Assessing the Security of Inter-App Communications in Android through Reinforcement Learning

Andrea Romdhana[a,b], Alessio Merlo[a], Mariano Ceccato[c], Paolo Tonella[d]

[a] *DIBRIS - Università degli Studi di Genova*
[b] *FBK-ICT, Security & Trust Unit*
[c] *Università di Verona*
[d] *Università della Svizzera italiana*

Abstract

A central aspect of the Android platform is Inter-Component Communication (ICC), which enables the reuse of functionality across apps and components via message passing. While a powerful feature, ICC still constitutes a serious attack surface. This paper addresses 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. Our approach, called RONIN, achieves better results than state-of-the-art and baseline tools, in the number of exploited vulnerabilities.

Keywords: reinforcement learning, security testing, Software security engineering

1. Introduction

Nowadays, mobile phones are the most pervasive electronic devices worldwide [1]. Among the available mobile operating systems, Android rose as the most used platform [2]. The reason for this success lies in the high number of available apps, whose number currently exceeds 3 billion at the time of writing [3]. The presence of app construction frameworks and rich libraries, as well as easy distribution via online app stores such as Google Play, have significantly lowered the barrier to entry in app development and deployment [4]. Bhattacharya et al. [5] claim that the low barrier to enter the market means apps (or app updates) are subject to limited scrutiny before dissemination, allowing error-prone apps through and therefore affecting also their security. Therefore, developers and designers of such apps need to utilize proper approaches, tools, and frameworks that assist them in creating secure apps.

Numerous methodologies have been developed to find security flaws in Android apps [6]. Most of these methods rely on static analysis of Android apps to find such vulnerabilities [6] [7] [8] [9], but there are also strategies that make use of dynamic analysis [10] [11]. A few methods detect vulnerabilities by combining static and dynamic analysis [12] [13] [14]. Although these methods and procedures have made it possible to detect vulnerabilities, it is frequently necessary for security analysts to decide manually whether those flaws are actually exploitable, possibly with the aid of dynamics tools such as Drozer [15], Inspeckage [16], and Objection [17]. However, such a manual task is laborious and slow. A method should ideally be able to automatically find vulnerabilities in Android apps and determine whether such vulnerabilities are exploitable. Discovering automatically exploitable security flaws would help software engineers choose which issues to address first. It would also give them information useful to fix the security bugs and to evaluate the *vulnerability risk*, i.e., likelihood and impact of the vulnerability.

One challenge to address to enable *automatic exploit generation for Android apps* is how to penetrate Android's specific attack surface, including its distributed event-based and message-based frameworks. In particular, asynchronous messages, or what Android refers to as Intents, are largely used for inter-component communication (ICC) within and between Android apps. Thus, it is essential to model the Android framework, especially the ICC interface. When generated, Intents accept parameters that specify which action the app should perform. Additionally, the Android framework offers several predefined components that respond to Intents in various ways. For example, to show the user a location on a map, a developer can use an Intent to request that another capable app show a specified location on a map; or can use Intents to start a service to download a file in the background.

Defining a method for automatically determining whether a vulnerability has been exploited is another challenge when generating automatic exploits for Android. Garcia et al. [18] proposed an approach called Letterbomb for automatically generating exploits for Android apps. 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. The focus of Letterbomb is ICC vulnerabilities. Specifically, it focuses on three types: 1) inter-process denial of service, 2) cross-

Email addresses: andrea.romdhana@dibris.unige.it (Andrea Romdhana), alessio.merlo@unige.it (Alessio Merlo), mariano.ceccato@univr.it (Mariano Ceccato), paolo.tonella@usi.ch (Paolo Tonella)

application scripting, 3) and Fragment injection. However, Letterbomb stimulates an app only by directly triggering Intents, ignoring the possible internal usage of Intents triggered by GUI events [19]. Due to this, Letterbomb misses specific possible true positives, which can only be exposed by a proper sequence of GUI events. Moreover, the exploitability of a vulnerability at a particular statement depends on the different program paths that may lead to reaching that statement. A specific path may reach a statement without exploiting the vulnerability, but there may be more paths that reach the same statement in the program, and only some of them may exploit the vulnerable statement. Hence, when we modify the payload of an Intent to exploit a vulnerability, its execution along a specific path may or may not trigger the vulnerability, depending on the chosen path. Letterbomb includes path-sensitive analyses for path selection but faces path explosion problem as the program grows due to the potentially exponential number of program paths to be analyzed. Another related challenge is the generation of the parameter values for the Intent. Letterbomb uses a Satisfiability Modulo Theory (SMT) solver to generate parameters for the Intent. The drawback of such an approach is that it loses efficiency when dealing with an increasing number of parameters. At last, it becomes inapplicable when the constraints to reach a certain path within the app are too difficult to handle for an SMT solver.

We propose a different approach to exploit generation, which we 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 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 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.

2. Background

This section introduces the basics of RL as well as the targeted ICC vulnerabilities in detail.

2.1. Reinforcement Learning

Reinforcement learning's [20] objective is to teach an agent how to interact with a specific environment to achieve a particular goal. The agent determines the environment's present state, takes actions that possibly affect the environment, and receives a positive, neutral or negative reward.

At each time step t, the agent takes an action a_t, according to an observation x_t, which may be a partial or full representation of the environment state s_t. The action a_t causes the transition of the environment from state s_t to state s_{t+1}, and its quality is measured by the reward function $R(x_t, a_t, x_{t+1})$. A *Markov Decision Process (MDP)* formally describes the agent environment. An MDP is 5-tuple, $\langle S, A, R, P, \rho_0 \rangle$, where: S is the set of states, A is the set of actions, R the reward function, P the transition probability function, and $\rho_0 \subseteq S$ is the set of initial states. The goal in RL is to learn a policy π, i.e., a rule for deciding which action to take, based on the perceived state s_t. The learned policy must maximize the so-called *expected return*. Earlier RL algorithms approximate states and actions using tables that store discrete values. As it may be impractical to describe all possible states and actions in a table, these tabular techniques cannot learn appropriate policies in vast or unbounded discrete spaces (e.g., programs with numerous inputs, constraints, and states to be modeled)[21]. Deep Reinforcement Learning is a novel technique that has emerged due to the advent of Deep Learning, which relies on the sophisticated function approximation capabilities of Deep Neural Networks [21] to learn an optimal policy even in the presence of large state and action spaces.

2.2. 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 [22]. 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 [23], 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 Intent's action should be carried out by the app. 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.

2.3. 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 [24]. Unsafe handling of incoming ICC data can result in different forms of attack. Below we list three of the main threats that we aim to discover.

Cross-Application Scripting (XAS). Similarly to Cross-Site Scripting (XSS) in the Web landscape, XAS [24] 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 1 illustrates the different ways 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 a 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 [24], 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).

```
1  public class MainActivity extends AppCompatActivity {
2      @Override
3      protected void onCreate (Bundle savedInstanceState) {
4          // ...
5      }
6
7      @Override
8      protected void onResume(){
9          super.onResume();
10         Button button = (Button) findViewById(R.id.get):
11         WebView webView = (WebView) findViewById(R.id.webView1) :
12         button.setonClickListener(new View.OnClickListener() {
13
14             @override
15             public void onclick(View v) {
16                 TextView textView = (TextView) findViewById(R.id.concat);
17                 Intent intent = getIntent():
18                 String a = intent.getStringExtra("s1");
19                 if ("URL".equals(a)) {
20                     /* If s2 contains a malicious site it will be loaded by the WebView
                          */
21                     webView.LoadUrl(intent.getstringExtra("s2"));
22                 } else {
23                     String b = intent.getStringExtra("s2");
24                     /* If String b is null the app will crash */
25                     if (b.equals ("www.test.com"))
26                         webView.LoadUrl(intent.getStringExtra("s2"));
27                 }
28             }
29     });
30 }
31
32 // ...
33
34 public static void getFragmentInstance(Intent my_intent){
35     /* If fame is not checked we can have a Fragment Injection */
36     String fname = my_intent.getStringExtra("fname");
37     return Fragment.instantiate(this, fname) ;
38     }
39 }
```

Listing 1: RONIN: An example app that contains ICC vulnerabilities

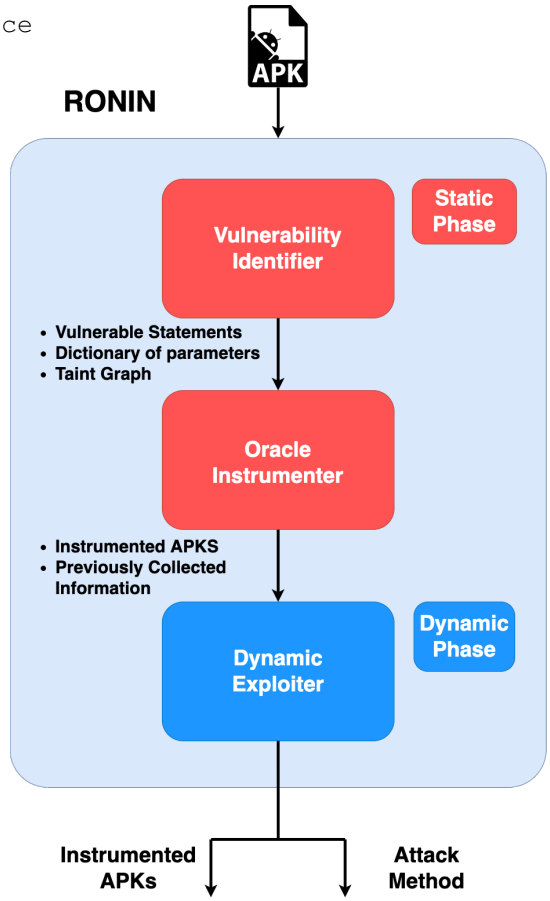

Figure 1: The RONIN workflow. At first RONIN analyzes the APK using static analysis and collects potential vulnerabilities. Then RONIN instruments the APK to verify whether a vulnerability has been reached or not. At last, RONIN leverages Intent generation through DeepRL and GUI stimulation to effectively exploit a vulnerability.

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 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.

3.1. Static Phase: Vulnerability Identifier

The *Vulnerability Identifier* analyzes an APK in search for 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 1). Subsequently, Vulnerability Identifier analyzes the taint graph in search for 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 we rely on the SEBASTiAn tool [25].

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. We describe how Oracle Instrumenter instruments the app for the three aforementioned vulnerability types. We only need to introduce the instrumentation once per identified vulnerability. After such one-time instrumentation, we 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 to instrument 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.

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 2 shows the workflow of this phase. The RL environment is represented by an app 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 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 vulnerability 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.

3.4. Dynamic Phase: Deep RL and GUI Events

The Dynamic Analysis of RONIN relies on Deep RL and random GUI events generation. 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.

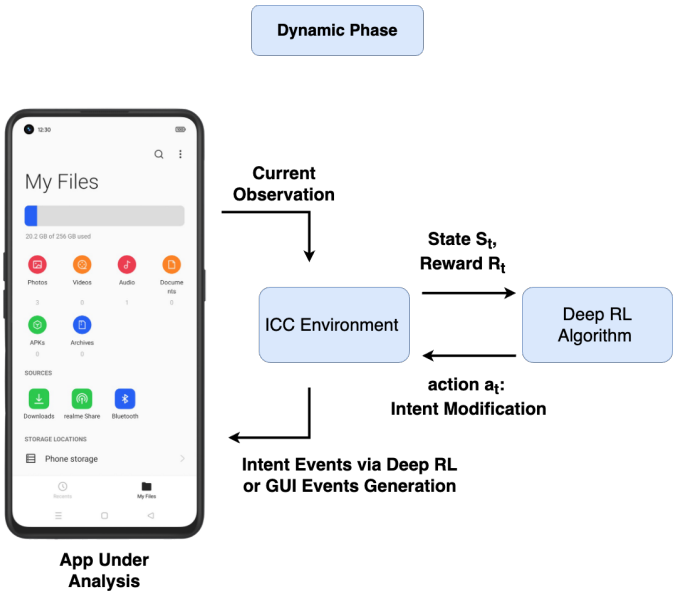

Figure 2: The dynamic phase workflow.

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
```

3.5. Dynamic Phase: Deep RL

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

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 ac-

tivity, 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 we set the corresponding node flag to 1; un-executed nodes have their flag set to 0.

Action Representation. Each time RONIN takes an 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, 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 an action. This is decided solely by the app's execution: RONIN observes the process passively, collecting the new state after the transition has occurred.

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

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

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

At 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 with respect to 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 with respect to the last execution. If the distance from the vulnerability remains the same as in the last execution, the reward is neutral ($\Gamma_0 = 0$).

4. Implementation

RONIN features a custom environment based on the OpenAI Gym[26] interface, which is a de-facto standard in the RL field. OpenAI Gym is a toolset with a number of built-in environments for building and comparing RL algorithms. It also includes instructions for defining custom environments. Our custom environment interacts with an Android app.

4.1. Tool Overview

As soon as it is launched, RONIN starts the static analysis on the target app in search for ICC vulnerabilities. The static analysis leverages the plugin that extracts ICC vulnerabilities in the SEBASTiAn tool [25], by extracting the vulnerable statements and the taint graph. Moreover, we added a method to the plugin to search and extract the parameters associated with the Intent functions.

Afterwards, if any vulnerability is present, RONIN starts the instrumentation phase. The Oracle Instrumenter is based on SOOT [27] and it uses the taint graph to decide where to inject log statements. Listing 2 shows an example of injected logs: each node that can potentially lead to the vulnerability is logged during the execution of the code.

RONIN generates as output many instrumented APKs, one per identified vulnerability. Then, RONIN starts the dynamic analysis on the instrumented APKs, leveraging the information collected during the static analysis: it builds the dictionary of values to use during Intent mutation, it instantiates a custom environment to interact with the application, and it starts the search for exploitations (Algorithm 1). At each time step, RONIN takes an action (i.e., it modifies the Intent) according to the current policy of the Deep RL algorithm. The action consists of crafting and launching the Intent, and possibly generating random GUI events on the target app. To generate random GUI events, we used the ARES tool [28]. Once the action has been fully processed, RONIN elaborates the log information, from which it computes observation and reward for the algorithm. RONIN arranges the whole testing session into finite-length episodes, the goal being to maximize the total reward received in each episode. Every episode lasts 100 time steps. Once an episode comes to an end, RONIN resets the Intent to the default value, and then it uses the acquired knowledge to reach the target node of the app in the next episode.

```
1 public void onClick(View v){
2    TextView textView = (TextView) findViewById(R.id.concat);
3    Intent intent = getIntent();
4    Log.v("{'method': 'onClick()', 'unit': 'intent.getStringExtra(..', 'id':
           '24503'}");
5    String example = intent.getStringExtra("example");
6    Log.v("{'method': 'onClick()', 'unit': 'textView.setText(..', 'id': '24504'}");
7    textView.setText("total length: " + Integer.toString(example.length()));
8 }
```

Listing 2: An example of instrumentation

4.2. ICC Environment

The ICC environment is responsible for handling the actions to interact with the app. 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 app to be analyzed and its setup. The second function is the `step(a)` function, which takes an action `a` as input and returns a list of objects, including observation (code coverage state) and reward.

4.3. Algorithm Implementation

RONIN leverages *Stable Baselines* [29], a modular library that adopts a plugin architecture to integrate the Deep RL algorithm to use. Currently, one Deep RL exploration strategy is integrated into RONIN: Soft Actor Critic (SAC) [30]. SAC

represents one of the state-of-the-art algorithms at the time of writing [31]. Its implementation comes from the Python library Stable Baselines. RONIN also implements a second exploration strategy, considered as a baseline in our empirical evaluation: random exploration, where the algorithm interacts with the target app by randomly selecting a mutation to perform on the Intent.

RONIN is publicly available as open source software at the url: `https://github.com/H2SO4T/RONIN`.

5. Evaluation

We 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. We evaluate RONIN's exploit generation capability by considering both exploits and unique exploits. We 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?* We 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. We evaluate the two tools on a benchmark to understand the key differences between them.
- **RQ3** *How does RONIN compare with the state-of-the-art on apps obtained from the wild?* We 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 Google Play Store.
- **RQ4** *How does RONIN behave when the generation of GUI events is disabled?* We 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?* We aim to evaluate RONIN's performance in detecting the three considered vulnerabilities when it generates Intents using a random algorithm (ablation study).

5.1. Evaluation Design

To evaluate the proposed approach, we used the software subjects from the Ghera dataset [19] 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, we used three ICC apps that contain four vulnerabilities related to IDOS, XAS, and FI. From the Google Play Store, we randomly selected 1500 apps 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.

5.2. Evaluation Procedure

With **RQ1**, we 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**, we 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 exploitation of the vulnerability. We investigate the reasons behind failures and the successes of both tools being compared.

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

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

In **RQ5**, we evaluate RONIN's performance (number of exploits and number of unique exploits) when generates Intents randomly. We also compare the Deep RL and random algorithm on the time necessary to generate the first exploit in each of the exploited vulnerabilities. To account for non-determinism, we applied the Wilcoxon non-parametric statistical test to draw conclusions on the difference between Deep RL and random algorithm, adopting the conventional p-value threshold at $\alpha = 0.05$.

6. Experimental Results

6.1. RQ1: Exploit Generation

Table 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, we 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, we 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, 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 1: Detected vulnerabilities and generated exploits

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

Table 2: Comparison between RONIN and Letterbomb on Ghera

6.2. RQ2: Comparison with Letterbomb on Ghera

Table 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 only in 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 3 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.

```
1  public class MyReceiver extends BroadcastReceiver {
2     @Override
3     public void onReceive (Context context, Intent intent) {
4        if (intent.getAction() != null && intent.getAction().equals("edu.ksu.cs.
               benign.myrecv")){
5           String number = intent.getStringExtra("number");
6           String text = intent.getStringExtra("text");
7           SmsManager smsManager = SmsManager.getDefault();
8           smsManager. sendTextMessage(number, null, "Benign: " + text, null, null);
9           Log.d("benign", "Message sent");
10       }
11    }
12 }
```

Listing 3: 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 2). RONIN successfully exploited all the statically detected vulnerabilities, while Letterbomb fails 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 4 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: then, 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 it 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           MainActivity.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 4: IDOS vulnerability occurring when the button instantiated at line 3 is pressed

6.3. 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 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. For FI, Letterbomb can not generate any exploit, while RONIN generates two exploits.

We 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. The remaining 7% of vulnerabilities are not covered by RONIN because its static analysis does not detect them. Moreover, 12% of the vulnerabilities that RONIN has exploited are related to GUI events that Letterbomb can not manage.

Listing 5 shows an example of vulnerability that Letterbomb does not exploit. This vulnerability is similar to the one presented for RQ2. At line 12, we 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 are missed by Letterbomb***.

```
1  call.setOnClickListener(new View.OnClickListener(){
2     @override
3     public void onClick(View view) {
4     Intent intent = getIntent();
5     String emergency_number = intent.getStringExtra("emergency_number");
6     if (ContextCompat.checkSelfPermission(getcontext(),
7        Manifest.permission.CALL_PHONE) != PackageManager.PERMISSION_GRANTED) {
8        ActivityCompat.requestPermissions(getActivity(), new String[]{Manifest.
               permission.CALL_PHONE}, REQUEST_CALL);
9     } else {
10       if (emergency_number.length() = 10) {
11          String dial = "tel:" + emergency_number;
12          startActivity(new Intent(Intent.ACTION_CALL, Uri.parse(dial)));
13       }
14    }
15    }
16 });
```

Listing 5: IDOS vulnerability occurring when the button attached to the `onClick`

6.4. RQ4: Disabling GUI Events

Table 3 shows the difference of behaviors when we 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 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 3: RONIN's reduced performance when GUI Events are disabled

6.5. RQ5: DeepRL vs Random

Table 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 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 the knowledge acquired so far. On the contrary, the random method does not have any memory of the past actions, resulting in poor performance compared to Deep RL.

Figure 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 4: RONIN's reduced performance when random Intent generation is adopted and Deep RL is disabled

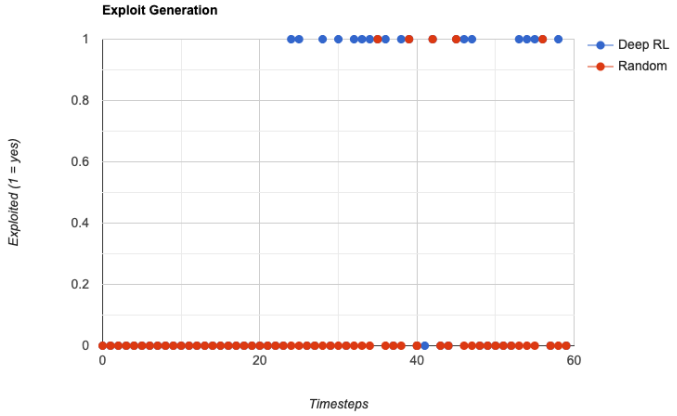

Figure 3: Time required by Deep RL vs Random to generate an exploit

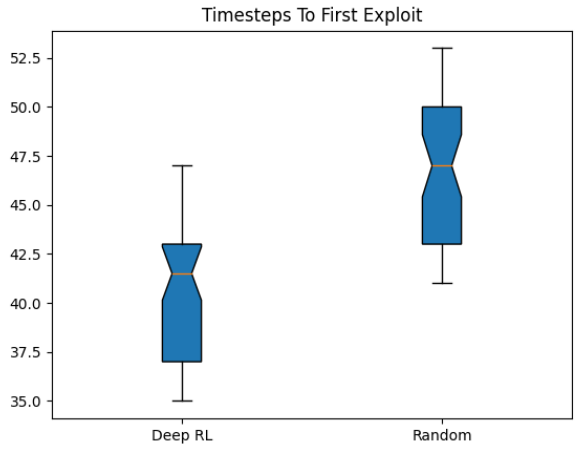

Figure 4: Distribution of the time steps required by Deep RL vs Random to generate the first exploit

7. Related Works

Several approaches have been developed to identify vulnerabilities in Android apps [32]. ComDroid [33] was one of the first significant works to target ICC-based vulnerabilities in detail, Epicc [34] and IC3 [35] extracted information about Intents in a flow-sensitive manner. IccTA [36] and COVERT [37] identified vulnerabilities involving interaction between apps rather than only individual apps. FlowDroid [38] performs a static taint analysis to identify flows and privacy leakages from Android API sources to sinks. Amandroid [7] is a static analysis approach based on Soot [27] 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 with respect to both. DroidPatrol [8] identifies a list of potential vulnerabilities and proposes quick fixes. MobSf [9] executes a plethora of security evaluations. However, none of these approaches can determine program paths and the Intents needed to execute them.

Another set of approaches relies solely on dynamic analysis to discover vulnerabilities. Buzzer [39] fuzzes Android system services to find flaws. Stowaway [40] detects permission overprivileged dynamically. Mutchler et al. [41] look for vulnerabilities in Android web apps. IntentDroid [24] 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 [42] 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 [43] is an OS-based security mechanism that evaluates an app's privileges as it gets requests from other apps. AppAudit [14] is primarily concerned with discovering privacy leakage vulnerabilities. However, it only conducts minimal Intent analysis (e.g., failing to account for various Intent attributes). AppCaulk [44] detects and stops data breaches through static and dynamic analysis and the ability to establish data leak policies. The DynaLog [10] 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. [11] 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 [12] [13] also detect vulnerabilities by combining static and dynamic analysis. Chao et al. [13] 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 by means of dynamic taint analysis and is reported to the user. Schindler et al. [12] 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 [45, 46] Demissie et al. present an approach based on static analysis and automated test case generation to generate exploits that target the Permission Redelegation vulnerabilities. To the best of our knowledge, Letterbomb [18] 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.

8. Conclusion

This paper introduces 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.

References

[1] Statista, Number of smartphone subscriptions worldwide from 2016 to 2027 (2022).
URL https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

[2] StatCounter, Mobile operating system market share worldwide (2022).
URL https://gs.statcounter.com/os-market-share/mobile/worldwide

[3] Statista, Number of available apps in the google play store from 2nd quarter 2015 to 2nd quarter 2022 (2022).
URL https://www.statista.com/statistics/289418/number-of-available-apps-in-the-google-play-store-quarte

[4] W. Enck, D. Octeau, P. D. McDaniel, S. Chaudhuri, A study of android application security., in: USENIX security symposium, Vol. 2, 2011.

[5] P. Bhattacharya, L. Ulanova, I. Neamtiu, S. C. Koduru, An empirical analysis of bug reports and bug fixing in open source android apps, in: 2013 17th European Conference on Software Maintenance and Reengineering, 2013, pp. 133–143. doi:10.1109/CSMR.2013.23.

[6] A. Sadeghi, H. Bagheri, J. Garcia, S. Malek, A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software, IEEE Transactions on Software Engineering 43 (6) (2017) 492–530. doi:10.1109/TSE.2016.2615307.

[7] F. Wei, S. Roy, X. 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) (2018) 1–32.

[8] M. A. I. Talukder, H. Shahriar, K. Qian, M. Rahman, S. Ahamed, F. Wu, E. Agu, Droidpatrol: a static analysis plugin for secure mobile software development, in: 2019 IEEE 43rd annual computer software and applications conference (COMPSAC), Vol. 1, IEEE, 2019, pp. 565–569.

[9] MobSF, Mobile security framework (mobsf) (2022).
URL https://github.com/MobSF/Mobile-Security-Framework-MobSF

[10] M. K. Alzaylaee, S. Y. Yerima, S. 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), IEEE, 2016, pp. 1–8.

[11] Y. He, X. Yang, B. Hu, W. Wang, Dynamic privacy leakage analysis of android third-party libraries, Journal of Information Security and Applications 46 (2019) 259–270.

[12] C. Schindler, M. Atas, T. Strametz, J. Feiner, R. Hofer, Privacy leak identification in third-party android libraries, in: 2022 Seventh International Conference On Mobile And Secure Services (MobiSecServ), IEEE, 2022, pp. 1–6.

[13] W. Chao, L. Qun, W. XiaoHu, R. TianYu, D. JiaHan, G. GuangXin, S. EnJie, An android application vulnerability mining method based on static and dynamic analysis, in: 2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC), IEEE, 2020, pp. 599–603.

[14] M. Xia, L. Gong, Y. Lyu, Z. Qi, X. Liu, Effective real-time android application auditing, in: 2015 IEEE Symposium on Security and Privacy, IEEE, 2015, pp. 899–914.

[15] M. InfoSecurity, Drozer (2022).
URL https://github.com/WithSecureLabs/drozer

[16] ac pm, Inspeckage (2022).
URL https://github.com/ac-pm/Inspeckage

[17] sensepost, Objection (2022).
URL https://github.com/sensepost/objection

[18] J. Garcia, M. Hammad, N. Ghorbani, S. Malek, Automatic generation of inter-component communication exploits for android applications, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 661–671.

[19] J. Mitra, V.-P. 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, Association for Computing Machinery, New York, NY, USA, 2017, p. 43–52. doi:10.1145/3127005.3127010.
URL https://doi.org/10.1145/3127005.3127010

[20] Sutton, Reinforcement Learning: An Introduction, MIT Press, 2014.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, Playing atari with deep reinforcement learning, arXiv preprint arXiv:1312.5602 (2013).

[22] W. Enck, M. Ongtang, P. McDaniel, Understanding android security, IEEE security & privacy 7 (1) (2009) 50–57.

[23] Google, Intent (2022).
URL https://developer.android.com/reference/android/content/Intent

[24] R. Hay, O. Tripp, M. Pistoia, Dynamic detection of inter-application communication vulnerabilities in android, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, 2015, pp. 118–128.

[25] F. Pagano, A. Romdhana, D. Caputo, L. Verderame, A. Merlo, SEBASTiAn: a Static and Extensible Black-box Application Security Testing tool for iOS and Android applications (10 2022). doi:10.36227/techrxiv.21261573.v1.
URL https://www.techrxiv.org/articles/preprint/SEBASTiAn_a_Static_and_Extensible_Black-box_Application_Security_Testing_tool_for_iOS_and_Android_applications/21261573

[26] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, Openai gym, arXiv preprint arXiv:1606.01540 (2016).

[27] S. R. Group, Soot - a framework for analyzing and transforming java and android applications (2022).
URL http://soot-oss.github.io/soot/

[28] A. Romdhana, A. Merlo, M. Ceccato, P. Tonella, Deep reinforcement learning for black-box testing of android apps, ACM Transactions on Software Engineering and Methodology (2022).

[29] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, Stable baselines, https://github.com/hill-a/stable-baselines (2018).

[30] T. Haarnoja, A. Zhou, P. Abbeel, S. Levine, Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor (2018). doi:10.48550/ARXIV.1801.01290.
URL https://arxiv.org/abs/1801.01290

[31] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, Stable baselines algorithms, https://stable-baselines3.readthedocs.io/en/master/guide/algos.html (2018).

[32] A. Sadeghi, H. Bagheri, J. Garcia, S. Malek, A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software, IEEE Transactions on Software Engineering 43 (6) (2016) 492–530.

[33] E. Chin, A. P. Felt, K. Greenwood, D. Wagner, Analyzing inter-application communication in android, in: Proceedings of the 9th international conference on Mobile systems, applications, and services, 2011, pp. 239–252.

[34] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, Y. Le Traon, Effective {Inter-Component} communication mapping in android: An essential step towards holistic security analysis, in: 22nd USENIX Security Symposium (USENIX Security 13), 2013, pp. 543–558.

[35] D. Octeau, D. Luchaup, M. Dering, S. Jha, P. McDaniel, Composite constant propagation: Application to android inter-component communication analysis, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1, IEEE, 2015, pp. 77–88.

[36] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, P. McDaniel, Iccta: Detecting inter-component privacy leaks in android apps, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1, IEEE, 2015, pp. 280–291.

[37] H. Bagheri, A. Sadeghi, J. Garcia, S. Malek, Covert: Compositional analysis of android inter-app permission leakage, IEEE transactions on Software Engineering 41 (9) (2015) 866–886.

[38] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, Acm Sigplan Notices 49 (6) (2014) 259–269.

[39] C. Cao, N. Gao, P. Liu, J. Xiang, Towards analyzing the input validation vulnerabilities associated with android system services, in: Proceedings of the 31st Annual Computer Security Applications Conference, 2015, pp. 361–370.

[40] A. P. Felt, E. Chin, S. Hanna, D. Song, D. Wagner, Android permissions demystified, in: Proceedings of the 18th ACM conference on Computer and communications security, 2011, pp. 627–638.

[41] P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, G. Vigna, A large-scale study of mobile web app security, in: Proceedings of the Mobile Security Technologies Workshop (MoST), Vol. 50, 2015.

[42] Y. Z. X. Jiang, Detecting passive content leaks and pollution in android applications, in: Proceedings of the 20th Network and Distributed System Security Symposium (NDSS), 2013.

[43] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, E. Chin, Permission re-delegation: Attacks and defenses., in: USENIX security symposium, Vol. 30, 2011, p. 88.

[44] J. Schutte, D. Titze, J. M. 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, IEEE, 2014, pp. 370–379.

[45] B. F. Demissie, M. 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, 2020, pp. 1–11.

[46] B. F. Demissie, M. Ceccato, L. K. Shar, Security analysis of permission re-delegation vulnerabilities in android apps, Empirical Software Engineering 25 (6) (2020) 5084–5136.