploration strategies can be easily added to the tool.

4.3 Fast Android Test Environment (FATE)

Deep RL algorithms require fine-tuning, which is expensive on real apps. Therefore, I developed FATE, a simulation environment for fast Android testing. FATE models only the navigation constraints of Android apps, so it can efficiently compare alternative testing algorithms and quickly tune their corresponding hyperparameters. After this algorithm selection and tuning phase through FATE is completed, the selected algorithms and their configurations are ported to ARES to test real apps.

4.3.1 FATE Design

In FATE, developers model an Android app by means of a deterministic Finite State Machine (FSM) $\mathcal{F} = (\Sigma, S, s_0, \delta, F)$, where Σ is a set of events, S a set of states with s_0 the initial state and F the set of final states, and δ the state transition function $\delta : S \times \Sigma \longrightarrow 2^S$. The *states* S of the FSM correspond to the activities of the app, while the *events* Σ trigger the transitions between activities, which in turn are modeled as a transition table δ. Events represent the clickable widgets (e.g., buttons) available in each activity. Transitions have access to a set of *global variables* and possess, among others, the following attributes: *ID*, *type*, *active* (boolean attribute), *guard* (boolean expression that prevents the transition from being taken if it evaluates to false), *set* (new values to be assigned to global variables), *destination* (target activity, i.e., value of δ). A prototype of a FATE model is shown in figure 4.2.

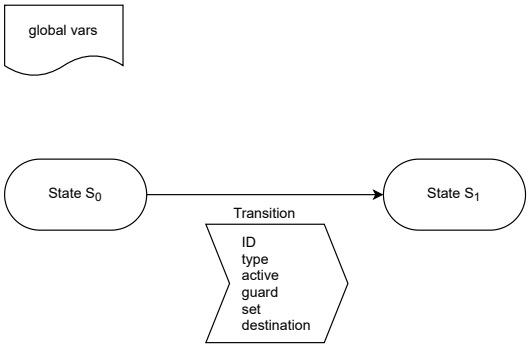

Figure 4.2: Prototype of a FATE model.

4.3.2 FATE Implementation

Figure 4.3 shows the FATE model of the prototypical app *Social Network*. To build such a model, developers can use Ptolemy [Lee09] to graphically draw an FSM that mimics the behavior of the application. While creating an FSM with Ptolemy is not mandatory in FATE, it simplifies the job of designing a logically correct model, thanks to the checks it performs. Then, FATE automatically translates the Ptolemy model (saved in XML format) into a JSON file that replicates the structure and behavior of the Ptolemy model. The JSON model translation has two main fields: `global_vars` and `nodes`. The first contains a list of global variables organized by `name` and `value`. The latter contains a list of all the activities. Each activity is characterized by a `node_id` and a list of corresponding node transitions, each including all the respective transition attributes.

The model of *Social Network* in Figure 4.3 contains a transition from `login` to `main_act` which is subjected to the guard `user_pass == real_pass`, i.e., the entered password must

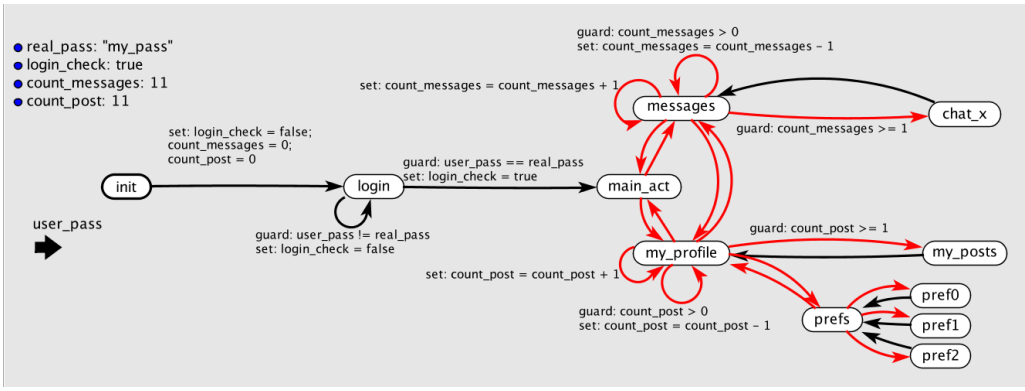

Figure 4.3: FATE model of *Social Network*: global variables are shown on the top-left; inputs on the bottom-left; red edges indicate non-deterministic transitions.

be correct in order for the transition to be taken. In the JSON model, the such transition is coded as:

```
"transition": {
    "transition_id": 0,
    "type": "button",
    "active": true,
    "guard": "user_pass == real_pass",
    "set": null,
    "destination": "main_act"
}
```

Another example of guarded transition is the one between `messages` and `chat_x`. The guard `count_messages >= 1` checks whether there exists at least one message from which a chat thread can be started. In the JSON model this is coded as:

```
"transition": {
    "transition_id": 0,
    "type": "button",
    "active": true,
    "guard": "count_messages >= 1",
    "set": null,
    "destination": "main_act"
}
```

In FATE, a Python environment compliant with the OpenAI Gym standard takes the JSON app model as input and tests it automatically using the selected algorithm. The available algorithms are: (1) Random, (2) Q-Learning, (3) DDPG, (4) SAC, (5) TD3. FATE was built with modularity in mind, and new exploration algorithms can be easily added to the tool. Compared to testing an Android app through Espresso or Appium, FATE makes test case execution significantly faster

because there is no need to interact with the app via its GUI. Moreover, the application navigation logic is simulated by the transition function δ, making it usually much faster to execute. Consequently, developers can run a large number of experiments, evaluate multiple algorithms, check various algorithm or application configurations, and find the optimal set of hyperparameters, all of which would be prohibitively expensive to execute on a standard Android testing platform. A limitation of FATE is that its effectiveness in hyperparameter tuning depends on the fidelity of the app models created by the developers. Despite not being the developer of the apps under test, I have been able to define models for them that turned out to be sufficiently faithful. In fact (see Section 6.1), the results obtained on the real apps with ARES and their FATE models are very close to each other. FATE is publicly available as open source software at `https://github.com/H2SO4T/ARES`.

4.3.3 Representative Family of Models

For fast evaluation of the Deep RL algorithms implemented in ARES, I modeled four Android apps using FATE. Each model represents the generalization of the apps belonging to a specific family, such as Shopping category. To obtain a set of app models representing the most common apps used in everyday life, I inspected AppBrain (a website that aggregates Google Play Store statistics and rankings) [app20a]. I selected four different and representative categories from the top ten: Music & Audio, Lifestyle, Business, and Shopping. From each category I then selected and modeled in FATE one prototypical app: *Player*, *Social Network*, *Bank* and *Market Place*. Each model is configurable with a variable degree of complexity.

The simplest scenario is *Player*. It features a wide number of activities arranged in a tree-like structure. It reflects the generalization of various applications to manage the settings and stream/add/remove media content, including apps or app components. *Social Network* (see Figure 4.3) starts by prompting a login activity with fields for username and password. Following the login activity, I have several activities that replicate a standard social network behavior, including a basic chat activity. The presence of inner password-protected operations characterizes the *Bank* model. *Market Place* models a typical app for e-commerce: the user can search for goods, login, purchase products, and monitor the orders. The four representative app models used in this work are publicly available inside the FATE tool.

4.4 Evaluation

I seek to address the following research questions, split between the following two studies:

Study 1 (FATE):

- **RQ1** *Are the results of synthetic apps comparable to those of their translated counterparts?*
- **RQ2** *Which Deep RL algorithm and which algorithm configuration performs better on the synthetic apps?*
- **RQ3** *How does activity coverage vary as the model of the AUT becomes increasingly difficult to explore?*
- **RQ4** *What are the features of the synthetic apps that allow Deep RL to perform better than Q-Learning?*

In Study 1, I want to understand if results obtained on synthetic app models run by FATE correlate with those obtained when executing the same apps (RQ1), once they are translated into real Android apps in Java code that ARES can execute. In particular, I translated three synthetic apps to Java/Android: Social Network, Bank, and Market Place. Due to its simplicity, I decided not to translate the "Player" model. I compare the rankings of the algorithms produced by FATE and ARES. Text inputs are chosen from a dictionary of 40 strings, which include credentials necessary to pass through the login activities. Since in this study I also use synthetic apps generated from models (i.e., Player, Social Network, Bank, and Market Place), coverage is measured at the granularity of Android activities. In fact, there is no source code implementing the business logic. Each run has a length of 4000-time steps, close to an hour of testing in a real Android test setting. With FATE, 4000 times steps are executed in approximately 200 seconds.

To answer RQ2, I take advantage of the fast execution granted by FATE to compare alternative RL algorithms on all synthetic apps and determine their optimal configuration (see Appendix 1). Text inputs are chosen from a dictionary of 20 strings, which include credentials necessary to pass through the login activities. To account for non-determinism, I executed each algorithm 60 times for each hyperparameter configuration of the algorithms and applied the Wilcoxon non-parametric statistical test to conclude the difference between algorithms and configurations, adopting the conventional p-value threshold at $\alpha = 0.05$. Since multiple pairwise comparisons are performed with overlapping data, the chance of rejecting true null hypotheses may increase (type I error). To control this problem, I adopt the Holm-Bonferroni correction [Hol79], which consists of using more strict significance levels when the number of pairwise tests increases.

To answer RQ3, I consider the best-performing configuration of the Deep RL algorithms, as selected from RQ2, and gradually increase the exploration complexity of the apps. Specifically, *20_strings, 40_strings, 80_strings* indicate an increasing size of the *string pool*. Such *string pool* is a dictionary of 20, 40, or 80 strings containing numbers and words, including the app's username and password, to use with a login activity. The string pool does not contain duplicates. *augmented_5* and *augmented_10* indicate an increasing size of the self navigation links (with 5 or 10 "dummy" buttons that do nothing) within the login activities.

I adopt the widely used metric AUC (Area Under the Curve) for the assessment, measuring the area below the activity coverage plot over time. To account for the non-determinism of

the algorithms, I repeated each experiment 30 times and applied the Wilcoxon non-parametric statistical test. In RQ4, I investigate qualitatively the cases where Deep RL is superior to Tabular Q-Learning.

Study 2 (ARES):

- **RQ5** *How do code coverage and time-dependent code coverage compare between Random, Q-Learning, DDPG, and SAC?*
- **RQ6** *What are the fault exposure capabilities of the alternative approaches?*
- **RQ7** *What features of the real apps make Deep RL perform better than Q-Learning?*
- **RQ8** *How does ARES compare with state-of-the-art tools in terms of coverage and bug detection?*

In Study 2, I use real apps and compare the alternative Deep RL algorithms between each other and with Random and Tabular Q-Learning. At last, I compare ARES to state-of-the-art testing tools.

To address RQ5-RQ6 and RQ7, I randomly selected 100 apps among the 500 most starred F-Droid apps available on GitHub, and 41 successfully compiled. I consider coverage at the source code level and compare both the final coverage and the coverage increase over time (RQ5). To obtain coverage data at the instruction level, I instrumented each app using JaCoCo [MRH20]. As in Study 1, I measured AUC concerning the code coverage and compared AUC values using the Wilcoxon statistical test with a significance level set to 0.05 (with correction). I exclude TD3 from the comparison since it performed consistently worse than the other RL algorithms on synthetic apps.

In addition to code coverage, I also report the number of failures (unique app crashes) triggered by each approach (RQ6). To measure the number of unique crashes observed, I parsed the output of Logcat and (1) removed all crashes that do not contain the package name of the app; (2) extracted the stack trace; (3) computed the hash code of the sanitized stack trace, to uniquely identify it. With RQ7, I qualitatively analyze the different performances of Deep RL vs. Q-Learning on real apps.

To address RQ8, I evaluate and compare ARES and state-of-the-art tools in terms of code coverage and the number of crashes using two different sets of apps under test, RQ8-a and RQ8-b, that accommodate the different requirements and constraints of the tools being compared. As state-of-the-art tools, I selected Monkey [Goo20c], Sapienz [MHJ16], TimeMachine [DBCR20b] and Q-Testing [PHW+20]. In RQ8-a, I compare ARES, Monkey, Sapienz, and TimeMachine on a set of 68 apps coming from AndroTest [CGO15c]. These apps are instrumented using Emma [emm06], the same coverage tool that is used in Sapienz and TimeMachine. In RQ8-b, I compare

ARES to Q-Testing on a set of ten apps instrumented using JaCoCo, the coverage tool supported by Q-Testing.

All experimental data were generated and processed automatically. Each experiment was conducted with a one-hour timeout and was repeated ten times for a total of 4560 hours ($\approx$ 190 days). The emulators involved in the study are equipped with 2 GB of RAM and Android 10.0 (API Level 29) or Android 4.4 only for the tool comparison.

4.4.1 Experimental Results: Study 1

Table 4.1 shows the ranking of the algorithms produced by ARES vs. FATE on the three apps translated from the synthetic FATE models to Java/Android. Below the ranking, Table 4.1 shows the AUC values obtained by the respective algorithms. The behaviors of the considered algorithms on synthetic (FATE) vs. translated (ARES) apps are very similar. The AUC values are quite close, and Spearman's correlation between AUC values across algorithms is 0.99 for Social, 0.89 for Bank, and 0.99 for Market; it is 0.95 overall. All correlations are statistically significant at level 0.05. ARES required 450 hours to complete the experiments. FATE required around 10 hours, reducing the computation time by a factor of 45 while producing similar results as ARES.

App	Tool	Ranking / AUC				
Social	ARES	Q-Learn	DDPG	Rand	SAC	TD3
		4: 7788	5: 6802	3: 9547	2: 9594	1: 15101
	FATE	Q-Learn	DDPG	Rand	SAC	TD3
		4:7737	5:7363	3:9291	2:10361	1:14451
Bank	ARES	Q-Learn	DDPG	Rand	SAC	TD3
		4: 8614	5: 7976	3: 9344	1: 12138	2: 10932
	FATE	Q-Learn	DDPG	Rand	SAC	TD3
		4: 7750	5: 6458	2: 9746	1: 16535	3: 9305
Market	ARES	Q-Learn	DDPG	Rand	SAC	TD3
		1: 16866	2: 16788	4: 15936	5: 15930	3: 15944
	FATE	Q-Learn	DDPG	Rand	SAC	TD3
		1: 16496	2: 16318	4: 15943	5: 15936	3: 15949

Table 4.1: Ranking of algorithms produced by ARES vs FATE; AUC values below ranked algorithms.

RQ1: The results obtained on synthetic apps are comparable to those obtained on their translated counterparts.

Figure 4.4 shows the coverage growth for the synthetic app Social. Each curve shows the mean of 60 runs. The shaded area around the mean represents the Standard Error of the Mean (*SEM* = $\sigma/\sqrt{n}$, where σ is the standard deviation and $n = 60$ the number of points). The highest activity coverage is obtained consistently by Deep RL algorithms, which have higher AUC values. Table 4.2 reports the AUC obtained on the synthetic apps in all tested configurations. Table 4.2 also

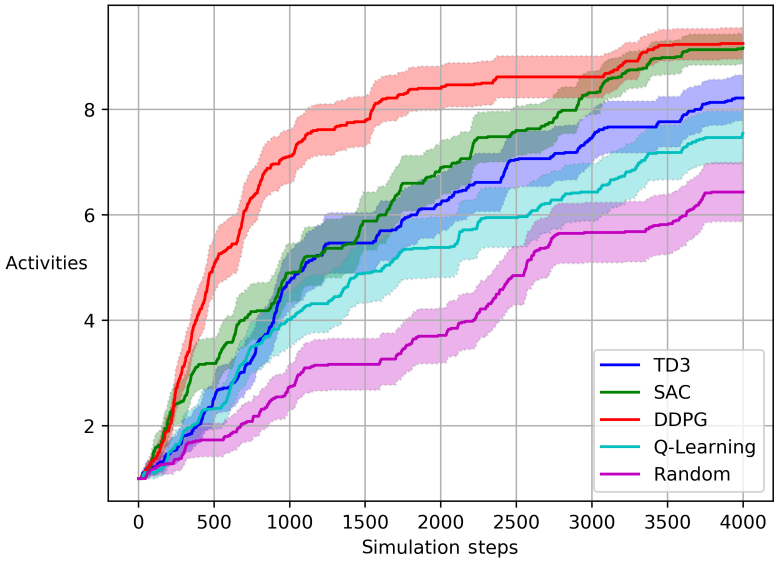

Figure 4.4: Activity coverage of the Social synthetic app in FATE.

shows the Vargha-Delaney effect size in the case of a statistically significant p-value $< \alpha/k$ where k is computed from the Holm-Bonferroni correction for multiple tests, between the winner algorithm (highest AUC) and the remainders.

Results show that Deep RL algorithms achieve higher coverage in most experiments. DDPG performs better in the simplest configuration, *20_strings*, while SAC performs better in almost all other configurations, including the most complex ones. Q-Learning prevails in only two scenarios belonging to *Market Place*, but the difference from the other algorithms is not statistically significant (p-value $> \alpha$).

> RQ2: Results lead us to select DDPG and SAC as best performing Deep RL algorithms to be involved in Study 2 experiments.

DDPG is selected due to its high performance in relatively simple scenarios; SAC because of its ability to adapt and maintain good performance in most scenarios.

> RQ3: While in simple situations (e.g., the Player app), all algorithms achieve a high level of coverage, when things get more complex (e.g., when the string pool increases), Deep RL algorithms retain higher coverage than all other algorithms.

I have manually inspected the step-by-step exploration performed by Q-Learning and by the

App	Config	Rand	Q-Learn	TD3	SAC	DDPG	Effect Size
Player	*20_str*	88719	89022	89903	89943	90337	-
Social	*20_str*	15840	20387	22809	25463	30008	L(Rand), M(Q)
	40_str	9291	7737	14451	10361	7363	S(DDPG)
	80_str	4535	5640	5730	7254	4774	-
	aug_5	13960	15400	13094	17402	13385	-
	aug_10	5291	3998	13737	11559	8870	M(Q, Rand)
Bank	*20_str*	22894	21622	29159	28016	36977	M(Q, Rand)
	40_str	9746	7750	9305	16535	6458	S(Q, DDPG)
	80_str	3998	4843	4776	5621	4798	-
	aug_5	12815	8634	8702	14914	11472	-
	aug_10	4121	6289	13289	14195	15361	M(Q, Rand)
Market	*20_str*	19236	18471	20980	23403	25923	-
	40_str	15943	16496	15949	15936	16318	-
	80_str	15944	15945	15935	15937	15932	-
	aug_5	18917	16377	16500	21208	16027	-
	aug_10	4121	6289	13289	14195	15361	-

Table 4.2: Mean AUC for synthetic apps: effect size between the winner (shaded cell) and other algorithms is reported only when p-value is statistically significant (S = Small; M = Medium; L = Large).

Deep RL algorithms. I found that login activities substantially complicate the exploration performed by Q-Learning. In fact, it is more difficult to reproduce the right username-password combination for a Tabular Q-Learning algorithm, which has limited adaptation capabilities. In contrast, Deep RL algorithms memorize the right combination in the DNN used to guide the exploration. In addition, large action spaces make it challenging for Q-Learning to learn an effective exploration strategy. The DNNs used by Deep RL algorithms can easily cope with large spaces of alternatives to choose from. The performance degradation of Q-Learning confirms this as the string pool increases in dimension or as new interactive elements ("dummy" buttons) are added, which confuse Q-Learning during its exploration.

RQ4: The performance of Q-Learning declines in the presence of blocking activities that require specific input combinations that must be learned from past interactions or when the input/action space becomes excessively large, while Deep RL can learn how to cope with such obstacles thanks to the DNN employed to learn the exploration strategy.

4.4.2 Experimental Results: Study 2

Table 4.3 shows coverage and crashes produced by each algorithm deployed in ARES. The highest average coverage and average number of crashes over ten runs are shaded in gray for each app.

I grouped the apps into three different size categories (Low-Medium, Medium, and High), depending on their ELOC (Executable Lines Of Code). Results show that the Deep RL algorithms arise more often as winners when the ELOC increase. Usually, larger size apps are more sophisticated and offer a richer set of user interactions, making their exploration more challenging for automated tools. I already know from Study 1 that when the action space or the observation space of the apps increase, Deep RL can infer the complex actions needed to explore such apps more easily than other algorithms. Study 2 confirms the same trend.

Overall, SAC achieves the best performance, with 42.61% instruction coverage and 0.3 faults detected on average. DDPG comes next, with 40.09% instruction coverage and 0.12 faults detected on average. To further investigate these results, I computed the AUCs reached by each algorithm, and I applied the Wilcoxon test to each pair of algorithms. Table 4.4 shows the AUCs achieved by the four algorithms and the Vargha-Delaney effect size between the winner and the other algorithms when the p-value is less than α. SAC results as the winner on 56% of the considered real apps, followed by Random (34%). Moreover, Table 4.4 confirms the trend observed in Table 4.3: as ELOC increase, a higher proportion of Deep RL algorithms produces the highest AUC. Figure 4.5 shows an example of code coverage over time for the app *Loyalty-card-locker*, averaged on 10 runs. SAC increases its coverage almost until the end of the exploration, while the other algorithms reach a plateau after around 35 minutes of exploration.

> RQ5: SAC reached the highest coverage in 24/41 apps, followed by DDPG (11 apps), Random (10 apps), and Q-Learning(1 app). SAC also has the highest AUC in 24/41 apps, followed by Random (13 apps), Q-Learning (11 apps), and DDPG (5 apps).

I suspect that the higher performance of SAC is related to its entropy regularization parameter. Thanks to the entropy regularization, in contrast to the other Deep RL algorithms that do not contain such parameters, SAC can maintain a high level of exploration even at the end of the testing phase, preventing the policy from converging to a bad local optimum.

Table 4.3 shows that SAC exposed the highest number of unique crashes (102), followed by Random (73), DDPG (52) and Q-Learning (43). The most common types of error exposed by SAC during testing are: *RuntimeException* (34 occurrences), *NullPointerException* (14), *IllegalArgumentException* (13). Figure 4.6 shows around a thirty percent overlap between the crashes found by SAC and the other algorithms. The overlap with Random is the highest. SAC discovered about 40% of unique crashes found by Random; however, SAC found many new crashes that Random did not find.

> RQ6: The SAC algorithm implemented in ARES generates the highest number of crashes, 102, in line with the results on coverage (RQ5), where SAC was also the best-performing algorithm.

I have manually inspected the coverage progress of the different algorithms on some of the real

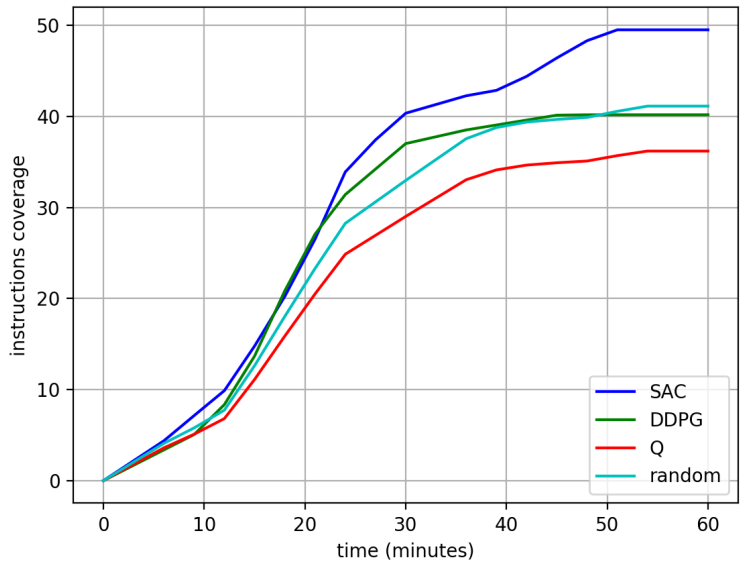

Figure 4.5: Instruction coverage over time for the app *Loyalty-card-locker*.

apps considered in Study 2. I have identified two structural patterns: concatenated activities (i.e., a sequence of nested activities possibly requiring some precondition to move from one to the next) and blocking activities (activities that require a specific input combination to enable the transition to the next activity). I observed that Deep RL algorithms achieve higher coverage than the other algorithms when it is necessary to replicate complex behaviors to: (1) overcome blocking activities, e.g., to create an item to be able to access its properties later or to successfully authenticate within the app; (2) to pass through concatenated activities without being distracted by already seen activities or ineffective buttons (high dimensional action/observation space); (3) reach an activity located deeply in the app. Such behaviors are possible thanks to the learning capabilities of the DNNs used by Deep RL algorithms, while they are hardly achieved by the other existing approaches, including Tabular Q-Learning.

RQ7: In the presence of blocking activities or complex concatenated activities (activities with a high number of widgets or located in depth) that require the capability to reuse knowledge acquired in previous explorations, the learning capabilities of Deep RL algorithms make them the most effective and efficient exploration strategies.

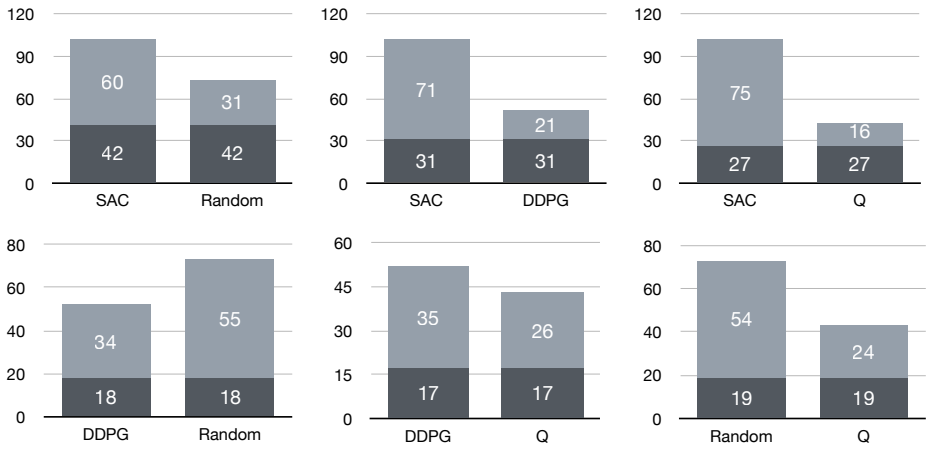

Figure 4.6: Comparison of the total number of unique crashes on the 41 apps involved in RQ5-6: dark gray areas indicate the proportion of crashes found by both techniques.

4.4.2.1 Comparison between ARES and state-of-the-art tools

RQ8-a. Table 4.5 shows the coverage reached and the faults exposed by each testing tool on 68 Android apps from AndroTest. Coverage data are summarized employing boxplots in Figure 4.7. The highest average coverage and average number of crashes over ten runs are highlighted with a gray background. ARES achieved 54.2% coverage and detected 0.48 crashes on average. TimeMachine achieved 50.4% code coverage and 0.42 faults on average. Sapienz reached a mean code coverage of 48.8% and discovered 0.22 faults on average. Monkey achieved 43.9% code coverage and discovered 0.11 faults. ARES achieved the highest code coverage on average, followed by TimeMachine, Sapienz, and Monkey. TimeMachine detected the most unique crashes (179), followed by ARES (171), Sapienz (103) and Monkey (51). Actually, ARES discovered less crashes than TimeMachine mostly because TimeMachine uses a system-level event generator, taken from Stoat [SMC+17a], which ARES does not support. However, system events present two major drawbacks: a) they vastly change depending on the Android version [SMC+17b] [Goo20b] (despite TimeMachine is compatible with Android 7.1, it uses only system-level actions compatible with Android 4.4 [DBCR20a]); and b) to execute them, root privileges are required. ARES does not require root privileges to work properly on any app (i.e., I recall that certain apps do not execute on rooted devices [TM17]). Analyzing the execution traces of each tool, I searched and identified the faults immediately after the generation of system events unrelated to the AUT. More than a third (63) of the crashes generated by TimeMachine come from system-level actions. Figure 4.8 shows a pairwise comparison of detected crashes among evaluated techniques. TimeMachine also finds only 20% of unique crashes found by ARES. For example, in the app *mnv*, only ARES generated a crash of the type *NullPointerException*, in which a missing control on input generates the failure of the conversion

function *CharSequence.toString()*. The text field from which the bug can be generated is not immediately available, but several interaction steps with the app are required. This shows that ARES can be used together with other state-of-the-art Android testing techniques (in particular, TimeMachine) to cover more code and discover more crashes jointly.

The good results of ARES in code coverage and exposed faults are due to the reinforcement mechanisms of the RL algorithms and the reward function that drives the testing tool through states of the app leading to hard-to-reach states. Search-based techniques such as Sapienz typically observe the program behavior over an event sequence that may be very long. Hence, the associated coverage feedback, used to drive search-based exploration, does not have enough fine-grained details to support good local choices of the actions to be taken. The fitness function used by these algorithms evaluates an entire action sequence and does not consider the individual steps in the sequence. TimeMachine improved this weakness by relying on the coverage feedback obtained at an individual state to determine which portion of the app is not still explored. The drawback of this kind of approach is a higher computational cost that requires a higher testing time. In fact, while the time to dump the GUI is in common both to TimeMachine and ARES, measuring the coverage as feedback at each timestep implies three steps:

- generation of the coverage files concerning the AUT,
- retrieval of coverage files from the Android device,
- coverage computation and processing.

These steps together take, on average, a second and a half to be completed. The strategy of ARES relies on monitoring the transition between activities taking on average 0.1 milliseconds rather than computing the code coverage at each time step. Hence, the latter approach offers a better trade-off between the granularity of the feedback and the computational cost required to obtain it.

RQ8-a: ARES achieved the highest code coverage in 41/68 apps, followed by TimeMachine (12/68), Sapienz (6/68), and Monkey (2/68). ARES triggered crashes more often than other tools in 23/68 apps, followed by TimeMachine (15/68), Sapienz (5/68), and Monkey (0/68). However, TimeMachine generated the highest number of unique crashes (179), but 63 of them came from system-level events. ARES generates 171 faults, Sapienz 103, and Monkey 51.

RQ8-b. Table 4.6 shows the coverage reached and the faults exposed by ARES and Q-Testing on 10 Android apps instrumented with JaCoCo. ARES achieved 64.3% coverage and exposed 0.41 faults on average; it detected 17 unique crashes. Q-Testing achieved 52.5% code coverage and 0.27 crashes on average, and it detected 16 unique crashes. ARES achieved the highest code coverage on almost all apps, and on average ARES covered 12% more code than Q-Testing.

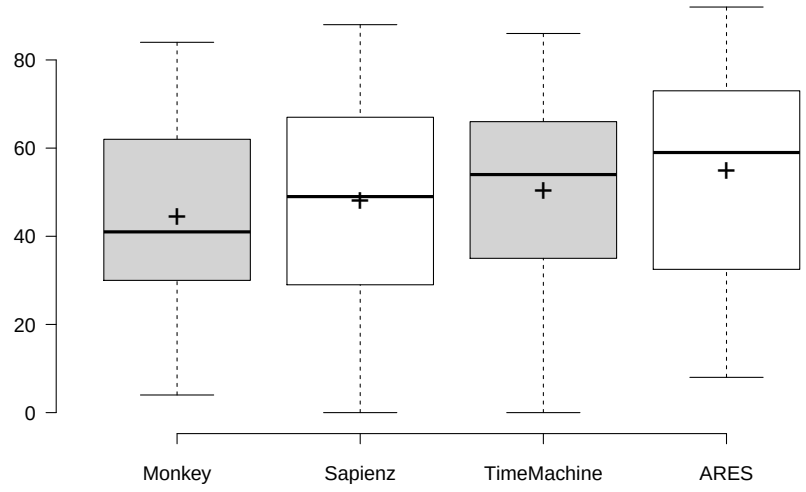

Figure 4.7: Code coverage achieved by ARES, TimeMachine, Sapienz, and Monkey.

Q-Testing generated six faults in common with ARES, while the other four faults are generated using the system-level events of Stoat. In the app *antennapod*, only ARES generated a *NumberFormatException* with a text field located deeply in the Settings submenu. In this comparison, the main advantage of ARES seems to be a better reward function that encourages the tool to visit the greatest number of activities within the same episode. Instead, Q-Testing determines the reward of the Q-Learning algorithm by computing the similarity between Android app states, which does not guarantee an efficient way to overcome blocking activities.

RQ8-b: ARES reached the highest code coverage on 9/10 apps, the highest average of crashes in 7/10 apps, and the highest number of unique crashes. Q-Testing, the state-of-the-art RL Android testing tool, reached the same level of exploration of ARES only in 1/10 apps and the highest average number of crashes on only 1/10 apps.

4.4.3 Threats to Validity

I adopted several strategies to enhance the internal validity of our results. I chose apps coming from a standard testing benchmark used in previous studies to mitigate risks of selection bias. I used default settings, given the same starting condition, and ran each tool several times under the same workload to ensure that no testing tool was at a disadvantage. I followed the Stoat protocol to identify unique crashes and manually checked the ones found. To measure coverage, I used JaCoCo, a standard coverage tool.

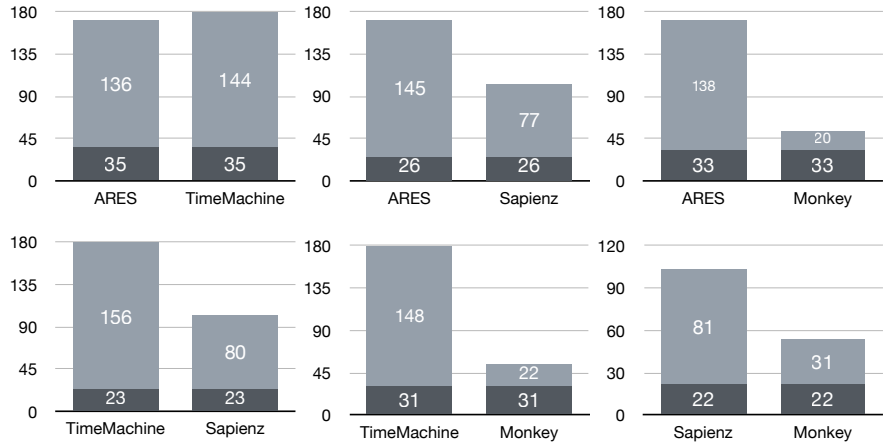

Figure 4.8: Comparison of total number of unique crashes involved in RQ8-a: dark gray areas indicate the proportion of crashes found by both testing tools.

4.5 Discussion

Using ARES as a testing tool involves several benefits:

- **black-box automated testing:** ARES relies only on the GUI of the AUT. This allows developers to test their production apps with no modifications. Available state-of-the-art tools, such as Sapienz and TimeMachine, rely on code coverage to drive exploration, and as recognized by the researchers who developed them, this makes testing less efficient.
- **wide compatibility:** ARES works on Windows, Linux, and MacOS. ARES can test apps in parallel on emulators or real devices with Android from 6.0 to 10.0.
- **policy reuse:** at the end of the testing phase, ARES saves the status of the neural network as a policy file. Instead of restarting ARES from scratch each time a new version of the AUT is launched, the policy can be loaded and reused on a later version of the AUT.
- **modularity:** within ARES, the app environment is decoupled from the RL algorithm used during the testing phase, allowing to deploy new algorithms easily.

Despite the advantages given by ARES there are also some limitations:

- **benefits on easy apps:** performance on simpler apps sometimes aligns with the performance of dummy methods such as random algorithms and does not justify using Deep RL.

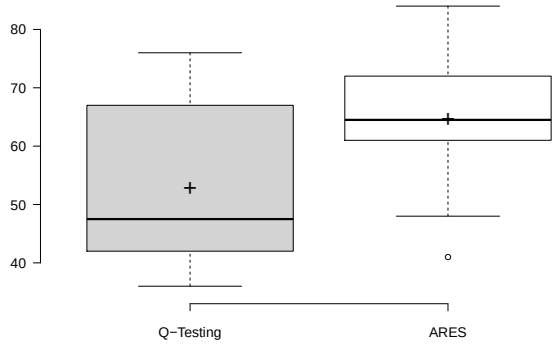

Figure 4.9: Code coverage achieved by ARES and Q-Testing.

- **system-level events:** ARES implements a limited set of events that act at the system level, including the most commonly supported ones, such as *toggle Internet connection* and *rotate screen*. The other system-level events require rooted devices to work, which brings some drawbacks, among which is the inability to work with apps that, for security reasons, perform a "root-check" and stop working if the device is rooted.

Applications	**ELOC**	**Rand**	**Q**	**SAC**	**DDPG**	**Rand**	**Q**	**SAC**	**DDPG**
		%Coverage(mean)				#Crashes(mean)			
Silent-ping-sms	263	41	41	41	41	0	0	0	0
Drawablenotepad	452	20	21	26	25	0.7	0.6	0.7	0.1
SmsMatrix	466	23	20	24	22	1.2	0	1.2	0.9
Busybox	540	75	73	74	76	0	0	0	0
WiFiKeyShare	627	37	36	37	37	0	0	0	0
Talalarmo	1094	69	71	71	71	0	0.5	0.5	0
AquaDroid	1157	55	55	55	55	1.0	0.4	0.8	0.3
Lexica	1215	72	72	74	75	0.3	0.1	1.5	1.2
Loyalty-card-locker	1228	41	37	50	41	0.5	0.4	0.8	0.1
Dns66	1264	58	58	58	58	0.1	0	0	0.2
Gpstest	1311	47	46	47	46	0	0	0	0
Memento	1336	77	76	74	77	0	0	0	0
Editor	1547	50	46	51	50	0	0	0	0
AndOTP	1560	20	25	27	20	0.5	0.5	0.7	0.2
BookyMcBookface	1595	26	25	25	24	0	0	0	0
Tuner	2207	80	74	79	75	0	0	0	0
WifiAnalyzer	2511	78	75	80	79	0	0	0	0
AdAway	3064	38	37	45	40	0	0	0.1	0.1
Gpslogger	3201	36	31	32	28	0	0	0	0.1
Connectbot	3904	26	25	28	18	0	0	0	0
Neurolab	3954	29	28	29	28	0	0.4	0.3	0.6
Anuto	4325	46	46	47	47	0	0	0	0
PassAndroid	4569	1	1	1	1	0	0	0	0
Markor	4607	51	43	53	41	0.3	0	0.4	0
Vanilla	4747	29	34	41	33	0	0	0	0
Average		45	43.84	46.76	44.32	0.15	0.12	0.28	0.15
Afwall	5130	12	12	16	13	0	0	0	0
OpenTracks	5260	45	42	44	45	0	0	0	0
Opentasks	5772	43	50	53	44	0	0	0.2	0
UserLAnd	5901	60	60	60	60	0.1	0.2	0.4	0.2
Simple-Solitaire	5907	10	30	31	31	0	0.4	0.4	0.2
Authorizer	5923	5	5	5	5	0	0	0	0
YalpStore	6734	35	34	38	33	0	0	0	0
CameraRoll	6836	32	31	31	32	0.8	0.1	1.6	0.1
AntennaPod	7975	46	40	48	38	0.5	0.1	0.8	0.4
Phonograph	8758	16	16	16	16	0	0	0	0
Average		30.4	30.5	34.2	31.7	0.14	0.08	0.34	0.09
MicroMathematics	10506	35	35	47	41	0	0	0	0
LightningBrowser	11961	35	36	43	37	0	0	0.4	0.1
Firefox-focus	12482	33	34	41	35	0.5	0.3	0.8	0.1
RedReader	12958	42	42	44	46	0	0	0.1	0
Wikipedia	23543	42	43	44	41	0	0	0	0
Slide	30483	19	17	18	19	0.8	0.3	1.2	0.3
Average		34.33	34.50	39.5	36.5	0.21	0.1	0.38	0.1
Total Average		39.62	39.58	42.61	40.09	0.17	0.1	0.3	0.12
Unique crashes						73	43	102	52

Table 4.3: Average coverage and the number of crashes observed on 41 real open-source apps in 10 runs of ARES.

App	Rand	Q	SAC	DDPG	Effect Size
Silent-ping-sms	0.36	0.36	0.38	0.37	S(DDPG),M(Rand),L(Q)
Drawable-notepad	0.13	0.14	0.20	0.18	L(Rand,Q)
SmsMatrix	0.13	0.13	0.11	0.11	-
Busybox	0.54	0.56	0.68	0.68	L(Q,Rand)
WiFiKeyShare	0.29	0.29	0.33	0.30	L(DDPG,Q,Rand)
Talalarmo	0.62	0.60	0.64	0.64	L(Q)
AquaDroid	0.529	0.526	0.531	0.522	L(Rand, Q, DDPG)
Lexica	0.63	0.61	0.66	0.65	L(Q,Rand)
Loyalty-card-locker	0.23	0.23	0.34	0.21	L(DDPG,Q,Rand)
Dns66	0.51	0.51	0.47	0.45	L(DDPG,SAC)
Gpstest	0.40	0.40	0.39	0.36	L(DDPG)
Memento	0.64	0.65	0.64	0.65	-
Editor	0.42	0.37	0.37	0.37	L(DDPG,Q,SAC)
AndOTP	0.16	0.18	0.23	0.15	M(Rand), L(DDPG)
BookyMcBookface	0.22	0.20	0.20	0.20	M(DDPG), L(SAC)
Tuner	0.68	0.66	0.60	0.60	L(DDPG,SAC)
WifiAnalyzer	0.56	0.56	0.67	0.58	L(DDPG,Q,Rand)
AdAway	0.25	0.25	0.27	0.25	-
Gpslogger	0.28	0.28	0.23	0.20	L(DDPG,SAC)
Connectbot	0.19	0.19	0.22	0.09	L(DDPG,Q,Rand)
Neurolab	0.23	0.23	0.22	0.22	-
Anuto	0.35	0.40	0.43	0.33	L(DDPG,Q,Rand)
PassAndroid	0.018	0.017	0.017	0.017	-
Markor	0.40	0.40	0.43	0.25	L(DDPG,Q,Rand)
Vanilla	0.17	0.23	0.26	0.23	L(Rand)
Afwall	0.09	0.09	0.13	0.10	L(DDPG,Q,Rand)
OpenTracks	0.37	0.35	0.35	0.35	-
Opentasks	0.18	0.46	0.46	0.29	L(DDPG,Rand)
UserLAnd	0.49	0.49	0.49	0.47	-
Simple-Solitaire	0.06	0.21	0.19	0.18	L(Rand)
Authorizer	0.05	0.046	0.047	0.049	S(SAC), M(Q)
YalpStore	0.28	0.28	0.31	0.26	L(DDPG,Q,Rand)
Camera-Roll	0.26	0.37	0.25	0.25	L(Rand,DDPG,SAC)
AntennaPod	0.33	0.33	0.35	0.22	L(DDPG)
Phonograph	0.085	0.077	0.075	0.076	S(Q,DDPG,SAC)
MicroMathematics	0.17	0.17	0.30	0.18	L(DDPG,Q,Rand)
Lightning-Browser	0.28	0.28	0.36	0.29	L(DDPG,Q,Rand)
Firefox-focus	0.27	0.28	0.43	0.35	L(Rand, Q)
RedReader	0.31	0.31	0.31	0.33	-
Wikipedia	0.30	0.30	0.33	0.28	-
Slide	0.13	0.13	0.12	0.14	-

Table 4.4: AUCs achieved on real apps; effect size between the winner and others when p-value $< \alpha$.

App	Coverage				Faults			
	Monkey	Sapienz	TM	ARES	Monkey	Sapienz	TM	ARES
a2dp	38	44	42	42	0	0.4	0.1	0.3
aagtl	16	18	17	19	0.3	1.0	1.7	1.5
aarddict	13	15	17	17	0	0	0.2	0.2
acal	18	27	27	28	0.5	0.8	1.0	1.0
addi	19	20	17	20	0.4	0.3	0.4	0.6
adsdroid	30	36	36	35	0.1	0.4	0.5	0.7
aGrep	45	-	59	56	0.1	-	0.3	0.2
aka	65	84	77	84	0.1	0.7	0.1	0.4
alarmclock	64	41	60	71	0.5	0.5	0.6	0.8
aLogCat	67	71	75	87	0	0	0	0
Amazed	36	66	63	89	0.1	0.3	0.2	0.2
AnyCut	63	65	63	73	0	0	0	0
anymemo	31	50	43	53	0.2	0.8	0.3	1.0
autoanswer	12	16	21	15	0	0	0.5	0.5
baterrydog	63	67	62	69	0	0.1	0.4	0.5
battery	73	78	77	92	0	0.4	0.5	0.4
bites	34	41	45	43	0.1	0.2	0.9	0.5
blokish	55	52	68	45	0	0.2	0	0.3
bomber	76	73	77	84	0	0	0	0
Book-Catalogue	27	29	27	25	0.1	0.2	0.8	0.5
CountdownTimer	74	62	77	84	0	0	0	0
dalvik-explorer	66	72	70	72	0.3	0.2	0.7	0.8
dialer2	39	42	42	44	0	0	0.3	0.4
DivideAndConquer	84	83	82	80	0	0.2	1.0	0.9
fileexplorer	41	49	55	64	0	0	0	0
frozenbubble	80	76	75	70	0	0	0	0
gestures	37	52	51	55	0	0	0	0
hndroid	7	15	18	18	0.1	0.4	1.1	1.3
hotdeath	75	75	72	74	0.1	0.2	0.8	0.9
importcontacts	40	39	40	42	0.1	0	0.6	0.8
jamendo	53	41	54	63	0	0.4	1.4	1.6
k9mail	6	7	8	8	0.4	0	1.8	1.2
LNM	47	-	-	75	0	-	-	0.2
lockpatterngenerator	75	79	74	78	0	0	0	0
LolcatBuilder	26	25	29	26	0	0	0.1	0
manpages	40	73	70	74	0	0.4	0.3	0.4
mileage	38	45	48	45	0.3	1.0	2.3	1.8
Mirrored	57	59	62	59	0.4	0	0.8	0.7
mnv	41	60	43	56	0.5	0.3	1.0	1.1
multismssender	34	59	61	73	0	0.2	0.3	0.4
MunchLife	67	72	71	88	0	0	0	0
MyExpenses	41	60	50	63	0	0	0.2	0.2
myLock	25	31	50	30	0	0	0.5	0.2
Nectroid	34	66	58	57	0	1.0	0.3	0.9
netcounter	43	70	58	69	0	0	0.4	0.5
PasswordMaker	53	58	55	59	0.3	0.8	0.6	0.9
passwordmanager	7	8	17	18	0	0	0	0
Photostream	30	34	35	29	0.1	0	0.4	0.8
QuickSettings	50	45	46	52	0	0.2	0	0.4
RandomMusicPlayer	53	58	58	63	0	0	0.6	0.8
Ringdroid	22	29	48	30	0	0.1	0.3	0.2
sanity	26	19	31	22	0.2	0.3	0.5	0.4
soundboard	42	32	59	61	0	0.6	0	0.4
SpriteMethodTest	58	80	73	88	0	0	0	0
SpriteText	60	60	57	60	0	0.4	0	0.5
swiftp	12	14	13	17	0	0.4	-	0.6
SyncMyPix	21	21	23	25	0	0	0	0
tippy	75	83	74	85	0	0.4	0.3	0.4
tomdroid	47	46	51	69	0	0.3	0	0.3
Translate	49	48	48	50	0	0	0	0
Triangle	-	-	-	-	-	-	-	-
weight-chart	63	67	66	71	0	0	0	0
whohasmystuff	61	68	66	81	0.1	0	0.9	1.0
wikipedia	31	32	33	35	0	0	0	0
Wordpress	4	5	7	8	0	0.5	1.5	1.0
worldclock	83	88	86	90	0	0	0.6	0.4
yahtzee	51	57	56	69	2	0.2	0.5	0.5
zooborns	30	16	35	37	0	0	0.1	0.5
Average	43.9	48.8	50.4	54.2	0.11	0.22	0.42	0.48
Sum					51	103	179	171

Table 4.5: Results on 68 open-source apps coming from AndroTest.

App	Coverage		Faults	
	ARES	**Q-Testing**	**ARES**	**Q-Testing**
Alogcat	84	76	0	0
Antennapod	48	42	0.8	0.4
AnyCut	72	67	0	0
batterydog	65	49	0.5	0.3
Jamendo	64	46	0.4	0.7
Multismssender	71	45	0.4	0
Myexpanses	63	36	0.4	0.2
talalarmo	74	74	0.8	0.5
Tomdroid	61	50	0.3	0.2
vanilla	41	40	0.5	0.4
Average	64.3	52.5	0.41	0.27
Sum			17	16

Table 4.6: Comparison between Q-Testing and ARES.

Chapter 5

Security Testing of Mobile Apps through Reinforcement Learning

In this chapter, I propose a different approach to exploit generation, which I call RONIN, based on Deep Reinforcement Learning. Deep Reinforcement Learning (Deep RL) is a machine learning technique that does not require a labeled training set as input since the learning process is guided by the positive or negative reward experienced during the tentative execution of a task. Hence, it can be used to dynamically learn how to build an Intent that exposes a specific vulnerability based on the feedback obtained during past successful or unsuccessful attempts. More specifically, RONIN manipulates the parameters of the Intents by applying a sequence of actions to them. Each action receives positive feedback if I move closer to the target statement (i.e., the vulnerable statement) upon execution of the Intent; neutral (zero) feedback if the minimum distance between the statements that I reached and the target statement does not change; negative feedback if I increase the distance from the target concerning the last Intent execution. RONIN uses a Deep Neural Network (DNN) to generate (initially random) actions during the training phase and observes their outcome (i.e., states and rewards). Then, RONIN leverages the collected information and iteratively trains the DNN to take a given action when in a given state. Our paper gives the following major contributions to the state of the art:

- The first Deep RL approach to Android security testing focused on ICC vulnerabilities. This approach applies to a wide range of Android apps by relying on feedback provided through dedicated instrumentation.
- RONIN, an open source tool, whose code is available at the url: `https://github.com/H2SO4T/RONIN`.
- An empirical study shows our approach's effectiveness compared with existing and baseline techniques.

5.1 Background

5.1.1 Android Background

The Android Software Development Kit (SDK) offers programmers a collection of communication components for building mobile applications. `Activities`, `Services`, `Broadcast Receivers`, and `Content Providers` are the four Android's pre-defined components. All such components (except dynamically registered Broadcast Receivers) are declared in the app's manifest file (`AndroidManifest.xml`).

An `Activity` is a GUI that an app displays to a user and that the user can interact with. A `Service` manages background tasks for an app. A `Content Provider` manages access to a central repository of data primarily intended to be used by other applications, allowing secure access. A `Broadcast Receiver` receives Intents that are broadcast by other apps or the Android framework (for example, informing the user that the battery is low). Intents can be exchanged between Activities, Services, and Broadcast Receivers. Activities might be made up of Fragments, each of which could be a user-viewable portion or a full screen. `Fragments` introduce modularity and reusability into the activity's UI by allowing developers to divide the UI into smaller, more manageable discrete pieces. Moreover, fragments support the dynamic composition of a GUI, allowing developers to add and remove fragments (and the layouts therein) dynamically and programmatically.

Android apps execute in a sandboxed environment to prevent malware from infecting the system and the hosted applications [EOM09]. The Android sandbox utilizes the isolation capabilities of the Linux kernel. Although sandboxing is an essential security feature, interoperability is negatively affected as a result. Apps need to be able to interact in a variety of ways. For instance, if the user points to the Google Play website, the browser app should be able to launch the Google Play app. To support interoperability, Android supplies high-level ICC mechanisms via the `Binder` class, implemented as a driver in the Linux kernel. ICC is achieved via `Messages` and `Intents`. `Intents` are messaging objects that contain both the payload and the target application component. Intents can either be implicit, which means that the target is not specified, or explicit, which means that a specific target is provided. Intents can be broadcast to Broadcast Receivers, invoke activities, or launch a Service. External parties can invoke an application component via an Intent if the manifest file allows that. The manifest also defines the permissions that the external party must possess.

According to the Android documentation, [Goo22], Intents contain actions, categories, and supplementary data that an app utilizes to decide how to carry out activities based on it. The attribute known as an Intent's action denotes the general action to be taken in response to an Intent (e.g., deliver data to some agent). The categories of Intent offer more details about how the app should carry out the Intent's action. A developer can declare categories in the application manifest, allowing the system to know if the application can handle a specific Intent category. For example,

by putting the CATEGORY_BROWSABLE category, an app specifies that a specific activity can be invoked through intent by a browser.

5.1.2 Vulnerabilities related to ICC Channels

We can refer to an Android component as *public* if 1) it is exported via Intent filters, either explicitly or implicitly; 2) it requires neither signed nor system permissions; or 3) unsanitized data originating from a public component flows into it. The presence of public components leaves a hole in the Android sandbox. They expose themselves to incoming data from malicious parties, which might lead to vulnerabilities if the data is not sanitized or validated. Fragments are also potentially vulnerable, as they can access incoming ICC data via their enclosing Activity and its initiating Intent. Malicious parties can be both local and remote. Malware is highly prevalent in Android and can interact directly with the public ICC interfaces through explicit Intents without special permissions [HTP15]. Unsafe handling of incoming ICC data can result in different forms of attack. Below I list three of the main threats that I aim to discover.

Cross-Application Scripting (XAS). Similarly to Cross-Site Scripting (XSS) in the Web landscape, XAS [HTP15] arises when script content (mostly JavaScript code) is injected into the HTML UI of a hybrid mobile application. Hybrid apps allow developers to write code based on platform-neutral web technologies and wrap them into a single native app that can render HTML/CSS content and execute JavaScript. This enables different forms of attack, including: 1) UI defacing/rewriting to trigger phishing attacks, 2) access to sensitive information, and 3) run native code via JavaScript. A concrete entry point for XAS attacks is the WebView class, which renders HTML content within a mobile app. The main method of WebView is `loadUrl`. If a malicious app can control the current URL, all the attacks above become potential threats. To exploit an XAS vulnerability, an attacker can inject JavaScript code using either the JavaScript URI scheme or the file scheme. The attacker creates a malicious HTML file and directs the target WebView object to load that file via an Intent.

Fragment injection. The static `instantiate(Context ctx, String fname, Bundle args)` method of class Fragment accepts as `fname` the name of the Fragment subclass to load reflectively. An attacker can leverage this to arbitrary load code obtainable through the class loader of `ctx`. A successful Fragment injection attack can result in loading an attacker-selected class into the context of the vulnerable app, which grants that class the same privileges and access rights as its host app. Otherwise, an exception is thrown, but before that, the class' static initializer and default constructor are executed, creating another attack vector. Another alternative is to load a Fragment already defined by the application or Android/Java framework but inject malicious initialization data into the Fragment. Fragments that are normally loaded by private Activities are more likely to trust rather than validate their initialization arguments, which renders them more exploitable to Fragment manipulation attacks.

Unhandled Exceptions (Denial of Service). Programming errors that trigger unchecked exceptions (like null dereference) will usually cause the target app to crash if the exception is missed. This presents an opportunity for Denial-of-Service (DoS) attacks and can generally drive the application into an unexpected state.

Running Example. The code in Listing 5.1 illustrates how incoming ICC messages are processed. An attacker can take advantage of an ICC-based vulnerability in an Android app by including actions, categories, or additional data with malicious payloads or by deleting these parameters. `getExtraString` retrieves the value of the custom string field of an Intent. The Activity contains two exploitable vulnerabilities that are reachable from the app's ICC interface. If the Activity receives an Intent whose `ExtraString s1` contains the string `URL` (line 20) the `WebView` of the Activity will load the string associated to `ExtraString s2` (line 22). This leads to the first XAS vulnerability. When `ExtraString s1` does not contain `URL`, the app performs a string comparison between `ExtraString s2` and a hardcoded string (line 27). If `ExtraString s2` is a null object, a `Null Pointer Exception` will be thrown, which results in the app crashing as the thrown exception is not caught. A malicious app can leverage this vulnerability to perform an *inter-process denial-of-service (IDOS)* attack on the MainActivity by periodically sending an Intent with no `ExtraString s2`. The function `getFragmentInstance` contains a Fragment injection (FI) vulnerability [HTP15], which occurs because within `getFragmentInstance` an incoming Intent with a string of extra data containing the key `fname` can be exploited by supplying as its value the name of a Fragment that resides in the corresponding app. This Fragment is then instantiated and loaded into the app (line 39).

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate (Bundle savedInstanceState) {
        // ...
    }

    @Override
    protected void onResume(){
        super.onResume();
        Button button = (Button) findViewById(R.id.get):
        WebView webView = (WebView) findViewById(R.id.webView1) :
        button.setonClickListener(new View.OnClickListener() {

        @override
        public void onclick(View v) {
            TextView textView = (TextView) findViewById(R.id.concat);
            Intent intent = getIntent():
            String a = intent.getStringExtra("s1");
            if ("URL".equals(a)) {
                /* If s2 contains a malicious site it will be loaded by the WebView*/
                webView.LoadUrl(intent.getstringExtra("s2"));
            } else {
                String b = intent.getStringExtra("s2");
                /* If String b is null the app will crash */
                if (b.equals ("www.test.com"))
                    webView.LoadUrl(intent.getStringExtra("s2"));
            }
```

```
        }
    });
}

// ...

public static void getFragmentInstance(Intent my_intent){
    /* If fname is not checked we can have a Fragment Injection */
    String fname = my_intent.getStringExtra("fname");
    return Fragment.instantiate(this, fname) ;
    }
}
```

Listing 5.1: RONIN: An example app that contains ICC vulnerabilities

5.2 Related Works

Several approaches have been developed to identify vulnerabilities in Android apps [SBGM16]. ComDroid [CFGW11] was one of the first significant works to target ICC-based vulnerabilities in detail, Epicc [OMJ+13] and IC3 [OLD+15] extracted information about Intents in a flow-sensitive manner. IccTA [LBB+15], and COVERT [BSGM15] identified vulnerabilities involving interaction between apps rather than only individual apps. FlowDroid [ARF+14] performs a static taint analysis to identify flows and privacy leakages from Android API sources to sinks. Amandroid [WRO18] is a static analysis approach based on Soot [Gro22] that performs inter-component and intra-component data flow point-to-point analysis. This methodology combines FlowDroid and IccTA approaches, resulting in more precise results concerning both. DroidPatrol [TSQ+19] identifies a list of potential vulnerabilities and proposes quick fixes. MobSf [Mob22] executes a plethora of security evaluations. However, none of these approaches can determine program paths and the set of Intents needed to execute them.

Another set of approaches relies solely on dynamic analysis to discover vulnerabilities. Buzzer [CGLX15] fuzzes Android system services to find flaws. Stowaway [FCH+11] detects permission overprivileged dynamically. Mutchler et al. [MDM+15] look for vulnerabilities in Android web apps. IntentDroid [HTP15] dynamically stimulates an app's Intent interface to find flaws. None of these strategies use static analysis, preventing them from finding many potential ICC-based program paths that may lead to a vulnerability.

A variety of approaches rely upon the conjunction of static and dynamic analysis to detect vulnerabilities. ContentScope [Jia13] examines Android app Content Providers to identify instances when data from those components leaked or was contaminated. This happens when one app manipulates the Content Provider of another app without the necessary permissions or authorization. To avoid privilege escalation threats, IPC Inspection [FWM+11] is an OS-based security mechanism that evaluates an app's privileges as it gets requests from other apps. AppAudit [XGL+15] is primarily concerned with discovering privacy leakage vulnerabilities. However, it only con-

ducts minimal Intent analysis (e.g., failing to account for various Intent attributes). AppCaulk [STDF14] detects and stops data breaches through static and dynamic analysis and the ability to establish data leak policies. The DynaLog [AYS16] framework leverages existing open-source tools to extract high-level behaviors, API calls, and critical events that can be used to examine an application. He et al. [HYHW19] developed a tool that can first identify the third-party libraries inside apps, then extracts call chains of the privacy source and sink functions during its execution, and finally evaluates the risks of privacy leaks of the third-party libraries according to the privacy leakage paths.

Methods such as [SAS+22] [CQX+20] also detect vulnerabilities by combining static and dynamic analysis. Chao et al. [CQX+20] propose an approach that uses a static analysis method to obtain some basic vulnerability analysis results for the application. Then, the application security vulnerability is verified using dynamic taint analysis and is reported to the user. Schindler et al. [SAS+22] combine free, open-source tools to support developers in checking that their application does not introduce security issues by using third-party libraries. None of these methods are thought to generate exploits. In [DC20, DCS20] Demissie et al. present an approach based on static analysis and automated test case generation to generate exploits that target the Permission Re-delegation vulnerabilities. To the best of our knowledge, Letterbomb [GHGM17] is the only tool that automatically generates exploits for IDOS, XAS, and FI vulnerabilities. Letterbomb relies on two phases. The first phase leverages combined path-sensitive symbolic execution-based static analysis. During the second phase, the tool tries to exploit the statically discovered vulnerabilities by generating an Intent and sending it to the analyzed app. However, Letterbomb only stimulates an app using the Intents, but Intent usage can also be triggered by other events coming from the GUI, which may result in missed vulnerabilities for Letterbomb. Moreover, Letterbomb becomes inapplicable when the constraints to reach a certain path within the app are too difficult to traverse. RONIN overcomes the limitations of Letterbomb by automatically triggering GUI events and by adopting a Deep RL algorithm to generate valid exploits. Our empirical evaluation showed the superiority of our new approach w.r.t. Letterbomb.

5.3 RONIN: Approach

This section describes **RONIN** (ReinfOrcement learning for security testiNg of INter-app communication), our approach to automatic generation of Inter-App Communication exploits. Figure 6.1 shows an overview of the approach. RONIN takes as input an app, and starts the *Vulnerability Identifier*. This analysis outputs all the statically identified vulnerable statements within the app. Moreover, the Vulnerability Identifier constructs a dictionary of the possible parameters that can be used as payloads in the Intents and the taint graph that leads to each identified vulnerability. According to the information coming from the taint graph, the *Oracle Instrumenter* injects code lines within the app to let RONIN's dynamic analysis verify its distance to the vulnerability during the exploitation of the app. The *Oracle Instrumenter* produces as output many instrumented

APKs, one for each identified vulnerability. At last, the *Dynamic Exploiter* leverages the information collected during the static phase and dynamically stimulates each instrumented APK with random GUI events and Intents crafted through Deep RL.

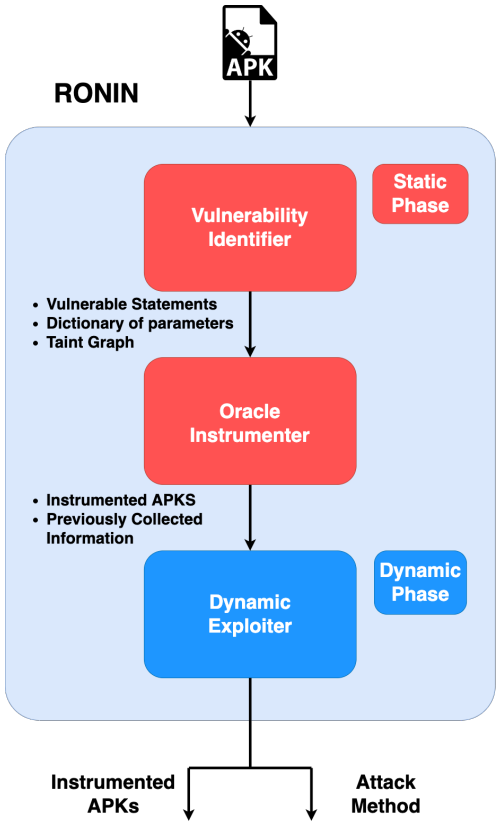

Figure 5.1: The RONIN workflow. At first, RONIN analyzes the APK using static analysis and collects potential vulnerabilities. Then RONIN instruments the APK to verify whether a vulnerability has been reached or not. At last, RONIN leverages Intent generation through DeepRL and GUI stimulation to effectively exploit a vulnerability.

5.3.1 Static Phase: Vulnerability Identifier

The *Vulnerability Identifier* analyzes an APK in search of ICC vulnerabilities. At first, it searches for a possible entry point. An entry point consists of an Intent function that can lead to an ICC vulnerability (an *Intent function* is any function that retrieves the parameters of Intents, such as `getStringExtra`). Once an entry point is found, Vulnerability Identifier taints the related Intent variable (e.g., variable `Intent` at line 18 in Listing 5.1). Subsequently, the Vulnerability Identifier analyzes the taint graph in search of possible improper usages of the tainted variables.

If so, RONIN produces: 1) a file that contains the vulnerable statements, 2) a dictionary of parameters related to the possible payloads of the Intent, and 3) the inter-components taint graph. To identify the vulnerable statements, I rely on the SEBASTiAn tool [PRC+22].

5.3.2 Static Phase: Oracle Instrumenter

To detect whether a generated Intent successfully exploits a vulnerability, RONIN instruments the app. The *Oracle Instrumenter* leverages the vulnerable statements and taint graphs previously produced by the *Vulnerability Identifier* to instrument the original APK. I describe how Oracle Instrumenter instruments the app for the three aforementioned vulnerability types. I only need to introduce the instrumentation once per identified vulnerability. After such one-time instrumentation, I can reuse it to detect multiple successful exploitations.

To instrument apps vulnerable to an IDOS attack, for each vulnerable statement RONIN adds a log instruction to record that the vulnerable statement has been executed. Additionally, RONIN instruments the statements can help the exploration during the dynamic phase, i.e., the statements that connect the Intent function to the vulnerability.

XAS instrumentation requires instrumenting each statement where a URL is loaded. Instrumentation must ascertain the URL loaded after the WebView's page has finished loading to verify whether the malicious URL injection was successful. To that end, RONIN logs the current HTML page loaded from the URL of a WebView once its page has finished loading.

For the FI instrumentation, RONIN injects logging statements to check for three conditions that together indicate a successful FI: (1) the target Activity has received the FI Intent, (2) the injected Fragment was instantiated successfully, and (3) the Activity is running without throwing any exception.

5.3.3 Dynamic Phase: Overview

This section describes the dynamic phase of RONIN, which leverages Deep RL to generate Intents for the app under test and uses a random algorithm to generate GUI events. Figure 5.2 shows the workflow of this phase. An app represents the RL environment under analysis, which is subject to several interaction steps. The objective is to successfully exploit the vulnerabilities discovered during the static phase. At each time step, RONIN observes the app state, computes the state s_t, and chooses an action a_t which modifies the current Intent. Then, it launches the Intent, and it optionally generates random GUI events. Subsequently, it iterates, receiving the new state s_{t+1} and the reward r_{t+1} (not shown in Figure 5.2).

Intuitively, if RONIN comes closer to the vulnerability, the reward is positive; it is neutral if the distance remains the same. Otherwise, the reward is negative if the distance from the vulnerabil-

ity increases.

The reward is used to update the neural network, which learns how to guide the Deep RL algorithm to generate Intents that exploit the app's vulnerabilities. The actual update strategy depends on the selected Deep RL algorithm.

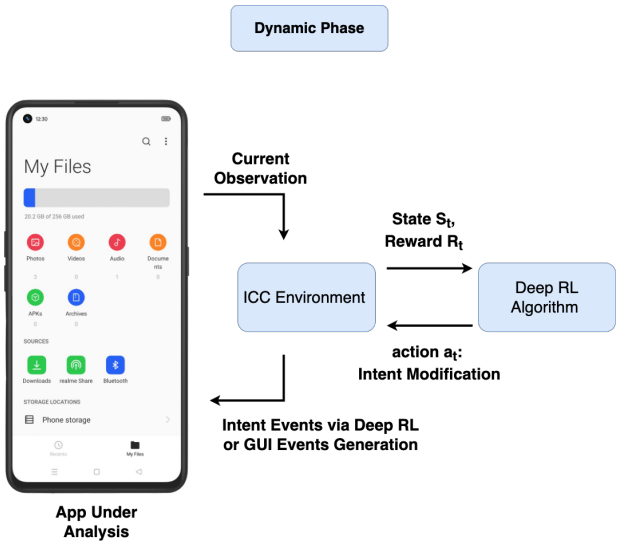

Figure 5.2: The dynamic phase workflow.

5.3.4 Dynamic Phase: Deep RL and GUI Events

The Dynamic Analysis of RONIN relies on the generation of Deep RL and random GUI events. Algorithm 1 represents the logic of the dynamic phase. The algorithm takes as input the instrumented APKs, the dictionary of the Intent parameters, and the taint graphs. The algorithm iterates on each vulnerability of each APK and tries to exploit it within a maximum time (10 minutes in our scenario) by mutating a default Intent and eventually generating a random GUI event. The GUI event is generated only if the Intent does not produce any log trace. At last, the algorithm computes the distance from the target vulnerability and the reward used to train the DeepRL algorithm. This process iterates until the timer expires, and the algorithm returns all the exploits generated during the execution.

5.3.5 Dynamic Phase: Deep RL

To apply RL, I have to map the problem of generating Android Intents to the standard mathematical formalization of RL: an MDP, defined by the 5-tuple, $\langle S, A, R, P, \rho_0 \rangle$.

Algorithm 1 : The Dynamic Phase

```
Input: Instrumented apks, Dictionary of Intent parameters, Taint Graphs
for apk ∈ Instrumented apks do
    for vulnerability ∈ apk do
        Start App Testing
        Intent = EmptyIntent
        while not Timer Expired do
            mutateIntent(Intent)
            Launch Intent
            LogTrace = CollectLogTrace()
            if LogTrace is empty then
                GenerateRandomGuiEvent()
                LogTrace = CollectLogTrace()
            end if
            Distance = DistanceFromVulnerability()
            Reward = ComputeReward(Distance)
            if Distance = 0 &
              LogTrace = Vulnerability-Exploited then
                Store Intent and Actions to Take
            end if
            trainDeepRLAlgorithm(Distance, Reward)
        end while
    end for
end for
```

State Representation. The state $s_t \in S$ is defined as a combined state $(a_0, ...a_n, node_0, ...node_m)$. The first part of the state $a_0, ...a_n$ is a one-hot encoding of the current activity, i.e., a_i is equal to 1 only if the currently displayed activity is the i-th activity; it is equal to 0 for all the other activities. The second part of the state vector, $node_0, ...node_m$ represents all the nodes of the paths that lead to a specific vulnerability. When a specific node is traversed by the last action generated by RONIN I set the corresponding node flag to 1; un-executed nodes have their flag set to 0.

Action Representation. Each time RONIN takes action, it manipulates a previously generated Intent. RONIN mutates an Intent by adding, removing, or modifying one of its parameters. Hence, an action $a = \langle a_0, a_1, a_2 \rangle$ is 3-dimensional: the first component a_0 specifies which type of action RONIN will apply to one of the Intent parameters. If zero, RONIN will remove a parameter from the Intent. Otherwise, if one is, it will add/modify the corresponding parameter. The second component a_1 encodes the index of the parameter to be manipulated. The third component a_2 specifies which payload is associated with the parameter selected by the previous action component. For example, RONIN can associate to a boolean parameter the payloads *True* or *False*. *Transition Probability Function.* The transition function P determines which state

the application can transit to after RONIN has taken action. This is decided solely by the app's execution: RONIN passively observes the process, collecting the new state after the transition.

Reward Function. The RL algorithm RONIN uses receives a reward $r_t \in R$ every time it executes an action a_t. I define the following reward function:

$$r_t = \begin{cases} \Gamma_1 & \text{if } dist\_from\_vuln()_t = 0 \\ \Gamma_2 & \text{if } dist\_from\_vuln()_t - dist\_from\_vuln()_{t-1} < 0 \\ -\Gamma_2 & \text{if } dist\_from\_vuln()_t - dist\_from\_vuln()_{t-1} > 0 \\ \Gamma_0 & \text{if } dist\_from\_vuln()_t - dist\_from\_vuln()_{t-1} = 0 \end{cases} \tag{5.1}$$

with $\Gamma_1 \gg \Gamma_2 \gg \Gamma_0$ (in our implementation $\Gamma_0 = 0$, $\Gamma_1 = 10$, $\Gamma_2 = 1$).

At the time t, the reward r_t is positive (Γ_1) if RONIN was able to trigger the selected vulnerability, i.e., the distance from the vulnerability is zero. When an action takes RONIN closer to the vulnerability without triggering it for the previous execution, the reward is slightly positive (Γ_2). The reward is negative ($-\Gamma_2$) when the action taken does not reach the target and decreases the distance from the vulnerability concerning the last execution. If the distance from the vulnerability remains the same as in the last execution, the reward is neutral ($\Gamma_0 = 0$).

5.4 Evaluation

I seek to address the following research questions:

- **RQ1** *To what extent can RONIN identify exploits for the three types of vulnerabilities described?* Exploit generation is the ultimate goal of RONIN. I evaluate RONIN's exploit generation capability by considering both exploits and unique exploits. I aim to evaluate the ability of RONIN to generate multiple different exploits.
- **RQ2** *How does RONIN compare with the state-of-the-art on a vulnerability benchmark?* I aim to evaluate the performance of RONIN in comparison with one baseline, Letterbomb. To the best of our knowledge, Letterbomb represents the state-of-the-art and produces a diversified set of exploits capable of triggering a vulnerability. I evaluate the two tools on a benchmark to understand their key differences.
- **RQ3** *How does RONIN compare with the state-of-the-art on apps obtained from the wild?* I aim to evaluate the performance of RONIN and Letterbomb in detecting the three considered vulnerabilities in the wild, considering a set of apps coming from the Google Play Store.

- **RQ4** *How does RONIN behave when the generation of GUI events is disabled?* I aim to evaluate RONIN's performance in detecting the three considered vulnerabilities when GUI event generation is disabled (ablation study).
- **RQ5** *How does RONIN behave when Deep RL is substituted with a random algorithm?* I aim to evaluate RONIN's performance in detecting the three considered vulnerabilities when it generates Intents using a random algorithm (ablation study).

5.4.1 Evaluation Design

To evaluate the proposed approach, I used the software subjects from the Ghera dataset [MR17] and the Google Play Store. Ghera contains benign apps with vulnerabilities related to Crypto, ICC, Networking, NonAPI, Permission, Storage, System, and Web APIs. From the Ghera dataset, I used three ICC apps that contain four vulnerabilities related to IDOS, XAS, and FI. I randomly selected 1500 apps from the Google Play Store among the 20k most downloaded apps. Such apps are the top free Android apps ranked by the number of installations according to Androidrank [21], and have been downloaded from the Google Play Store between Dec. 2021 and Jan. 2022.

5.4.2 Evaluation Procedure

With **RQ1**, I evaluated RONIN in terms of: 1) How many apps coming from Google Play Store are vulnerable; 2) How many exploits can RONIN generate; 3) how many of them are unique exploits.

In **RQ2**, I compare RONIN to Letterbomb on three apps from the Ghera dataset. These apps contain four vulnerabilities (i.e., 3 IDOS, 1 FI). The objective is to successfully detect and then exploit the vulnerabilities within the apps, obtaining a correct sequence of actions that fulfill the exploitation of the vulnerability. I investigate the reasons behind the failures and successes of both tools being compared.

In **RQ3**, I compare RONIN to Letterbomb on the number of generated exploits and unique exploits. Moreover, I evaluate whether the exploits generated by the two tools differ among them or belong mostly to the same set.

In **RQ4**, I disable GUI events generation in RONIN to evaluate their impact on the overall performance (number of exploits and number of unique exploits).

In **RQ5**, I randomly evaluate RONIN's performance (number of exploits and number of unique exploits) when generating Intents. I also compare the Deep RL and random algorithm on time necessary to generate the first exploit in each of the exploited vulnerabilities. To account for non-

determinism, I applied the Wilcoxon non-parametric statistical test to conclude the difference between Deep RL and the random algorithm, adopting the conventional p-value threshold at α = 0.05.

5.5 Experimental Results

5.5.1 RQ1: Exploit Generation

Table 5.1 (top) shows the results of RQ1, split by each of the three considered vulnerability types (i.e., IDOS, XAS, and FI). In Column 2, I report the number of Google Play Store apps for which RONIN statically detected a vulnerability, followed by the number of detected vulnerabilities. In Column 4, I report the number of apps for which RONIN successfully generated an exploit (Expl. Apps), followed by the number of successfully generated exploits and the number of *unique exploits*, where an exploit is unique if it either reaches a unique vulnerable statement or, in the case of FI, it successfully injects a unique Fragment.

RONIN successfully exploited 25 apps containing IDOS vulnerabilities, ten apps containing XAS vulnerabilities, and one with an FI vulnerability. RONIN obtained 46 unique exploits and 180 total exploits for IDOS, 10 unique and 18 total exploits for XAS, and two unique and six total exploits for FI. It should be noticed that a vulnerable statement may be exploited from more than one program path, resulting in multiple non-unique exploits for the same vulnerable statement. These results indicate that ***RONIN is capable of producing a sizeable number of exploits***.

RONIN					
Vuln. Type	**Apps**	**Vulnerabilities**	**Expl. Apps**	**Exploits**	**Unique Expl.**
IDOS	537	867	25	180	46
XAS	158	134	10	18	10
FI	28	31	1	6	2
Letterbomb					
IDOS	1232	1523	12	74	15
XAS	174	231	3	10	3
FI	84	119	0	0	0

Table 5.1: Detected vulnerabilities and generated exploits

Bench. App	**Vuln.**	**Letterbomb**		**RONIN**	
		Detected	**Exploited**	**Detected**	**Exploited**
FragmentInjec.	IDOS	✗	✗	✓	✓
	FI	✗	✗	✓	✓
UnhandledExc.	IDOS	✓	✗	✓	✓
UnprotectedBroad.	IDOS	✗	✗	✓	✓

Table 5.2: Comparison between RONIN and Letterbomb on Ghera

5.5.2 RQ2: Comparison with Letterbomb on Ghera

Table 5.2 shows the comparison between RONIN and Letterbomb on the apps from the Ghera dataset. RONIN detects the known ICC vulnerabilities in all the apps, while Letterbomb just only one. In most cases of false negatives, Letterbomb fails to detect the lack of null checks when executing backward data-flow analysis along the use-def chains. Listing 5.2 shows the IDOS vulnerability of app *UnprotectedBroadcastRecv-PrivEscalation-Lean* at line 8. This vulnerability can be exposed when the Intent provided to the *BroadcastReceiver* does not contain one of the extra strings at lines 5-6. RONIN successfully detects the IDOS, while Letterbomb can not identify the function `sendTextMessage` as a possible cause of a crash.

```
public class MyReceiver extends BroadcastReceiver {
	@Override
	public void onReceive (Context context, Intent intent) {
		if (intent.getAction() != null && intent.getAction().equals("edu.ksu.cs.benign.myrecv")){
			String number = intent.getStringExtra("number");
			String text = intent.getStringExtra("text");
			SmsManager smsManager = SmsManager.getDefault();
			smsManager. sendTextMessage(number, null, "Benign:␣" + text, null, null);
			Log.d("benign", "Message␣sent");
		}
	}
}
```

Listing 5.2: IDOS vulnerability occurring when either of the extra strings at lines 5-6 is not supplied

Let us consider the exploited vulnerabilities (Columns 4 and 6 in Table 5.2). RONIN successfully exploited all the statically detected vulnerabilities, while Letterbomb failed in exploiting the single IDOS it was able to detect statically. The latter failure happened because Letterbomb does not generate GUI events when trying to exploit a vulnerability. Listing 5.3 shows that the vulnerability at line 12 in the app *UnhandledException-DOS-Lean* is triggered when the button instantiated at line 3 is pressed. Once pressed, the button consumes the Intent (line 9), then extracts the extra values (lines 10-11), and at last calls the function `length` on both extra values. Suppose one of the extra values is unavailable; the app crashes. RONIN can exploit such a vulnerability, firstly sending the correct Intent and secondly clicking on the button that uses the payload coming from the Intent.

In summary, ***RONIN can detect and exploit all vulnerabilities in the Ghera benchmark, while Letterbomb can detect just one vulnerability but can not exploit it.***

```
1
2 public void onResume() {
3 	super.onResume();
4 	Button button = (Button) findViewById(R.id.get);
5 	button.setOnClickListener(new View.OnClickListener(){
6 		@Override
7 		public void onclick(View v) {
8 			TextView textView = (TextView);
9 			MainActivitv.this.findViewById(R.id.concat);
10 			Intent intent = MainActivity.this.getIntent();
11 			String a = intent.getStringExtra("s1");
12 			String b = intent.getstringExtra("s2") ;
13 			textView.setText("total␣length:" + Integer.toString(a.Length() + b.Length()));
14 		}
15 	});
16 }
```

Listing 5.3: IDOS vulnerability occurring when the button instantiated at line 3 is pressed

5.5.3 RQ3: Comparison with Letterbomb in the Wild

RONIN is the only approach that generates exploits in the wild for all the three types of vulnerabilities that both tools target. Table 5.1 (top vs bottom) shows the results of RONIN's vulnerability comparison with Letterbomb in the wild. For IDOS and XAS, RONIN generates three times more unique exploits than Letterbomb. Letterbomb can not generate any exploits for FI, while RONIN generates two exploits.

I also compared the two sets of vulnerabilities exploited by the two tools and found that 93% of the vulnerabilities exploited by Letterbomb are also triggered by RONIN. RONIN does not cover the remaining 7% of vulnerabilities because its static analysis does not detect them. Moreover, 12% of RONIN's vulnerabilities are related to GUI events that Letterbomb can not manage.

Listing 5.4 shows an example of a vulnerability that Letterbomb does not exploit. This vulnerability is similar to the one presented for RQ2. At line 12, I have an IDOS vulnerability contained within the function attached to a button. If RONIN sends an Intent that does not contain the extra parameter `emergency_number`, the if condition at line 12 will generate a crash, raising a *NullPointerException*. Letterbomb misses it because it does not generate GUI events to reach vulnerable code.

In summary, ***RONIN can generate three times more unique IDOS/XAS exploits than Letterbomb and can generate FI exploits that Letterbomb misses***.

```
call.setOnClickListener(new View.OnClickListener(){
        @override
        public void onClick(View view) {
        Intent intent = getIntent();
        String emergency_number = intent.getStringExtra("emergency_number");
        if (ContextCompat.checkSelfPermission(getcontext(),
                Manifest.permission.CALL_PHONE) != PackageManager.PERMISSION_GRANTED) {
                ActivityCompat.requestPermissions(getActivity(), new String[]{Manifest.permission.CALL_PHONE}, REQUEST_CALL);
        } else {
                if (emergency_number.length() = 10) {
                                String dial = "tel:" + emergency_number;
                                startActivity(new Intent(Intent.ACTION_CALL, Uri.parse(dial)));
                        }
                }
        }
});
```

Listing 5.4: IDOS vulnerability occurring when the button attached to the `onClick`

5.5.4 RQ4: Disabling GUI Events

Table 5.3 shows the difference in behaviors when I disable GUI events in RONIN. The overall number of unique exploits found by RONIN decreases by 7, 6 IDOS, and 1 XAS, respectively

(see Column 4 in Table 5.3). The number of exploited apps also decreases, specifically by five apps (for IDOS). These results confirm that ***adding GUI event generation is extremely useful for covering a broader range of vulnerabilities***.

RONIN Without GUI Events			
Vuln. Type	**Expl. Apps**	**Exploits**	**Unique Expl.**
IDOS	20 (-20%)	162 (-10%)	40 (-13%)
XAS	10	14 (-22%)	9 (-10%)
FI	1	6	2

Table 5.3: RONIN's reduced performance when GUI Events are disabled

5.5.5 RQ5: DeepRL vs Random

Table 5.4 shows the vulnerabilities exploited by RONIN when using random Intent generation instead of Deep RL. The two approaches perform similarly regarding the number of exploited apps and unique exploits. RONIN with the random method only misses two vulnerabilities and one app. We can appreciate the difference between the two methods when looking at the total number of generated exploits. Actually, RONIN with Deep RL generates 48 more exploits than RONIN with the random method.

Let us now consider the time required by either version of RONIN to generate an exploit. Figure 5.3 shows the comparison between Deep RL and Random on one of the analyzed apps. When the point lies on the $x - axis$ ($y = 0$), the action did not generate any exploit at the corresponding time step. When y equals 1, one of the two algorithms generates a valid exploit for the analyzed app. The Deep RL approach remains more consistent than Random in generating exploits after generating the first one, and it generates the first exploit earlier than random. The reason is that once the Deep RL algorithm generates the first exploit, it is encouraged to generate new exploits similar to the previous one, leveraging previous knowledge. On the contrary, the random method does not have any memory of past actions, resulting in poor performance compared to Deep RL.

Figure 5.4 shows the distribution of the number of time steps needed to generate the first exploit for each of the apps under analysis. The Deep RL approach employs fewer time steps to generate the first exploit, driven by the negative or the positive reward it receives. Instead, the random approach could not leverage any information collected during the dynamic phase, so the occurrence of the first exploit is unpredictable and is not consistent across runs. Moreover, the Wilcoxon non-parametric statistical test demonstrates that the difference between the algorithms is statistically significant (p-value $< \alpha$).

In summary, ***while RONIN with random Intent generation can still generate almost the same number of unique exploits as RONIN with Deep RL, the latter generates the first exploit much earlier than Random, and it then continues to generate valid exploits much more consistently than Random***.

RONIN Random			
Vuln. Type	**Expl. Apps**	**Exploits**	**Unique Expl.**
IDOS	24 (-4%)	137 (- 24%)	44 (-5%)
XAS	10	14 (-22%)	10
FI	1	5 (-17%)	2

Table 5.4: RONIN's reduced performance when random Intent generation is adopted and Deep RL is disabled

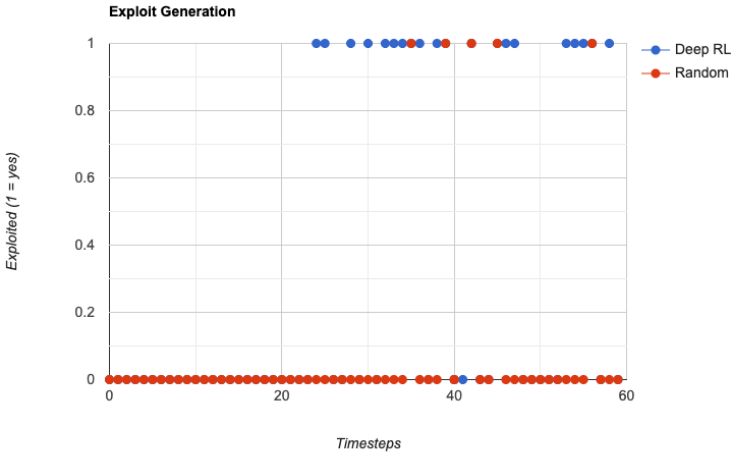

Figure 5.3: Time required by Deep RL vs Random to generate an exploit

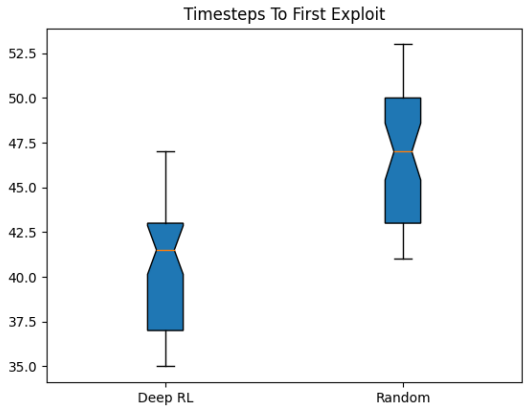

Figure 5.4: Distribution of the time steps required by Deep RL vs Random to generate the first exploit

5.6 Discussion

This chapter presents RONIN, an approach for generating exploits for Android ICC vulnerabilities through static analysis, Deep Reinforcement Learning-based dynamic analysis, and software instrumentation. RONIN, achieves better results than state-of-the-art and baseline tools, improving the number of exploited vulnerabilities. RONIN can generate three times more unique IDOS/XAS exploits than Letterbomb and can generate FI exploits that are missed by Letterbomb.

Chapter 6

Focused Testing through Reinforcement Learning

This chapter addresses the issue of developing a high-quality test suite to repeatedly cover a given point in a program, with the ultimate goal of exposing faults affecting the given program point. my approach, IFRIT, uses Deep Reinforcement Learning to generate diverse inputs while keeping a high level of reachability of the desired program point. IFRIT achieves better results than state-of-the-art and baseline tools, improving reachability, diversity, and fault detection. The key algorithmic advantage of IFRIT over DFT is that it requires only runtime coverage information from the subject under test. On the contrary, DFT requires that the subject under test can undergo static symbolic execution and that the generated path constraints, once relaxed by means of parametrization, can be solved by an SMT tool. The requirements of DFT limit its practical applicability to simple numeric functions only. On the contrary, IFRIT requires minimal runtime information on the coverage of the target statements, used to provide feedback to the RL agent during training. Hence, it is generally applicable to any, arbitrarily complex, software system. In terms of reachability and diversity, my empirical results show that IFRIT is equally effective as or more effective than DFT when executed on subjects to which DFT is applicable (i.e., the benchmark used in the original paper [MJS+21] that proposed DFT). Results show also that IFRIT is applicable to programs that cannot be handled by DFT. On them, IFRIT outperforms the only available baseline, which is random test input generation.

IFRIT gives the following major contributions to the state of the art:

- The first Deep RL approach to focused testing. By relying just on runtime coverage feedback, this approach is applicable to a wide range of programs.
- IFRIT, an open source tool, whose code is available at the URL: `https://github.com/H2SO4T/IFRIT`.

- An empirical study showing the effectiveness of my approach in comparison with existing and baseline techniques.

6.1 Approach

IFRIT (reInFoRcement learnIng for focused Testing) is an approach to focused testing based on Deep RL. Figure 6.1 shows an overview of IFRIT. The RL *environment* is represented by a program P under test, which is subject to several interaction steps. The objective is to generate inputs that reach a program point $pp \in P$ (see Section II). At each time step, IFRIT observes the code coverage measured on the program, computes the state s_t, the reward r_t, and chooses an action a_t used to generate a new input for the program. Then, it iterates, receiving the next code coverage s_{t+1} and reward r_{t+1} (not shown in Figure 6.1).

Intuitively, if the new state s_{t+1} reaches the target point in the program with new input, the reward is positive; it is neutral if such input is not new. Otherwise, if the target is not reached, the reward is negative.

The reward is used to update the neural network, which learns how to guide the Deep RL algorithm to generate useful inputs for the program. The actual update strategy depends on the selected Deep RL algorithm.

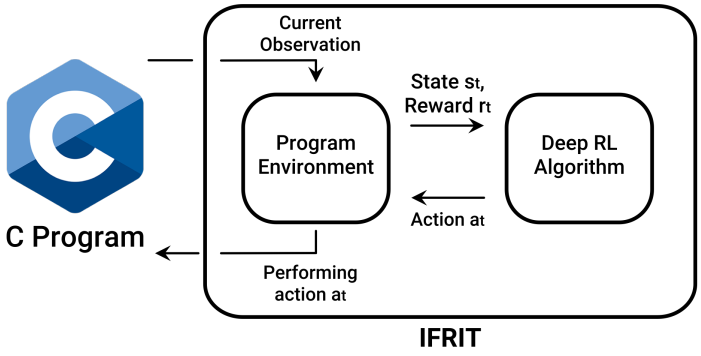

Figure 6.1: The IFRIT testing workflow. Code coverage is extracted (e.g., by gcovr, a utility for generating summarized code coverage results), from which IFRIT generates the state s_t and then determines the reward r_t for the chosen action a_t. By choosing an action a_t, IFRIT generates a new input that stimulates the program under test.

6.1.1 Instantiating RL for Focused Testing

To apply RL, I have to map the problem of focused testing to the standard mathematical formalization of RL: an MDP, defined by the 5-tuple, $\langle S, A, R, P, \rho_0 \rangle$. Moreover, I have to map the testing problem onto an RL task divided into several finite-length episodes.

State Representation

The state $s_t \in S$ is defined as a combined state $(b_0, ...b_n, i_0, ...i_m)$. The first part of the state $b_0, ...b_n$ represents the frequency of branch coverage during the last program execution, i.e., b_i is equal to $k > 0$ if the i-th branch of the program has been taken k times; it is equal to 0 if it was not traversed at all. The second part of the state vector, $i_0, ...i_m$ is equal to the last vector of input values generated by IFRIT. This means that the last execution of program P was performed by calling $P(i_0, ..., i_m)$, where $\langle i_0, ...i_m \rangle$ is called the *input vector*.

Action Representation

Each time IFRIT takes action, it manipulates a previously generated input (e.g., a number or a string) by using modifiers. Modifiers mutate an input value based on the type of such input. Hence, an action $a = \langle a_0, a_1, a_2 \rangle$ is 3-dimensional: the first component a_0 encodes the index of the input vector to be manipulated. In fact, in the general case a program accepts a vector of input values as input, and an action a will manipulate only one of them. The second component a_1 specifies the data to use to manipulate the input, depending on the context. The third component a_2 specifies how to use the second component on the input, i.e., what operation to apply using the second component as a parameter for such operation.

Numeric Input Manipulation. When managing numerical inputs, the starting input vector contains only zeros. The mutation of numeric input is done by using scale factors and operands. The first component (`i`) of the action indicates which portion of the input vector to modify (`input[i]`). The second component (`scale_factor`) indicates the scale factor, and the third specifies the operation (`<op>`) associated with that scale factor: `input[i] <op> scale_factor`, where `<op>` can be any arithmetic operator among +, -, *, /.

String Input Manipulation. When managing string inputs, the starting input is a vector of empty strings. IFRIT mutates the string by iteratively adding or removing characters to/from the input. The first component of the action indicates which portion of the input vector to modify. The second component indicates which char to use, and the third specifies the operation to perform (i.e., add char at the beginning, append char at the end, and remove char at the beginning/end). In the case of remove, the second component is not used.

Transition Probability Function

The transition function determines which state the program can go to after IFRIT has taken action. In our case, this is decided solely by the program's execution: IFRIT observes the process passively, collecting the new state after the program's execution has occurred.

Reward Function

The RL algorithm used by IFRIT receives a reward $r_t \in R$ every time it executes an action a_t. I define the following reward function:

$$r_t = \begin{cases} \Gamma_1 & \text{if } input(s_{t+1}) \notin inputs(E_j) \ \wedge \\ & x \in X_{pp} \\ \Gamma_0 & \text{if } input(s_{t+1}) \in inputs(E_j) \ \wedge \\ & x \in X_{pp} \\ -\Gamma_1 & x \notin X_{pp}. \end{cases} \tag{6.1}$$

with E_j the current episode and $\Gamma_1 > \Gamma_0 \geq 0$ (in our implementation $\Gamma_0 = 0$, $\Gamma_1 = 1$). This structure of the reward function and the chosen reward values are widely used in the literature in several different contexts [MKS+13] [LHP+15].

The exploration of IFRIT is divided into *episodes*. At time t, the reward r_t is positive (Γ_1) if IFRIT was able to reach the selected program point pp with an input never generated during the current episode E_j (i.e., the current input does not belong to the set of inputs generated so far in E_j): if a new episode E_{j+1} is started at $t + 1$, its set of inputs is reset: $inputs(E_{j+1}) = \emptyset$.

When an input that reaches the target has already been generated before in the same episode, the reward is zero (Γ_0), as it is no more useful during the current episode, but it remains a good input for the given task.

Resetting the set of generated inputs at the beginning of each new episode is a technique that encourages IFRIT to generate a high number of different inputs in each episode, which in turn makes the reward positive several times in the episode. In contrast, if I provide the algorithm a significant, positive reward only a few times (i.e., "sparsely"), e.g., because I have seen all new inputs already in previous episodes, the information to learn the optimal state-action combinations might be insufficient. The algorithm might fail to reproduce the sequence of actions leading to a high reward in the future, and the performance of the algorithm results be poor. On the contrary, another pattern to avoid is rewarding every successful input, regardless of its novelty, as this would encourage cycling behaviors [CA16]. Our definition of the reward function tries to balance the frequency of positive rewards and the avoidance of cycling behaviors by rewarding

positive inputs that are novel just in the current episode, not across all past episodes. The reward is negative ($-\Gamma_1$) when the input does not reach the target (i.e., the selected program point pp).

6.2 Implementation

IFRIT features a custom environment based on the OpenAI Gym [BCP+16] interface, which is a de-facto standard in the RL field. OpenAI Gym is a toolkit for designing and comparing RL algorithms and includes several built-in environments. It also contains guidelines for the definition of custom environments. Our custom environment interacts with a C program.

6.2.1 Tool Overview

As soon as it is launched, IFRIT automatically generates a configuration file that contains information about the C program under test. The configuration file includes several data useful for compilation and the execution of the program (e.g., how many parameters the program takes as input, the type of each input, the target file, the target program point pp). Afterward, IFRIT instantiates a custom environment compatible with the C program and starts the testing phase. At each time step, IFRIT takes an action (i.e., modifies the input vector) according to the behavior of the exploration algorithm. Once the action has been fully processed, which includes the execution of the C program, IFRIT elaborates the code coverage information, from which IFRIT computes the observation and the reward for the algorithm.

IFRIT organizes the whole testing session into finite-length episodes. The goal of IFRIT is to maximize the total reward received during each episode. Every episode lasts 24000-time steps. To select the ideal episode boundaries, I conducted a preliminary experiment on a subset of programs coming from CodeFlaws (CF)[TYM+17]. On this subset, I trained the same Deep RL algorithm by varying the episode length. Training characterized by short episodes results in poor performance due to the impossibility of exploring the input space. Similarly, long episodes took too much training time. Once an episode comes to an end, IFRIT resets the input vector to the default value, and then it uses the acquired knowledge to reach the target of the C program in the next episode. Figure 6.2 shows an example of numeric input manipulation. IFRIT generates an action that contains the rules to modify the previous input vector (a). I select the parameter of (a) indicated by the action and add to it the third scale factor (i.e., three). The new input vector (b) is now used to stimulate the program under test (not shown in the figure).

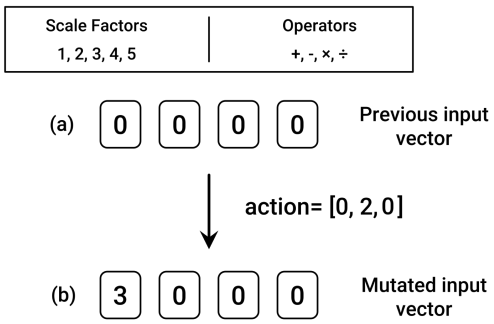

Figure 6.2: The IFRIT action vector $[0, 2, 1]$ indicates that the input element at index 0 should be modified by applying the operator at index 1 (i.e., '+') with the scale factor at index 2 (i.e., '3'). The new input element at index 0 becomes thus equal to 3 (i.e., 0+3).

6.2.2 Program Environment

The application environment is responsible for handling the actions to interact with the program. Since the environment follows the guidelines of the Gym interface, it is structured as a class with two key functions. The first function `init(configuration_file)` is the initialization of the class. The additional parameter `configuration_file` consists of a dictionary containing the program tester setup and the program to be tested. The second function is the `step(a)` function, that takes an action `a` as input and returns a list of objects, including `observation` (code coverage state) and `reward`.

6.2.3 Algorithm Implementation

IFRIT exploits *Stable Baselines* [HRE+18], a modular library that adopts a plugin architecture to integrate the RL algorithm to use. In the current implementation, IFRIT provides two exploration strategies: (1) random, and (2) PPO. The random algorithm interacts with the program by randomly selecting a mutation to perform on the input vector. Currently, one Deep RL algorithm is available in IFRIT: PPO. Its implementation comes from the Python library Stable Baselines. IFRIT is publicly available as open-source software at the URL: `https://github.com/H2SO4T/IFRIT`.

6.3 Evaluation

I seek to address the following research questions:

- **RQ0** *Which scale factors provide the best configuration of IFRIT?* Selecting proper scale factors is essential to maximize the performance in terms of reachability and uniformity of IFRIT. I conducted a preliminary experiment to determine the optimal scale factors to use.
- **RQ1** *What proportion of inputs generated by IFRIT reaches the target point of the program under test and how uniform are such inputs?* To evaluate the usefulness of IFRIT, I analyzed two key aspects: uniformity and reachability.
- **RQ2** *How do the test suites generated by IFRIT compare to those generated by the state-of-the-art tool DFT and random generators in terms of uniformity and reachability? What is the time required to produce them?* I aim to evaluate the reachability and diversity of IFRIT in comparison with three baselines, namely, a pseudo-random generator (Random), Random Mutations, and DFT. To the best of our knowledge, DFT is the only test input generator with focused testing, producing a diversified set of test inputs capable of reaching the target. Random performs random uniform sampling of the input domain. It has the highest possible (100%) uniformity level by construction, but it might produce only a few or no inputs that reach the target. Random Mutations perform a uniform selection of the input mutation to apply to a previously generated input. This means that it uses the same mutation operators as IFRIT but applies them randomly instead of learning how to apply them by means of RL.
- **RQ3** *What is the quality of the test suites generated by IFRIT concerning the state-of-the-art, in terms of mutation killing capability and fault detection?* Fault detection is the ultimate goal of focused testing. I evaluate IFRIT's fault detection by considering both mutants and real faults. Mutation testing introduces small syntactic alterations to the program (i.e., mutants). It measures the ability of a test suite to detect the errors caused by these alterations, i.e., the ability to kill the mutants. I aim to evaluate the ability of IFRIT to detect these artificial faults, in comparison with three baselines, Random, Random Mutations, and DFT. I also evaluate fault detection on real faults.

6.3.1 Evaluation Design

To evaluate the proposed approach, I used the software subjects coming from two open-source software repositories: the Software-artifact Infrastructure Repository (SIR) [GRD06], and CodeFlaws (CF) [TYM+17]. SIR contains C programs that accept numeric or string inputs. From the SIR repository, I used `tcas` as a numeric program. It is used to avoid collisions in aircraft systems. The `tcas` the program has 135 LoC and 40 branch statements. As programs that accept string input, I used `printokens`, `printokens2`, `flex`, `gzip`, `grep`. The programs have 570 to 10459 LoC and 45 to 1065 branch statements. The CF repository includes 7,436 C programs in total. Each of them has, on average, 50 LoC. Among these, around 3900 programs have both a buggy and a fixed version. I randomly selected 100 pairs of buggy/fixed programs

for our experiment. These programs have an average of 10 branch statements. To identify the program points pp on which I should focus the test generation process, I applied a tree edit distance algorithm to each program's buggy and fixed versions. Finally, I evaluated the performance of IFRIT, by computing reachability and uniformity.

6.3.1.1 Tree Alignment

The target point in our experiments is the one that contains the fault or the mutation. When this error is fixed by adding new lines or deleting existing lines of code, the identification of the target program is rather complex. Using Zhang and Shasha's algorithm [ZS89], the alignment process converts the programs into their abstract syntax trees and calculates the tree edit distance between them. This algorithm provides the shortest sequence of edit commands at the node granularity that converts one tree into another. Each node is labeled with the edit operation corresponding to it, which can be *transform, insert, delete,* or *keep*. I set the program point after the modified node in the fixed version using this information. In the presence of multiple modified lines, I treat them all as target program points, with the beginning of each area serving as the program point of interest. This ensures that IFRIT targets all modified code.

```
1  int function(int x) {
2      x--; //keep
3      x+=2; // del
4      //pp
5      if(x>5) //keep
6          x = x+1; //keep
7      }
```

```
1  int function(int x) {
2      x--; //keep
3
4      //pp
5      if(x>5) //keep
6          x = x+1; //keep
7      }
```

Figure 6.3: Example of alignment

Figure 6.3 shows an example of target point identification. In this example, the fix of the fault requires deleting a line. I set the program point at line 5, immediately after the deleted node.

6.3.2 Evaluation Procedure

To answer RQ0, I conducted a preliminary experiment on numeric input manipulation. The objective is to determine the optimal number of scale factors to use. I selected a subset of 10 programs coming from CodeFlaws (CF) [TYM+17]. I tested each program in 5 different groups of scale factors. I selected scale factors that can ensure a good exploration of the input space once combined, considering the input domain and the episode length during the testing phase. Configurations A-B-C uses small scale factors that allow to easily generate inputs close

among them. Configurations D-E feature both small both large-scale factors that allow to quickly traverse the input space.

To answer RQ1, I measure the performance of IFRIT in terms of reachability and uniformity. In RQ2, I compare the same metrics concerning other baseline techniques. Every experiment was repeated 20 times per program to account for the non-determinism of the algorithms involved. Moreover, I checked whether there is a statistically significant difference between these distributions by using the Wilcoxon rank sum test. When the p-value is smaller than 0.05, I consider that there is a significant difference between IFRIT and the best-performing generator between the opponents.

The last part (RQ3) compares the ability of IFRIT to detect mutants and real faults for the baseline tools. I dedicate one hour of testing time per technique for each program point.

6.4 Experimental Results

6.4.1 Reachability and Uniformity

Configuration	Scale Factors	Reachability
A	1,2,3	90%
B	1,2,3,4,5	100%
C	3,4,5,6,7	92%
D	1,2,5,10	87%
E	1,2,5,10,100	83%

Table 6.1: The Configurations

Table 6.1 shows the reachability obtained with different configurations of scale factors of IFRIT. Configuration *B* achieves the highest score in terms of reachability. When only a few scale factors are used (i.e., *Configuration A*), IFRIT does not explore enough of the input space of the programs and does not generate enough different inputs within the same episode. The problem with the other configurations (i.e., *Configurations C-D-E*) is that they tend to generate sparse inputs, which do not explore accurately the neighborhoods of the inputs that are close to those reaching the program point. Configuration *B* has enough scale factors to explore the input space at large, but at the same time, it has factors that are small enough to avoid overly big jumps in the input space.

RQ0: Configuration *B* achieves the highest score in terms of reachability. Hence it has been selected as the default configuration of IFRIT.

Repo		IFRIT			DFT			Random			Random Mutations			
	Size	**Reach.**	**Unif.**	**Time**	**Reach.**	**Unif.**	**Time**	**Reach.**	**Time**	**T. To IFRIT**	**Reach.**	**Unif.**	**Time**	**T. To IFRIT**
CF	2000	100%	**100%(M)**	468	100%	85%	158	48%	8	17	60%	90%	12	20
	6000	100%	**95% (S)**	468	99%	85%	483	51%	25	49	70%	90%	28	40
	12000	88%	**95% (M)**	468	82%	80%	683	52%	53	90	53%	85%	59	98
	24000	84%	**90% (L)**	468	81%	60%	1071	53%	97	153	51%	85%	108	178
tcas	2000	**95% (S)**	95%	477	85%	90%	161	9%	23	243	24%	85%	29	115
	6000	**93% (M)**	90%	477	83%	85%	476	10%	70	651	20%	90%	82	381
	12000	**90% (M)**	**95% (S)**	477	80%	85%	657	13%	140	969	19%	85%	157	744
	24000	**86% (S)**	**95% (S)**	477	78%	85%	1034	13%	280	1852	14%	85%	284	1745

Table 6.2: Comparison on mean reachability and mean uniformity across the considered tools. Boldface highlights the best results, when statistical significance is reached (the effect size is summarized in brackets: N = Negligible; S = Small; M = Medium; L = Large).

Table 6.2 shows the comparison between IFRIT and baseline tools when dealing with programs that accept only numeric inputs, as DFT can handle only this type of program. I chose a program point at the beginning of each branch of each program in our corpus to measure reachability and uniformity. When there is a statistically significant difference between IFRIT and the second best performing generator according to the Wilcoxon test (p-value < 0.05), I show the performance metric in boldface, including the Vargha-Delaney effect size in brackets (N = Negligible; S = Small; M = Medium; L = Large).

To measure reachability, I read the traces produced by the instrumentation and verified that the flag of the program point is active for the given test case. I calculated the percentage of tests that reached their target program points for each test suite and showed their descriptive statistic in Table 6.2 (mean). This table shows that the Random has the worst reachability results (around or lower than 50%), and Random Mutations behave similarly. DFT performs well on the programs coming from CodeFlaws. The metric "Time to IFRIT" (i.e., T. to IFRIT) in Table 6.2 is computed for Random as the hypothetical time in minutes needed to produce the same number of inputs that reach the target as IFRIT by continuing random generation beyond the test suite size limit (indicated in column 2). For instance, if the test suite size is 2000 and the IFRIT reachability is 90% (Random reachability is 40%), I know that in total IFRIT has been able to produce 1800 (Random: 800) inputs that reach the target. Hence, the execution time of Random should be multiplied by a factor $\times 2.25$ (i.e., 1800/800) to generate the same number of reaching inputs produced by IFRIT.

There is no advantage in using IFRIT rather than a random algorithm on simple programs like the ones contained in CF. In fact, the time to achieve reachability with Random is lower than the time that IFRIT takes to generate the same amount of inputs that reach the target. Instead, on *tcas* it is clear that a random generator can not match the performance of IFRIT in a reasonable amount of time. Considering DFT, its reachability on CF is close to that of IFRIT. This could be due to the simplicity of the programs that the CF repository contains. On *tcas*, IFRIT performs better than DFT, and the difference is statistically significant. This could be due to the higher complexity of the program under test, which the symbolic execution component of DFT can not handle efficiently.

I measured the uniformity of the approaches by running the L2-test, which requires the definition of a distance metric and the associated ϵ-margin. I used $\epsilon = 0.05$. This distance is smaller than the traditional distance from the state-of-the-art experiments, where it is generally around 0.25 [DGPP16], hence setting a stricter criterion for the L2-test. I limited the number of samples generated for the experiments considering the following domain sizes: 2,000, 6,000, 12,000, and 24,000, respectively. The L2-test results are shown in Table 6.2. Percentages represent the proportion of test suites generated for each program point that passed the L2-test. Because it samples directly from the uniform distribution, Random passes the L2-test in all cases by construction (its value is always 100%). Hence, it is not included in Table 6.2. IFRIT achieves significant improvements in almost every scenario w.r.t. DFT.

Table 6.3 shows the comparison between IFRIT and baselines tools when dealing with programs that accept string inputs. I could not test DFT in this scenario since it only generates numeric inputs. The Random string generator, which builds up a random string length at first, and then it generates random characters to fill the string, shows the worst reachability results. I do not report its uniformity results because it generates a uniform distribution by construction. Random Mutations, which randomly mutates previously computed strings, perform better than Random. Still, the reachability of IFRIT is higher than that of Random Mutations, with statistical significance and a large effect size. Looking at the metric "Time To IFRIT", on simple programs (e.g., *printtokens*, and *printtokens2*), both Random and Random Mutations are more convenient to use than IFRIT. IFRIT becomes more convenient to use when dealing with more complex programs (e.g., *gzip* and *grep*) and a test suite of size 6000. Figure 6.4 shows the reachability of the different generators on `tcas` (with test suite size: 12,000) over time. Random is the fastest to terminate, but its reachability is low compared to the other generators. DFT reaches a plateau after 500 minutes and remains quite stable until the end. This plot confirms that IFRIT is faster than DFT, and achieves better results than Random and DFT.

RQ1: IFRIT produces test suites with a close to uniform distribution and it reaches the target program point 85% of the times on average.

RQ2: IFRIT improves reachability and uniformity with respect to the baselines and state-of-art tools.

Qualitative Analysis

Figure 6.5 shows an excerpt from `tcas`, which includes a target program point controlled by the condition `enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped)`. The boolean variables that appear in this condition are defined in previous statements as boolean expressions that involve input variables (identifiers made of words starting

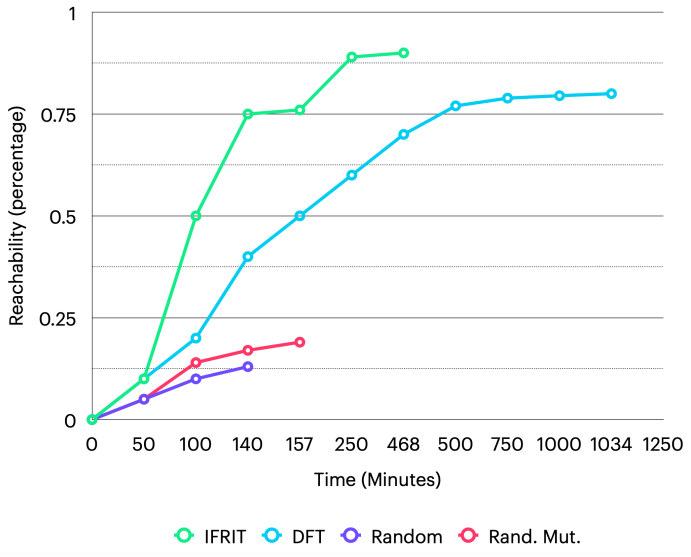

Figure 6.4: Reachability over time for IFRIT, DFT, and Random on `tcas` (with test suite size: 12,000).

with a capital letter and continuing with lower case letter, such as `Other_Capability`) and constants (upper case identifiers, such as `TCAS_TA`).

The symbolic execution step of DFT will replace all program variables (e.g., `tcas_equipped`) with the expressions defining them along the path of interest and will repeat the process recursively until only input variables and constants are left. In the example in Figure 6.5, such symbolic substitution produces a rather complex boolean expression for the condition controlling the target. Indeed, the resulting symbolic condition contains 6 distinct input variables and 4 distinct constants and involves 8 boolean or relational operators. Then, DFT relaxes the constraints by introducing parameters to be optimized by a search algorithm. However, due to the complexity of the parameterized expression, the SMT solver executed to find a solution either finds no solution (8% of the cases) or finds a diverging solution, i.e., a solution that respects the parameterized constraints but does not lead to the target.

On the contrary, IFRIT's incremental input mutation process can find a solution by exploring the neighborhood of previously attempted candidate inputs. While initially such a search process is mostly random, once any viable solution is found, the positive reward received by the RL algorithm is consolidated into its exploration policy. Hence, in the next iterations, the RL algorithm will exploit such accumulated knowledge to select the actions (input mutations) that are more likely to lead to the target. At the same time, as a larger reward, Γ_1 is granted only when new inputs that reach the target are generated, the learned policy will avoid the mere repetition of previous actions, which would re-generate the same inputs multiple times (this is associated with a smaller reward Γ_0), and will promote diversity in the generation process.

Repo		IFRIT			Random Mutations				Random		
	Size	Reach.	Unif.	Time	Reach.	Unif.	Time	T. To IFRIT	Reach.	Time	T. To IFRIT
printtokens	2000	**89% (L)**	90%	523	51%	100%	70	123	40%	58	129
	6000	**86%(L)**	85%	501	60%	95%	130	186	39%	112	246
printtokens2	2000	**88% (L)**	85%	502	56%	85%	74	116	37%	62	147
	6000	**84%(L)**	90%	490	65%	85%	134	173	39%	117	252
flex	2000	**83% (L)**	80%	496	22%	85%	69	260	13%	65	415
	6000	**80% (L)**	85%	478	32%	90%	123	307	15%	123	656
gzip	2000	**70% (L)**	85%	533	14%	85%	87	435	8%	75	656
	6000	**70% (L)**	85%	520	22%	85%	160	509	10%	137	959
grep	2000	**72% (L)**	85%	487	12%	80%	92	552	9%	81	648
	6000	**71% (L)**	85%	483	24%	85%	174	514	13%	143	781

Table 6.3: Reachability and uniformity across the considered tools. Boldface highlights the best results, when statistical significance is reached (the effect size is summarized in brackets: N = Negligible; S = Small; M = Medium; L = Large).

Overall, the reachability of IFRIT for this target branch was 92%, while DFT had reachability of 80%. At the same time, IFRIT passed the uniformity L2 test 90% of the time.

6.4.2 Mutation Score and Faults Detected

I evaluate IFRIT in terms of mutation score and number of faults detected and compare it with the baseline techniques. I created up to 100 mutants per program using Milu [JH08]. I did not filter the generated mutants. I used the same test suite on both the mutant and the original program to see if they produce different results. When the test results differ, I conclude that the test suite strongly kills the mutant. The mutation score represents the percentage of mutants that were killed. I used the alignment algorithm described in Section VI to determine the program point where the mutation or the fault is located. I allocate one hour of testing time per technique for each program point.

Table 6.4 shows the mutation score and percentage of faults detected for each technique and repository. On numeric inputs, I compare Random, Random Mutation, DFT, and IFRIT. On CF, IFRIT performs statistically better than DFT in killing mutants. The effect size is small. On `tcas`, IFRIT is statistically better both in killing mutants and in fault detection. Considering the programs that accept string inputs, IFRIT reaches up to 61% improvement in mutation score and up to 59% improvement in detecting real faults.

Figure 6.6 shows the asymptotic behavior of the different generators on `tcas`. DFT reaches a plateau and remains quite stable earlier than IFRIT, which keeps a positive derivative all over the allotted time budget, while Random and Random Mutations are not enough powerful to kill

Repo	Mutants				Faults			
	IFRIT	**DFT**	**Random**	**Rand. Mut.**	**IFRIT**	**DFT**	**Random**	**Rand. Mut.**
CF	**89%(S)**	79%	71%	72%	81%	80%	74%	72%
tcas	**85%(M)**	73%	7%	10%	**65% (S)**	55%	10%	13%
	IFRIT	**Rand. Mut.**	**Random**		**IFRIT**	**Rand. Mut.**	**Random**	
printtokens	**80% (L)**	32%	23%		**73% (L)**	40%	26%	
printtokens2	**74% (L)**	38%	27%		**75% (L)**	35%	28%	
flex	**70% (L)**	32%	25%		**81% (L)**	28%	23%	
gzip	**74% (L)**	15%	13%		**78% (L)**	12%	17%	
grep	**70% (L)**	14%	9%		**73% (L)**	9%	14%	

Table 6.4: Mutation Score and Faults Detected on the two repositories in a Fixed Time Budget (One hour). Boldface highlights the best results, when statistical significance is reached (the effect size is summarized in brackets: N = Negligible; S = Small; M = Medium; L = Large).

```
int alt_sep_test(){
    bool enabled, tcas_equipped, intent_not_known;
    bool need_upward_RA, need_downward_RA;
    int alt_sep;

    enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV)
              && (Cur_Vertical_Sep > MAXALTDIFF);
    tcas_equipped = Other_Capability == TCAS_TA;
    intent_not_known = Two_of_Three_Reports_Valid && Other_RAC == NO_INTENT;

    alt_sep = UNRESOLVED;

    if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped)){
       need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
       need_downward_RA = Non_Crossing_Biased_Descend() && Own_Above_Threat();
       ...
    }
    ...
}
```

Figure 6.5: Example of a conditional branch from `tcas`, where the DFT parameterized constraints are unsatisfiable or divergent, while IFRIT input mutations are guided toward the target.

a significant proportion of mutants. This plot confirms that IFRIT outperforms the rest of the tools, achieving better results even on the SIR repository.

RQ3: On every repository, IFRIT performs better than the baselines, with an improvement between 18% and 78% on mutation score, and between %7 and %61 on fault detection.

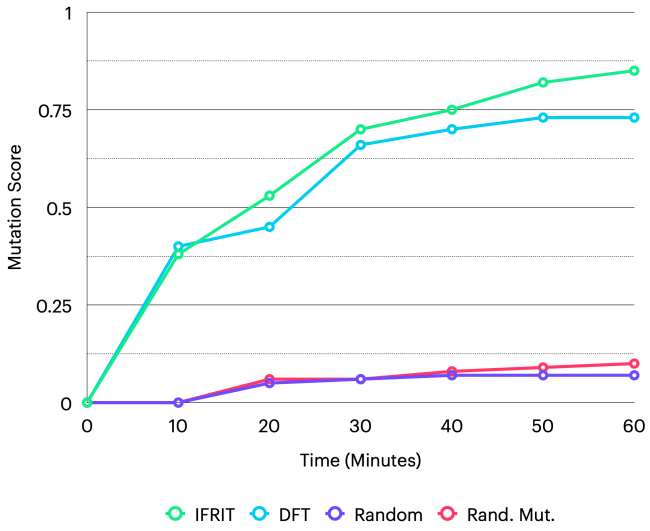

Figure 6.6: Mutation score over time for IFRIT, DFT, and Random on `tcas`.

6.5 Threats to Validity

Construct Threats. Our definition of diversity may pose a threat to construct validity. Due to its grounding in information theory, I used uniformity as a measure of diversity. Other authors, on the other hand, use test case similarity metrics. It is possible that different findings would have been obtained if similarity measures had been used instead of relying on diversity/distance.

External Threats. Our experiments are performed on open-source code repositories. Although our subjects have been previously used in the literature [MJS+21, CKMT10, SDM+18], our results might not generalize to other subjects or programming languages.

6.6 Discussion

IFRIT improves the quality of fault detection and mutation killing when a focused test suite that reach a given target is to be generated automatically. By using Deep RL, our approach enhances the diversity of the test suite, making the input distribution close to a uniform distribution. Empirical results show that the quality of the test suites generated by IFRIT significantly outperforms random generation and the state-of-the-art tool DFT. Moreover, IFRIT is faster in generating big test suites since it does not rely on symbolic execution.

6.6.1 Future extensions

The current implementation of IFRIT is limited to the manipulation of numeric and string inputs only. However IFRIT can be extended to deal with types such as structures and pointers, by recursively applying the generators for strings and numbers, for structure fields that belong to these two types, and by choosing abstract memory references to typed data structures from a pool of available memory references, when pointers are used as structure fields. This extension of IFRIT is part of the ongoing tool development.

Another potential extension of IFRIT would be to add support for other programming languages. Given the black-box nature of the RL algorithm being used, the main limitations are technological (e.g., how to collect coverage information) rather than conceptual.

Chapter 7

Conclusion

This thesis addresses the problem of automatically generating test cases for applications through Deep Reinforcement Learning. This problem has been investigated from different viewpoints. This thesis explored both GUI testing and security testing for test cases generation. Subsequently, this thesis presented a focused testing approach capable of increasing the number of test cases that can reach a specific target to better exhibit the bugs exposed by previous testing techniques.

I studied the problem of automated GUI testing of Android apps, proposing an approach based on Deep RL. The approach is implemented in the open-source tool ARES. The AUT represents the RL environment subject to several interaction steps. At each time step, assuming the GUI state is s_t, and the reward is r_t, ARES first takes an action a_t. Then, it receives the new GUI state s_{t+1} of the AUT, and a reward r_{t+1}. If the new state s_{t+1} is similar to the prior state s_t, the reward is negative. Otherwise, the reward is positive. In this way, ARES promotes the exploration of new states in the AUT, assuming that this is useful for testing the application more thoroughly. The reward is used to update the neural network, which learns how to guide the Deep RL algorithm to explore the AUT. The AUT's best exploration strategy is automatically learned as the test progresses. ARES outperformed all the considered baselines in terms of coverage achieved over time and exposed bugs.

In this thesis, I also studied the issue of generating exploits for a subset of Android ICC vulnerabilities (i.e., IDOS, XAS, and FI) through static analysis, Deep Reinforcement Learning-based dynamic analysis, and software instrumentation. More in detail, RONIN takes as input a mobile app. It starts the static analysis that outputs all the identified vulnerable statements within the AUT. According to the information coming from the static analysis, the AUT is instrumented to let RONIN's dynamic analysis verify whether a vulnerability has been triggered. During the dynamic analysis, RONIN stimulates the AUT with Intents crafted through Deep RL and random GUI events to confirm whether the statically identified vulnerabilities can be exploited. The RONIN approach achieved better results than state-of-the-art and baseline tools in the number of

exploited vulnerabilities.

At last, I studied the issue of developing a test suite to repeatedly cover a given point in a program, with the ultimate goal of exposing faults affecting the given program point. The approach, IFRIT, uses Deep Reinforcement Learning to generate diverse inputs while maintaining a high reachability of the desired program point. IFRIT achieves better results than state-of-the-art and baseline tools, improving reachability, diversity, and fault detection.

7.1 Open Research Directions

The results of this thesis open new research directions toward the automatic generation of effective test cases for interactive and non-interactive applications.

RONIN: Potential Extensions RONIN approach proved effective in exploiting certain classes of ICC vulnerabilities in Android apps. However, the current version of RONIN only supports the automatic exploitation of three categories of vulnerabilities. Potential future extensions include the application of RONIN to other complex attack patterns, e.g., multi-app attack scenarios such as colluded applications [RCT+14] and confused deputy attacks [WCB+15].

RL-based Black-box Testing of IoT Devices through Mobile Apps. The huge pervasiveness of interacting IoT devices is enabling unprecedented application scenarios. Although the IoT paradigm is increasing in terms of features and supporting frameworks, the same maturity is still not achieved from a cybersecurity standpoint. More specifically, I argue that a missing security enabler is a methodology that supports the systematic detection of vulnerabilities in IoT firmware and apps in a technology-agnostic way. Investigating an approach that leverages Reinforcement Learning techniques to test IoT devices without accessing their firmware could be interesting. The key idea is that IoT devices are controlled through their companion mobile app. "Command and control" attacks are carried on thanks to companion apps.

Bibliography

[Ach18] Josh Achiam. Key concepts in rl, 2018.

[AFT+12] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. Using gui ripping for automated testing of android applications. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. IEEE, 2012.

[AFT+14] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE software*, 32(5):53–59, 2014.

[AGB+16] Iftekhar Ahmed, Rahul Gopinath, Caius Brindescu, Alex Groce, and Carlos Jensen. Can testedness be effectively measured? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 547–558, New York, NY, USA, 2016. Association for Computing Machinery.

[AGGC16] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christi. Generating focused random tests using directed swarm testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 70–81, 2016.

[AH12] Nadia Alshahwan and Mark Harman. Augmenting test suites effectiveness by increasing output diversity. In *Proceedings of 34th International Conference on Software Engineering, ICSE*, pages 1345–1348, 2012.

[ANHY12] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[app20a] Appbrain, 2020.

[App20b] Appium, 2020.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[atlay] atlasVPN. Over 60% of android apps have security vulnerabilities. `https://atlasvpn.com/blog/over-60-of-android-apps-have-security-vulnerabilities`, Accessed in January 16, 2023.

[AYS16] Mohammed K Alzaylaee, Suleiman Y Yerima, and Sakir Sezer. Dynalog: An automated dynamic analysis framework for characterizing android applications. In *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*, pages 1–8. IEEE, 2016.

[Bac96] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[BBGM10] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE symposium on security and privacy*, pages 332–345. IEEE, 2010.

[BCF+08] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401, 2008.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[BGZ18] Nataniel P Borges, Maria Gómez, and Andreas Zeller. Guiding app testing with mined interaction models. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 133–143. IEEE, 2018.

[BM95] Justin A Boyan and Andrew W Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in neural information processing systems*, pages 369–376, 1995.

[BP14] Marcel Böhme and Soumya Paul. On the efficiency of automated testing. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 632–642, 2014.

[BSGM15] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. Covert: Compositional analysis of android inter-app permission leakage. *IEEE transactions on Software Engineering*, 41(9):866–886, 2015.

[BSRT19] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Diversity-based web test generation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*, pages 142–153, 2019.

[CA16] Jack Clark and Dario Amodei. Faulty reward functions in the wild, 2016.

[CCYW16] Lin Cheng, Jialiang Chang, Zijiang Yang, and Chao Wang. Guicat: Gui testing as a service. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 858–863, 2016.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, pages 209–224, 2008.

[CFGW11] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252, 2011.

[CGLX15] Chen Cao, Neng Gao, Peng Liu, and Ji Xiang. Towards analyzing the input validation vulnerabilities associated with android system services. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 361–370, 2015.

[CGO15a] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet?(e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440. IEEE, 2015.

[CGO15b] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet?(e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440. IEEE, 2015.

[CGO15c] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet?(e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440. IEEE, 2015.

[CKMT10] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[CMV13] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. A scalable approximate model counter. In *International Conference on Principles and Practice of Constraint Programming*, pages 200–216. Springer, 2013.

[Cov99] Thomas M Cover. *Elements of information theory*. John Wiley & Sons, 1999.

[CQX+20] Wang Chao, Li Qun, Wang XiaoHu, Ren TianYu, Dong JiaHan, Guo GuangXin, and Shi EnJie. An android application vulnerability mining method based on static and dynamic analysis. In *2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC)*, pages 599–603. IEEE, 2020.

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[DBCR20a] Zhen Dong, Marcel Bohme, Lucia Cojocaru, and Abhik Roychoudhury. Github repository: Timemachine, 2020.

[DBCR20b] Zhen Dong, Marcel Bohme, Lucia Cojocaru, and Abhik Roychoudhury. Time-travel testing of android apps. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, pages 481–492, 2020.

[DC20] Biniam Fisseha Demissie and Mariano Ceccato. Security testing of second order permission re-delegation vulnerabilities in android apps. In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pages 1–11, 2020.

[DCS20] Biniam Fisseha Demissie, Mariano Ceccato, and Lwin Khin Shar. Security analysis of permission re-delegation vulnerabilities in android apps. *Empirical Software Engineering*, 25(6):5084–5136, 2020.

[Det22] CVE Details. Vulnerabilities by date, 2022.

[DGPP16] Ilias Diakonikolas, Themis Gouleakis, John Peebles, and Eric Price. Collision-based testers are optimal for uniformity and closeness. *arXiv preprint arXiv:1611.03579*, 2016.

[emm06] Emma, 2006.

[EOM09] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding android security. *IEEE security & privacy*, 7(1):50–57, 2009.

[FA13] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Trans. Software Eng.*, 39(2):276–291, 2013.

[FCH+11] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638, 2011.

[FPCY16] Robert Feldt, Simon M. Poulding, David Clark, and Shin Yoo. Test set diameter: Quantifying the diversity of sets of test cases. In *Proceedings of IEEE International Conference on Software Testing, Verification and Validation, ICST*, pages 223–233, 2016.

[FvHM18] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[FWM+11] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX security symposium*, volume 30, page 88, 2011.

[GHGM17] Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek. Automatic generation of inter-component communication exploits for android applications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 661–671, 2017.

[GKKP09] Svetoslav Ganov, Chip Killmar, Sarfraz Khurshid, and Dewayne E Perry. Event listener analysis and symbolic execution for testing gui applications. In *International Conference on Formal Engineering Methods*, pages 69–87. Springer, 2009.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[Goo19a] Google. Broadcasts, 2019.

[Goo19b] Google. Safety net, 2019.

[Goo20a] Google. Android emulator, 2020.

[Goo20b] Google. System-level events api 25, 2020.

[Goo20c] Google. Ui/application exerciser monkey, 2020.

[Goo22] Google. Intent, 2022.

[GR11a] Oded Goldreich and Dana Ron. On testing expansion in bounded-degree graphs. In *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation*, pages 68–75. Springer, 2011.

[GR11b] Oded Goldreich and Dana Ron. On testing expansion in bounded-degree graphs. In *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation*, pages 68–75. Springer, 2011.

[GRD06] Alex Kinneer Gregg Rothermel, Sebastian Elbaum and Hyunsook Do. Software-artifact infrastructure repository, 2006.

[Gro22] Sable Research Group. Soot - a framework for analyzing and transforming java and android applications, 2022.

[GSM+19] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. Practical gui testing of android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 269–280. IEEE, 2019.

[GSS+20] Wunan Guo, Liwei Shen, Ting Su, Xin Peng, and Weiyang Xie. Improving automated gui exploration of android apps via static dependency analysis. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 557–568, 2020.

[GTDR18] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. Android testing via synthetic symbolic execution. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 419–429. IEEE, 2018.

[Har07] Mark Harman. The current state and future of search based software engineering. In *Future of Software Engineering (FOSE'07)*, pages 342–357. IEEE, 2007.

[HCL22] HCL. Appscan, 2022.

[HCM10] Si Huang, Myra B. Cohen, and Atif M. Memon. Repairing gui test suites using a genetic algorithm. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 245–254, 2010.

[HJ01] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.

[HMZ12] 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.

[HN11] Cuixiong Hu and Iulian Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, page 77–83, New York, NY, USA, 2011. Association for Computing Machinery.

[Hol79] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70, 1979.

[HRE+18] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[HTP15] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 118–128, 2015.

[HYHW19] Yongzhong He, Xuejun Yang, Binghui Hu, and Wei Wang. Dynamic privacy leakage analysis of android third-party libraries. *Journal of Information Security and Applications*, 46:259–270, 2019.

[HZAL18] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[IH14] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering, ICSE*, pages 435–445, 2014.

[JH08] Yue Jia and Mark Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. In *Testing: Academic Industrial Conference - Practice and Research Techniques (taic part 2008)*, pages 94–98, 2008.

[Jia13] Yajin Zhou Xuxian Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[KLP17] Rody Kersten, Kasper Luckow, and Corina S. Păsăreanu. Poster: Afl-based fuzzing for java with kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2511–2513, New York, NY, USA, 2017. Association for Computing Machinery.

[KSM+18] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. Qbe: Qlearning-based exploration of android applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 105–115. IEEE, 2018.

[LBB+15] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291. IEEE, 2015.

[LDR16] Sungho Lee, Julian Dolby, and Sukyoung Ryu. Hybridroid: Static analysis framework for android hybrid applications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 250–261, 2016.

[Lee09] Edward A. Lee. Finite state machines and modal models in ptolemy ii. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, Nov 2009.

[LHP+15] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[Li17] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[Lin93] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[LL05] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX security symposium*, volume 14, pages 18–18, 2005.

[LYGC19] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Humanoid: a deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1070–1073. IEEE, 2019.

[MAZ+15] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. GRT: program-analysis-guided random testing (T). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 212–223, 2015.

[MBN03] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, pages 260–269. Citeseer, 2003.

[MBNR13] Atif Memon, Ishan Banerjee, Bao N Nguyen, and Bryan Robbins. The first decade of gui ripping: Extensions, applications, and broader impacts. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 11–20. IEEE, 2013.

[McM04] Phil McMinn. Search-based software test data generation: a survey. *Softw. Test. Verification Reliab.*, 14(2):105–156, 2004.

[MDM+15] Patrick Mutchler, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. A large-scale study of mobile web app security. In *Proceedings of the Mobile Security Technologies Workshop (MoST)*, volume 50, 2015.

[MHJ16] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105, 2016.

[Mic20] Microsoft. your device is rooted and you can't connect-android, 2020.

[MJS+21] Héctor D Menéndez, Gunel Jahangirova, Federica Sarro, Paolo Tonella, and David Clark. Diversifying focused testing for unit testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–24, 2021.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[MMM14] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 599–609, 2014.

[MMP05] Y Marhuenda, D Morales, and MC Pardo. A comparison of uniformity tests. *Statistics*, 39(4):315–327, 2005.

[Mob22] MobSF. Mobile security framework (mobsf), 2022.

[MPRS12] Leonardo Mariani, Mauro Pezze, Oliviero Riganelli, and Mauro Santoro. Autoblacktest: Automatic black-box testing of interactive applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 81–90. IEEE, 2012.

[MR17] Joydeep Mitra and Venkatesh-Prasad Ranganath. Ghera: A repository of android app vulnerability benchmarks. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE, page 43–52, New York, NY, USA, 2017. Association for Computing Machinery.

[MRH20] Evgeny Mandrikov Marc R. Hoffmann, Brock Janiczak. Jacoco code coverage, 2020.

[MTN13] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234, 2013.

[MX05] Atif M Memon and Qing Xie. Studying the fault-detection effectiveness of gui test cases for rapidly evolving software. *IEEE transactions on software engineering*, 31(10):884–896, 2005.

[OLD+15] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 77–88. IEEE, 2015.

[OMJ+13] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective {Inter-Component} communication mapping in android: An essential step towards holistic security analysis. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 543–558, 2013.

[PHW+20] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. Reinforcement learning based curiosity-driven testing of android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 153–164, 2020.

[PKT18] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Trans. Software Eng.*, 44(2):122–158, 2018.

[Pla83] Robin L Plackett. Karl pearson and the chi-squared test. *International Statistical Review/Revue Internationale de Statistique*, pages 59–72, 1983.

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering, ICSE*, pages 75–84, 2007.

[PRC+22] Francesco Pagano, Andrea Romdhana, Davide Caputo, Luca Verderame, and Alessio Merlo. SEBASTiAn: a Static and Extensible Black-box Application Security Testing tool for iOS and Android applications. 10 2022.

[PY08] Mauro Pezzè and Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[RCT+14] Tristan Ravitch, E Rogan Creswick, Aaron Tomb, Adam Foltzer, Trevor Elliott, and Ledah Casburn. Multi-app security analysis with fuse: Statically detecting android app collusion. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, pages 1–10, 2014.

[Rie05] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer, 2005.

[SAS+22] Christian Schindler, Müslüm Atas, Thomas Strametz, Johannes Feiner, and Reinhard Hofer. Privacy leak identification in third-party android libraries. In *2022 Seventh International Conference On Mobile And Secure Services (MobiSecServ)*, pages 1–6. IEEE, 2022.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SBGM16] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering*, 43(6):492–530, 2016.

[SDM+18] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 130–140. IEEE, 2018.

[SLH+14] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 387–395, Bejing, China, 22–24 Jun 2014. PMLR.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272, 2005.

[SMC+17a] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based GUI testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations*

of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, pages 245–256, 2017.

[SMC+17b] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. System-level events api 19, 2017.

[Sta22] Statista. Number of available applications in the google play store. `https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store`, 2022.

[STDF14] Julian Schutte, Dennis Titze, and José María De Fuentes. Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 370–379. IEEE, 2014.

[SYB18] Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. A theoretical and empirical study of diversity-aware mutation adequacy criterion. *IEEE Trans. Software Eng.*, 44(10):914–931, 2018.

[Syn22] Synopsys. The cost of poor software quality in the us: A 2020 report, 2022.

[TH02] Mustafa M Tikir and Jeffrey K Hollingsworth. Efficient instrumentation for code coverage testing. *ACM SIGSOFT Software Engineering Notes*, 27(4):86–96, 2002.

[TM17] Vincent F Taylor and Ivan Martinovic. Short paper: A longitudinal study of financial apps in the google play store. In *International Conference on Financial Cryptography and Data Security*, pages 302–309. Springer, 2017.

[TSQ+19] Md Arabin Islam Talukder, Hossain Shahriar, Kai Qian, Mohammad Rahman, Sheikh Ahamed, Fan Wu, and Emmanuel Agu. Droidpatrol: a static analysis plugin for secure mobile software development. In *2019 IEEE 43rd annual computer software and applications conference (COMPSAC)*, volume 1, pages 565–569. IEEE, 2019.

[TYM+17] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 180–182. IEEE, 2017.

[VKCF+15] Tanja EJ Vos, Peter M Kruse, Nelly Condori-Fernández, Sebastian Bauersfeld, and Joachim Wegener. Testar: Tool support for test automation at the user interface level. *International Journal of Information System Modeling and Design (IJISMD)*, 6(3):46–83, 2015.

[Voa92] Jeffrey M. Voas. PIE: A dynamic failure-based technique. *IEEE Trans. Software Eng.*, 18(8):717–727, 1992.

[WCB+15] Jianliang Wu, Tingting Cui, Tao Ban, Shanqing Guo, and Lizhen Cui. Paddyfrog: systematically detecting confused deputy vulnerability in android applications. *Security and Communication Networks*, 8(13):2338–2349, 2015.

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[WRO18] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)*, 21(3):1–32, 2018.

[XGL+15] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *2015 IEEE Symposium on Security and Privacy*, pages 899–914. IEEE, 2015.

[Zal22] Michal Zalewski. American fuzzy lop (afl), 2022.

[ZS89] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989.