Static Analysis of Android Applications for Intent Extras

AIORT – Android Intent-Oriented Robustness Tester

Nicola Busato - 2119291
dept. of Mathematics, School of Science
University of Padua
Padua, Italy
nicola.busato.4@studenti.unipd.it

Jacopo Momesso - 2123874
dept. of Mathematics, School of Science
University of Padua
Padua, Italy
jacopo.momesso@studenti.unipd.it

***Abstract*—Dynamic analysis tools for Android applications often struggle to correctly handle app entry points, especially those reachable through inter-component communication (ICC). This limitation undermines their ability to explore realistic execution paths, leaving potentially vulnerable code untested. ICC-based entry points, such as exported activities, services, and broadcast receivers, represent a particularly attractive vector for attackers due to their accessibility from other apps or the system. As a result, a significant number of Android vulnerabilities are discovered through improperly handled `Intents`. Despite this, current automated testing tools typically overlook the nuanced logic guarding these entry points or fail to provide the precise input parameters required to traverse deeper code paths.**

To address this gap, we present TOOL_NAME a semi-automated analysis pipeline for Android applications that systematically generates and validates `Intents` to explore code paths and uncover runtime failures. Our approach integrates three key components: (1) *static analysis* using the Soot framework to extract control-flow graphs (CFGs) from exported activities and identify intent-dependent program points; (2) *symbolic execution* with the Z3 SMT solver to translate conditional branches into path constraints and derive concrete extra values and actions; and (3) *dynamic testing* via Android emulators (ADB/Drozer) coupled with LLM-based log analysis to verify which generated `Intents` trigger exceptions or anomalous behaviors. We evaluate our system on nine real-world APKs from the AndroTest benchmark, dispatching 158 valid `Intents`. Performance measurements show end-to-end analysis time on average is under ten seconds, with CFG construction and path solving remaining efficient in the majority of cases. Our results highlight the efficacy of combining static path extraction with constraint solving to focus testing on meaningful input conditions and reduce redundant exploration.

***Index Terms*—Static Analysis, Symbolic Execution, Z3 solver, Intent Fuzzing, LLM-based log analysis**

I. Introduction

Android applications are a frequent target of security research, with studies focusing on vulnerabilities, privacy violations, and implementation errors in mobile apps [1, 2]. One common approach to uncovering such issues is through the manual crafting of `Intent`-based inputs that exercise specific code paths.

Android's Inter-Component Communication (ICC) mechanisms, such as Intents, are integral to app functionality but also introduce significant security challenges. The complexity and flexibility of ICC can lead to vulnerabilities like component hijacking, privilege escalation, and unauthorized data access [3]. For instance, Wang and Wu [4], highlight that malicious applications can exploit ICC channels to perform unauthorized actions by colluding with other apps, thereby bypassing standard security measures. Additionally, Gadient et al. [5], identify various "security code smells" in Android applications that serve as indicators of potential ICC-related vulnerabilities, emphasizing the need for developers to be vigilant during the development process.

Manually crafting `Intent`-based inputs can be effective in triggering specific code paths, but it typically requires extensive reverse engineering to understand the app's control-flow structure and input handling, making the process labor-intensive and hard to scale.

In this project, we present a semi-automated pipeline that analyses Android applications to generate valid `intents` capable of reaching specific code paths and potentially triggering runtime exceptions. The system combines static analysis (via Soot[1]), constraint solving (via Z3[2]), and automated testing in an emulator, followed by log evaluation using a Large Language Model (LLM). The ultimate goal is to accelerate the discovery of proof-of-concept exploits by automating the generation and validation of `intents`.

Our approach is inspired by FlowDroid [1], a highly precise static taint-analysis tool that models the Android lifecycle and leverages context-, flow-, field-, and object-sensitivity to detect data leaks. While FlowDroid focuses on identifying information flow from sources to sinks, our pipeline instead tracks `intent`-related parameters (i.e., extras and actions), propagates their influence through control-flow graphs, and uses Satisfiability Modulo Theories (SMT) solving to determine valid input values that satisfy path conditions.

[1]https://github.com/soot-oss/soot
[2]https://github.com/Z3Prover/z3

This paper introduces **TOOL_NAME**[3], a novel framework that combines static analysis, symbolic execution, and log-based dynamic validation to generate semantically valid `intents` for testing Android applications. To our knowledge, this is the first lightweight and modular pipeline designed to automatically extract path-sensitive constraints from exported activities and solve them to derive valid ICC inputs with minimal manual intervention. Unlike prior work that either focuses on taint tracking or relies on random or coverage-guided fuzzing, our system offers a principled method for exploring deep app behaviours triggered by external inputs.

The remainder of this paper is structured as follows: In Section II, we provide background on Android ICC, static analysis, symbolic execution, and existing frameworks. Section III reviews prior work in taint analysis, symbolic execution for Android, and intent fuzzing. Section IV describes the overall architecture of our pipeline. Section V presents implementation details, including our use of Soot, Z3, and emulator automation. Section VI evaluates the system on synthetic and real-world applications, measuring performance and testing effectiveness. Section VII discusses the limitations and potential improvements. Finally, Section VIII summarises our contributions and outlines future directions.

II. Background

Android is a mobile operating system based on a modified Linux kernel, designed primarily for touchscreen devices such as smartphones and tablets. Android applications are primarily written in Java or Kotlin and are packaged in Android Package Kit (APK) files containing compiled bytecode, resources, and a manifest file. Android isolates apps from one another through the sandboxes and requires each app to explicitly request permissions (e.g., camera, contacts) before accessing sensitive data, letting users safely run a banking app alongside a casual game. Rather than reinventing common features, apps can delegate tasks like sending support emails to specialized apps through `Intents`, the Android Inter-Component Communication (ICC) mechanism [6].

Each APK includes an `AndroidManifest.xml` file that declares the app's components, required permissions, and access policies. This manifest serves as a blueprint for how the application interacts with the Android system and other apps. It defines the core components of the application, including activities (UI screens), services (background tasks), broadcast receivers (event handlers), and content providers (data interfaces). Crucially, it also specifies which components are accessible to other apps via the `android:exported` attribute.

A. Android Activities and Intents

The most common component in Android applications is the `Activity`, which represents a single screen with a user interface. Activities can be invoked internally by the app itself or externally by other apps or the system, depending on their visibility and export status. It also works seamlessly with other Android components such as `Broadcast Receivers` and `Services`.

An *access point* is any component that can be triggered externally by another app or system service. In our context, we focus on *exported activities*, including activities, broadcast receivers, and services, which are explicitly marked as accessible in the manifest file using the `android:exported="true"` attribute. These components become entry points into the application logic and may expose code paths that are unintentionally reachable.

When an exported component (whether an activity, broadcast receiver, or service) is triggered, it can receive `Intents` from external sources. These intents can specify:

- an `action`, which describes the operation to be performed (e.g., `android.intent.action.VIEW`);
- a set of `extras`, which are key-value pairs containing additional input data;

In Android, an `Intent` is a messaging object that facilitates communication between application components, including activities, services, and broadcast receivers. Intents allow one component to request an action from another. Intents may carry payloads in the form of key-value pairs, known as `extras`, which influence component behaviour at runtime. These extras can include various data types, such as primitives, strings, or complex objects like those implementing the `Serializable` or `Parcelable` interfaces.

Since exported components can be triggered by any application on the system, analyzing how intents and their extras are accessed and propagated within the code is essential for identifying potential vulnerabilities or privacy leaks [6]. By tracking intent-related operations in all components, we ensure that our analysis is comprehensive and applicable to the full spectrum of Android application functionality.

B. Static Analysis for Android

Static analysis examines an app's code and resources without executing it, enabling early detection of bugs, misconfigurations, and security vulnerabilities. In the Android context, static analysis must model the lifecycle of Activities and other components to handle callbacks correctly. Tools such as FlowDroid [1] achieve this by building precise call- and control-flow graphs, tracking taint from sensitive sources (e.g., GPS, contacts) to sinks (e.g., network, file I/O).

Soot [7] is a framework for statically analysing Android and Java applications. Android apps are compiled into Dalvik bytecode, a register-based format designed to run on the Android Runtime. Soot converts this low-level bytecode into Jimple, an intermediate representation that simplifies the code into a more readable and structured form. Jimple expresses each operation as a simple three-address statement, making it easier to analyse program logic and reason about variable values and conditions.

[3]Repository: https://github.com/Nicola-01/ATCNS

C. Symbolic Execution

Symbolic execution is a program analysis technique that treats inputs as symbolic variables instead of concrete values. As the program executes, it collects logical constraints—known as path conditions—based on the operations and branches encountered. By systematically exploring multiple execution paths, symbolic execution enables the generation of inputs that trigger different program behaviours, making it useful for coverage-guided testing and vulnerability detection [8].

D. Z3 SMT Solver

Z3 [9] is a high-performance Satisfiability Modulo Theories (SMT) solver that decides the satisfiability of logical formulas over theories such as bit-vectors, arrays, and uninterpreted functions. In the context of Android analysis, Z3 can solve constraints derived from path conditions over Intent extras, producing concrete values that satisfy these conditions and thus enabling automated input generation.

E. Android Debug Bridge and Drozer

Android Debug Bridge (ADB) [10] is a versatile command-line tool provided by Google as part of the Android SDK. It facilitates communication between a host computer and an Android device, allowing developers and testers to perform various actions such as installing applications, accessing device logs, and executing shell commands. ADB operates using a client-server architecture, comprising a client, a daemon (adbd) running on the device, and a server managing communication between the two.

Drozer [11] is an open-source security testing framework for Android, developed by WithSecure Labs. It enables security analysts to identify and exploit vulnerabilities in Android applications by simulating the behavior of malicious apps. Drozer operates by interacting with the Android runtime environment, allowing testers to assess the security posture of applications through dynamic analysis.

III. Related Work

A. Taint- and Lifecycle-Aware Static Analysis

FlowDroid [1] pioneered precise, lifecycle-aware taint tracking for Android apps by modeling callbacks and Activity lifecycles with context-, flow-, field-, and object-sensitive analysis. It tracks sensitive sources (e.g., GPS) to sinks (e.g., network) to report privacy leaks at high precision. Building on this, IccTA [12] extends FlowDroid's framework to catch inter-component leaks across Activities, Services, and BroadcastReceivers by linking data-flows through Intent boundaries. Complementarily, Avdiienko *et al.* [13] mine a large corpus of real-world Android apps to identify abnormal usages of sensitive data such as GPS or contacts by combining static taint-analysis with statistical outlier detection to flag apps whose data-access patterns deviate significantly from the norm.

ReproDroid [14] provides a common harness to benchmark six major taint analyzers (Amandroid, DIALDroid, DidFail, DroidSafe, FlowDroid and IccTA) against DroidBench, inferring ground truth to compare precision and recall under uniform conditions.

More recently, Samhi *et al.* [15] evaluate 13 state-of-the-art static analyzers on 1,000 real apps, revealing that an average of 61% of dynamically-executed methods are missing from static call graphs and that increased precision often comes at the cost of soundness.

In our pipeline, we adopt similar call-graph and control-flow extraction techniques to statically identify paths influenced by external Intent inputs, but refocus the analysis toward generating concrete path constraints rather than pure information-flow.

B. Symbolic Execution for Android

SymDroid [16] applies symbolic execution directly to Dalvik bytecode, translating it into a simpler μ–Dalvik IR and modeling core Android APIs and lifecycle callbacks; it systematically explores feasible program paths to identify permission-use and control-flow vulnerabilities. Later frameworks, such as the interaction-based declassification checker, leverage symbolic execution to enforce user-driven information-flow policies by symbolically modeling UI events and secret inputs [17].

More recently, Luo *et al.* [18] present Centaur, a tainting-assisted and context-migrated symbolic execution framework that targets the Android Framework itself (rather than just apps). By migrating concrete snapshots of the system-service execution context into a detached symbolic executor and then selectively exploring only those service-interface entry points, Centaur can automatically discover and even generate proof-of-concept exploits for real framework vulnerabilities with high precision and scalability.

C. Intent-Focused Fuzzing

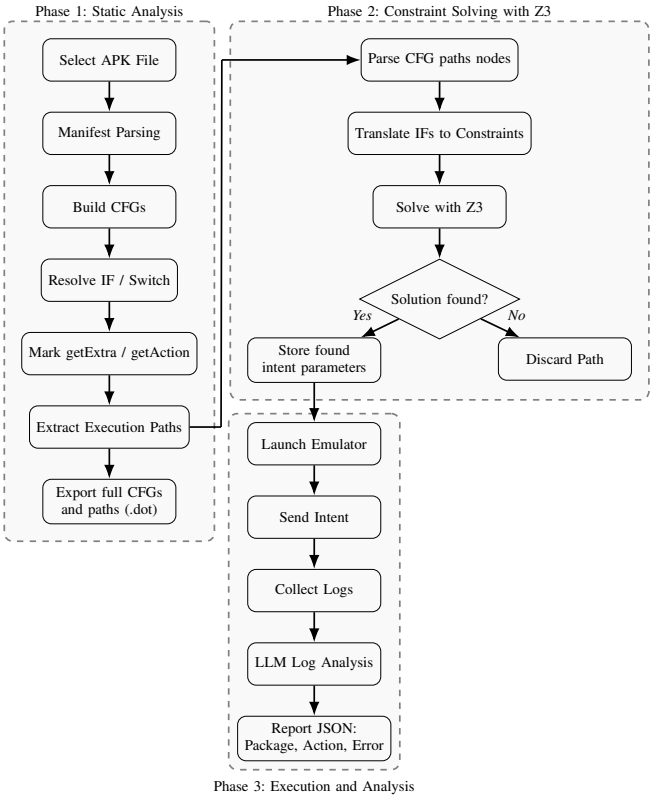
Targeted fuzzing of Intent handlers has also received attention. MALintent [6] is a grey-box fuzzer that instruments apps for coverage feedback and drives mutated Intents through exported components, finding security bugs in intent-processing code. Other works apply coverage-guided fuzzing to Intent-based networking and IPC services, using customizable bug oracles to detect logic and security flaws in Intent handlers [19]. Compared to these, our pipeline statically derives path constraints over Intent extras and then uses an SMT solver to generate valid payloads before dynamic validation, reducing the reliance on coverage-guided mutations.

While FlowDroid and IccTA [1, 12] provide highly precise taint-analysis foundations, and symbolic executors like SymDroid [16] enable deep path exploration, fully automated, Intent-focused testing remains challenging. Input generators such as Dynodroid, Sapienz, and MALintent improve coverage and bug-finding but either lack path-specific guidance or depend on extensive runtime instrumentation. Our work bridges this gap by integrating static path-constraint extraction (via Soot), symbolic solving (via Z3), and dynamic execution to generate and validate Intents that target specific code paths with minimal human effort.

IV. Design

This section presents the overall design of our system, structured as a three-phase pipeline for the automated analysis and testing of Android applications. The system statically inspects the app's code to extract execution paths influenced by external `Intent` inputs, symbolically solves the constraints governing these paths to generate valid test cases, and dynamically executes these tests within an emulated environment. Figure 1 illustrates the modular architecture and flow of data through each phase.

Fig. 1: System architecture divided into three phases



A. Phase 1: Static Analysis

The pipeline begins with the selection of a mobile application. The application's manifest and its metadata (i.e., SDK version, package name, etc.) are extracted to determine its declared components, exported access points, and system requirements. The system identifies the exported activities that act as entry points for the application due to the possibility of external calls from other applications.

Then, the system extracts the global variables declared in the application package. For each exported activity, the Jimple code of each method is extracted and used to construct the corresponding control-flow graph (CFG). A CFG is a directed graph in which each node represents a single Jimple instruction, and edges represent possible execution transitions between instructions (see Figure 2a). In the case of conditional nodes, such as `if` statements, the control flow branches into multiple successor nodes, each corresponding to a possible outcome of the condition (e.g., true or false), thereby capturing the program's decision structure.

All methods belonging to the same application package are considered eligible for analysis, and for each control-flow graph, the system parses every node to identify method calls. If the method call refers to another method within the same application package, the system inlines the callee's control-flow graph into the caller's CFG, effectively embedding the method's CFG within the main graph.

Global variables and static fields encountered in the graph are resolved and substituted with their known values. The resulting graph is then normalised: constructs such as switch statements are transformed into cascaded conditionals, and all operations related to intent handling (e.g., `getExtra()`, `getAction()`) are explicitly marked to facilitate tracking during the symbolic execution phase. A CFG is retained only if it contains at least one node that retrieves data from an `Intent`.

From these normalised graphs, execution paths are identified and enumerated. Each CFG may yield multiple paths, especially in the presence of conditionals or loops. Each CFG is then split into a set of linear paths, where each path corresponds to a sequence of nodes representing one possible execution scenario, based on the evaluation of conditional branches (see Figure 2). The system extracts and retains only the paths that contain at least one node dependent on intent inputs for further processing.

These paths are exported in a structured format (i.e., DOT format), where each file begins with metadata such as the target SDK version, component name, and action label. Each DOT file encodes all the execution paths extracted from a single method. Consequently, the output of the static analysis phase is a directory containing a collection of DOT files, one for each method belonging to exported activities that involve intent-related operations.

B. Phase 2: Constraint Solving

In this phase, each linear execution path extracted from the static analysis phase is examined to identify conditional expressions and variable derivations. Conditions may involve equality, inequality, type checks, string length, or other properties that influence control flow.

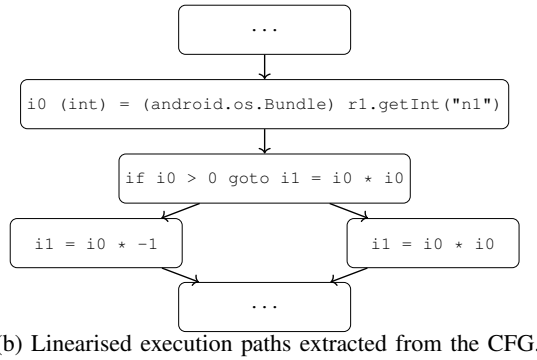
The system tracks all variables dependent on `Intent` inputs, especially those involved in `if` statements. These conditions are translated into logical constraints expressed in a symbolic language, allowing formal representation of restrictions on input variables.

The constraints are provided to a symbolic solver that attempts to find assignments of values to variables satisfying all conditions simultaneously. Paths for which no consistent solution exists are discarded, while satisfiable paths yield sets of valid values for intent parameters such as actions and extras.

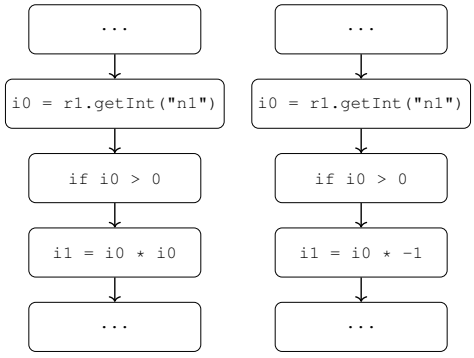
The output of this phase consists of a collection of `Intent` configurations represented in a textual format, including appli-

Fig. 2: Example of control-flow analysis and path extraction. The original CFG is shown in (a), from which each path in (b) is derived based on specific conditional branches.

(a) Control-flow graph (CFG) with a conditional branch.



(b) Linearised execution paths extracted from the CFG.



cation metadata (SDK version, package name, activity, action) and concrete assignments for relevant input parameters. These values include both constrained parameters and unconstrained ones. In cases where a value is unconstrained (denoted as `[no lim]`), a representative set of values is automatically generated during the next phase to explore input diversity.

Finally, special cases such as parameters of type `Serializable` are handled with a dual approach: a null instance is always considered alongside a non-null instance that satisfies the symbolic constraints on the path. This ensures a more comprehensive evaluation of input configurations for the subsequent execution phase.

C. Phase 3: Execution and Log Analysis

The final phase of the system is responsible for validating the feasibility of satisfiable paths through dynamic execution. For each path that has been symbolically solved, a corresponding intent configuration, composed of an action and a set of key and value extras, is sent and logged in a controlled runtime environment. This environment mimics a real Android system and is dynamically configured to match the application's SDK requirements, ensuring compatibility with the binary under analysis.

When a parameter is unconstrained (denoted as `[no lim]`), the system instantiates multiple representative values in order to test a range of runtime behaviours. These intent instances are then sequentially send to the application.

The execution of each intent is accompanied by system-level monitoring. Logs are collected during and immediately after the invocation, capturing messages, warnings, and potential error traces emitted by the application. These logs serve as the primary diagnostic signal for evaluating application behaviour under test inputs.

Rather than relying on hard-coded crash detection, the system delegates the analysis of log output to a large language model (LLM), which is prompted with structured information about the invoked action and observed logs. The model is instructed to detect and summarise any indication of failure, producing a structured JSON output that includes the tested action, extras, error flag, and a human-readable explanation.

V. Implementation

This section details the implementation of the static analysis and path generation system written in Java using the Soot framework and in Python using Z3. The pipeline consists of three major components: APK parsing and Soot setup, control flow graph construction and filtering, and execution path generation and export. We describe each in dedicated subsections.

A. Phase 1: Static Analysis and CFG Construction

a) APK Selection: The user is prompted to select an application to analyse. The system offers multiple selection modes to facilitate flexibility and reproducibility. For detailed instructions, please refer to the project README[4].

Once selected, the APK is unpacked, and its manifest and compiled code are loaded for further processing.

b) Manifest Parsing: To extract the `AndroidManifest.xml` from the selected APK, the system uses `Apktool` in decode mode with the following command:

```
apktool d <apkPath> -o manifest/<appName>
-f --no-src
```

After decompilation, the system parses the `AndroidManifest.xml` to extract critical metadata, including:

- **Package Name**: Identifies the unique application identifier.
- **Target SDK Version**: Determines the API level the application targets.
- **Activities**: Lists all declared activities, along with their attributes.
- **Exported Activities**: Identifies components marked as accessible to external applications. An activity is considered exported if the `android:exported` attribute is explicitly set to `true`, or, for applications targeting SDK versions below 31, if the attribute is omitted but the activity includes at least one `intent-filter`[5].

[4]https://github.com/Nicola-01/ATCNS/tree/master?tab=readme-ov-file#1-static-analysis-java--soot

[5]Since Android SDK 31, the `android:exported` attribute is mandatory for all components that use `intent-filters`; otherwise, the app will fail to install.

c) Platform JAR Resolution and Soot Setup: In order to enable correct resolution of Android API references during the analysis, the system dynamically retrieves the appropriate platform library for the application's target SDK version. This is done by checking whether the corresponding Android SDK JAR file is available locally; if not, it is automatically downloaded from a public repository[6]. The downloaded file is stored in a predefined directory and reused across multiple analyses to reduce overhead.

This platform JAR is then supplied to the static analysis engine, Soot, which is configured to analyse the APK in its original DEX bytecode form. Soot converts the bytecode into its intermediate representation, Jimple, a simplified, typed three-address code designed to facilitate precise data-flow and control-flow analysis.

d) Global Variable Collection: To support accurate static resolution and symbolic analysis, the system extracts global variables defined within the application package. Specifically, it collects fields that are either declared as `static final` or `static` non-final. These fields are typically used as constants or configuration flags and may influence control flow or variable assignments within the code. The values of these variables, when resolvable at analysis time, are recorded and substituted into the control-flow graph to avoid undefined references during path extraction. All collected variables are also saved to a text file for traceability and debugging purposes.

e) Control Flow Graph Construction: For each method defined in an exported activity, the system constructs a control-flow graph that represents the possible execution flow within that method. This analysis is applied to all methods in classes whose fully qualified names begin with the package name of the application and are exported. Each method body is processed through a transformation phase that registers its control-flow structure in a map indexed by a unique identifier composed of the class name, method name, and parameter types.

After the initial CFG is generated, it is passed to a `FilteredControlFlowGraph` constructor, which enhances the structure through a series of static transformations. These include resolving calls to methods defined within the same application (up to a fixed and configurable depth), and inlining their control flow to capture interprocedural logic. The transformation also replaces static fields with known constant values when possible, ensuring that conditional logic depending on global configuration is correctly represented in the graph.

To simplify the analysis and make the structure more regular, complex control constructs such as `goto` and `switch` are rewritten into standard conditional branches. At the same time, each node is renamed and reformatted according to a standardised naming convention to improve the readability and consistency of the resulting output (e.g., inside the node, the "equals" method is replaced with "=="). This normalisation ensures that downstream modules can rely on a stable structure when extracting constraints or building symbolic paths.

f) Intent-Related Node Filtering: After simplification and normalisation, each control-flow graph is scanned to identify nodes that involve operations related to `Intent` data access. The system applies a set of regular expressions to the textual representation of each instruction to detect direct or transitive calls to methods such as `getIntExtra()`, `getStringExtra()`, `getAction()`, and similar APIs that interact with `Intent` or `Bundle` objects. These regex-based patterns are defined in a dedicated utility component and allow flexible detection without relying on explicit method resolution. Only graphs containing at least one intent-related instruction are retained, as they represent methods whose behaviour depends on externally supplied input. Graphs without such dependencies are discarded to focus the analysis on execution paths relevant to symbolic input modelling and testing.

g) Execution Path Extraction and Renaming: Once the filtered control-flow graph is finalised, the system computes all simple execution paths from root to leaf nodes. This process is handled by a dedicated component that performs a depth-first traversal of the CFG, generating every acyclic path that starts at an entry node and terminates at a node with no successors. To reduce redundancy, only paths containing intent-related operations, such as access to extras or actions, are retained. The relevant nodes are identified through static matching patterns, and paths are deduplicated using a hash of the filtered node subset to ensure semantic uniqueness.

To enable accurate symbolic constraint generation, each path undergoes a variable renaming phase. This transformation ensures that variables assigned multiple times along the path are uniquely versioned using a sequential suffix. For example, a variable `i0` appearing in successive assignments is renamed to `i0_1`, `i0_2`, and so on, reflecting its temporal evolution along the path. The renaming is applied consistently to both left-hand-side and right-hand-side occurrences, preserving semantic coherence and avoiding aliasing conflicts during constraint resolution.

The final output of this phase is a DOT file containing all valid, intent-relevant paths extracted from a given method. Each path is represented as a subgraph, annotated with the sequence of instructions and control transitions. Additionally, the file includes metadata such as the SDK version, package name, activity name, and the action associated with the component. This structured format serves as the direct input for the constraint-solving stage, where each path is independently analysed for satisfiability.

B. Phase 2: Constraint Solving with Z3

a) Parsing Intent Parameters: The solver first scans each `.dot` subgraph for nodes that extract `extras` or `actions` from an `Intent` or `Bundle`. Matching is done via regular expressions, and each parameter is assigned a Z3 variable of the appropriate sort (`Int`, `String`, etc.), with names resolved via lookup or default inference.

[6]https://github.com/Sable/android-platforms/

b) Condition Extraction from Control Flow: Conditionals (`if` statements) in the path are translated into logical expressions by inspecting node labels and successor edges. Edge annotations (`true`/`false`) determine whether the condition or its negation should be added. The system supports string comparisons, numeric bounds, `null` checks, and composite expressions involving `Length()`, arrays, and custom types. We opted to extract and encode conditions directly from the normalized CFG path nodes to maintain semantic fidelity with the original application logic. This avoids the abstraction loss common in purely taint-based analyses and ensures that generated constraints reflect the precise control logic expressed by the developer. This high-fidelity modeling is essential to produce realistic and valid input values that correspond to actual feasible execution paths in the app.

c) Custom Type Handling and Sort Declaration: Android applications frequently use complex data types in their Intent extras, particularly objects implementing the `Serializable` or `Parcelable` interfaces. To accommodate these without requiring full semantic modeling, we introduce *uninterpreted sorts* in Z3 via `DeclareSort` statements. For each custom class (e.g., `MyClass`), the system declares an abstract type in Z3 and assigns symbolic variables of this sort to represent intent parameters of the corresponding type. To model nullability, which is often crucial in control-flow decisions, we introduce a dedicated constant `null_MyClass` for each custom type. Conditions comparing a parameter against `null` are then expressed using equality or inequality against this constant. This mechanism allows the solver to explore both the presence and absence of object-type extras, such as: This approach enables us to reason about object existence without requiring knowledge of internal fields or methods, which would otherwise make constraint solving intractable.

d) Constraint Encoding and Solving: All extracted conditions are assembled into a conjunction of constraints using Z3's Python API. The solver is invoked and, if `sat`, a model is extracted. Parameter names are mapped back to their user-visible keys (e.g., extra names) and formatted for output.

e) Result Formatting and Export: Each satisfiable path results in a line of output containing the package, activity, action, and all inferred extras with their types and values. If a serializable field is present, associated conditions are appended as metadata. Results are saved per-path and per-APK in a structured text format.

C. Phase 3: Automated Testing and Log Evaluation

This phase automates the dynamic testing of Android applications by injecting generated `Intent` messages into an emulated runtime environment and analysing their effects. The system takes as input both the APK file and the output of the static analysis phase, including metadata specifying the package name, activity, action, and required Android SDK version.

a) Emulator Setup and API Compatibility: A dedicated module initialises a virtual device matching the app's target SDK version. If the specified version is unavailable, the system incrementally moves forward to the next available version. Each emulator is launched with minimal boot animations and no audio to reduce resource consumption. The emulator is verified to be online, and if required, the screen is unlocked programmatically. If no compatible AVD exists, a new one is created on the fly using an appropriate system image with Google APIs.

b) App Installation and Launching: The APK is deployed to the emulator using `adb install`. For SDK versions `>= 23`, runtime permissions are granted automatically at install time. If the application is already present, it is uninstalled and reinstalled to ensure a clean test environment. The main activity is then launched, and the process ID (PID) is retrieved to enable fine-grained log filtering.

c) Intent Generation and Dispatch: The analysis output includes, for each path, a set of key-value pairs representing the required `Intent` extras. These values are parsed and used to construct valid `adb` or `drozer` commands. If a value is marked as unconstrained (denoted as `[no lim]`), the system instantiates multiple representative values from a predefined set, such as the empty string, a generic alphanumeric sequence (e.g., `"123 abc"`), and one containing special characters (e.g., `"123@abc"`). For each `[no lim]` entry, three distinct intents are sent, enabling broader behavioural coverage. Depending on the SDK version, the system dynamically chooses whether to send the intent via `adb` or via `drozer`[7], the latter being used for enhanced inspection on devices with SDK `>= 17`. Intents are dispatched sequentially without resetting the app state between tests.

d) Log Collection and LLM Evaluation: Before each intent is issued, the `adb logcat` buffer is cleared to isolate new logs. After execution, logs are retrieved and filtered based on the package name and process ID. These logs, along with the executed intent command, are passed to a large language model (LLM), which is prompted using a structured template including the APK package name, the intent command, and the full log output. The LLM is instructed to determine whether the intent caused an error, crash, or was ignored. The model replies in JSON-only format, with fields specifying the package, action, extras (including type and value), an `error` flag, and a short explanation. The format is enforced through a strict system prompt and no markdown is allowed in the output.

e) Batch Testing and Summary: The complete test workflow is orchestrated by a control script that recursively processes all analysis files within a directory. For each path, intents are constructed and dispatched, and results are aggregated into a single JSON file. At the end of execution, the system reports key statistics such as total intents sent, number of detected errors, and overall execution time. These outputs form the final result of the pipeline and support downstream validation, reporting, or triaging tasks.

[7]Drozer support is available only for applications targeting Android SDK version 17 or higher.

VI. Evaluation

In this section, we describe the experimental setup, present the results of our analyses, and evaluate the effectiveness and limitations of our approach.

A. Experimental Setup

All experiments were performed on two host systems: (1) Kali GNU/Linux 2025.1 and (2) Ubuntu 24.04.2. The Kali Linux machine is equipped with an Intel Core i7-10700K CPU and 32 GB of DDR4 RAM, while the Ubuntu machine is equipped with a Ryzen 7 7700X CPU and 32 GB of DDR5 RAM. The analysis pipeline is implemented in Java (OpenJDK 21) using Soot (latest version) for static analysis, while the symbolic execution is implemented in Python (version 3.13.3) using Z3 (latest version, installed via `pip install z3-solver`) for constraint solving. Android APKs are unpacked with apktool (v2.7.0-dirty) and supplemented with platform stubs matching each target SDK. For dynamic validation, we launch Android Virtual Devices (AVDs) through the Android Emulator, using system images corresponding to each app's declared API level; emulators are booted with disabled animations and unlocked screens to minimize startup overhead. Generated intents are dispatched via `adb` (and Drozer for SDK ≥ 17), with runtime output captured through `adb logcat` and subsequently analyzed by an LLM (model llama-3.3-70b-versatile) to automatically detect crashes or anomalous behaviors. Unless otherwise specified, no per-phase timeouts or memory limits are enforced; reported timings cover the full end-to-end workflow from manifest parsing through log-based error analysis.

B. Test Applications

To validate our system, we employed a two-phase testing strategy involving both synthetic and real-world Android applications.

First, we developed approximately nine custom APKs during the early stages of implementation. These synthetic apps were specifically designed to test individual components of the pipeline under controlled conditions.

Second, we evaluated our system on a subset of the **AndroTest** benchmark suite, a collection of 68 open-source Android applications. From this set, only nine applications were ultimately selected for testing. This selection was driven by two main constraints: (i) many applications did not expose any exported activities, and (ii) others triggered resource overflows or excessive path generation during static analysis. In particular, several apps caused exponential state expansion and memory issues during control-flow graph construction, a known challenge in Android static analysis. As noted by Arzt et al. [20], even with large memory resources and extended analysis time, tools like FlowDroid have been shown to fail when processing apps with highly dynamic or interprocedural control flow.

Table I lists the final set of real-world applications selected from AndroTest, together with their target SDK version and APK size.

TABLE I: List of real-world APKs tested (SDK version from manifest)

App Name	SDK	APK Size
aard2	31	4.2 MB
aGrep	21	353.2 kB
anymemo	29	4.5 MB
jamendo	10	4.5 MB
mileage	10	605.6 kB
mirrored	19	380.9 kB
nectroid	4	202.1 kB
passwordmaker	28	4.7 MB
zoobornss	13	41.7 kB

C. Performance Considerations

To evaluate the runtime performance and scalability of our system, we measured the execution time required for different phases of the pipeline across all tested applications. This includes manifest parsing, CFG construction, path extraction, constraint solving, and intent injection.

Table II shows the number of total and exported activities for each APK, along with the time needed to analyse each activity and extract the relevant metadata from the manifest. This step is relatively lightweight, typically completing in under 1.2 seconds per application.

TABLE II: Manifest analysis and number of exported entry points

APK	Activities	Exported	Time (sec)
aard2	5	5	1.02
aGrep	6	2	0.77
anymemo	29	5	1.15
jamendo	16	16	0.91
mileage	70	1	0.84
mirrored	4	1	0.81
nectroid	6	1	0.82
passwordmaker	9	2	1.09
zooborns	2	2	0.77

Table III reports the number of methods containing intent-related operations ("extras") for which a filtered CFG was constructed, along with the corresponding analysis time. This phase includes CFG creation, method expansion, simplification, filtering, and path extraction. While most apps remain under 4 seconds, a few show higher values due to interprocedural branching and call expansion.

Note: While `aard2` declares five exported activities, our analysis identifies six methods as interacting with Intent extras. This discrepancy arises because the exported activity count refers to the number of Android components (classes) marked as externally invocable in the application manifest, whereas our pipeline operates at the method level. For each exported activity, we extract a control-flow graph (CFG) for every method within that class. A method is flagged as "extra-relevant" if it contains (or, depending on the configured inlining depth, transitively calls) any statement that accesses data from an Intent—typically via `getIntExtra()`, `getStringExtra()`, or similar calls. Therefore, an app can have more extra-relevant methods than exported activities, especially if individual activities delegate input parsing to

helper functions or exhibit significant internal modularity. The same reasoning applies to other cases in which the number of extras exceeds the number of exported activities.

TABLE III: Soot analysis time and number of intent-relevant methods

App Name	Extras	Time (sec)
aard2	6	4.08
aGrep	2	2.12
anymemo	5	0.81
jamendo	15	1.83
mileage	1	0.04
mirrored	3	4.32
nectroid	1	0.12
passwordmaker	2	0.36
zoobornss	4	0.39

The final step of static analysis involves solving the corresponding input constraints using Z3. Table IV summarises the number of satisfiable paths generated per application and the total solving time. While most apps contain fewer than 100 paths, applications such as `aard2` exhibit significant path explosion due to nested conditionals and repeated use of intent-based logic.

TABLE IV: Path extraction and constraint solving times

App Name	Paths	Time (sec)
aard2	1403	8.573
aGrep	21	0.126
anymemo	11	0.055
jamendo	88	0.351
mileage	2	0.009
mirrored	21	0.060
nectroid	3	0.009
passwordmaker	9	0.027
zoobornss	29	0.138

Overall, our implementation remains efficient on small to medium-sized applications, with most per-phase times well below ten seconds. However, applications with high conditional complexity or poorly modularised code structures can incur higher analysis costs, especially during symbolic path generation.

D. Evaluation Metrics

To assess the impact of our system in practical scenarios, we collected preliminary metrics related to the number of dispatched intents and the presence of runtime errors. For each analysed APK, we recorded the number of valid `Intent` instances generated from satisfiable paths and injected into the application using either `adb` or `drozer`. We then monitored application behaviour through log collection and passed the logs to a language model tasked with identifying whether an error or crash occurred.

Table V summarises the results of this evaluation. The column *Valid Intents* indicates the number of concrete inputs dispatched during testing, while *Crash Detected* counts the number of those intents flagged as having caused a failure by the LLM. The *Crash Rate* is computed as the percentage of failing intents over the total dispatched for each application.

TABLE V: Intent injection results and error detection across APKs

APK	Valid Intents	Crash Detected	Crash Rate (%)
aard2	122	0	0.00
aGrep	4	0	0.00
anymemo	22	0	0.00
jamendo	14	3	21.43
mileage	1	0	0.00
mirrored	5	0	0.00
nectroid	3	0	0.00
passwordmaker	2	0	0.00
zooborns	0	0	0.00

VII. Discussion

A. Interpretation of Results

The experimental results demonstrate that our pipeline is effective at automatically generating valid Intents for a variety of Android applications, with a total of 158 valid Intents dispatched across nine real-world APKs and only one application (`jamendo`) exhibiting a non-zero crash rate of 21.43%. This suggests that while most apps were either resilient to malformed input or implemented effective input sanitisation, `jamendo` likely includes control branches that are insufficiently guarded against certain extra values.
To further investigate the anomalies detected in `jamendo`, we analysed the crash-inducing intents. The errors were caused by two primary issues:

- **Unhandled Extras and Activity Misrouting:** One crash was triggered when the boolean extra `"handled" = False` was passed to an activity that did not anticipate this parameter. Specifically, the `PlayerActivity` either lacked null-checks or failed to validate the expected structure of the received Intent, resulting in a runtime failure. In a similar case, the `IntentDistributorActivity` crashed due to a `NullPointerException`, indicating that internal logic assumed the presence of data that was either absent or incorrectly typed.
- **Boundary Value Exploitation:** Another failure was linked to the integer extra `"flipper_page" = -2147483648` (i.e., `Integer.MIN_VALUE`), a classic edge case. This value may have caused logic relying on positive indices to break, for instance when used in array access, pagination, or offset calculations without proper bounds checking. The corresponding logs confirmed that the application raised an error when this value was processed.

These results highlight the value of symbolic constraint solving: the system was able to generate semantically valid but edge-case inputs that uncovered real robustness flaws, which would likely be missed by random or mutation-based fuzzers.

Compared to random or mutation-based Intent fuzzers such as MALintent [6] and Dynodroid [21], which rely on coverage feedback or heuristic mutations, our method systematically derives path constraints and concrete inputs, reducing redundant or invalid tests and focusing on semantically meaningful edge

cases. This targeted generation likely contributes to the high precision of our injected tests, as evidenced by the low false-positive crash detections.

B. Challenges & Limitations

Despite its strengths, our tool exhibits some limitations. First, applications with highly complex or poorly modularized code (e.g., deep interprocedural branches) suffer from path explosion, as seen in `aard2`, which generated over 1,400 paths and incurred an 8.6 s solving time. Similarly, several AndroTest APKs had to be excluded due to resource overflows or excessive CFG growth during static analysis. Second, our current support is limited to primitive extras (String, int, boolean) and does not yet handle types such as arrays or complex Parcelable objects, potentially missing vulnerabilities in apps relying on richer data structures. Finally, delegating crash detection to an LLM introduces dependency on model accuracy and prompt quality; misclassifications in log analysis could lead to undetected faults or false alarms.

From a theoretical perspective, we also acknowledge that the problem of determining whether a specific input will reach a given program point is, in general, undecidable. According to Rice's Theorem [22], all non-trivial semantic properties of programs, i.e., properties concerning behaviour rather than syntax, are undecidable. In our context, this implies that it is fundamentally impossible to guarantee whether a given `Intent` will activate a particular control path. Consequently, our pipeline necessarily over-approximates the set of potentially relevant inputs, accepting the risk of false positives or infeasible paths. Nonetheless, this conservative approach enables sound analysis under partial information and remains effective at uncovering real vulnerabilities, as demonstrated by the failures observed in `jamendo`.

C. Future Work

To address these challenges and extend our contributions, we propose the following directions:

- **Scalability improvements**: Integrate path-pruning heuristics or summary-based interprocedural analysis to mitigate path explosion for large or deeply nested applications.
- **Richer data-type support**: Expand constraint encoding to better handle additional Java types such as Parcelable, Serializable, custom objects, and array-based objects. This enhancement would enable more comprehensive and realistic generation of Intent payloads for testing complex Android applications.
- **Benchmarking on larger datasets**: Evaluate the pipeline on broader benchmarks (e.g., full AndroTest, real-world app markets) and compare systematically against state-of-the-art tools such as FlowDroid [1], Dynodroid [21], and Sapienz [23] to quantify improvements in coverage and fault detection.

These enhancements will further solidify our pipeline's applicability to real-world Android security testing and open new avenues for automated, semantics-aware fuzzing of mobile applications.

VIII. Conclusion

We have introduced ***Tool_Name***, a novel `Intent`-focused testing framework that bridges static analysis, symbolic solving, and dynamic validation to automate the discovery of error-triggering inputs in Android apps. By leveraging Soot to build precise CFGs, Z3 to solve path conditions, and an LLM to interpret runtime logs, our system generates semantically guided `Intents` and verifies their effects with minimal manual effort. Experimental results on nine benchmark apps illustrate both the scalability of our approach, maintaining sub ten seconds analysis for most methods, and its effectiveness at uncovering subtle crashes that may evade random or coverage-guided fuzzers.

Our evaluation on nine benchmark applications illustrates both the scalability and practical effectiveness of the approach. *Tool_Name* maintained fast analysis times (typically under 10 seconds per phase) and successfully revealed logic flaws in real-world apps, including crash-inducing edge cases related to unchecked extras and invalid input boundaries. While challenges remain, such as handling complex data types, path explosion, and LLM misclassification, our methodology provides a reproducible framework for testing.

Future improvements will aim to expand support for richer data types, incorporate interprocedural summarisation to curb path growth, and benchmark against broader datasets. Overall, *Tool_Name* lays a foundation for principled, automated input testing in mobile app security, offering a compelling alternative to existing fuzzing and taint-analysis tools.

References

[1] Steven Arzt et al. "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 2014, pp. 259–269.

[2] William Enck et al. "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones". In: *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)*. USENIX Association. 2010, pp. 393–407.

[3] Daoyuan Wu et al. *SCLib: A Practical and Lightweight Defense against Component Hijacking in Android Applications*. 2018. arXiv: 1801.04372 [cs.CR]. URL: https://arxiv.org/abs/1801.04372.

[4] Jice Wang and Hongqi Wu. *Android Inter-App Communication Threats, Solutions, and Challenges*. 2018. arXiv: 1803.05039 [cs.CR]. URL: https://arxiv.org/abs/1803.05039.

[5] Pascal Gadient et al. "Security code smells in Android ICC". In: *Empirical Software Engineering* 24.5 (Dec. 2018), 3046–3076. ISSN: 1573-7616. DOI: 10.1007/s10664-018-9673-y. URL: http://dx.doi.org/10.1007/s10664-018-9673-y.

[6] Ammar Askar et al. “MALintent: Coverage Guided Intent Fuzzing Framework for Android”. In: Jan. 2025. DOI: 10.14722/ndss.2025.230125.

[7] Patrick Lam et al. “The Soot framework for Java program analysis: a retrospective”. In: *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*. Vol. 15. 35. 2011.

[8] Cristian Cadar and Koushik Sen. “Symbolic execution for software testing: three decades later”. In: *Communications of the ACM* 56.2 (2013), pp. 82–90.

[9] Leonardo De Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[10] Android Developers. *Android Debug Bridge (adb)*. https://developer.android.com/tools/adb.

[11] WithSecure Labs. *Drozer: The Leading Security Assessment Framework for Android*. https://github.com/WithSecureLabs/drozer.

[12] Li Li et al. “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. DOI: 10.1109/ICSE.2015.48. URL: https://doi.org/10.1109/ICSE.2015.48.

[13] Vitalii Avdiienko et al. “Mining Apps for Abnormal Usage of Sensitive Data”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 426–436. DOI: 10.1109/ICSE.2015.61.

[14] Felix Pauck, Eric Bodden, and Heike Wehrheim. “Do Android taint analysis tools keep their promises?” In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, 331–341. ISBN: 9781450355735. DOI: 10.1145/3236024.3236029. URL: https://doi.org/10.1145/3236024.3236029.

[15] Jordan Samhi et al. “Call Graph Soundness in Android Static Analysis”. In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2024. Vienna, Austria: Association for Computing Machinery, 2024, 945–957. ISBN: 9798400706127. DOI: 10.1145/3650212.3680333. URL: https://doi.org/10.1145/3650212.3680333.

[16] Jinseong Jeon, Kristopher K Micinski, and Jeffrey S Foster. “SymDroid: Symbolic execution for Dalvik bytecode”. In: *University of Maryland, Tech. Rep* 7 (2012).

[17] Kristopher Micinski et al. “Checking interaction-based declassification policies for android using symbolic execution”. In: *European Symposium on Research in Computer Security*. Springer. 2015, pp. 520–538.

[18] Lannan Luo et al. “Tainting-Assisted and Context-Migrated Symbolic Execution of Android Framework for Vulnerability Discovery and Exploit Generation”. In: *IEEE Transactions on Mobile Computing* 19.12 (2020), pp. 2946–2964. DOI: 10.1109/TMC.2019.2936561.

[19] Jiwon Kim, Benjamin E Ujcich, and Dave Jing Tian. “Intender: Fuzzing {Intent-Based} networking with {Intent-State} transition guidance”. In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, pp. 4463–4480.

[20] Vitalii Avdiienko et al. “Mining Apps for Abnormal Usage of Sensitive Data”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 426–436. DOI: 10.1109/ICSE.2015.61.

[21] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. “Dynodroid: An Input Generation System for Android Apps”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: Association for Computing Machinery, 2013, 224–234. ISBN: 9781450322379. DOI: 10.1145/2491411.2491450. URL: https://doi.org/10.1145/2491411.2491450.

[22] Henry Gordon Rice. “Classes of recursively enumerable sets and their decision problems”. In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366.

[23] Ke Mao, Mark Harman, and Yue Jia. “Sapienz: Multi-Objective Automated Testing for Android Applications”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: Association for Computing Machinery, 2016, 94–105. ISBN: 9781450343909. DOI: 10.1145/2931037.2931054. URL: https://doi.org/10.1145/2931037.2931054.