University of Padova

Department of Mathematics "Tullio Levi-Civita"

Master Thesis in Cybersecurity

Control Flow Graph-based Path Reconstruction in Android applications

Supervisor
Prof. Eleonora Losiouk
University of Padova

Master Candidate
Samuele Doria

Student ID
2057059

Academic Year
2022-2023

"There are three things that have meaning for life. They are the motivational factors for everything in your life, for anything that you do, or any living thing does: The first is survival, the second is social order, and the third is entertainment. Everything in life progresses in that order. And there is nothing after entertainment. So, in a sense, the implication is that the meaning of life is to reach that third stage. And once you've reached the third stage, you're done. But you have to go through the other stages first."

— Linus Torvalds

Abstract

Over the years, the field of Android security research has faced significant limitations due to the absence of reliable methods for achieving automated interaction with mobile applications. The lack of such tools has resulted in the widespread use of automatic exercising software, which randomly interfaces with apps in the hopes of obtaining desired outcomes. However, this approach cannot always be considered a satisfactory solution, as it lacks solid criteria and fails to provide any Proof-of-Reachability. In the context of my thesis, I employed Control Flow Graphs to reconstruct pathways that lead to specified target methods within Android applications. This approach allowed me to extract high-level instructions that automatic interaction software can accurately and reliably execute in order to reach a designated endpoint. Tests and evaluations conducted on this technique demonstrate its potential to facilitate more precise and goal-oriented testing. Its applications in the future could span from fuzzing and exploitation to aiding in the disclosure of privacy violations.

Contents

Listing of figures

Listing of tables

Listing of acronyms

OS . . . . . Operating System

CFG . . . . . Control-Flow Graph

GAPS . . . . . Graph-based Automated Path Synthesizer

PoR . . . . . Proof-of-Reachability

ART . . . . . Android Runtime

API . . . . . Application Programming Interface

HAL . . . . . Hardware Abstraction Layer

DEX . . . . . Dalvik Executable

UI . . . . . User Interface

VM . . . . . Virtual Machine

DVM . . . . . Dalvik Virtual Machine

APK . . . . . Android Package Kit

ODEX . . . . . Optimized DEX

ICC . . . . . Inter-Component Communication

BFS . . . . . Breadth-First Search

ADB . . . . . Android Debug Bridge

1 Introduction

Android, developed by Google, stands as a prominent Operating System (OS) in the realm of mobile technology. It has gained substantial traction owing to its prevalence in smartphones, tablets, diverse digital devices and even cars. For each of the previously mentioned instances, Android supports their core functionality and enables the execution of a myriad of applications, encompassing communication, gaming, navigation, and beyond. The extensive user base and widespread adoption of Android highlight its significance in modern computing. The ecosystem of Android, characterized by its open architecture, has prompted a rich landscape of research endeavors aimed at enhancing its security. Researchers diligently investigate methods to secure Android against a plethora of threats, encompassing malware, data breaches, and unauthorized access. Moreover, the preservation of user privacy and the safeguarding of sensitive information occupy a pivotal role in this research domain. Through these scientific undertakings, the robustness and trustworthiness of the Android platform continue to evolve, thereby fostering a more secure digital landscape for its diverse user base.

However, over the last few years the Android security community has suffered from a lack of an important contribution that, as stated in recent works [1] [2] [3], would greatly improve this research landscape: path reconstruction and precise automatic interaction. These topics are extremely relevant whenever reachability is in question. Some of the most import examples are in vulnerability analysis and exploitation, or even testing. Notably, this is most prominent in Android apps with a large code base; in the scenario where a vulnerability is found in one of them, a reverse-engineer would need to spend enormous manual effort in order to prove that

the bug can be actively triggered by an attacker. The intended objective of this endeavor is to establish a systematic approach that facilitates the seamless and lucid delineation of a path, starting from an initial entry point and culminating in a specific target method. The predominant challenge manifests when considering the growing intricacy inherent in Android applications, an evolution that has been ongoing since their initial iteration. Furthermore, despite similar efforts have been made in the past in other domains, the distinctiveness of Android apps from other forms of software enhances the difficulties encountered in this pursuit. In addition to this, another missing feature in the community at the moment is the capacity for automated interaction with the application, including carrying out high level operations (e.g., tapping) under the path's guidance, leading potentially to Proof-of-Reachability.
Over the years, some research publications have attempted to propose remedies in these particular fields. Worth noting are Flowdroid[4] and Amandroid[5], which, while initially designed for the purpose of taint analysis, also allow generating Control-Flow Graphs (CFGs), from which paths could be extracted. However, both were introduced respectively in 2014 and 2018 and nowadays encounter challenges in terms of scalability due to the increasing complexity of newer Android applications. Regarding automatic interaction, instead, solutions such as Monkey [6] and ARES [7] are heavily employed in pseudo-random testing. This practice, commonly referred to as *exercising*, seeks to increase code coverage to cause crashes or trigger specific app's functionalities. This is achieved through pseudo-random or model-guided interactions with the graphical components of the application. However, a notable drawback in this process is related to the absence of a Proof-of-Reachability, emerging from the lack of guidance during the testing procedure. Consequently, this can potentially lead to inaccurate outcomes.
In my Master thesis, I was able to build a prototype solution that aims to solve the reachability problem in Android app's analysis with a static approach. The tool, called GAPS (Graph-based Automated Path Synthesizer), seeks to make the following contributions:

- Provide a reliable and scalable method for generating Control-Flow Graphs of applications
- Statically validate the reachability of methods by constructing paths backwards starting from a target method to an entry point, from the CFG
- Address paths with conditional statements by finding and validating states in the app that satisfy the circumstances
- Produce high-level instructions to guide interaction and reproduce the execution of the crafted paths

- Autonomously execute the extracted operations and ascertain Proof-of-Reachability (PoR) upon successful contact with the target method.

Finally, GAPS was tested against AndroTest, an app dataset that is widely acknowledged within the research community [8] [3] [9] [10] [11] [12] as the benchmark in the domain of automatic testing. Furthermore, GAPS was able to reconstruct 25% of paths over the total that lead to an entry point and 33% of them, during automated testing, were able to produce a PoR. Similarly, the module that generates paths that are conditionally-satisfied was separately tested and achieved 60.71% reachable paths and 11.96% with a Proof-of-Reachability.

2
Background

2.1 Android's Architecture

Android's architecture, as referenced in Figure 2.1, encompasses a set of layers with their own specific functionalities. This distinction allows to clearly separate and assign responsibilities to different parts of the OS in a precise manner. Starting from the top:

- **System Apps** - This layer contains all the apps that are pre-installed in the Android device, some examples are those used for e-mail, SMS messaging, calendar, and more. Their functions can all be replaced by third-party applications, both downloaded or created by the users.
- **Java API Framework** - The Java API Framework layer contains all the Java classes in Android that can be used directly from the applications' code.
- **Native C/C++ Libraries** - This layer contains some binaries built on native code, mainly C and C++, that are useful for some core components, such as the Android Runtime and the Hardware Abstraction Layer.
- **Android Runtime (ART)** - Since Android 5.0, each app runs inside its own process and with its own instance of the Android Runtime (ART). ART executes the application's Dalvik Executable (DEX) files, a byte code format designed for Android, by relying on the native libraries.
- **Hardware Abstraction Layer (HAL)** - This layer provides a standardized interface for hardware components to be used by the Java API Framework. Some notorious examples

are the camera and Bluetooth modules: when access is required, the respective libraries are loaded by the system.

- **Linux Kernel** - The Linux Kernel is the foundation of the Android OS. A modified version is adapted for mobile devices.

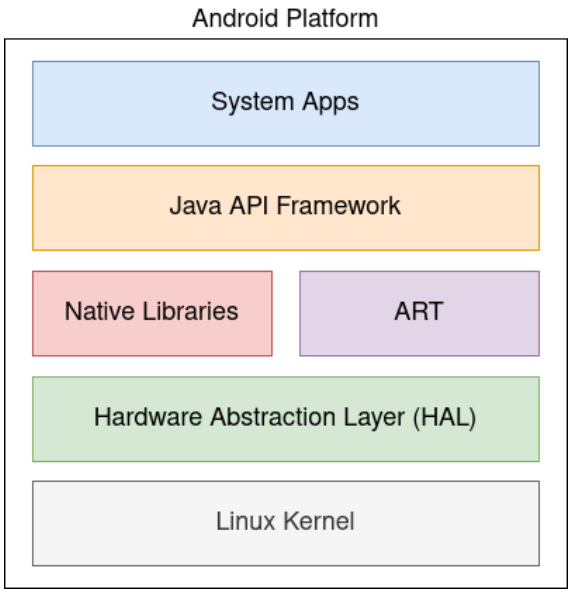

Figure 2.1: Layers of Android platform architecture

On the other hand, Android apps can be developed mainly using Java or Kotlin, while they can also implement their own Native Libraries using C or C++. Additionally, every application also includes a wide variety of XML files that are introduced to statically define User Interface (UI) elements or components. The most important among them is the *Android Manifest*, that is always present and contains essential information about the app. Alternatively, most of the statically defined features can be replaced by a programmatic approach.

2.2 App Components

Android applications, as expressed by the official developer guide [13], can be described by the interaction of four components. These building blocks can be found in most apps and are responsible for creating an internal ecosystem, interacting with the underlying OS and other software installed in the device. Each of them has its own life cycle that defines how it is created and destroyed (e.g., an Activity starts with `onCreate`, and it finishes with `onDestroy`).

The most popular one is the **Activity** component that represents the entry point in the app for the user. It consists in a single screen with a user interface but, despite being independent, when multiple are present they can work together to form a cohesive user experience. Every Activity can adopt and show a wide range of graphical components that add dynamic behavior in the app, some of the most popular examples are buttons, menus, switches, and many more. Every application must define a *Main Activity*, a special kind that is launched whenever the user taps on the app's icon, and that is defined in the *Android Manifest* XML file.
Services are general-purpose entry-points to an app that are used to run a task in the background. They can be defined in two different ways:

1. Started services, that tell the system to keep them running until their work is complete. A common examples is when music is being reproduced.

2. Bound services, that can be requested by other apps or the system itself. Their usage is similar to one of APIs and resemble them in how they are also able to build dependencies.

A **Broadcast Receiver**, instead, is a component that lets the app receive system-wide messages through a special type of object called *Intent*. They are also intended as another entry-point to the application, even when it is not currently running.
Lastly, **Content Providers** allow managing app data in the file system and define permissions for it. They also serve as an additional entry-point.
The one common denominator between all these elements are the *Intent* objects, used as messaging entities to request "actions" and that constitute Inter-Component Communication (ICC) on Android. Every component can register an *IntentFilter* either dynamically or statically and decide to receive these requests, both from other apps, the system or even other components in the same app. Notably, this is carried out by creating an *Intent* object and expressing either a fully qualified class name as a destination (e.g., an Activity) or a specific "action", which is a string used to reference a component (i.e., Broadcast Receivers, Services). Finally, to send the object it is passed as an argument to specifically designed Android API methods, for example `startActivity(Intent)` can be used to show on the screen an Activity.
The possibility of defining and characterizing components both programmatically and through XML files renders the practice of static analysis more complex, since the code used for the app is not the only source that needs to be considered. Additionally, applications differ vastly from any other type of software in the way that, for example, every C program must always define a `main` function, from which every execution starts from. On the other side, in Android there can be many entry points, as highlighted, from which different behaviors may carry out.

2.3 Smali Overview

Apps, after being written either in Java or Kotlin, go through an intermediate compilation process where an artifact is produced (e.g., `.class` file for Java). In Android, as represented in Figure 2.2, this is further adapted for the architecture through another compilation stage where a Dalvik Executable (DEX) file is obtained, in the format of Dalvik Bytecode. Subsequently, the code is executed by the Dalvik Virtual Machine (DVM), which nowadays is replaced by the Android Runtime (ART) on newer OS versions. [14] The last compilation process into a DEX file can be reversed to obtain an assembler-like representation of the app's code, in the form of *Smali* instructions. Smali [15] can be considered an equivalent to the assembly language for C programs. Since every Android application must contain at least one DEX file, the Smali code can always be retrieved and is able to offer a common representation, even if they were originally written using either Java or Kotlin.

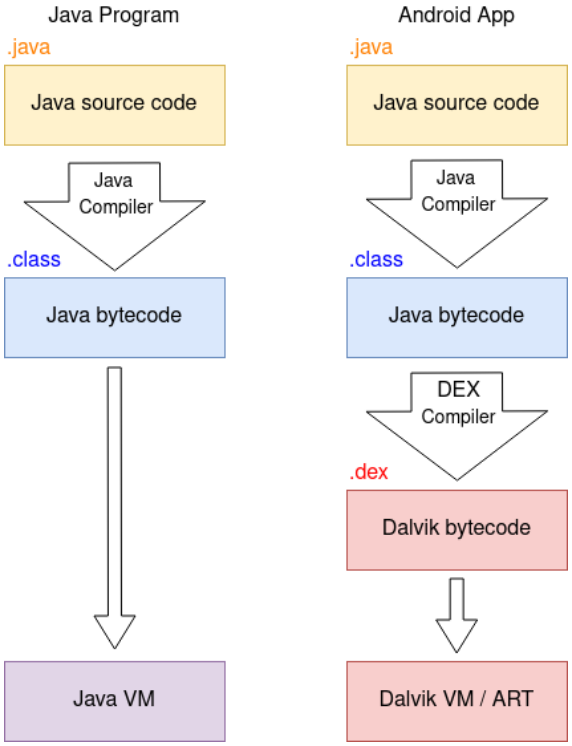

Figure 2.2: Android app compilation process compared to a Java program

Furthermore, code is structured around 32-bit registers, which are analogous to variables in higher-level languages. Registers are designated with the prefix "v" followed by a number (e.g.,

`v0`, `v1`). Smali employs a set of instructions that manipulate registers (e.g., `move`, `move-result`), perform arithmetic operations (e.g., `add`, `sub`), control execution flow (e.g., `goto`, `if` statements), and more. Moreover, smali is organized into methods, each corresponding to a function in high-level languages. Each procedure is defined with labels, and their parameters and return values are also represented through registers. Additionally, to reflect on the Java and Kotlin counterpart, it also supports Object-Oriented constructs. For instance, the invocation of a non-static method (e.g., an `invoke-direct` instruction) contains in its first register the reference to the caller object.

3 Design

GAPS is a static analysis tool capable of processing input applications, with the primary goal of reconstructing clear and accurate paths, culminating in desired methods, and subsequently verify their reachability. It is designed to be fully modular and customizable to make use of specific functionalities, such as: indicating a specific target method to test its reachability, generating interaction-oriented high-level instructions and deriving paths that satisfy conditional statements.

3.1 Illustrative Example

In order to better illustrate the way GAPS works on Android apps, an example can be considered. For instance, an app can be taken into account, containing the invocation of the `target_method` in a class *ActivityA*. More specifically, its execution depends on the tap of a button, represented by the `onClick` callback, that also contains a conditional statement that verifies the value of a boolean variable. Listing 3.1 provides a code sample.

```
1 public class ActivityA extends AppCompatActivity {
2     ...
3
4     protected void onStart() {
5         super.onStart();
6         buttonA.setOnClickListener((view) -> {
```

```
            if(access == True){
                target_method();
            }
        });
    }
}
```

Listing 3.1: *ActivityA* contains the invocation to `target_method`

Furthermore, *ActivityA* is shown on screen whenever the user click on another button in the *MainActivity*, which is one of the entry points of the app. More specifically, an *Intent* message is built in the callback and is subsequently sent through the *startActivity* API. For this case, Listing 3.2 can be consulted.

```
public class MainActivity extends AppCompatActivity {
    ...

    protected void onStart() {
        super.onStart();
        buttonMain.setOnClickListener((view) -> {
            Intent myIntent = new Intent(MainActivity.this, ActivityA.class);
            startActivity(myIntent);
        });
    }
}
```

Listing 3.2: The *MainActivity* starts *ActivityA*

In the next sections, this app example will be used to illustrate GAPS' inner working throughout its phases.

3.2 Requirements

In order for GAPS to produce the expected results, the following requirements need to be met:

- **Requirement 1** - Generate intra-procedural Control-Flow Graphs
- **Requirement 2** - Query the data structures to extract intra-procedural paths
- **Requirement 3** - Extract information related to Inter-Component Communication
- **Requirement 4** - Analyze the data flow

- **Requirement 5** - Build inter-procedural path
- **Requirement 6** - Retrieve IDs associated to graphical elements

3.3 Overview

The first step GAPS undertakes during its analysis is to extract for each method in the input app its smali code and build data structures that allow to traverse their Control-Flow Graphs (Requirement 1). Furthermore, from every instruction its numerical address can be retrieved, and it is considered in order to visit the method's CFG backwards and build linear sequences of directives in the form of paths (Requirement 2). The query system has a pivotal role also for Requirement 3, thus allowing to extract the app's code responsible for the creation and the dispatch of inter-procedural messages. On top of that, these objects are analyzed by considering their intra-procedural data flow (Requirement 4), in order to understand senders and recipients. Finally, intra-procedural paths are used as building blocks and assembled to form complete sequences that from a target instruction are able to lead backwards to an entry point (Requirement 5). Subsequently, the graphical elements' ID is retrieved (Requirement 6), thanks to Requirement 2 and 4, to achieve automatic interaction under the execution flow guidance.

3.4 Control-Flow Graphs Generation

After loading and analyzing the Android Package Kit (APK) or DEX file supplied in input, every method defined in the app is processed to obtain an intra-procedural CFG where every node is a smali instruction, thus satisfying Requirement 1. The edges are added by considering the progressive execution order that is followed normally, while more than one branch can also be present in case of control flow instructions (e.g., `if` statements, `goto`).
A pseudocode can be found in Algorithm 3.1.
The concept of external methods is mentioned in line 6 and is applied to refer to all the Android and Java API methods that are not present statically in the DEX files and instead are loaded during runtime. This category is, therefore, excluded at this stage. After that, for each method, its basic blocks can be retrieved and parsed (line 9). A basic block is defined as a sequence of contiguous instructions executed without interruptions. The same principles are applied to all the additional edges (e.g., conditional behavior, jumps), mentioned as *"childs"*.

Algorithm 3.1 Dalvik Disassembler

```
1: percentage_chance ← 0.20
2: max_random_methods ← 50
3: random_methods_counter ← 0
4: incremental_offset ← 0
5: for each method in app.get_methods()
6:     if method is external
7:         continue
8:     method_offset ← incremental_offset
9:     for each method_basic_block in method.get_basic_blocks()
10:        instruction_address ← method_basic_block.get_address() + method_offset
11:        instructions ← method_basic_block.get_instructions()
12:        for each instruction in instructions
13:            address_to_instruction[instruction_address] ← instruction
14:            if instruction is method_invocation or instruction is assignment_type
15:                instruction_signature ← instruction.get_signature()
16:                signature_to_address[instruction_signature].add(instruction_address)
17:            if instruction is method_invocation and "Intent" in instruction_parameters
18:                icc_methods_addresses.add(instruction_address)
19:            if target_method_search is True
20:                if instruction is method_invocation and instruction == target_method
21:                    instruction_signature ← instruction.get_signature()
22:                    starting_points[instruction_signature].add(instruction_address)
23:            else
24:                if random.random() < percentage_chance and random_methods_counter <
   max_random_methods
25:                    instruction_signature ← instruction.get_signature()
26:                    starting_points[instruction_signature].add(instruction_address)
27:                    random_methods_counter + +
28:            instr_to_parent_method[instruction_address] ← method
29:            next_instruction_address ← method_offset + instruction.get_length()
30:            instr_cfg[next_instruction_address] ← instruction_address
31:            instruction_address ← next_instruction_address
```

```
32:     for each child in method_basic_block.childs
33:         child_address ← child + method_offset
34:         instr_cfg[child_address].add(instruction_address)
35:         instr_to_parent_method[child_address] ← method
36:     incremental_offset + +
```

Consequently, all smali operations are used to build three essential data structures:

- `address_to_instruction`, allowing to translate from numerical to literal representation (line 13),
- `instr_to_parent_method`, which correlates every instruction address to the signature of the parent's method (line 28),
- `instr_cfg`, a dictionary that defines edges between the nodes (i.e., instructions' addresses) in the Control-Flow Graph (line 30).

Specifically, the last one is of essential importance during path reconstruction and is built such that every address used as key points to its predecessors (i.e., a destination points to its sources). This, more specifically, allows performing a backwards-search, meaning that the path reconstruction starts from the objective and, in reverse, tries to reach an entry-point.
Moreover, during disassembly, more information are stored and pre-fetched for easier access in later stages. These are represented by the following data structures:

- `signature_to_address`, that maps the signature of a method or related to an assignment instruction to its addresses (line 16),
- `icc_methods_addresses`, that acts similarly but for all the methods related to Inter-Component Communication (line 18),
- `starting_points`, which holds for every method invocation their corresponding numerical value (i.e., address) (line 22, 26).

These dictionaries allow taking advantage of the dynamic programming paradigm, that is: storing results of sub-problems instead of re-computing them later. More specifically, `icc_methods_addresses` guides the ICC mapping of the entire Android app subsequently, while `starting_points` bootstraps the path reconstruction phase. Furthermore, all of them prove to be an optimization with an `O(1)` cost, as opposed to performing a linear search over all the instructions (i.e., `O(N)`).

For instance, in the illustrative example, every method defined in the app will be disassembled and processed. More specifically, the address corresponding to the `startActivity` invocation in Listing 3.2 will be saved in `icc_methods_addresses`, and similarly the address of `target_method` from Listing 3.1 in `starting_points`.

3.5 Intra-Procedural Graph Visit

The data collected during the app's disassembly helps to bootstrap the intra-procedural path reconstruction process, which consists in performing a backwards visit of the Control-Flow Graph of a single method body. This process stands at a pivotal role for GAPS, and it is represented by Requirement 2. Each of the starting points found during the initial phase, whether they correspond to the target method invocation or by random selection, provide a set of addresses to start the backwards search for each method. To better illustrate the process, Algorithm 3.2 is provided. Aside from specifying a target method and class (line 6 to 9), the path reconstruction can also be guided by stating the fully qualified class name and signature method (line 4 and 5). Additionally, the `acyclic` and `avoid_explosion` boolean parameters, optionally specified, allow breaking cycles when a search has already been performed for the same path (line 10), and limit the size and number of alternative paths during the graph visit in the CFG (line 25), respectively. Going into more depth, the process involves the initial construction of a query identifier aimed at verifying whether it has been previously encountered among the elements requested (line 10). This search key subsequently also serves the purpose of retrieving partial outcomes from earlier saved invocations (line 12-13). This approach, akin to dynamic programming, proves effective in preventing redundant computations. Alternatively, if starting addresses have not been provided, either a linear search can be performed over all the instruction (line 17 to 19) or, if the signature is known, direct access to `signature_to_address` can be made (line 22). Once a set of addresses is found, a Breadth-First Search (BFS) over the Control-Flow Graph is performed (line 25). By choosing BFS over other techniques, such as Depth-First Search, the exploration prioritizes visiting most of the graph until a root is reached. In this instance, the search is happening intra-procedurally, and consequently halts upon reaching the initial instruction when no further additions are feasible. The algorithm used for visiting follows the standard BFS implementation, using the aforementioned `instr_cfg` data structure. However, it has been customized, especially when the `avoid_explosion` parameter is enabled, to constrain each path to a maximum of 200 smali instructions and permits the storage of up to 10 alternative paths simultaneously.

Algorithm 3.2 Smali Path Finding

Input

target_method: string
target_class: string, optional
target_signature: string, optional
acyclic: boolean, optional. default=False
starting_addresses: list, optional
avoid_explosion: boolean, optional. default=True

Output

paths: list

```
1:  if not target_method
2:      return array()
3:  search ← None
4:  if target_signature
5:      search ← target_signature
6:  else if target_class
7:      search ← target_class + target_method
8:  else
9:      search ← target_method
10: if acyclic is True and search in requested_queries
11:     return array()
12: if search in search_results
13:     requested_queries.add(search) return search_results[search]
14: if not starting_addresses
15:     starting_addresses = set()
16:     if not target_signature
17:         for each instruction in address_to_instruction
18:             if target_method == instruction.method and target_class == instruction.class
19:                 starting_addresses.add(address)
20:     else
21:         if target_signature in signature_to_address
22:             starting_addresses = signature_to_address[target_signature]
```

```
23: if starting_addresses is empty
24:     return array()
25: return breadth_first_search_graph(starting_addresses, acyclic, avoid_explosion, search)
```

This measure is designed in order to maintain a lighter memory during execution. Since it is performing a backwards visit, there might be cases where a path might unexpectedly terminate before reaching the method's top. For example, this can happen due to a `goto` instruction pointing to a previously visited section. Therefore, the ultimate collection of paths returned from this task is determined from the ones that come closest to the root.
This scenario is also present in the illustrative example, as the intra-procedural path starting from the invocation of `target_method` in Listing 3.1 depends on a conditional statement. Therefore, there will be two branches in the Control-Flow Graph, depending on whether the condition is either true or false, but only the variant that will lead to the desired execution will be saved.
Furthermore, the `instr_to_parent_method` dictionary consistently provides the ability to discern the parent method's signature based on the instruction's address. This dictionary is instrumental in completing the sequence effectively, since this signature is always considered in order to concatenate more paths inter-procedurally. In the example's case, this will correspond to the `onClick` signature.
Once the resulting paths are reconstructed, the `search` string is added to the requested queries and used to save the intermediate results under its key.

3.6 Inter-Component Communication Analysis

As mentioned in Chapter 2, Inter-Component Communication in apps is fundamentally important. Therefore, to properly build paths, this needs to be accounted for, as also represented by Requirement 3. More specifically, GAPS is able to map ICC based on the knowledge that to send the Intent object, it needs to be passed as an argument to a specific Android API. Under this basis, during disassembly, a data structure called `icc_methods_addresses` is created to save the method's addresses and retrieve their intra-procedural path subsequently. Furthermore, the objective is to analyze them and retrieve the sender and receiver of the message. As delineated in Algorithm 3.3, the initial step undertaken by GAPS involves the static retrieval of all actions attributed to the components (line 1 and 2).

Algorithm 3.3 ICC Information Retrieval

```
1:  for each component in app_components
2:      get_static_intent_filters(component)
3:  register_receiver_paths ← find_path_smali(registerReceiver)
4:  for each path in register_receiver_paths
5:      arguments ← get_instruction_arguments(path)
6:      for each argument in arguments
7:          if argument is object_name
8:              destination_class ← get_class_from_object(argument)
9:              intent_action ← get_action_intent_filter(path)
10:             icc[destination_class].add(intent_action)
11: icc_paths ← find_path_smali_icc()
12: for each path in icc_paths
13:     if intent_construction in path
14:         arguments ← get_instruction_arguments(path)
15:         for each argument in arguments
16:             if argument is class_name
17:                 icc[argument].add(path)
18:             else if argument is string
19:                 destination_class ← translate_action_to_class(argument)
20:                 icc[destination_class].add(path)
```

Subsequently, intra-procedural smali paths containing *registerReceiver* invocations are extracted (line 3). These invocations accept both a *BroadcastReceiver* and an *IntentFilter* objects as parameters. In the case of the former, the `get_class_from_object` method is employed (line 8) to ascertain the destination class, as this class is also responsible for implementing the receiver. Conversely, for the latter situation, the corresponding action is obtained in `get_action_intent_filter` (line 9). This acquired knowledge is subsequently used to create a mapping within the `icc` dictionary, associating the destination class with the corresponding action (line 10). To further collect all the dynamically reachable components, all the smali paths from `icc_methods_addresses` are used to find, during Intent construction, which receiver is specified, both explicitly (i.e., through a fully qualified class name) (line 16 and 17) or implicitly (i.e., through an action) (line 18 to 20). These paths are then saved in the aforemen-

tioned `icc` data structure under the corresponding destination class key.

3.7 Data Flow Analysis

In Algorithm 3.3 a specific technique is used in order to retrieve the arguments passed as parameters to a function, represented by invocations to `get_instruction_arguments`. This approach uses Data Flow Analysis (Requirement 4) by taking advantage of the register-based smali syntax. Furthermore, every invocation instruction for a method mentions a list of registers, where the first one can be recognized as the caller object in case of non-static calls, and the rest correspond to the parameters. Subsequently, GAPS takes advantage of the intra-procedural smali instructions in the paths to find deterministic register assignments and recover the method's arguments. A pseudocode of this functionality can be consulted in Algorithm 3.4.
More specifically, this procedure requires the intra-procedural path that is being considered along with the index where the target instruction resides. From this point, the registers associated are extracted (line 3), and the goal is to find for each of them the assigning directive.
Optionally, it is possible to provide the `ignore_caller` argument to exclude the first argument (e.g., caller of the method if it is not static) or `only_caller` to be the only one considered. Subsequently, every instruction in the path is parsed to check its registers and type (line 10 to 13), more specifically, all those that overwrite values. For each, the instruction itself and its index are saved (line 42 to 44) in the dictionary that is returned at the end of the function. Additionally, if the instruction is interacting with a register without overwriting it, this information is also saved and tagged as "additional" (line 46), which is a useful feature for studying arguments behavior before they enter an app's method. This procedure is also adopted to find paths that satisfy conditional statements but in a *constant propagation* fashion. Constant propagation is a compiler optimization technique used to enhance the efficiency and performance of code execution. It aims to replace variables with their constant values whenever possible, reducing unnecessary computations and improving the overall speed of the program. This feature is implemented by recursively using the data flow analysis on paths by tracking value assignments.
In the example app, this process, combined with the ICC one previously mentioned, allows to accurately map the creation of the Intent object that permits the *MainActivity* to transition to *ActivityA*.

Algorithm 3.4 Data Flow Analysis

Input

path: list
start_from: int
ignore_caller: boolean, default=False
only_caller: boolean, default=False

Output

arguments: dictionary

```
1:  results ← dictionary()
2:  target_instruction ← path[start_from]
3:  registers ← get_registers(target_instruction, ignore_caller, only_caller)
4:  for each register in registers
5:      if register not in path[start_from+1:]
6:          registers.remove(register)
7:  if registers is empty
8:      return results
9:  to_translate ← dictionary()
10: for (i=start_from+1; i < path.length; i++)
11:     instruction ← path[i]
12:     instruction_type ← get_instruction_type(instruction)
13:     instruction_registers ← get_registers(instruction)
14:     if instruction_registers[0] in registers
15:         register ← instruction_registers[0]
16:         to_remove ← None
17:         instruction_found ← None
18:         instruction_found_index ← −1
19:         if instruction_type is "move result"
20:             to_remove ← register
21:             instruction_found ← path[i + 1]
22:             instruction_found_index ← i + 1
23:         else if instruction_type is "move between registers"
24:             registers.append(instruction_registers[len(instruction_registers) − 1])
25:             to_remove ← register
```

```
26:         if register not in to_translate
27:             to_translate[instruction_registers[len(instruction_registers) − 1]] = instruction_registers[0]
28:         else
29:             prev,eg ← to_translate[register]
30:             to_translate[instruction_registers[len(instruction_registers) − 1]] = old_reg
31:             to_translate.remove(register)
32:     else if instruction_type is "overwriting register"
33:         to_remove ← register
34:         instruction_found ← path[i]
35:         instruction_found_index ← i
36:     else
37:         instruction_found ← path[i]
38:     if instruction_found
39:         register_found ← register
40:         if register in to_translate
41:             register_found ← to,ranslate[register]
42:         if to_remove
43:             results[register_found][instruction] ← instruction_found
44:             results[register_found][instruction_index] ← instruction_found_index
45:         else
46:             results[register_found][additional_instructions].append(instruction_found)
47:     if to_remove
48:         registers.remove(to_remove)
49:   if registers is empty
50:     break
51: return results
```

3.8 Path Reconstruction

The partial intra-procedural paths previously obtained can be intended as "seeds" to start the inter-procedural reconstruction that also takes advantage of the pre-computed ICC information stored. This process is represented by Requirement 5. The pseudocode for this procedure can be consulted in Algorithm 3.5.
In this phase, path reconstruction is performed in BFS fashion too, since alternative paths are stored in order to resume their visit in the next iterations. In the innermost loop (line 10 to 24), the path is built by first considering the retrieval of graphical elements IDs (line 12) that might be associated to a callback (e.g., *onClick*). Subsequently, the `find_next_paths` function tries and concatenate new "building blocks" to the intermediate results (line 13) by considering explicit invocations of the last path's parent method. To further elaborate, this means that the parent method's signature will be used as a reference to find its usages over the whole app code, which consists in finding the possible callers. In Android, this is not always possible, since there might be classes that instead use implicit calls, for example those associated to components' life cycles. Therefore, GAPS interprets their instances based on their correct usage to append the corresponding code portions. For instance, *AsyncTasks* are used for multithreaded execution, therefore if while building a path the last path placed is one of its methods (e.g., `onPostExecute`), GAPS searches and add the portions of code that initialize and start the object (i.e., the method `execute`). The same practice is repeated similarly for *Handler* and *Thread* classes. Additionally, the both dynamic and static nature of the *Fragment* is also accounted for, offering additional insights to address its dual nature. This process is iteratively performed and the `add_partial_paths` method adds the first intra-procedural result to the current path (i.e., corresponding to `path_index`) (line 16 and 22), while the rest, if more than one is present, is used to generate alternatives that are completely visited in the next cycles. These additions are prioritized over the ICC ones through the `skip_icc` variable (line 18). Moreover, Inter-Component paths are retrieved using the `icc` data structure mentioned in Section 3.6 only when other programmatic interactions cannot be found, as previously explained. Finally, if during the last iteration there were not any new additions (line 23 and 24), the last path appended is checked to see if any root in the graph was reached (line 25). GAPS considers a root either reaching the app's main activity or a component that is reachable from outside (i.e., *BroadcastReceiver*). Furthermore, the initiation of conditionally-satisfiable path generation hinges on two conditions: the `conditional` flag being set to True and the attainment of an entry point (line 26 and 27).

Algorithm 3.5 Inter-procedural Path Reconstruction

Input

partial_paths: list

conditional: boolean

Output

paths list

```
1:  set_paths ← set()
2:  paths ← array()
3:  path_index ← 0
4:  for each partial_path in partial_paths
5:      paths.append(partial_path)
6:      last_path ← partial_path
7:      complete ← False
8:      while not complete
9:          path_available ← True
10:         while p ath_available is True
11:             skip_icc ← False
12:             find_graphical_id(last_path)
13:             paths_to_add ← find_next_paths(last_path)
14:             last_path ← array()
15:             if paths_to_add is not empty
16:                 last_path ← add_partial_paths(paths_to_add, paths, path_index)
17:                 if last_path is not empty
18:                     skip_icc ← True
19:             if skip_icc is False
20:                 icc_path ← find_icc_paths(last_path.last_class_name)
21:                 if icc_path is not empty
22:                     last_path ← add_partial_paths(icc_path, paths, path_index)
23:             if last_path is empty
24:                 path_available ← False
25:         root_reached ← is_root_reached(last_path)
26:         if conditional is True and root_reached is True
27:             find_conditional_paths(paths, path_index)
28:         set_paths.add(paths[path_index])
```

```
29:       if path_index != paths.length
30:         path_index + +
31:         last_path ← paths[path_index].last_path
32:       else
33:         complete ← True
34:     path_index + +
35: return list(set_paths)
```

Consequently, in the event that alternative paths are queued for completion, the subsequent iteration selects one and formulates the next path (line 29 to 31). Conversely, if no alternative paths await (line 33), the next partial path is considered, also known as the next "seed" (line 34). The set of paths collects all the results and return them as a list of unique elements (line 28). At this stage, the illustrative example analysis has a clear understanding of the execution flow, obtained by assembling the building blocks retrieved in the previous examples. More specifically, the information at hand would be found in the following order:

- `target_method` is executed in *ActivityA*'s `onClick` method,
- *ActivityA* is reachable by the *MainActivity*, which is one of the app's entry points and constructs an Intent for this purpose,
- the Intent construction and dispatch is bound to another `onClick` event, on the *MainActivity*'s UI.

3.9 Graphical Elements' ID Retrieval

In Android app development, *callbacks* are methods that are invoked whenever certain events are triggered. The most common examples are the implementation of buttons in applications that execute a series of operations whenever they are clicked. As mentioned in Section 3.8, GAPS during the path reconstruction phase tries to retrieve the graphical elements' ID in order to accessorize the results. This additional information, represented by Requirement 5, allows carrying out automatic interaction and guide a more precise and objective oriented app exercising. In this scenario, it needs to be noted that visual objects can be defined both dynamically and statically, too. In the former case, knowledge about how callbacks are used is needed, as each of them is associated with a listener class that triggers the action whenever the requested

event happens. For example, the *onClick* method is implemented inside an *OnClickListener* object, that is passed as argument to the *setOnClickListener* method. Therefore, GAPS, using its smali intra-procedural path retrieval capabilities and data flow analysis, can retrieve the portion of code that sets the listener and find the caller, which is also the graphical element's object. This one, when created, is also associated with an ID, therefore leading to the desired objective. A practical code samples can be consulted in Listing 3.3.

```
Button button = (Button) findViewById(R.id.button1);

button.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        target_method();
    }
});
```

Listing 3.3: Graphical Element example

This mechanism is able to work on every graphical element that is similarly implemented, which includes buttons but also switches, text fields, checkboxes and more. On top of this, GAPS is also capable of recognizing callbacks associated to menu screens. In Android, these can usually be found in the top-left corner and, when clicked, can show a list of entries to choose from. In order to decide which one would be the correct one to interact with, in smali the ID of the graphical element is compared and, depending on the value, execution jumps to a different section of the code. Therefore, since path building is performed backwards, GAPS can match the current invocation investigated to the matching entry's ID.
Lastly, callbacks can also be statically specified in the activities XML files, where all the graphical elements are listed. In these cases, a lookup is performed in the corresponding record, where matching the method name also leads to the ID retrieval as it is one of the values in the XML tag.
Following the example app's analysis, during this step, the aforementioned operations are performed in order to discern the two separate IDs connected to the `onClick` callbacks invocations.

3.10 Conditionally-Satisfiable Path Reconstruction

During path reconstruction, as mentioned in Section 3.8, it is possible to analyze the conditional statements encountered and retrieve paths that satisfy them. Just like in many program-

ming languages, these instructions involve the comparison between two elements or against zero. Currently, GAPS supports conditional statements that involve:

1. variables, primitive types such as `int`, `String`, `float`, and more
2. objects, which can be compared against `null`, represented by `0` in smali, or against another object (i.e., aliasing, pointing to the same address)
3. methods, in case they are recognized as "getters", thus considering the "setter" counterpart (e.g., `getString` becomes `setString`)

The tool uses the dynamic flow analysis previously mentioned to first retrieve the signatures of the elements involved. Then, by using this identifying string, GAPS can take advantage of the information stored during disassembly (Chapter 3), referenced by the `signature_to_address`, to retrieve the intra-procedural paths. Furthermore, the register corresponding to the value appointed is analyzed and, if the element gets an assignment from another variable, constant propagation is performed in order to recursively resolve for all the bindings in the whole app's code. Finally, when all constant values are retrieved, a check is performed for the satisfiability of the conditional statement. In the case that the outcome is positive, then the partial paths can be fully reconstructed to provide, as a final result, a set of initial instructions that need to be performed before the main path is executed. Additionally, if the conditional statement can be satisfied in more than one unique way, alternative conditionals paths are also appended and can act as a fallback approach.
For example, the sample app mentioned in Section 3.1 will see a combination of data flow analysis and constant propagation applied in order to find a satisfying state in the code that is able to fulfill the conditional statement, shown in Listing 3.1. More specifically, every assignment involving the `access` variable will be analyzed to find those that set its value to `True`. If the condition depends on another assignment, the process is recursively performed until a constant is found. This procedure is used in order to improve chances of reachability for a target method.

3.11 Automatic Interaction

The last step of this generation phase is the extraction of high-level instructions, which encompass all the previously retrieved graphical elements' ID that need to be tapped during automatic interaction. Additionally, if the path requires contacting an exported component through an intent, specifications in order to do this are also included. Subsequently, these operations are

performed by a separate GAPS's module that is able to recognize the current activity and tap elements on the screen based on their ID, along with assembling and sending Intents, supporting both real devices and emulators.
For the example followed throughout the chapter, the automatic interaction steps that would need to be performed consist in:

- performing any action that would guarantee `access` to be `True`, as a prerequisite
- launching the app, i.e., starting from the *MainActivity*
- clicking the correct button in the screen
- tapping another button in *ActivityA* corresponding to the ID found

Subsequently, a Proof-of-Reachability of the correct execution of `target_method` is expected.

4 Implementation

GAPS is a Python-based tool that is able of extracting the application's smali code to derive and summarize their inner workings, ultimately to reconstruct paths and abstract high-level instructions out of them for automatic interaction purposes. A search can be guided by a target method specified through a command line argument, or, if it is not indicated, a random set of 50 invocations is chosen during the disassembling phase.

4.1 CFG and Path Generation

At its core, the Androguard Python library is used. Androguard[16] is a reverse-engineering and pentesting tool for Android applications. It can be used by command line, through a graphical frontend or purely as a library to implement custom scripts. Furthermore, it supports all major app formats, most notably APK, DEX and Optimized DEX (ODEX), offering also the possibility of decompiling and disassembling their code, and analyze XML and resource files. Androguard in GAPS is used to load and analyze the APK or DEX files in input through some built-in functions: `AnalyzeAPK` and `AnalyzeDex`, that permit accessing information contained in the app. More specifically, the `Analysis` object obtained from this procedure allows analyzing the `MethodAnalysis` construct for each of the methods and, consequently, access their basic blocks. During this disassembly phase, showcased in Chapter 3, all the data structures that are created are saved to a *GAPS* class instance, that orchestrates all operations in the framework and represents the current analysis. These objects are used in many occurrences,

some examples are the intra-procedural graph visit, the ICC analysis and the path reconstruction, mentioned in Chapter 3. It is important to acknowledge that the actual addresses of smali instructions are substituted with an alternative system utilizing an incremental offset. This adjustment is necessary due to instances where the same address would recur multiple times. The reason behind this behavior becomes evident upon considering that all APKs can encompass multiple DEX files and a single one has the limitation of accommodating only up to 65,536 methods. Moreover, this constraint results in occasional overlapping of the address space in different DEX files.

4.2 Graphical Elements' ID retrieval

The app disassembly is required for graphical elements' ID retrieval and is also performed by external tools, more specifically *Apktool*[17] for APKs and *baksmali*[15] for DEX files. All the resulting files from this process are temporarily saved in the `/tmp` directory and accessed during static analysis phases that require information that, currently, cannot be retrieved solely through Androguard. For instance, this is required when retrieving the graphical elements' ID, since in smali, after the process explained in Section 3.9, a numerical version is obtained. In order to convert this to a literal representation, GAPS consults a file produced by Apktool called *public.xml*, where the string IDs are stored.
Furthermore, menus in Android are usually implemented through a single method with a `switch-case` construct that confronts the identifier connected to the entry and executes the corresponding portion of code. Meanwhile, the smali counterpart is represented by a `sparse/packed-switch` that complicates a bit the syntax by using labels and a table where for each ID a jump corresponds. Currently, Androguard is not able to provide this degree of information, therefore static analysis on the smali files disassembled by Apktool is performed. More specifically, the label associated to the targeted portion of code is retrieved and used as a reference to find the graphical elements' ID.
Apktool also generates a decoded version of the Android Manifest file that is consulted during the Inter-Component analysis to recognize components that are exported, meaning they can be contacted with an Intent from outside the application.

4.3 Automatic Interaction

The GAPS module for automatic interaction uses the high-level instructions generated during path reconstruction in a json format to test the target's reachability. This script requires a device or an emulator connected via Android Debug Bridge (ADB), which is used initially to install the app's APK. Then, for each entry in the json file, the key of the internal dictionary represents the signature of the method considered, and the values are the set of operations that should lead to it. All the paths should either start from the main activity, which is also the default behavior, or by sending an intent to an exported component (e.g., an Activity) or through an implicit intent broadcasted to the system. Subsequently, each tap on a graphical element is represented by a pair of strings, consisting in the name of the Activity where it needs to performed and the ID itself. In order to check that the current UI displayed is the correct one, the ADB utility, through the command `adb shell dumpsys window`, can provide information about the focused Activity name. Since apps may display unpredictable behaviors (e.g., loading screens, ads), attempts to recognize the correct screen are performed every 5 seconds for a total of 5 times. Then, the AndroidViewClient [18] Python library is adopted for tapping the correct graphical element based on the specified identifier. Optionally, through the *frida* command line argument for the module, it is possible to exercise the app and receive a Proof-of-Reachability response when the method is executed through Frida [19], a dynamic instrumentation framework that can trap invocations to modify their behavior. When enabled, if the device is rooted and Frida is installed, GAPS can automatically create a Javascript file to add a hook for the target method and receive a message upon arrival, proving that the path was correctly visited.

5 Evaluation

Evaluation of automatic interaction tools on Android, in the last few years, has been largely performed on a popular dataset called "AndroTest", as attested by the numerous mentions in literature [8] [3] [9] [10] [11] [12]. Since AndroTest was originally created in 2015 and is not available anymore through its authentic source, I manually re-created the dataset by downloading the applications in their latest versions. In some cases, however, this was not possible since the original developer deleted their repository. The lack of some of the original applications was compensated by adding additional newer ones found on F-Droid[20], an open-source app repository. The full list of apps used in the evaluation can be consulted in Table 5.1. Using this set is especially useful for tools like GAPS because:

- it is composed of real-world open source applications published by developers on the F-Droid repository,
- apps are simple enough to test some of the most popular functionalities in Android (e.g., interactions with graphical components) but without the struggle of managing complex UIs (e.g., login screens, advertisement),
- obfuscation can be ignored, since apps can be compiled from their source code.

To further illustrate on the last point, obfuscation is a term commonly used in Android app development to refer to the practice of reducing DEX file sizes by shortening names of classes and members. Although this feature does not currently constitute a limitation for GAPS, in

this initial prototypical phase I decided to avoid the increasing complexity that this would bring while analyzing the results.

#	App name	App Size	PlayStore Downloads
1	A2DP Volume	2.7M	100.000+
2	aagtl	328K	not available
3	Aard2	4.1M	10.000+
4	Addi	2.2M	not available
5	ADSDroid	100K	not available
6	aGrep	348K	10.000+
7	AirGuard	21M	100.000+
8	Alarm Clock	180K	10.000+
9	aLogCat	148K	not available
10	Amazed	29M	not available
11	Androidmatic Keyer	88K	not available
12	AnyCut	32K	not available
13	AnyMemo	4.7M	100.000+
14	App Cache Cleaner	6.3M	100.000+
15	Auto Answer	100K	not available
16	Based Cooking	23M	50+
17	Battery Dog	24K	10.000+
18	Bites	96K	1000+
19	Blokish	12M	100+
20	Book Catalogue	3.3M	100.000+
21	DeepL	2.1M	5 Mln+
22	DuckDuckGo	64M	10 Mln+
23	Easy XKCD	17M	50.000+
24	F-Droid Client	12M	not available
25	Flipper Zero	38M	100.000+
26	Frozen Bubble	8.4M	1 Mln+
27	Hacki	26M	1000+
28	HotDeath	7.7M	not available
29	KeePass2	32M	1 Mln+
30	Linux Command Library	22M	1 Mln+

#	App name	App Size	PlayStore Downloads
31	Lock Pattern Generator	132K	5 Mln+
32	LolCatBuilder	44K	not available
33	ManPages	60K	1000+
34	Mileage	376K	50.000+
35	Mini Note Viewer	164K	not available
36	Multi SMS Sender	84K	500.000+
37	Munch Life	420K	not available
38	MyExpenses	39M	1 Mln+
39	MyLock	52K	10.000+
40	Nectroid	200K	not available
41	PasswordMaker	4.5M	5000+
42	PCAPDroid	11M	100.000+
43	Sanity	628K	500+
44	Stoic Reading	9.1M	10+
45	SwiFTP	6.1M	1000+
46	Weight Chart	116K	not available
47	WhoHasMyStuff	2.8M	5000+
48	Wikipedia	24M	50 Mln+
49	WorldClock	1.2M	100.000+

Table 5.1: List of apps used for GAPS evaluation

In the following sections, results of GAPS' analysis are illustrated, more specifically:

- the first measurements concern path reconstruction capabilities, assessing the analysis time and the number of paths that are able to reach an entry point over the total generated
- the latter is related to the assessment of the automatic interaction, that will test the reachability of the previously generated paths to gather a Proof-of-Reachability for each of them.

In each case, the analyses were carried out both with and without the conditionality module enabled. The motivation behind this choice is that finding paths that satisfy conditional statements can be quite computationally cumbersome, and can therefore be optionally enabled by the user through the command line arguments.

5.1 Evaluation on Path Reconstruction

In the initial testing phase, GAPS selects for each app a total of 50 random target methods, and its path reconstruction feature is evaluated by measuring:

- the time in seconds it takes to complete the building of paths and the extraction of high-level instructions,
- the total number of paths generated,
- the number of paths that are considered reachable, meaning that starting from an entry point in the app (i.e., main activity or an exported component) the target method is accessible.

Tests were run on an Ubuntu 20.04 VM on a server with 64GB of RAM.

5.1.1 GAPS without Conditionality Module

All the applications in the dataset are correctly analyzed, and their statistics can be consulted in Table 5.2.

App name	Analysis time (seconds)	Reachable paths	Total paths
A2DP Volume	90,81	845	1338
aagtl	7,74	66	179
Aard2	43,10	0	531
Addi	12,16	18	67
ADSDroid	5,23	0	177
aGrep	4,02	51	90
AirGuard	194,73	0	204
AlarmClock	5,15	38	141
aLogCat	5,39	94	104
Amazed	4,62	34	138
Androidmatic Keyer	4,69	17	52
AnyCut	3,12	23	26
AnyMemo	64,33	0	291
App Cache Cleaner	73,27	59	78
Auto Answer	3,30	4	7

App name	Analysis time (seconds)	Reachable paths	Total paths
Based Cooking	16,16	0	480
Battery Dog	2,76	5	7
Bites	3,90	51	55
Blokish	61,21	136	290
Book Catalogue	23,54	0	231
DeepL	37,76	0	457
DuckDuckGo	292,29	0	218
Easy XKCD	85,75	0	82
F-Droid Client	191,76	48	308
Flipper Zero	324,87	0	70
Frozen Bubble	10,33	255	657
Hacki	35,72	0	56
HotDeath	5,07	0	82
KeePass2	150,28	24	720
Linux Command Library	99,89	0	65
Lock Pattern Generator	3,53	29	43
LolCatBuilder	3,78	32	40
ManPages	3,00	15	18
Mileage	7,37	0	156
Mini Note Viewer	5,17	190	220
Multi SMS Sender	3,62	31	35
Munch Life	2,94	4	4
MyExpenses	510,74	0	56
MyLock	3,64	23	73
Nectroid	3,93	42	84
PasswordMaker	57,78	3	53
PCAPDroid	134,21	55	464
Sanity	5,85	60	247
Stoic Reading	68,12	44	48
SwiFTP	70,97	9	188
Weight Chart	3,54	34	74
WhoHasMyStuff	35,62	2	63

App name	Analysis time (seconds)	Reachable paths	Total paths
Wikipedia	164,73	0	52
World Clock	3,79	18	24

Table 5.2: Results on GAPS path reconstruction without conditionality enabled

Furthermore, some statistics can be observed in Table 5.3. Overall, it took GAPS around 50 minutes to inspect all the samples, averaging 60.31 seconds per app. Moreover, a total of 9143 paths were generated in total, around 186 each, and among those, the paths that are considered to be reachable are 2359, which are 48.14 per app on average. Additionally, 25% of the paths reconstructed are able to draw a clear connection between one of the app's entry points and a randomly selected target method.

	Analysis time (seconds)	Reachable paths	Total paths
Total	2955,28	2359	9143
Average	60,31	48,14	186,59

Table 5.3: Statistics on GAPS path reconstruction without conditionality enabled

5.1.2 GAPS with Conditionality Module

By enabling the conditionality module, GAPS' analysis complexity grows. More specifically, this depends on how intricate the conditional statements is, since it may involve, for instance, two variable whose value involves a long chain of assignments. In these cases, constant propagation is recursively performed in order to retrieve the possible values assigned over the whole app code. Therefore, this can quickly become exponentially complex and lead to out-of-memory crashes. For this reason, not all the apps from the dataset have been correctly analyzed in this modality, thus fully analyzing 39 out of 49. A list can be consulted in Table 5.4. It needs to be noted that the set of target methods tested were randomly selected and, therefore, are different from the ones previously mentioned.

App name	Analysis time (seconds)	Reachable paths	Conditional paths	Total paths
A2DPVolume	1100,79	817	3179	3594
Aard2	168,57	0	0	917
Addi	10,72	0	21	74

App name	Analysis time (seconds)	Reachable paths	Conditional paths	Total paths
ADSDroid	5,95	0	1	293
Air Guard	466,22	0	8	258
Alarm Clock	25,67	1	915	1013
aLogCat	9,59	395	439	452
Amazed	4,26	0	53	302
Androidmatic Keyer	5,90	16	21	68
AnyCut	2,91	0	21	24
AnyMemo	251,04	0	0	324
App Cache Cleaner	114,07	621	703	724
Auto Answer	3,14	0	9	22
Based Cooking	17,18	0	0	440
Battery Dog	3,58	2	13	21
Bites	3,93	0	65	74
Book Catalogue	111,60	0	0	387
DeepL	66,48	0	2	1184
DuckDuckGo	17684,11	0	0	377
Easy XKCD	296,19	0	0	221
Frozen Bubble	68,63	162	2445	2760
Hacki	67,16	0	0	291
Linux Command Library	120,42	0	0	183
Lock Pattern Generator	3,22	0	38	68
LolCatBuilder	6,24	54	122	133
Man Pages	3,16	0	21	23
Mileage	10,39	0	0	276
Mini Note Viewer	36,35	8	960	1060
Multi SMS Sender	4,04	3	56	111
Munch Life	3,36	0	4	4
MyExpenses	1715,99	0	0	77
MyLock	20,86	271	1356	1398
Nectroid	12,21	108	264	330
Password Maker	165,15	176	397	509
Sanity	15,73	27	215	516

App name	Analysis time (seconds)	Reachable paths	Conditional paths	Total paths
Stoic Reading	77,94	0	49	74
Weight Chart	4,73	1	33	120
WhoHasMyStuff	47,44	0	2	111
World Clock	3,96	2	40	48

Table 5.4: Results on GAPS path reconstruction with conditionality enabled

Compared to previous analysis modality, Table 5.5 shows that the execution time grows drastically, lasting around 6.5 hours and averaging almost 10 minutes per app. Furthermore, out of the total number of paths reconstructed, 60.71% are reachable from an entry point. A subset of 2664 of these has a set of paths found by GAPS that satisfy conditional statements, which out of 11452, constitutes the 23%.

	Analysis time (seconds)	Reachable paths	Conditional paths	Total paths
Total	22738,90	11452	2664	18861
Average	583,05	293,64	68,31	483,62

Table 5.5: Statistics on GAPS path reconstruction with conditionality enabled

5.1.3 Limitations

Several of the most common factors contributing to the inability to fully reconstruct a path and reach an entry point are intrinsically linked to the complexity of components within Android apps. GAPS is currently supporting the most frequently used constructs, such as AsyncTasks and Fragments, but every application can implement their own and have them behave uniquely. Furthermore, at the current state, Inter-Component Communication is only working when the Intent object is built inside the same method that sends it (i.e., as a result of intra-procedural analysis). While at the moment these limitations are influencing negatively the results, with enough engineering effort GAPS can overcome them and improve in the future.
In the case of the conditionality module enabled, the increase in computational complexity also needs to be accounted for. Therefore, future improvements should consider limiting the amount of paths parsed, in order to reduce memory usage.

5.2 Evaluation on Automatic Interaction

The second evaluation performed on GAPS consists on measuring correctness in producing Proof-of-Reachability (PoR) following the high-level instructions generated during the path reconstruction phase. Therefore, the following results are a direct consequence of the previously illustrated sections. Tests were performed on a rooted Pixel 2 device with Frida running Android 8.1.

5.2.1 GAPS without Conditionality Module

A comprehensive list of all the apps can be found in Table 5.6.

App name	Paths with PoR	Total interactive paths
A2DP Volume	31	63
aagtl	0	0
Aard2	0	0
Addi	0	1
ADSDroid	0	0
aGrep	13	44
Air Guard	0	0
AlarmClock	6	21
aLogCat	15	19
Amazed	0	0
Androidmatic Keyer	0	0
AnyCut	0	11
AnyMemo	0	0
Appca Cache Cleaner	0	8
Auto Answer	0	2
Based Cooking	0	0
Battery Dog	0	0
Bites	0	0
Blokish	0	16
Book Catalogue	0	0
DeepL	0	0
DuckDuckGo	0	0

App name	Paths with PoR	Total interactive paths
Easy XKCD	0	0
F-Droid Client	0	9
Flipper Zero	0	0
Frozen Bubble	9	23
Hacki	0	0
HotDeath	0	0
KeepPass2	0	24
Linux Command Library	0	0
Lock Pattern Generator	7	9
LolCatBuilder	0	0
ManPages	1	8
Mileage	0	0
Mini Note Viewer	5	31
Multi SMS Sender	8	12
Munch Life	0	0
MyExpenses	0	0
MyLock	0	10
Nectroid	1	1
Password Maker	0	0
PCAPDroid	0	51
Sanity	1	49
Stoic Reading	43	44
SwiFTP	1	9
Weight Chart	0	21
WhoHasMyStuff	0	0
Wikipedia	0	0
World Clock	6	10

Table 5.6: Results on GAPS automatic interaction without conditionality enabled

In Table 5.7 it is shown that 445 unique paths with high-level instructions were extracted, from which 33% (147) produced a Proof-of-Reachability through Frida upon executing the target method.

	Paths with PoR	Total interactive paths
Total	147	445
Average	3,06	0,27

Table 5.7: Statistics on GAPS automatic interaction without conditionality enabled

5.2.2 GAPS with Conditionality Module

Similarly, apps that completed the path reconstruction analysis were also tested, and the full list is provided with Table 5.4.

App name	Paths with PoR	Total interactive paths
A2DPVolume	31	495
Aard2	0	0
Addi	0	0
ADSDroid	0	0
Air Guard	0	8
Alarm Clock	6	17
aLogCat	15	46
Amazed	0	0
Androidmatic Keyer	0	10
AnyCut	0	10
AnyMemo	0	0
App Cache Cleaner	0	25
Auto Answer	0	4
Based Cooking	0	0
Battery Dog	0	2
Bites	0	0
Book Catalogue	0	0
DeepL	0	0
DuckDuckGo	0	0
Easy XKCD	0	0
Frozen Bubble	9	108
Hacki	0	0

App name	Paths with PoR	Total interactive paths
Linux Command Library	0	0
Lock Pattern Generator	7	9
LolCatBuilder	0	2
Man Pages	0	7
Mileage	0	0
Mini Note Viewer	5	31
Multi SMS Sender	9	20
Munch Life	0	1
MyExpenses	0	0
MyLock	0	130
Nectroid	1	31
Password Maker	0	24
Sanity	0	31
Stoic Reading	43	49
Weight Chart	0	21
WhoHasMyStuff	0	0
World Clock	6	22

Table 5.8: Results on GAPS automatic interaction with conditionality enabled

In this instance, performances are worse as the percentage of paths with PoR are only 11.96% over 1103 paths with possible interactions.

	Paths with PoR	Total interactive paths
Total	132	1103
Average	3,38	28,28

Table 5.9: Statistics on GAPS automatic interaction with conditionality enabled

5.2.3 Limitations

A major limitation in this category is the missing support for additional graphical elements, from which to retrieve the IDs for automatic interaction. Furthermore, currently GAPS sup-

port all the UI components that are defined through a combination of XML and Java/Kotlin code (i.e., buttons, switches, text views, and more). Although these are natively introduced by Android and recognized as the most popular types, others can be used in applications nowadays and support needs to be manually introduced through engineering effort. Additionally, some components can be recognized as reachable entry points through implicit Intent, whose actions are considered privileged by Android. Despite this being a feature of these apps to collect information about the device current state (e.g., an Intent is sent whenever the device is charging), some can only be sent by the system itself, otherwise exceptions are thrown. Therefore, in the future these cases will need to be managed to avoid testing paths that are, consequently, categorized as unreachable.

5.3 ICC Analysis Comparison with Amandroid

The Inter-Component Communication module implemented in GAPS can be directly compared to Amandroid [5], a state-of-the-art tool that provides the same type of information, among other things. During these analyses, I generated only the ICC-related data with GAPS over the AndroTest dataset Table 5.1, and similarly for Amandroid in its two variants:

- component, that performs an analysis based on knowledge about Android components, deemed to be more complex,
- work unit, alternative that uses a different "work-unit" analysis, based on Amandroid original implementation.

More specifically, since the analyses can take a lot of time I established a timeout of one hour and a half (5000 seconds). In this section, for each of the solutions and their variants, the ICC retrieval capabilities are measured by assessing:

- analysis time, measured as the required amount in order to produce the output or as the timeout, if it is exceeded,
- total number of Intent links, that are retrieved from the Json files that are produced by each solution.

5.3.1 Analysis Time

In Table 5.10 a full list of the applications analyzed can be consulted along with the Analysis Time (A.T.) required for each solution to terminate and produce their results.

App name	GAPS A.T.	Amandroid Component A.T.	Amandroid WorkUnit A.T.
A2DP Volume	22,71	753,96	1021,03
aagtl	3,43	2310,44	5000,01
Aard2	42,85	1548,94	365,80
Addi	4,68	994,31	1004,23
ADSDroid	1,97	165,42	97,24
aGrep	0,62	112,85	232,29
AirGuard	166,52	1061,29	879,76
Alarm Clock	0,87	70,66	65,01
aLogCat	1,92	39,71	40,50
Amazed	1,07	8,00	7,65
Androidmatic Keyer	1,72	419,77	762,44
AnyCut	0,47	22,37	28,15
AnyMemo	65,61	5000,00	5000,01
App Cache Cleaner	61,99	390,05	230,24
Auto Answer	0,47	17,67	19,44
Based Cooking	6,09	74,42	56,98
Battery Dog	0,57	15,67	26,38
Bites	0,82	177,68	51,70
Blokish	49,73	457,23	1033,71
Book Catalogue	19,50	5000,00	5000,02
DeepL	13,44	127,85	229,13
DuckDuckGo	225,08	5000,01	5000,02
Easy XKCD	72,87	1687,32	164,63
F-Droid Client	112,85	5000,00	5000,01
Flipper Zero	368,90	5000,00	5000,02
Frozen Bubble	1,82	1417,06	2516,70
Hacki	25,08	2554,92	747,67
HotDeath	1,62	82,74	410,81
KeePass2	141,21	808,41	679,73

App name	GAPS A.T.	Amandroid Component A.T.	Amandroid WorkUnit A.T.
Linux Command Library	82,22	59,03	48,70
Lock Pattern Generator	0,72	41,35	74,37
LolCatBuilder	0,52	39,58	44,37
ManPages	0,57	35,99	43,11
Mileage	3,13	5000,00	53,25
Mini Note Viewer	1,32	5000,00	5000,02
Multi SMS Sender	0,77	184,07	252,98
Munch Life	0,42	24,04	26,19
MyExpenses	506,10	5000,00	5000,01
MyLock	0,57	61,02	65,90
Nectroid	1,12	154,66	565,20
PasswordMaker	55,71	642,12	437,01
PCAPDroid	67,41	1480,15	913,98
Sanity	2,28	1879,89	740,63
Stoic Reading	33,51	5000,00	5000,03
SwiFTP	63,67	71,78	101,97
Weight Chart	0,87	74,33	125,54
WhoHasMyStuff	29,60	61,91	28,54
Wikipedia	164,22	5000,00	5000,02
WorldClock	0,87	177,26	29,01

Table 5.10: ICC analysis time (seconds) comparison between GAPS, Amandroid Component and Amandroid WorkUnit

Overall, GAPS never exceeded the one hour and a half timeout. Furthermore, the app that took most of the time is "MyExpenses", with 506.10 seconds (8.43 minutes). Instead, Amandroid Component and WorkUnit exceeded the time limit in 10 instances, sharing 9/10 cases. Moreover, Table 5.11 shows that GAPS required 40.53 minutes to complete the analysis, averaging 49.63 seconds. This result is also coherent with the path reconstruction statistics previously mentioned, that requires 60.31 seconds per app, without the conditionality module enabled. Therefore, the Inter-Component Communication information retrieval is a key step during the pre-processing phase of GAPS' analysis, occupying 82% of the average time, combined with the disassembling operation.
Meanwhile, Amandroid Component and WorkUnit completed their analysis in 19.52 and

17.83 hours, respectively. Furthermore, averaging 23.91 and 21.84 minutes correspondingly.

	GAPS A.T.	Amandroid Component A.T.	Amandroid WorkUnit A.T.
Total	2432,09	70305,94	64222,14
Average	49,63	1434,82	1310,66

Table 5.11: Statistics on ICC analysis time (seconds) comparison between GAPS, Amandroid Component and Amandroid WorkUnit

5.3.2 Intent Links Retrieval

The results of the ICC analysis for all the solutions are produced in the form of Json files, where all the retrieved Intent Links (I.L.) are shown in output. In Table 5.12 the amount of information collected is compared across GAPS, Amandroid Component and WorkUnit.

App name	GAPS I.L.	Amandroid Component I.L.	Amandroid WorkUnit I.L.
A2DP Volume	33	172	55
aagtl	2	4	5
Aard2	11	3	3
Addi	5	6	8
ADSDroid	1	2	2
aGrep	8	10	9
AirGuard	34	0	1
Alarm Clock	10	1	1
aLogCat	5	1	4
Amazed	1	0	0
Androidmatic Keyer	3	3	3
AnyCut	4	5	4
AnyMemo	88	0	3
App Cache Cleaner	2	0	3
Auto Answer	4	0	3
Based Cooking	16	0	0
Battery Dog	4	2	2
Bites	9	0	0
Blokish	5	4	6

App name	GAPS I.L.	Amandroid Component I.L.	Amandroid WorkUnit I.L.
Book Catalogue	62	0	54
DeepL	6	1	3
DuckDuckGo	59	0	0
Easy XKCD	22	0	4
F-Droid Client	151	0	32
Flipper Zero	48	0	1
Frozen Bubble	7	6	6
Hacki	27	0	0
HotDeath	1	2	2
KeePass2	44	110	1
Linux Command Library	5	1	1
Lock Pattern Generator	3	2	3
LolCatBuilder	0	0	0
ManPages	1	1	2
Mileage	22	0	1
Mini Note Viewer	14	0	15
Multi SMS Sender	7	6	6
Munch Life	1	1	1
MyExpenses	101	0	0
MyLock	21	4	5
Nectroid	12	23	13
PasswordMaker	11	4	5
PCAPDroid	29	1	21
Sanity	40	184	16
Stoic Reading	633	0	632
SwiFTP	41	0	6
Weight Chart	5	8	10
WhoHasMyStuff	8	0	1
Wikipedia	46	0	0
WorldClock	2	2	2

Table 5.12: ICC information (Intent Links) retrieval comparison between GAPS, Amandroid Component and Amandroid WorkUnit

Moreover, Table 5.13 shows that GAPS acquired 1674 Intent Links, 294% and 175% more compared to Amandroid Component and WorkUnit. Furthermore, on average, GAPS found 34.16 ICC-related information in each app, while the counterparts 11.61 and 19.49 respectively.

	GAPS I.L.	**Amandroid Component I.L.**	**Amandroid WorkUnit I.L.**
Total.	1674	569	955
Average	34,16	11,61	19,49

Table 5.13: Statistics on ICC information (Intent Links) retrieval comparison between GAPS, Amandroid Component and Amandroid WorkUnit

5.3.3 Considerations

Overall, GAPS, compared to Amandroid Component and WorkUnit, is able to extract a larger amount of information related to Inter-Component Communication in the test applications and in a fraction of their time (3.45% and 3.78% respectively). However, currently there is not an available automatic mean of testing the results' correctness across all the solutions, aside from manually reverse-engineering the apps to test the validity of the results, which can require a lot of time. In the future, these output should be actively compared to measure their accuracy, either through automated tests or by manual effort. Furthermore, GAPS is currently only retrieving ICC information through intra-procedural paths and is planned to also work inter-procedurally, similarly to how Amandroid does it. The addition of this feature can greatly increase the number of Intent Links retrieved.

6 Use Cases

At the current state, GAPS can be used to reconstruct paths to test the method's reachability for an Android application. The process through which this is achieved involves:

- generating the Control-Flow Graph of app's methods,
- finding Inter-Component Communication information,
- performing data flow analysis on method arguments and variable assignments,
- retrieving IDs associated to graphical elements to draw precise automatic interaction.

For each of the aforementioned tasks, further enhancements can be made in the future that can adapt GAPS to numerous use cases.

6.1 Alternative Static Analysis Approaches

Although static analysis has been largely explored over the years, especially among the Android research community [21] [22] [23] [24], the adoption of smali in this domain has been overlooked compared to examining the source and decompiled Java code. Furthermore, the rigorous smali's semantic can be advantageous for its clear usage of registers in handling operations. Therefore, GAPS by relying on this can provide insights that, compared to other approaches, can appear clearer and less subject to intricate syntax. Furthermore, since path reconstruction

works in a backwards fashion starting from a target method and is based on the Control-Flow Graph, conditional branches are also considered as they lead to uniquely different sets of instructions. This allows to study alternative intra-procedural code in the form of a linear sequence of directives. Moreover, this renders static analysis much simpler and affordable thanks to the direct relationship between instruction's addresses that is established during the app disassembly, which is not as easy when considering decompiled Java code. These features, in addition to path reconstruction and data flow analysis that GAPS provides, can open up room for unexplored approaches in static analysis that can benefit of these scalable and accurate techniques.

6.2 ICC Analysis and Testing

Inter-Component Communication in Android is a complex mechanism that has been extensively studied [25] [26] [27] [28] [29] [30] [31] [32] for its nature and tendency to leading to security threats. Applications that are registering as available in receiving Intents from external sources and handling their requests can inadvertently expose themselves to a range of attacks, such as Denial of Services, Spoofing and Hijacking. Furthermore, users can consequently become victim of data leaks and privilege escalation. GAPS can statically recover ICC information over the whole app and precisely and automatically map senders and receivers involved for each message. Moreover, the data flow analysis used to retrieve this information can be further extended and utilized to study the messages' construction, such as to detect additional data that can be appended. Consequently, this knowledge can help uncover vulnerabilities in apps, for instance it could be used in fuzz testing and use the Intents format to accurately generate inputs that can lead to crashes.

6.3 Taint Analysis

Currently, GAPS can perform intra-procedural data flow analysis using the registers involved in the smali instructions and resolve variable assignments through constant propagation. This functionality can be further extended thanks to the path reconstruction feature to perform this type of analysis over the whole path, therefore inter-procedurally. The expected use case would be similar to those of Flowdroid [4] and Amandroid [5], which perform taint analysis by using the Control-Flow Graph of the app. Furthermore, this feature could be applied to GAPS for discovering and studying information leaks inside Android applications and, combined with

the extraction of high-level instructions, automatically trigger the target methods and measure the user's exposure to threats.

6.4 Integration with Vulnerability Detection Tools

The Android research community, over the years, has produce a multitude of solutions that aim to provide automatic vulnerability detection through static analysis, such as SPECK [33]. However, one key limitation is that every flaw that can be proven to be a threat also needs to be manually verified to be a reachable portion of code by malicious entities. On top of that, Proof-of-Concept exploits should be usually implemented and this can cause trouble in the cases where, in order to be effective, one or more conditions need to be triggered through interactions with the vulnerable app. In the past, in order to try and solve this issue application exercisers have been used, such as Monkey [6] and ARES [7], although they cannot provide a guarantee of reaching the desired state. This scenario is the optimal use case for GAPS that would use the information found by the vulnerability detection solutions and draw a path to prove the reachability of the end point, along with instructions for automatically interacting with the app and generate a Proof-of-Reachability. Subsequently, launching an exploit that previously did not produce the expected results might then prove to be effective. On top of that, GAPS is built as a Python module that can be easily integrated in any project.

7 Related Works

Understanding the existing body of research in Android security is crucial for contextualizing the current study and identifying shortcomings that warrant further investigation. In this section, an overview of relevant literature that inspired GAPS is provided, focusing on recent studies related to static analysis and automatic interaction of applications. By reviewing the current state of knowledge, the goal is to identify areas where GAPS contributes and addresses existing inadequacies.

7.1 Control-Flow Graph Generation

Notoriously, Flowdroid [4] and Amandroid [5] are static taint analysis tools that generate the app's Control-Flow Graph in order to carry their tasks and draw a connection between sources and sinks. Both of them employ data flow analysis techniques to develop a CFG in a forward-visit approach, that is, every branch is visited during the initial phases. Moreover, this approach has proven to be resource-intensive when integrated with the escalating intricacy of Android applications. As a result, the computational demands have increased significantly, causing these tools to frequently encounter crashes due to memory exhaustion. Similarly, Soot [34] also provides Control-Flow Graph generation for Android apps by creating an intermediate representation of the Java bytecode, called "Jimple", which contains information about the relationship with other parts of the app's code.
Compared to these solutions, GAPS is based on the generation of intra-procedural CFGs for

each of the app's methods through Androguard and then paths are reconstructed in a backwards fashion, therefore starting from the endpoint to all possible entry points. Backdroid [35] also uses a similar technique to perform taint analysis. Moreover, this ensures that computations are performed strictly when required to avoid wasting additional resources in branches that should be never traversed. Consequently, compared to the counterparts, GAPS build its inter-procedural Control-Flow Graph on-demand and only for the building blocks required along the pathway, as opposed to doing it over the whole app during pre-processing.

7.2 Mapping Inter-Component Communication

In the past, the problem of mapping communication between components in the app and with the OS itself has been largely discussed in literature. For instance, Octeau et al. [36] in 2013 presented a scalable solution to statically identify ICC vulnerabilities in the Android platform. Furthermore, ICC-Inspect [37] similarly profiles Intents dynamically to trace call graphs and collects statistics. Elish et al. [38] applied these techniques to detect and prevent malicious tasks involving Intents. On top of that, IccTA [30] and Flair [26] have explored the matter of Inter-Component information leakage and vulnerabilities, respectively. Moreover, Amandroid [5] also allows retrieving knowledge about ICC links and created a benchmark suite to test other tools efficiencies in this domain. GAPS also intends to make a contribution in this regard by generalizing the search for methods accepting Intents as parameters and studying their intra-procedural code to gather information about their construction. Furthermore, in the future this feature could be evaluated using the Amandroid's ICC test suite and, most importantly, real-world applications.

7.3 Automatic Interaction

Automated interaction constitutes a relevant subject within Android research due to the complex challenge posed by the management of various graphical components and corresponding events, which can vary significantly based on the specific goal. Currently, one important employment is app exercising with the goal of increasing code coverage. Some of the most popular examples in these cases are Monkey [6], which is able to generate pseudo-random interactions with the app UI, and MonkeyRunner [39], which instead can be used to specify commands to perform and create custom scripts, both released officially by Google. Furthermore, recent trends in the last few years consist in trying to make smarter explorations by integrating Ma-

chine and Deep Learning models, such as Reinforcement Learning in the case of ARES [7]. Moreover, similar attempts have also been performed using stochastic models [9]. However, this type of pseudo-random app exercising cannot always produce the expected results, as bigger real-world apps usually have a lot of code that would need to be covered.
GAPS, as opposed to previous work in literature, aims to combine path reconstruction and automatic interaction to obtain a precise app exercising tool that can be adopted to reach an app state deterministically. A similar attempt was made by Liu et al. [40] in 2018, but they limited their analysis to consider only buttons, while GAPS abstracts their behavior and applies it also to other graphical elements.

8 Conclusions

In this Master thesis, I present GAPS, a static analysis tool that aims to offer path reconstruction through the generation of Android app's Control-Flow Graphs, and abstract from them high-level instructions that can be used for automatic interaction. This solution is built using the Androguard Python module to access the Dalvik bytecode and uses a backwards technique to draw the sequence of smali instructions and method invocations that are required to reach a target method. The path reconstruction phase concatenates app's invocations between each other, along with information related to Inter-Component Communication. Additionally, paths that satisfy conditional statements can be generated, to further enhance chances of reaching the desired end point. Furthermore, IDs associated to graphical UI elements are retrieved thanks to Apktool and automatic interaction is achieved through the AndroidViewClient Python package. Ultimately, GAPS was tested against the AndroTest open-source app dataset that is widely recognized as the standard for automatic interaction tools. In light of the results obtained, future improvements can be made to enhance performances, such as adding support for more Android components, graphical elements, and more. Furthermore, data flow analysis based on smali registers is used solely intra-procedurally at the moment; performances could greatly benefit from extending this functionality to also work across more invocations. Through engineering effort, these improvements can facilitate GAPS' adoption in many uses cases, that range from integration to vulnerability and information leakage detection tools, to also work as a standalone static and taint analysis solution.

References

[1] A. Pradeep, Álvaro Feal, J. Gamba, A. Rao, M. Lindorfer, N. Vallina-Rodriguez, and D. Choffnes, "Not your average app: A large-scale privacy analysis of android browsers," in *Proceedings on Privacy Enhancing Technologies Symposium*, 2022. [Online]. Available: https://doi.org/10.56553/popets-2023-0003

[2] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for android: Are we really there yet in an industrial case?" in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 987–992. [Online]. Available: https://doi.org/10.1145/2950290.2983958

[3] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 429–440. [Online]. Available: https://doi.org/10.1109/ASE.2015.89

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *AMC SIGPLAN Notices*, vol. 49, no. 6, p. 259–269, jun 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594299

[5] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Transactions On Privacy and Security*, vol. 21, no. 3, apr 2018. [Online]. Available: https://doi.org/10.1145/3183575

[6] Monkey. [Online]. Available: https://developer.android.com/studio/test/other-testing-tools/monkey

[7] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep reinforcement learning for black-box testing of android apps," *ACM Transactions on Software*

Engineering and Methodology, vol. 31, no. 4, jul 2022. [Online]. Available: https://doi.org/10.1145/3502868

[8] P. Bose, D. Das, S. Vasan, S. Mariani, I. Grishchenko, A. Continella, A. Bianchi, C. Kruegel, and G. Vigna, "Columbus: Android app testing through systematic callback exploration," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1381–1392. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00121

[9] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256. [Online]. Available: https://dl.acm.org/doi/10.1145/3106237.3106298

[10] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 94–105. [Online]. Available: https://doi.org/10.1145/2931037.2931054

[11] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 481–492. [Online]. Available: https://doi.org/10.1145/3377811.3380402

[12] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 224–234. [Online]. Available: https://doi.org/10.1145/2491411.2491450

[13] Android developer guide. [Online]. Available: https://developer.android.com/guide/components/fundamentals.html#Components

[14] Owasp mobile application security testing guide. [Online]. Available: https://mas.owasp.org/

[15] smali and baksmali. [Online]. Available: https://github.com/JesusFreke/smali/wiki

[16] Androguard. [Online]. Available: https://github.com/androguard/androguard

[17] Apktool. [Online]. Available: https://apktool.org/

[18] Androidviewclient. [Online]. Available: https://github.com/dtmilano/AndroidViewClient

[19] Frida. [Online]. Available: https://frida.re/

[20] F-droid. [Online]. Available: https://f-droid.org/

[21] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584917302987

[22] Y. Pan, X. Ge, C. Fang, and Y. Fan, "A systematic literature review of android malware detection using static analysis," *IEEE Access*, vol. 8, pp. 116 363–116 379, 2020. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.3002842

[23] Étienne Payet and F. Spoto, "Static analysis of android programs," *Information and Software Technology*, vol. 54, no. 11, pp. 1192–1201, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584912001012

[24] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak, "Static analysis of executables for collaborative malware detection on android," in *2009 IEEE International Conference on Communications*, 2009, pp. 1–5. [Online]. Available: https://doi.org/10.1109/ICC.2009.5199486

[25] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, ser. MoMM '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 68–74. [Online]. Available: https://doi.org/10.1145/2536853.2536881

[26] H. Bagheri, J. Wang, J. Aerts, N. Ghorbani, and S. Malek, "Flair: efficient analysis of android inter-component vulnerabilities in response to incremental changes,"

Empirical Software Engineering, vol. 26, no. 3, p. 54, Apr 2021. [Online]. Available: https://doi.org/10.1007/s10664-020-09932-6

[27] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Assessing the security of inter-app communications in android through reinforcement learning," *Computers & Security*, vol. 131, p. 103311, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404823002213

[28] P. Gadient, M. Ghafari, P. Frischknecht, and O. Nierstrasz, "Security code smells in android icc," *Empirical Software Engineering*, vol. 24, no. 5, pp. 3046–3076, Oct 2019. [Online]. Available: https://doi.org/10.1007/s10664-018-9673-y

[29] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer, "An empirical study of the robustness of inter-component communication in android," in *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, ser. DSN '12. USA: IEEE Computer Society, 2012, p. 1–12. [Online]. Available: https://doi.org/10.1109/DSN.2012.6263963

[30] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 280–291. [Online]. Available: https://doi.org/10.1109/ICSE.2015.48

[31] A. Kalysch, M. Deutel, and T. Müller, "Template-based android inter process communication fuzzing," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ser. ARES '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3407023.3407052

[32] M. Auer, A. Stahlbauer, and G. Fraser, "Android fuzzing: Balancing user-inputs and intents," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2023, pp. 37–48. [Online]. Available: https://doi.org/10.1109/ICST57152.2023.00013

[33] Speck. [Online]. Available: https://github.com/SPRITZ-Research-Group/SPECK

[34] Soot. [Online]. Available: https://soot-oss.github.io/soot/

[35] D. Wu, D. Gao, R. H. Deng, and C. Rocky K. C., "When program analysis meets bytecode search: Targeted and efficient inter-procedural analysis of modern android apps in backdroid," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 543–554. [Online]. Available: https://doi.org/10.1109/DSN48987.2021.00063

[36] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, "Effective Inter-Component communication mapping in android: An essential step towards holistic security analysis," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 543–558. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/octeau

[37] J. Jenkins and H. Cai, "Icc-inspect: Supporting runtime inspection of android inter-component communications," ser. MOBILESoft '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 80–83. [Online]. Available: https://doi.org/10.1145/3197231.3197233

[38] K. O. Elish, D. Yao, and B. G. Ryder, "On the need of precise inter-app icc classification for detecting android malware collusions," in *Proceedings of IEEE mobile security technologies (MoST), in conjunction with the IEEE symposium on security and privacy*. Citeseer, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:18538908

[39] Monkeyrunner. [Online]. Available: https://developer.android.com/studio/test/monkeyrunner/

[40] Y. Liu, S.-C. Wang, Y. Yang, Y.-C. Chen, and H.-M. Sun, "An automatic ui interaction script generator for android applications using activity call graph analysis," *Eurasia Journal of Mathematics, Science and Technology Education*, vol. 14, pp. 3159–3179, 05 2018. [Online]. Available: https://doi.org/10.29333/ejmste/91668

Acknowledgments