MALintent: Coverage Guided Intent Fuzzing Framework for Android

Ammar Askar[†*], Fabian Fleischer[†*], Christopher Kruegel[‡], Giovanni Vigna[‡] and Taesoo Kim[†]
[†]Georgia Institute of Technology
[‡]University of California, Santa Barbara
{aaskar, fleischer, taesoo}@gatech.edu, {chris, vigna}@cs.ucsb.edu

***Abstract*—Intents are the primary message-passing mechanism on Android, used for both communication between intra-app and inter-app components. Intents go across the trust boundary of applications and can break the security isolation between them. Due to their shared API with intra-app communication, apps may unintentionally expose functionality leading to important security bugs. MALintent is an open-source fuzzing framework that uses novel coverage instrumentation techniques and customizable bug oracles to find security issues in Android Intent handlers. MALintent is the first Intent fuzzer that applies greybox fuzzing on compiled closed-source Android applications. We demonstrate techniques widely compatible with many versions of Android and our bug oracles were able to find several crashes, vulnerabilities with privacy implications, and memory-safety issues in the top-downloaded Android applications on the Google Play store.**

I. Introduction

The Android ecosystem allows applications to add rich sets of functionalities on top of the operating system (OS). Unlike most desktop threat models, these applications are isolated from one another, preventing unilateral access to each other's data and code. However, for a seamless user experience, applications need to be able to trigger and re-use functionality in one another. For example, a photo viewer application may want to allow the user to share a photo through an email application. The photo viewer may also want to allow the user to view where the photo was taken with a map application.

The primary Inter-Process Communication (IPC) mechanism that enables this invoking of functionality across apps on Android is called *Intent*s. Components in applications that handle these intents act as an entry point and as a trust boundary between apps. This security boundary is critical and can have a significant security impact. For example, (1) an application might allow sensitive functionality to be invoked, like sending an email without user interaction, or (2) an application might call an image parsing library with a memory-safety issue. In these cases, a malicious application on the phone could trigger this intent to send an email without the user's consent or violate the security isolation between applications by exploiting the memory-safety bug.

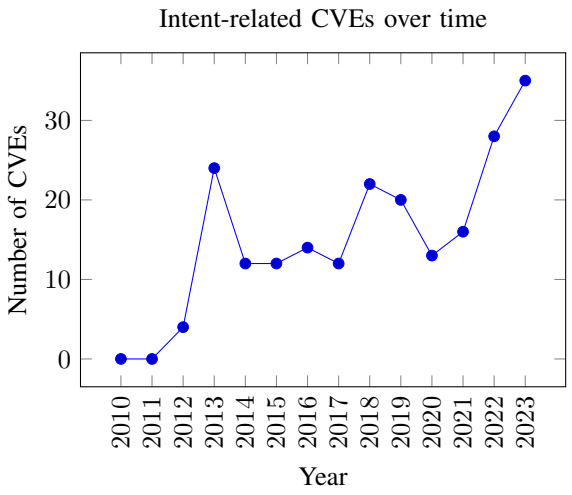

Fig. 1: Number of CVEs caused by Intent related issues over time. Each issue with "android" in the CPE configuration was checked.

To further complicate matters, aside from inter-process communication, applications launch and call components internal to themselves with Intents as well [1]. This can cause developers to fail to realize that a component or code path could be reachable from outside the application, accidentally exposing a sensitive operation only meant to be used internally [2] by the application itself. Figure 1 shows a steady trend of Intent-related issues in the form of CVEs over time.

The primary intuition behind MALintent is that these intent handlers are an effective entry point for fuzzing Android apps. They use fairly structured data and, unlike fuzzers that target the UI of apps, intents can be sent without user interaction from malicious low-privileged apps purely through code.

In summary, the main contributions of MALintent are:

- Demonstrating how specifications for intents can be gathered from an application using static and dynamic analysis. This enables MALintent to power an intent mutation system that can explore deep application behaviors.
- Developing a new coverage instrumentation technique for Android applications that improves intent fuzzing and, we believe, is helpful in general. MALintent uses

* These authors contributed equally to this work.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.230125
www.ndss-symposium.org

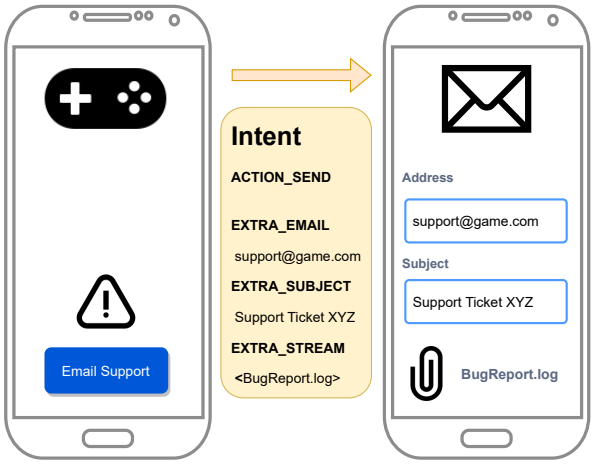

Fig. 2: A gaming app using an intent to trigger the user's preferred email app. The intent carries data to pre-fill the recipient address, the subject of the email, and to attach a file.

```
void contactSupportEmail() {
  Intent i = new Intent(Intent.ACTION_SEND);
  i.setData(Uri.parse("mailto:"));
  i.putExtra(Intent.EXTRA_EMAIL, "support@game.com");
  i.putExtra(Intent.EXTRA_SUBJECT, "Support Ticket XYZ");

  File bugReport = generateBugReportLog();
  i.putExtra(Intent.EXTRA_STREAM, Uri.fromFile(bugReport));

  // Start the user's preferred email app to send email.
  startActivity(i);
}
```

Listing 1: An example of launching an intent that would cause the user's email app to pre-fill an email with a subject line, recipient's address and attachment.

```
<activity android:exported="true"
          android:name=".EmailComposeActivity">
  <intent-filter>
    <action android:name="android.intent.action.SEND" />
    <data android:scheme="mailto" />
  </intent-filter>
</activity>
```

Listing 2: A snippet from an app's `AndroidManifest.xml` file showing how an app subscribes to receiving email intents.

this instrumentation to be the first greybox intent fuzzer capable of fuzzing closed source Android applications.
- Introducing and evaluating three novel intent bug-oracles for MALintent written by the authors that aim to find (1) application crashes, (2) basic privacy violations, and (3) memory safety issues in native code called through the Java Native Interface (JNI).

II. Background

Unlike most desktop OSes, Android's security model demands that apps are isolated from each other, lacking access to each other's data. Each app also needs to request permissions [3] from the OS to gain access to the user's private data such as their camera, microphone, or memory card. This isolation keeps users safe despite the millions of apps on the Android app store. Users can install a critical health or banking app alongside a lower-trust gaming app without needing to audit its security. The permission model grants higher privileges to trusted apps, enabling users to give location permissions to a preferred maps app but not to a random game.

Instead of implementing all functionality internally, Android apps can request services from each other. For instance, a gaming app can let users send an email to the support through their preferred email app, as illustrated in Figure 2, rather than implementing email functionality itself. This approach encourages the use of mature apps, ensures a consistent user experience with their preferred apps, and maintains the principle of least privilege by granting permissions only to trusted apps. This interaction between apps is enabled by an inter-process communication (IPC) mechanism called an *Intent*.

A. Intents

An *Intent* is the message in the message passing system on Android. Intents contain an action to perform, such as *view an image* (`ACTION_VIEW`) or *dial a number* (`ACTION_DIAL`), and data: the image to view or the phone number to dial. This data is often encoded in the form of a URI and its query parameters but can contain any serializable Java data through an arbitrary key–value map called Intent extras.

Listing 1 shows how the example from Figure 2 can be implemented: a gaming app allowing the user to send an email through their preferred email app. The app sets up an intent with an action of `ACTION_SEND` and sets extra fields in the intent to fill the recipient email address, the subject line, and attach a file. When an intent is provided to the Android OS via the `startActivity` method, the OS delivers it to the user's chosen email app. This is an *implicit intent*, where the OS selects the app. Alternatively, the intent sender can specify the target app, known as an *explicit intent*.

App components that receive intents must be declared in `AndroidManifest.xml` (see Listing 2). The `android:exported` property specifies if the component can receive intents from other apps or only internally. Tags filter the desired actions, categories, and data schemes from incoming intents, further detailed in §III-A1.

B. Intent Vulnerabilities

To understand potential vulnerabilities in an app's intent handlers, we examine how it handles an incoming intent. Listing 3 shows how an email app would handle the intent from Listing 1. The handler retrieves "extras," which are key-value pairs. In our example, the email app extracts extras from the intent and populates the appropriate fields in the email. There are two potential security bugs in this receiver:

- The email app accepts attachment files from the intent's `EXTRA_STREAM` and uses the method `renderPreviewIfImage` to render the image. This method calls a native C function to render the image but naïvely uses the height and width specified in the image's header, which may lead to a buffer overflow in the native code due to the lack of bounds checking.

```java
private void handleIntent(Intent i) {
  // ...
  if (i.hasExtra(Intent.EXTRA_EMAIL)) {
    setRecipient(i.getStringExtra(Intent.EXTRA_EMAIL));
  }
  if (i.hasExtra(Intent.EXTRA_SUBJECT)) {
    setSubject(i.getStringExtra(Intent.EXTRA_SUBJECT));
  }
  if (i.hasExtra(Intent.EXTRA_STREAM)) {
    Uri uri = i.getParcelableExtra(Intent.EXTRA_STREAM);
    File attachment = new File(uri.getPath());
    addAttachment(attachment);

    renderPreviewIfImage(attachment);
  }
  if (i.hasExtra(EXTRA_EMAIL_SERVER)) {
    sendWithServer(
      i.getString(EXTRA_EMAIL_SERVER));
  }
}

private void renderPreviewIfImage(File f) {
  // ...
  // Use GIFLib through the JNI.
  GifInfoHandle handle = new GifInfoHandle(f);
  byte[] pixels = new byte[header_height * header_width];
  handle.renderFrame(pixels);
}
```

Listing 3: An example of receiving an email-sending intent. The email app pre-fills the email recipient and subject and attaches the requested attachments.

- The email app uses the extra `EXTRA_EMAIL_SERVER` to decide which email server to send messages with. This is used internally when switching sender accounts but it is important to prevent external apps from setting this field to ensure that a malicious app cannot intercept the email written by the user.

C. Coverage-driven Grey-box Fuzzing

Coverage-driven grey-box fuzzing [4], [5], [6] is an automated software testing technique that effectively uncovers vulnerabilities in target programs. Unlike grammar-based approaches, mutation-based fuzzing requires little prior knowledge of the target program and generates new inputs by modifying existing seed inputs. The general algorithm used in coverage-driven mutation fuzzers involves maintaining a queue of inputs, initially populated by known seeds [7]. The fuzzer then mutates the inputs in various ways, tests the program with the mutated inputs, and reports any crashes or bugs encountered. Inputs that reveal new code coverage are added back to the queue, and the process continues by mutating inputs from the queue.

III. Design

Due to the critical security boundary and declarative nature of intents, their handlers are ideal for fuzzing. This leads us to the overall design of MALintent (see Figure 3):

1) MALintent performs a static analysis on the tested app, generating a list of intent handlers, actions, categories, data, and statically detectable intent extras (§III-A).
2) For coverage-driven fuzzing, MALintent uses Android's official debugger mechanisms to instrument the target app, ensuring wider compatibility (§III-B).
3) After identifying intent targets and collecting coverage, fuzzing begins. An intent matching static properties from Step 1 is generated and sent to the target app. The intent is randomly mutated, and those with higher coverage are added to the fuzzing queue for further mutation (§III-C).
4) During fuzzing, the app may exhibit behaviors indicating security issues. MALintent uses bug oracles to identify intents triggering security issues, such as crashes or memory-safety bugs, akin to sanitizers in traditional fuzzers (§III-D).

A. Static Analysis

MALintent starts by statically examining the Android PacKage (APK) of the application, which includes metadata, compiled Java/Kotlin code, native code, and media resources. Using the jadx [8] decompiler, we unpack the APK to analyze its metadata and bytecode. This analysis identifies the handlers that can receive intents, the data types and keys of extras in the intent, and the action/category/data filters. The resulting intent specifications are used as fuzzing seeds [7] to create initial intents for fuzzing, ensuring they are well-formed and pass Android's intent filtering mechanism described in §III-A1.

1) Application Manifest: The analysis begins with the `AndroidManifest.xml` file, the primary metadata file for Android applications. It contains the app name, required permissions, developer information, and critically for us, a list of intent-receiving components and their accessibility to other apps. We look for components that can receive intents:

- `<activity>` Activities act as the primary user-interface component on Android. The screen to compose an email in an email app would, for example, be an *Activity*.
- `<activity-alias>` Aliases for activities are often used for fine-grain intent filters and backwards compatibility.
- `<service>` Background services may be used, for example, to poll an email server for new emails.
- `<receiver>` Broadcast *Receiver*s listen to intents broadcast by the system, such as the WiFi state changing.

Next, we filter the components based on their `android:exported` property being set to true and the `android:permission` attribute being unset. These matched components can be invoked by all other applications without any specific permissions needed.

Once these intent-receiving components have been identified, MALintent analyzes their child `<intent-filter>` tags. The intent filter tags instruct the Android OS on what types of intents the component is willing to receive. This feature is called *intent matching* and allows apps to declaratively filter intents. Intents that do not pass the filtering will not be delivered to the destination application.

The `<action>` and `<category>` tags within the `<intent-filter>` tag specify action and category names as strings. In terms of filtering, the `<action>` tags allow the application to specify that an intent's action should either match one of the actions specified in the tags or be empty. The `<category>` tags require that an intent's category list must *at least* include *all* the required categories, or be empty.

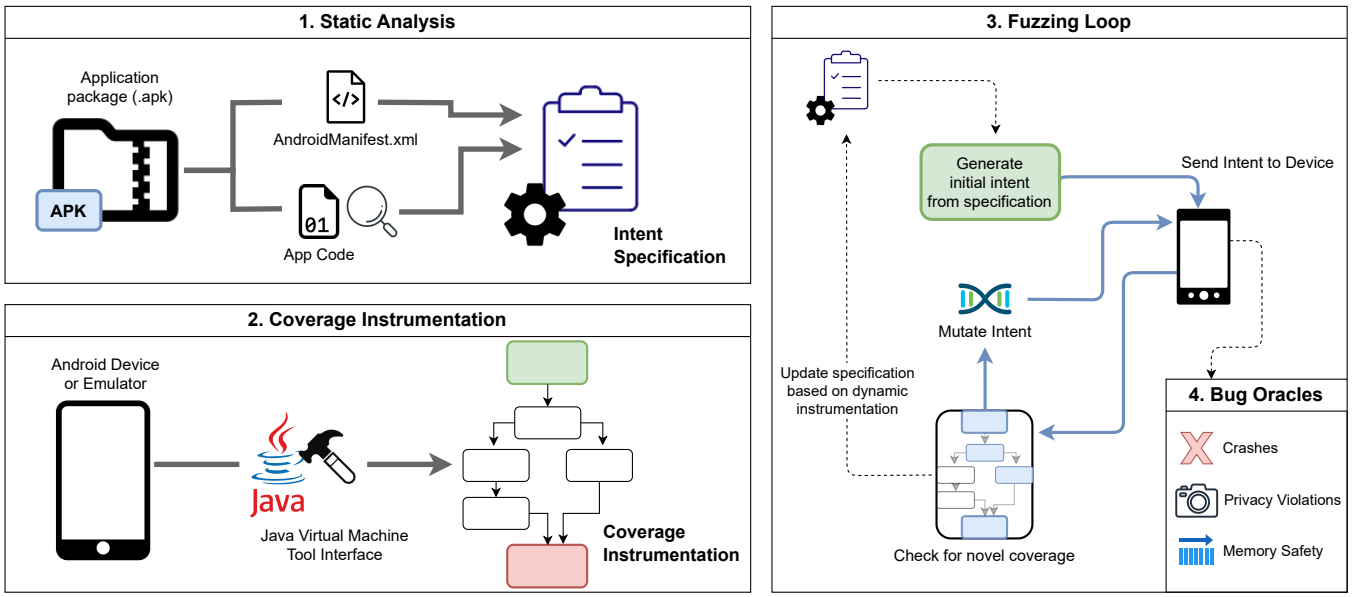

Fig. 3: Overall design of MALintent. **1.** Static analysis is performed on the packaged application in order to discover the components that can receive intents and their requirements. This is used to generate intent specifications for the application. **2.** MALintent leverages the interface available for Java Virtual Machine debuggers to instrument the app on a device or emulator for coverage feedback. **3.** A traditional fuzzing loop is seeded from the intent specifications. Coverage feedback from step 2 is used to mutate and drive the fuzzer. **4.** During the execution of the intents by the fuzzer, bug oracles detect different security issues based on the state of the device and MALintent records these issues.

The `<data>` child tag allows filtering based on the intent's "data" field, which is a Uniform Resource Identifier (URI), and the "type" field, representing the MIME type [9] of the data. Collectively, these two fields are used to pass files across application boundaries.

The constituent parts of a URI are:

`<scheme>://<host>:<port><path>`

When the data tag specifies only an `android:mimeType`, Android implicitly accepts schemes of `content` and `file`. The `android:scheme` mandates that the URI matches at least one of the required schemes (e.g., `http` and `ftp`). For filtering other URI properties such as `host`, `port`, and `path`, a `scheme` filter must be present. The `android:host` property mandates that the URI's hostname matches one of the required hosts, and it allows wildcard characters at the beginning of the host (e.g., `*.google.com`). The `android:port` property requires the URI's port to match one of the specified ports. Lastly, the `android:path` requires the URI's path to match one of the specified path filters. Path filtering options include `pathPrefix`, `pathSuffix`, and `pathPattern`, allowing for matching a path with a specific prefix, suffix, or wildcard anywhere in the filter, respectively.

2) Intent Handling Code: After going through the manifest, MALintent uses a visitor for jadx's abstract syntax tree to determine the names and types of intent extras. This is accomplished by examining method invocations on the `Intent` type that retrieve intent extras (e.g., `getStringExtra` and `getFloatExtra`). By statically determining the value of the string argument, MALintent stores the key and its type. Consequently, it becomes possible to generate a predetermined list of valid intent extra keys and types for the fuzzer, eliminating the need to discover them gradually based on coverage changes during the fuzzing process.

3) Intent Specification: Once MALintent has both analyzed the application's manifest and performed static analysis on its intent-handling code, it outputs an intent specification file with the intent-receiving components and their constraints for the fuzzer to use. Listing 4 shows an intent specification generated from a real-world Android application: *Samsung Internet Browser*.

```
{
  "package": "com.sec.android.app.sbrowser",
  "name":
    "com.sec.android.app.sbrowser.SBrowserLauncherActivity",
  "component": "<activity>",
  "action": "android.intent.action.VIEW",
  "categories": [
    "android.intent.category.DEFAULT",
    "android.intent.category.BROWSABLE",
  ],
  "data": {
    "scheme": ["http", "https", "about", "javascript"],
  },
  "extras": {
    "android.intent.extra.REFERRER_NAME": "string",
    "create_new_tab": "boolean",
    "trusted_application_code_extra": "string",
    "com.android.browser.headers": "bundle",
    "// ..."
    "// More extras omitted for space."
  },
}
```

Listing 4: An example of a shortened intent specification generated by analyzing the Samsung Browser application.

The intent specification contains one specification object per

`intent-filter` encountered. This specification object states (1) what type of *component* the filter is from: `<activity>`, `<activity-alias>`, `<service>` or `<receiver>`; (2) the name of the component, which is required to target it for an explicit intent; (3) the *action*, *category*, and *data* filters from the manifest; and (4) the intent extras as well as their types.

B. Coverage Instrumentation

To drive the fuzzer towards deeper and more interesting behaviors, we utilize coverage-driven (grey-box) fuzzing [10]. Traditionally, grey-box fuzzers instrument the target program at the compiler level if the source code [6] and compilation tool chain are available. Otherwise, such as in the case of most popular Android applications, binary instrumentation must be applied. Binary instrumentation usually involves using a framework like DynamoRIO [11], PIN [12], or QEMU [13] to hook instructions or modifying the executable binary code to include the instrumentation [14].

MALintent utilizes a binary instrumentation approach primarily because most Android applications are closed source. Past approaches [15], [16], [17] take the APK of the application and disassemble it to Android's bytecode format smali [18], optionally lifting it to Java [19]. This decompiled code is then altered to collect coverage, after which the APK is recompiled and used in place of the original. We forego this alternation of the original APK in MALintent due to its critical drawbacks as discussed in §V-A.

MALintent, instead, uses the Java Virtual Machine Tools Interface (JVMTI) [20] to implement its binary instrumentation. Notably, the JVMTI is the low-level application programming interface (API) that is used to implement debuggers in the Java Virtual Machine (JVM) and allows access to JVM internals when executing code (e.g., setting breakpoints and examining thread state). MALintent uses methods that allow the mutation of Java bytecode when it is loaded. At a high level, the process is:

1) MALintent attaches a JVMTI agent to be loaded on the target application's startup. The JVMTI agent registers a `JVMTI_EVENT_CLASS_FILE_LOAD_HOOK` callback to be called when a class is loaded into the JVM.
2) When the class load callback is triggered, MALintent checks its cache in the file system for an instrumented version of the class. If the instrumented version is found in the cache, MALintent retrieves and returns it.
3) If the class is not cached, MALintent generates a control flow graph (CFG) from the bytecode. This CFG is then used to instrument the start of each basic block to collect basic block-level coverage information.
4) MALintent starts up a communication channel in the form of a Unix or TCP socket to provide coverage information back to the fuzzer.

Dynamic Bytecode Instrumentation. Our JVMTI agent receives the Dalvik bytecode for every loaded Java class. We employ dexter [21] (maintained by Android) to alter the bytecode of these classes. As part of the modification, dexter generates a CFG. MALintent traverses the basic blocks of the CFG and inserts a call to an instrumentation method in each one. MALintent also increments the virtual register count of the method to account for the basic block ID argument.

The modified smali bytecode highlighting the added instrumentation is shown in Listing 5. Specifically, we include two instructions: one to assign the basic block's ID as an argument, and another to invoke the instrumentation method. This instrumentation method is statically implemented in a Java class that is loaded during the agent initialization. It updates the coverage map by adding an edge between the previous and current blocks, as shown in Listing 6. To ensure unique identification for each basic block, IDs are generated deterministically during the instrumentation process. We base this generation on the method name and signature, which provide a distinctive identifier for each method.

```
1 .method public d(Lcom/dropbox/common/taskqueue/c;)V
2 -     .registers 2
3 +     .registers 3
4 +     const v0, 0xd46f
5 +     invoke-static/range {v0 .. v0}, Lcom/<redacted>/
  ↪  coverageagent/Instrumentation;->reachedBlock(I)V
6       iget-object p1, p0, Lcom/dropbox/product/dbapp/
        ↪  downloadmanager/b$b;->b:Ldbxyzptlk/dq0/b;
7       invoke-interface {p1}, Ldbxyzptlk/dq0/b;->onCancel()V
8       return-void
9 .end method
```

Listing 5: Smali bytecode of a single-basic block Java function with coverage instrumentation. Lines 3 and 4 are added by MALintent and the register count incremented by 1.

```
1 public static void reachedBlock(int blockId) {
2     coverageMap[(blockId ^ prevBlock) % COVERAGE_MAP_SIZE]++;
3     prevBlock = blockId >> 1;
4 }
```

Listing 6: The instrumentation method in Java. Adds an edge between the previous and current block to the coverage map. This is the algorithm used in the AFL [5] fuzzer.

C. Fuzzing Loop

MALintent crafts an initial set of intents based on the specifications generated from the static analysis step in §III-A3. These intents act as the seeds to start the fuzzing loop. The following initial intents are created:

- An intent with no *action* or *categories* but fulfilling the *data* filter requirements.
- An intent with no *action* but with all required *categories* that fulfills the *data* filter requirements.
- One intent per *action* with no *categories* that fulfills the *data* filter requirements.
- One intent per *action* with all required *categories* that fulfills the *data* filter requirements.

These are all the possible intents that pass the intent filters in the application's manifest and get delivered to the target application as explained in §III-A1. The *action* and *category* filters allow for cases where they are empty but the *data* filter requires that the data URI always meet the requirements.

1) Sending Intents: MALintent uses the Android Debug Bridge (ADB) to communicate with an Android emulator or device in order to send intents to the target application. We invoke the Android `ActivityManagerService` [22] through ADB, which then dispatches the intent to the target application. The intent is serialized by the fuzzer and sent across ADB in the form of command-line arguments. How the values are passed depends on the extra type: for regular data types such as integers and strings, we pass the values as command-line arguments. For extras of the content type, we provide the fuzzing data in a file or via a content provider, which is Android's recommended interface for sharing larger data among applications. In addition, the data field of the intent can hold either of the three input types: a direct value as a command-line argument, the path of a file, or the URI of an object provided by a content provider.

To collect accurate feedback information, MALintent needs to gather coverage from the point the intent is delivered until the application becomes idle after processing the intent. We use the `ActivityThread`, which oversees the main thread, activities, and broadcast operations, to detect when the application reaches an idle state. Once the `Idler` in the activity thread identifies the target activity as idle, the coverage agent of MALintent stops recording executed basic blocks, marking the completion of intent execution.

After the intent has been delivered and the application finished processing it, MALintent communicates with the coverage instrumentation system in §III-B to retrieve the latest coverage map. The fuzzer then compares this coverage information with that of prior runs to determine if the fired intent discovered new coverage.

Aside from coverage, the instrumentation system also sets callbacks to collect data on how `Intent` objects get used dynamically. This allows MALintent to update its intent specification as it reaches deeper code paths. For example, runtime usages of `hasExtra` are monitored to gather additional intent keys and `registerReceiver` is intercepted to identify broadcast receivers that are registered at runtime. Listing 7 demonstrates code where MALintent dynamically identifies the types and keys for intent extras that were not constants.

```
int index = 0;
while (intent.hasExtra(URI_EXTRA + "_" + index)) {
  Uri uri = Uri.parse(
    intent.getStringExtra(URI_EXTRA + "_" + index)
  );
  mediaItems.add(
    createMediaItemFromIntent(uri, intent)
  );
  index++;
}
```

Listing 7: Intent handling code from a media player application where the intent extra key is not a constant. MALintent's dynamic instrumentation allows it to identify the key at runtime.

2) Mutating Intents: After an intent has been determined to add new coverage, it is added back to the fuzzing queue to be mutated. Our approach mutates the input intents in a structure-aware way. We implement several mutation strategies in MALintent to mutate the different parts of the intent.

Data Mutator. Most importantly, MALintent mutates the data field of the intent. This field can directly hold a sequence of bytes, the path of a file, or the URI of an object provided by a content provider. Our mutator modifies both the type of the input and its contents.

MIME Type Mutator. With the MIME type, the sender of the intent can specify the type of the intent data. MALintent comes with a mutator to change the MIME type.

Flag Mutator. MALintent also mutates the *flag* field of the intent, which impacts how the Android OS delivers the intent to the target application. (e.g,. the flag field controls the back button behavior of the launched activity).

Extra Mutators. MALintent incorporates a series of mutators to modify the extra keys. One mutator introduces additional extras, while another alters the keys of the existing extras, utilizing keys derived from both the static analysis and dynamic discovery. MALintent also includes mutators for the extra content, type, and suffix of files, applicable when the extra content is a file path or a URI. As discussed in §III-C1, the content of the extra could be a standard type like an integer, string, or byte array, which are directly mutated. Alternatively, it could be a file path or URI, where the file content is mutated. Another mutator adjusts the input type—file, content provider, or direct input. Lastly, a mutator modifies the suffix of the file or URI using a list of common file types.

D. Bug Oracles

MALintent uses multiple different oracles to detect whether the execution of an intent has triggered a security issue. Denial-of-service issues in the form of crashes are detected by the crash oracle. The privacy oracle uses dynamic dataflow analysis to find privileged API calls such as placing phone calls or accessing the filesystem where inputs are controllable through an intent. The novel JNI memory safety oracle uses dynamic traces of native code in order to fuzz these code components and find memory safety issues inside them.

1) Crashes: The crash oracle aims to identify intents that crash the target application. This is usually caused by app developers utilizing poor error handling practices, in particular, around nullability. Many developers directly use objects from the intent extras without checking if they are present with `hasExtra` as shown in Listing 8.

```
String text = intent.getStringExtra(Intent.EXTRA_TEXT);
if (text.startsWith("tel:")) {
    // ...
```

Listing 8: Intent handling code that throws a null pointer exception when the extra `TEXT` is not present.

Such a crash allows a malicious app to be able to continuously deny a user access to an app. If such a crash is present in a critical system app such as the dialer, this can lead to a user being unable to call emergency services.

MALintent monitors the Android logcat utility to check the system crash log buffer. When an application crashes, Android dumps its Java stack trace to the logcat buffer. By taking this stack trace and hashing the call stack, MALintent deduplicates unique crashes.

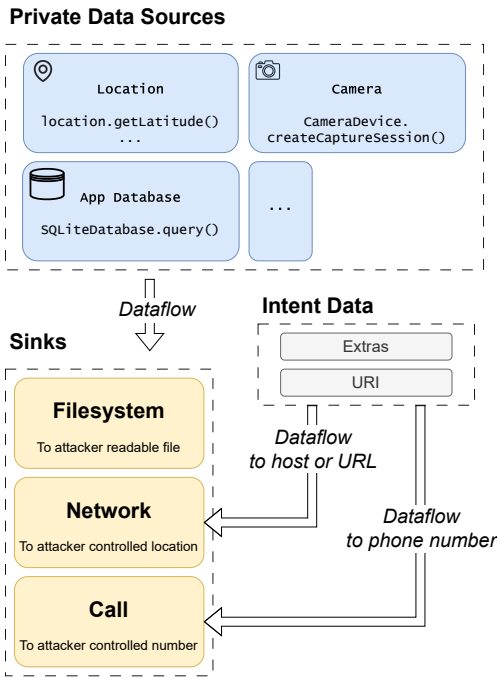

Fig. 4: Overview of the sources and sinks in MALintent's privacy oracle. Dataflow is tracked from private sources such as the user's location and camera to sinks where the attacker can read the data.

2) Privacy Violations: The privacy violation oracle aims to find bugs where the user's private data such as their location, camera, or microphone can be determined by the attacker from an unprivileged application without the user's consent. The latter part is key, while it is expected for an app to be able to request that the user take a photo through their camera, or pre-fill a phone number to dial, it is not acceptable for the app to unilaterally take a photo or place a call unless it has been permitted to do so.

MALintent approaches this problem by performing dynamic taint analysis [23] on the application. In our analysis, we consider our *source*s to be APIs that access privacy-sensitive data such as the camera or location. The *sink*s we track are filesystem locations and network locations that can be read by the attacker application. Since MALintent performs no GUI interaction with the target application, any path from a source to a sink constitutes a leak of private data: from an application with high privilege to the attacker application. An overview of these sources and sinks is shown in Figure 4.

While fuzzing, if privacy sensitive data flows from their respective APIs to places readable by the attacker application, a privacy violation is detected. Concretely, MALintent uses taint analysis sources identified in the past work FlowDroid [24]. This includes Android APIs such as `Location.getLatitude`, access to the application's internal database with `SQLiteDatabase.query`.

For sinks, MALintent uses a more conservative set of easily verifiable sinks to minimize false positives. In particular, it tracks filesystem access and considers any files created that would be readable by the attacker application to be a valid sink. This means an application writing to its own private data directory will not be considered but writing to a world readable folder would be.

MALintent also checks for situations where there is a dataflow from a privacy sensitive function to the network *and* the network location (socket connection host, URL, etc.) has a dataflow path from the intent. This captures behaviors where the application can be tricked into sending private data to a server controlled by the attacker. Lastly, MALintent also tracks dataflow from intents to `SMSManager.sendTextMessage` and `TelecomManager.placeCall` to detect telephony and messaging apps placing calls or sending messages without the user's consent to attacker controlled numbers.

3) JNI Memory Safety: While Android applications are typically built in Java or Kotlin, both memory-safe languages, Android allows applications to include native code written in memory-unsafe languages such as C/C++. This native code can then be called from Java using the JNI [25]. This feature allows optimized code to improve the performance of critical code paths such as graphics rendering. Developers may also prefer to reuse mature C/C++ libraries such as *giflib* [26] instead of rewriting them in Java. Third parties also often distribute proprietary libraries as closed-source shared objects for advertising frameworks [27] and digital rights management [28]. Writing and using Java wrappers around native libraries can be challenging as developers have to ensure the Java code upholds the invariants of the native code. Due to this difficulty and the high impact of memory corruption issues, native code has been a valuable target for attackers in the past [29], [30], [31].

MALintent, therefore, incorporates an oracle that observes JNI code invoked during the execution of intents to discover memory safety issues in the native code. From dynamic execution traces, the oracle generates fuzzing harnesses for native code included in the application. The fuzzing harnesses wrap the native libraries at the JNI boundary, which allows us to fuzz the libraries. A major component of the fuzzing harnesses is the emulation of the JNI environment, which the libraries use to interact with the state of the JVM.

Dynamic Invocation Sequences. MALintent collects invocation sequences of Java methods implemented in native code, also called native methods. For each input of the fuzzing corpus, MALintent creates a sequence of the native methods invoked at run time. Each entry includes the invoked native method, its concrete argument, and return values which MALintent later uses to generate the fuzzing harnesses. Additionally, MALintent enhances the traces by applying two checks to determine if the input is passed via a file. In particular, MALintent marks an argument in the trace as file input, if:

- The argument type is a string, and the argument contains the path of an existing, regular file, or
- The argument type is an integer, and the argument value, interpreted as a file descriptor, refers to an existing, regular file in the file system.

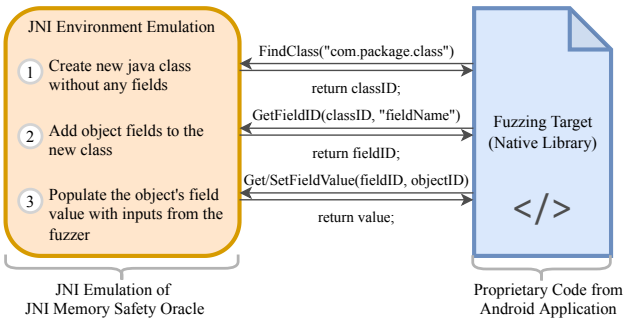

Fig. 5: JNI Environment Emulation. The native library, which is called from the fuzzing harness (not displayed), interacts with the JNI environment. Our emulation creates objects with lazy instantiation and keeps track of objects with which the native library interacts.

Harness Generation. Using the dynamic invocation sequences, MALintent constructs fuzzing harnesses for an Android application's native libraries. The invocation sequence is filtered according to a targeted Java class, which clusters corresponding native functions semantically. Native methods are only included in the harness if they are invoked, that is when an entry in the invocation sequence exists. Invocation sequences are unified simplistically by forming a tree of invocations. Two invocations from different sequences are represented by the same node in the tree, if they share the same list of previous invocations. MALintent then flattens the tree and automatically creates a harness in C utilizing a template we created. The harness then gets compiled and linked against the native library intended for fuzzing. This harness also includes the emulated execution environment.

Execution Environment Emulation. We design an execution environment that emulates the runtime of the native libraries, namely the JNI environment. Our environment includes an emulation stub for each of the 233 JNI Environment (JNIEnv) functions of the `JNIEnv` structure. The emulation stubs emulate the behavior of the JVM and, by default, return values from the fuzzing input stream. We have replaced 40 of those stubs with more complex functions to consistently model Java objects. The subsequent paragraphs describe the aspects of the JNIEnv emulation.

JVM state and lazy object instantiation. For correct emulation of the JNIEnv, it is necessary to maintain the state of the emulated Java objects and consistently return the same values in subsequent calls that request the same resources. Additionally, the full state of the JVM is unknown since our emulation cannot be aware of the entire state of the JVM; yet, we need to emulate those unknown objects as they are accessed.

Our emulated environment uses the design of the JNIEnv to employ lazy object instantiation and internally maintains the state of the objects that were already used. For instance, to get or set the value of an object member, the fuzzing target needs to call multiple JNIEnv functions. Figure 5 illustrates a typical interaction with the emulated JNIEnv.

1) `FindClass`. First, the fuzzing target requests the ID of the Java class, given the class name as a string. If not previously used, the emulation will create a new opaque ID for the class, add the class with the ID to the list of all classes, and return the class's ID.
2) `GetFieldID`. Second, the fuzzing target gets the ID of the desired field of the Java class, given the class ID and the field name as a string. Similarly to the class ID, the emulation adds the new field with a new opaque ID to that class and returns the field ID.
3) `Get/Set<Type>Field`. Last, the fuzzing target obtains or sets the value of a field, given the field and object IDs. When first accessing a field, the emulation populates the field value with fuzzing input. Subsequent get and set operations will read or write the value stored in the emulated Java object.

This design makes the JNI emulation generic so it can provide reasonable fuzzing input even for objects that are initialized in the Java code.

IV. Implementation

MALintent is implemented in four distinct components: **(1)** The coverage instrumentation JVMTI client consists of 500 lines of C++ code. **(2)** The static analysis for the APK is done in 400 lines of Kotlin code. **(3)** The intent fuzzer itself is implemented using LibAFL [32] in 2500 lines of Rust. **(4)** Crash and privacy oracles are written in 2000 lines of Kotlin code, with the privacy oracle requiring modifications to Android itself. The JNI memory safety oracle consists of 400 lines of C++ code for the dynamic instrumentation, 1400 lines of Python to generate the harness, and 3100 lines of C for the execution environment emulation.

V. Evaluation

To evaluate MALintent, we performed experiments and gathered data to answer the following research questions related to MALintent's main contributions:

RQ1 *Is MALintent's coverage instrumentation method more widely compatible than past approaches on Android?*
RQ2 *Does MALintent's coverage feedback mechanism allow it to reach deeper behavior?*
RQ3 *Does MALintent improve over the state of the art?*
RQ4 *Does MALintent's modular bug oracle system allow it to discover bugs from past intent bug discovery works?*
RQ5 *How much static overhead does MALintent require?*
RQ6 *Was MALintent able to discover novel bugs related to intent security? What is the impact of those bugs?*

For our experiments, we ran the fuzzer on servers with AMD Ryzen 9 3900X 12-Core CPUs (3.8 GHz) and 32 GB RAM. The servers run Ubuntu 18.04 as the host OS. On the servers, we used virtual Android devices based on a dockerized version of Android 13 [33] with Google Play services.

A. Coverage Instrumentation

To answer RQ1, we conducted an empirical study on approaches to instrumenting closed-source apps for coverage

Coverage System	Supported Android Versions	Incompatibility Reason
MALintent	**8 – 14**[a]	pre-jvmti Android versions
COSMO	2 – 5	multi-dex, dex format
ACVTool	1 – 5	multi-dex
ELLA	1 – 5	multi-dex, dex format
BBoxTester	2 – 5	multi-dex, dex format

[a]Latest Android version at time of writing

TABLE I: Comparison of MALintent's coverage instrumentation compatibility with major Android versions compared to prior coverage instrumentation systems.

feedback on Android. From a literature review, we collected the following systems used in prior works:

- COSMO: *Code Coverage Made Easier for Android*. [16]
- ACVTool: *Fine-grained code coverage measurement in automated black-box Android testing*. [15]
- ELLA: *A Tool for Binary Instrumentation of Android Apps*. [34]
- BBoxTester: *Towards Black Box Testing of Android Apps*. [35]
- InsDal: *A safe and extensible instrumentation tool on Dalvik byte-code for Android applications*. [36]
- CovDroid: *A Black-Box Testing Coverage System for Android*. [37]

Note that it was not possible to evaluate InsDal or CovDroid because the authors did not make their tool available and their source code is not public.

We evaluated which major versions of Android each coverage instrumentation system is fully compatible with. This was done by compiling an application with each possible Android `compileSdkVersion` and then using each tool to try to instrument it for coverage. These results are shown in Table I. MALintent works with a wide range of Android versions up to the latest version of 14 at the time of writing starting at the version where the JVMTI API was made available.

ACVTool, the most mature of the binary coverage instrumentation systems available is not fully compatible with Android version 5 and up due to its lack of support for Android's new multi-dex bytecode format [38]. ELLA also lacks support for the new multi-dex format and additionally uses a dex bytecode writing system that does not support the newest instruction format. COSMO and BBoxTester both rely on the tool `dex2jar` to first lift Android bytecode to a `.jar` file and then use traditional Java instrumentation tools. The `dex2jar` version used by them does not support the multi-dex format and COSMO's evaluation excludes all multi-dex apps for this reason. Additionally `dex2jar` does not support dex bytecode instructions from Android 8 and upwards [39].

> MALintent is compatible with Android versions 8 to 14 (the latest version as of writing), surpassing other coverage tools in terms of compatibility.

B. Effect of Coverage Feedback on Fuzzing

For RQ2, we evaluate whether the use of coverage feedback helps MALintent to reach deeper behavior in apps. Therefore, we run the MALintent intent fuzzer twice for each target application: one time with coverage feedback and another time without feedback. We compared how many edges of the CFG the fuzzer covered with and without our feedback. We conducted this experiment on two different datasets:

- *Top Google Play (TGP)*: The top-50 overall most popular and the top-50 productivity Android apps on the Google Play store as of March 2023 (those sets did not overlap at that time).
- *F-Droid (FDROID)*: 500 randomly sampled applications from the F-Droid Android app repository [40].

For TGP, we the fuzzer was run for 4 hours on each app component; for the larger FDROID dataset, we allocated 4 hours of fuzzing for each app.

On the TGP dataset, we found that MALintent covered additional edges on 81.8% of all applications with coverage enabled. On average, the coverage feedback helped to discover 3.1 times more edges, with a mean of 4.6% overall more edges (standard deviation of 9.6).

On the FDROID dataset, MALintent was able to cover additional 16.5% edges on average, with a mean of only 1.4% more edges (standard deviation of 0.7). We found that the smaller improvement on the FDROID dataset stems from the simplicity of those apps: In general, applications on F-Droid have fewer functionalities and less code. Thus, on these simpler apps, a fuzzer without feedback is able to trigger a larger part of the intent handling code such that the use of the coverage feedback helps cover more code only on fewer apps. Nonetheless, MALintent was able to cover additional edges on 56.5% of the apps in FDROID due to our coverage feedback.

> We conclude that the coverage feedback significantly increases the code coverage in our system MALintent, increasing the edge coverage on 81.8% and 56.5% apps of the Google Play and F-Droid datasets, respectively.

C. Comparison to State of the Art

To address RQ3, we evaluate the performance of MALintent in comparison to previous work measuring code coverage. For this purpose, we conducted multiple runs on each app to determine the significance of the results, opting for longer runs on a smaller set of applications to ensure reliability.

From the TGP dataset, we randomly sampled six applications and executed each fuzzer—MALintent and IccDroid [41]—on each app five times. We excluded an older tool based on MATE [42] because we were unable to get it running; the reproduction package failed to instrument the provided APKs. Despite contacting the authors and access to the source code, we were unable to use the tool to instrument real-world APKs and perform the fuzzing.

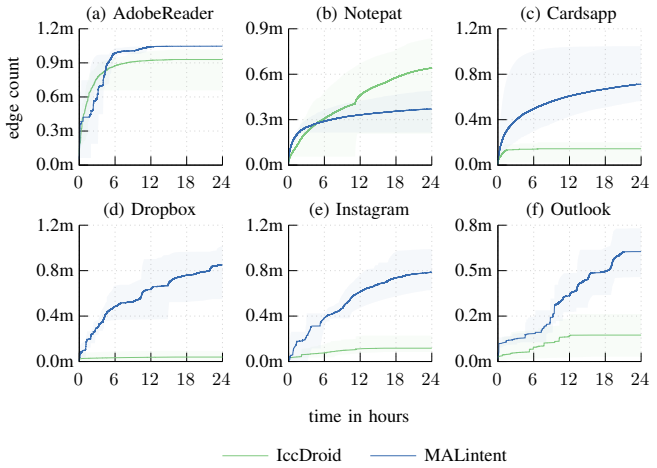

Fig. 6: Coverage comparison between MALintent and IccDroid.

The experiment results are shown in Figure 6, depicting the average code coverage achieved by each tool, with transparent areas indicating the range from minimum to maximum coverage across all runs. Our findings show that MALintent outperforms IccDroid in coverage for all applications except Notepat. A one-sided Mann-Whitney U test returned $p < .01$ for all but Notepat, which had a p-value of .07215 making the result on Notepat insignificant. Thus, we reject the null hypothesis, confirming that MALintent significantly outperforms IccDroid on five out of six apps.

For Notepat, where IccDroid achieved higher average coverage, we found that IccDroid reached additional activities through graphical user interface (GUI) interactions, enabling it to cover more code. However, interacting with the GUI contradicts MALintent's goal of identifying privacy leaks without user interaction. Future research will explore detecting GUI interactions that do not break MALintent's threat model and extending MALintent to trigger those interactions.

> We conclude that MALintent significantly ($p < .01$) covers more code than the state of the art, IccDroid, on most apps.

D. Bug Oracles

To answer RQ4, we evaluate MALintent against past research work trying to automate finding bugs in Android intent handlers. We demonstrate that through its oracles and the generic nature of the fuzzing framework MALintent provides, it was able to rediscover all the bugs found in those projects.

In particular, we found the following past works that focus on security issues in intents:

- *Intent Fuzzer: Crafting Intents of Death* [43]
- *DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag* [44]
- *Identifying Android inter-app communication vulnerabilities using static and dynamic analysis* [45]

System	Coverage Instrumentation	Bug Types
MALintent	Yes	Crashes, Privacy, and Memory Safety
IccDroid [41]	*Only w/ Source Code*	Crashes
Intents of Death [43]	*Only w/ Source Code*	Crashes
DroidFuzzer [44]	No	Crashes, Resource Exhaustion
Demissie *et al.* [45]	No	Privacy
AndroidIntentFuzzer [46]	No	Crashes
MindMacIntentFuzzer [47]	No	Crashes

TABLE II: Comparison of the types of bugs detected and whether the fuzzer uses coverage instrumentation against prior work and open source projects. MALintent is able to use grey-box instrumented fuzzing for all applications and detects a wide variety of bug types.

Prior Work	Bugs Reported	Time to Discover
Intents of Death [43]	10 Crashes	1m
DroidFuzzer [44]	14 Crashes and DoS	2m
Demissie *et al.* [45]	9 Privacy Violations	38m

TABLE III: Evaluation of MALintent's bug findings against application datasets from past works. The time column is the average time in minutes to find each bug.

To summarize, the *Crafting Intents of Death* paper fuzzes to find intents that crash the target application, similar to our crash oracle described in §III-D. It uses a black-box fuzzing approach based on data from APK files. *DroidFuzzer* also uses black-box fuzzing using only the `<data>` portions of the intent filter to find crashes in applications. *Identifying Android inter-app communication vulnerabilities* by Demissie *et al.* uses static data-flow analysis to find flows from intents in an activity to privileged Android OS calls such as the one to dial a phone call. Table II compares these works to MALintent in terms of coverage instrumentation and the type of bugs detected.

We ran MALintent against the datasets reported in each paper to see if our oracles could find the bugs they did. Table III shows the number of vulnerabilities in each application found by those projects and how much time it took to find with MALintent on average.

The crashes described in *Crafting Intents of Death* were found in all 10 of the applications with their reported versions. For *DroidFuzzer* we could not find an `apk` for the exact version of an application they used (MX Player) version `1.7.14`, so we used the next available version `1.7.15`. We confirmed that each of the crashes and resource exhaustion bugs reported in the paper were triggerable by MALintent and detected by its crash oracle. Triggering the bugs described in the work by by Demissie *et al.* took longer as they involved deeper bugs in the application's behavior. Demissie *et al.* indicated that it took 5 hours on average per app for them to discover bugs, showing that our gray-box fuzzing may be a faster approach here.

MALintent's oracles were able to rediscover all issues from past automated intent-security issue-finding works, sometimes faster than them.

JNI Memory Safety Oracle. The JNI memory safety oracle of MALintent tests native code of the Android application, which previous work did not test. To evaluate the effectiveness of the JNI memory safety oracle, we compare the harnesses generated by our framework against baseline harnesses, which we created by removing the data dependencies that MALintent detected. More specifically, MALintent internally uses an intermediate representation (IR), which we designed to describe the native harnesses—for the baseline harnesses, we removed data dependencies from the harness IR and let MALintent continue with the modified IR.

Our evaluation is based on nine target libraries, which MALintent found when running on the TGP dataset and six libraries from the FDROID dataset. We fuzzed each of the harnesses with AFLPlusPlus [48] for 24 hours.

Code Coverage. We evaluate whether MALintent is able to help explore the target libraries. Therefore, we compare the coverage of MALintent and baseline harnesses on the TGP and FDROID datasets, as illustrated in Figure 7 and Figure 8, respectively. We find that our harnesses gain significantly more coverage in most cases, and similar coverage in some cases. In one instance (sqlcipher), the reported coverage is significantly higher in the baseline harness compared to MALintent.

We found that sqlcipher's baseline harness shows inflated coverage due to unstable feedback from handling incorrect inputs. One of the input arguments is a string representing a SQL database file path. If the string points to files modified by the fuzzer between runs, it causes different paths to be triggered, leading to instability and inflated coverage. In contrast, MALintent detects the argument as a file path and provides a file input, but the fuzzer fails to generate the correct SQL database file format in the given time, resulting in poor coverage. Generating input seeds to match the target's expectations is a separate issue and is considered future work.

The baseline harness for xzinputstream couldn't run due to immediate false positive crashes. Overall, we conclude that MALintent effectively generates harnesses that cover the targeted native code.

False Positives. Harnesses based on heuristics can cause false positive crashes by incorrectly modeling API invocation. To assess MALintent's performance, we compared false positive crashes of MALintent and the baseline fuzzer for each harness (see Figure 9). In four out of five cases, MALintent outperformed the baseline, with two cases showing zero false positives and crashes in `pdfjni` and `pdfjni2` being non-reproducible. For `nativedocument`, both MALintent and the baseline had higher false positives due to different root causes. The baseline lacks constraints present in the MALintent harness, while MALintent crashes are non-reproducible or due to incomplete JNIEnv emulation, unlike the baseline. We plan to extend our emulation environment to fix this.

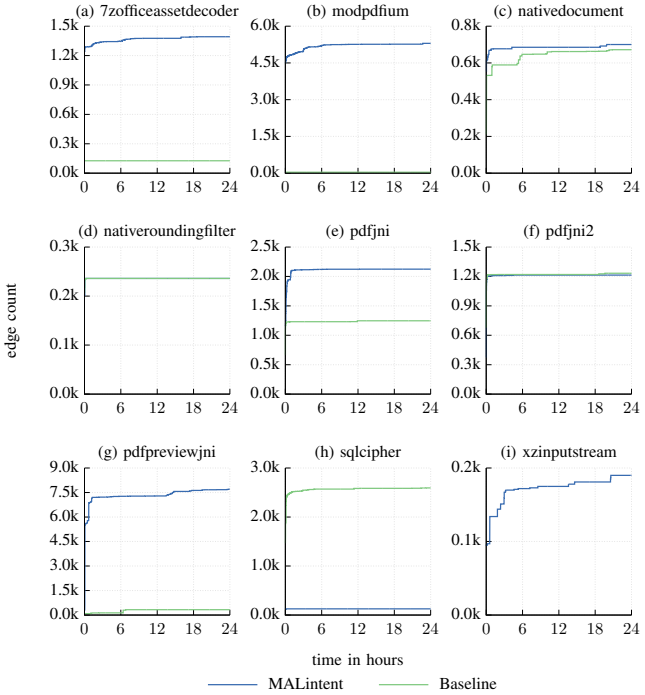

Fig. 7: Coverage comparison between MALintent and baseline JNI fuzzing (Top Google Play dataset).

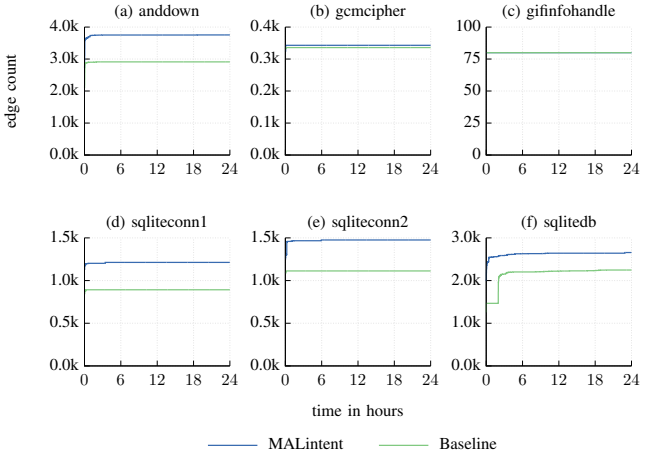

Fig. 8: Coverage comparison between MALintent and baseline JNI fuzzing (F-Droid dataset).

Other baseline crashes result from missing dependencies and incorrect input formats, which MALintent handles by using dynamic invocation sequences for harness generation.

The memory safety oracle in MALintent reduces false positives and increases coverage compared to a naïve fuzzer running against JNI entry points.

E. Static Overhead

To address RQ5, we evaluated the execution time of the static analysis component of MALintent on each dataset. For the larger, simpler FDROID dataset, the average execution time was 7.4 s (median 6 s). The maximum time recorded was

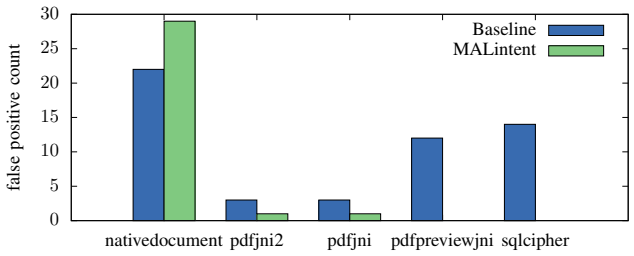

Fig. 9: False positive crashes comparison between MALintent and baseline JNI fuzzing. Lower is better. Only targets that resulted in false positive crashes are included.

75 s, with 75% of applications completing in 10 s or less. On the smaller but more complex TGP dataset, the average execution time increased to 62 s (median 53 s). The longest execution time recorded was 4.5 minutes, though 75% of applications completed in 77 s or less.

> We conclude that the static analysis is well within acceptable time frames, with most apps completing in around one minute, even for complex real-world cases.

F. Bug Discovery

To answer RQ6, we ran MALintent against real-world Android applications to investigate whether it could detect security issues. For these experiments, we ran MALintent on both the TGP and FDROID datasets. We fuzzed each application using MALintent for 16 hours.

Overall Bug Findings. The overall bug findings are presented in Table IV. During its fuzzing, MALintent was able to find 1 memory safety bug, 9 privacy issues, and 49 crashes.

Crashes. The 49 crashes were generally caused by null pointer exceptions in applications using methods like `getStringExtra` before checking `hasExtra`. There were also some index out of bounds exceptions caused by assumptions of the length of arrays passed as intent extras. Each crash report has an exact intent to trigger the bug and had no false positives.

Privacy Violations. Privacy violations were found in 9 apps. We filtered out 4 invalid reports where our dataflow analysis incorrectly showed sensitive data in attacker accessible files even though it had been redacted. We analyze and describe four privacy violations in greater detail:

(1) In the Chrome web browser, there was an exposed `TracingControllerAndroidImpl` broadcast receiver used to begin GPU profiling data in the browser. This receiver accepted an intent with a string key of `file` corresponding to where to store the profiled data. This profiling data reads from the app's private database and contains sensitive user information including the URLs of the tabs open in the browser as well as their headers (including session cookies).

(2) In WhatsApp, a chat application, there is an exported `CameraActivity` component. When sent with an action of `uri` pointing to a local file (or content provider) and with the boolean extra `more_images` set to true, the camera immediately takes one picture and stores it in a location where an attacker app can read it.

(3) The TextNow application, an alternative dialer that utilizes Voice over Internet Protocol (VoIP), contains a `DialerActivity` that can be sent intents. If an intent is sent containing a boolean extra `answer_call` with a value of true and a `phone_number`, the application calls the phone number without the user's consent.

(4) In Forbis VideoCall, the main `SplashActivity` can receive an intent extra `api_url` nested inside a `SharedIntent.extras` bundle. This causes the application to make a connection to the URL specified with some metadata from the application.

Bugs 1 and 4 leak privileged data to the attacker. Bug 2 allows a malicious application to access the user's camera without permission. Bug 3 can cause an unprivileged app to access the user's microphone by having them call the attacker.

JNI Memory Safety. The memory safety bug was found in Fresco, a common GUI framework used by Facebook and Instagram. The framework provides an API to add rounded corners to an image. This feature is implemented in native code and invoked through the JNI. The native implementation uses `memset` to modify the bitmap data of the provided image. However, MALintent found an off-by-one error in the calculation of the destination pointer that is passed as argument to `memset`. The off-by-one lets the destination pointer point past the bitmap array resulting in an out-of-bounds write. *Four other reports were found to be duplicated false positive crashes from MALintent's incomplete JNIEnv emulation.*

> MALintent was able to discover 47 crashes, 9 privacy violations and 1 memory safety issue in the Top Google Play and F-Droid datasets.

Responsible Disclosure. We reported all findings to their respective vendors, where applicable. We did not scan for bugs in applications other than the ones specified in the evaluation. All findings were confirmed or fixed by developers except 3 privacy bugs and 12 crashes in umaintained applications where we did not receive a response.

VI. Discussion

A. Coverage Instrumentation

Coverage instrumentation for closed-source applications on Android ecosystem has been a consistent challenge. Prior research becomes hard to evaluate when coverage tools only support outdated versions of Android [49]. These old versions of Android are usually unable to run the latest applications limiting evaluations to older code with less mature security and engineering practices. In order to improve this situation, we have released MALintent's coverage instrumentation tool as

Application	Component	Oracle	Bug Description
Instagram	`libnative-filters.so`	JNI Memory Safety	Out-of-bounds write in the Fresco GUI framework used by Instagram and Facebook.
Chrome Browser	`TracingController`	Privacy Violation	Exposed memory profiler leaks private browser data: URL and headers including cookies.
WhatsApp	`CameraActivity`	Privacy Violation	Sending an intent with the `uris` extra pointed to an attacker controlled content provider and `add_more_images` set to true causes a camera image to be saved.
TextNow	`DialerActivity`	Privacy Violation	An intent with the boolean extra `answer_call` set to true and `phone_number` causes the app to call up a number without user interaction or consent.
Forbis VideoCall	`SplashActivity`	Privacy Violation	Creates a connection to an attacker controlled URL through the `api_url` extra.
Easy-phone	`RealCallActivity`	Privacy Violation	Calls an attacker controlled phone number passed through the `NUMBER_TO_CALL` extra.
OpenGPX	`CacheListActivity`	Privacy Violation	App accepts an arbitrary URI to copy gpx map files. Path traversal allows copying app's files including user location.
AndrOBD GpsProvider	`GpsProvider`	Privacy Violation	AndrODB allows different plugins to provide data to a central application. The GpsProvider plugin allows leaking of GPS data.
RethinkDNS	`HomeScreenActivity`	Privacy Violation	Allows restoring config from backup, can set a malicious proxy and intercept all traffic.
OpenCamera Sensors	`MainActivity`	Privacy Violation	Sending `VIDEO_CAPTURE` with `DURATION_LIMIT` extra captures video without interaction.
WhatsApp	`HomeActivity`	Crash	Sending an intent with the `NDEF_DISCOVERED` action and no `code` extra causes a null pointer exception.
WhatsApp	`HomeActivity`	Crash	Sending an intent with an empty `NDEF_MESSAGES` extra causes an index out of bounds.
47 other crashes omitted for space			

TABLE IV: Security bugs found by running MALintent against the top-50 productivity, top-50 most popular apps on the Google Play Store (TGP) and 500 randomly sampled F-Droid apps (FDROID).

an independent open source project that can be used by others at https://github.com/sslab-gatech/AndroidCoverageAgent.

As described in §III-B, MALintent's binary instrumentation method uses the Android JVMTI API instead of decompiling and recompiling application `apk`s like past approaches. MALintent's technique carries significant advantages, first being its broad compatibility with Android versions.

Secondly, some apps check their own signatures before executing. Developers also utilize Android's permissions system where certain permissions are granted to all of the apps by a single developer. For example, Google Docs grants special permissions to Google Calendar to render document previews. In these situations, it is impossible to re-sign the `apk` with the developer's original key causing them not to function. Since MALintent does not recompile applications, they continue to function even with their signature checks.

B. Limitations

The static analysis (§III-A) used by MALintent to generate initial intent specifications is quite simple. If the app is using dynamic code execution techniques such as reflection [50] or obfuscation [51] in its intent handlers, MALintent may be unable to fuzz it efficiently. Additionally, Android allows the registration of `BroadcastReceiver`s at runtime with a `IntentFilter` object in code instead of declaring it in the application manifest [52]. These dynamic receivers are not currently handled by MALintent. Supplementing the static analysis by introspecting the intent receivers through the OS at runtime could be a solution to this problem.

C. Future Work

While MALintent implements a complete approach to fuzzing and mutating intents, there is a wealth of new fuzzing mutation techniques. These could be used to improve the performance of the fuzzer. We built three oracles for MALintent. However, our implementations of the oracles are only prototypes to show the potential of our approach. We release our fuzzer to allow the community to develop additional oracles and explore further possibilities.

For example, our privacy oracle uses a very conservative set of sinks to minimize false positives. More mature dynamic taint analysis work on Android applications would allow the privacy oracle to potentially find more bugs. Another beneficial technique for assisting the fuzzer may be symbolic execution.

Also, our JNIEnv emulation only supports a subset of JNI environment functions. Our emulation is not complete and may not support all native libraries. In addition, our environment emulation does not support the invocation of Java methods through the JNIEnv. While this was not needed in our evaluation, there may exist libraries that would require the correct invocation of Java methods.

Broadly, intents are not the only entry point to Android applications: user input, network events, and Firebase cloud messages also exist. However, in those scenarios, the threat model is not as clearly defined. It depends on the target application: if an attacker can control, for example, a Firebase cloud message or the contents of a network packet used by the target application. Thus, additional research needs to identify, which parts of the input may be controlled by an attacker.

While a new fuzzer would need to be developed, our coverage instrumentation can be readily used in those scenarios.

VII. Related work

GUI Fuzzing. Using automated tools to explore and interact with the graphical interface of applications to maximize coverage has been a widely explored area of research [53], [54], [55], [56], [57], [58], [59], [60]. GUI fuzzing aims to systematically explore the program space to find crashes and bugs rather than security issues. Some GUI fuzzers are coverage-guided but as per our evaluation of MALintent in §V-A, their coverage instrumentation is not robust and breaks with Android updates.

Android Coverage Instrumentation. In the space of coverage instrumentation for closed-source Android applications, there have been attempts that try to lift the bytecode to a Java `.jar` files in order to use traditional Java instrumentation tools. [16], [35] Other tools disassemble the bytecode and then reassemble it back into an application [15], [34], [36], [37]. MALintent uses an approach similar to these tools, however, it modifies the bytecode directly in the Android runtime using the JVMTI instead of having to extract and repackage applications.

Intent Vulnerabilities. In the space of outlining and detecting bugs related to intents, Enck *et al.* [61] pointed out intent broadcast receivers that were unintentionally exposed allowing malicious apps to trigger functionality inside them. ComDroid [1] takes a deeper look at intent vulnerabilities and offers a static analysis tool to detect intent hijacking (a malicious app receiving intents it should not be able to).

More recent static analysis tools around intents have worked to find code paths from intent handlers to privileged OS functions [45], [62], [63], [64]. This includes finding intent handlers that forward intents at a higher privilege level [65].

Intent Fuzzing. Prior work on fuzzing intent handlers has greatly explored black-box intent fuzzing on Android [43], [44], [66], [67], [68], [69]. This existing work generates intents but has no feedback mechanism to explore deeper code behaviors like MALintent. Previous approaches also detect only crashes in the target application rather than finding other types of security bugs like with MALintent's oracles.

JNI Security. Several tools leverage static and dynamic analyses at the JNI level to detect errors in JNI usage [70], [71], [72], [73]. Those tools employ static checks with type-state analysis [70], track exception states with data-flow analysis to detect unsafe JNI operations [71], and provide static and dynamic checks to guarantee type safety in JNIEnv interactions [72]. Those analyses are limited to specific categories of bugs, mainly consider interactions with the JNIEnv only, cannot be used for Android apps, and do not use fuzz testing.

The closest to our JNI-fuzzing component is a Quarkslab blog post [74], which describes setting up a JNI fuzzer with AFLPlusPlus Frida mode. While MALintent also uses AFLPlusPlus, we developed our fuzzer before the blog post was published and, thus, the internals are different: We automatically generate harnesses from a custom harness IR, which we designed only for MALintent internally. In addition, our fuzzer emulates the JNI environment allowing MALintent to automatically instantiate objects, which need to be manually initialized with the Quarkslab setup.

VIII. Conclusion

We introduce MALintent, a framework for fuzzing Intent handlers in Android applications. MALintent uses a novel coverage instrumentation method for coverage-guided fuzzing on Android. MALintent was able to rediscover all intent bugs from previous works as well as 49 new crashes, 9 privacy violations, and 1 memory safety issue in the top-50 overall most popular and top-50 productivity on Google Maps, as well as 500 random F-Droid applications. MALintent's coverage instrumentation is open source and available at https://github.com/sslab-gatech/AndroidCoverageAgent. MALintent's intent fuzzer is open source and available at https://github.com/sslab-gatech/MALintent.

Acknowledgments

We thank the anonymous reviewers for their constructive comments and valuable suggestions that helped improve the quality of this paper.

This material is based upon work supported by the Office of Naval Research under awards N00014-23-1-2387 and N00014-23-1-2095, by the National Science Foundation under grants no. 2229876 and CNS-1749711, by the Defense Advanced Research Projects Agency under grant N66001-21-C-4024, and is supported in part by funds provided by the National Science Foundation, by the Department of Homeland Security, by the Technology Innovation Institute (UAE), by IBM, by Facebook, by Mozilla, by Intel, by VMware, and by Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or its federal agency and industry partners.

References

[1] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 239–252. [Online]. Available: https://doi.org/10.1145/1999995.2000018

[2] Curesec, "CVE-2013-6271: Remove device locks from android phone," *Curesec Security Research Blog*, Nov 2013, accessed 27-01-2024. [Online]. Available: https://curesec.com/blog/article/blog/CVE-2013-6271-Remove-Device-Locks-from-Android-Phone-26.html

[3] I. Mohamed and D. Patel, "Android vs ios security: A comparative study," in *2015 12th International Conference on Information Technology - New Generations*, 2015, pp. 725–730.

[4] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1032–1043. [Online]. Available: https://doi.org/10.1145/2976749.2978428

[5] M. Zalewski, "American fuzzy lop," 2017, accessed 27-01-2024. [Online]. Available: http://lcamtuf.coredump.cx/afl

[6] R. Swiecki, "Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options," 2017. [Online]. Available: https://github.com/google/honggfuzz

[7] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 861–875. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[8] skylot, "jadx: Dex to Java decompiler," 2023, accessed 27-01-2024. [Online]. Available: https://github.com/skylot/jadx

[9] E. Levinson, "RFC 2387: the mime multipart/related content-type," Network Working Group, Tech. Rep., 1998.

[10] P. Godefroid, "Fuzzing: Hack, art, and science," *Commun. ACM*, vol. 63, no. 2, p. 70–76, jan 2020. [Online]. Available: https://doi.org/10.1145/3363824

[11] D. Bruening and Q. Zhao, "Tutorial: Building dynamic instrumentation tools with dynamorio," in *International Symposium on Code Generation and Optimization (CGO 2011)*, 2011, pp. xxi–xxi.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, p. 190–200, jun 2005. [Online]. Available: https://doi.org/10.1145/1064978.1065034

[13] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41. Califor-nia, USA, 2005, p. 46.

[14] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1683–1700. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/nagy

[15] A. Pilgun, O. Gadyatskaya, S. Dashevskyi, Y. Zhauniarovich, and A. Kushniarou, "An effective android code coverage tool," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2189–2191. [Online]. Available: https://doi.org/10.1145/3243734.3278484

[16] A. Romdhana, M. Ceccato, G. C. Georgiu, A. Merlo, and P. Tonella, "Cosmo: Code coverage made easier for android," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021, pp. 417–423.

[17] A. Pilgun, O. Gadyatskaya, Y. Zhauniarovich, S. Dashevskyi, A. Kushniarou, and S. Mauw, "Fine-grained code coverage measurement in automated black-box android testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, jul 2020. [Online]. Available: https://doi.org/10.1145/3395042

[18] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth, "Slicing droids: Program slicing for smali code," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 1844–1851. [Online]. Available: https://doi.org/10.1145/2480362.2480706

[19] G. Sharma and D. Hiran, "Reverse engineering for potential malware detection: Android apk smali to java," *Journal of Information Assurance & Security*, vol. 15, no. 1, pp. 26–34, 2020.

[20] J. Howarth, I. Altas, and B. Dalgarno, "Information flow control using the java virtual machine tool interface (jvmti)," in *2010 International Conference on Availability, Reliability and Security*, 2010, pp. 689–695.

[21] A. O. S. Project, "platform/tools/dexter," 2023, accessed 27-01-2024. [Online]. Available: https://android.googlesource.com/platform/tools/dexter/

[22] Q. Gan and H. Wu, "The research of android broadcast intercept technology based on priority," in *2012 Fourth International Conference on Multimedia Information Networking and Security*, 2012, pp. 556–559.

[23] W. You, B. Liang, W. Shi, P. Wang, and X. Zhang, "Taintman: An art-compatible dynamic taint analysis framework on unmodified and non-rooted android devices," *IEEE Trans. Dependable Secur. Comput.*, vol. 17, no. 1, p. 209–222, jan 2020. [Online]. Available: https://doi.org/10.1109/TDSC.2017.2740169

[24] 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," *SIGPLAN Not.*, vol. 49, no. 6, p. 259–269, jun 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594299

[25] Oracle, "Java native interface specification," 2021, accessed 27-01-2024. [Online]. Available: https://docs.oracle.com/en/java/javase/17/docs/specs/jni/index.html

[26] E. S. Raymyond, "giflib," 2019, accessed 27-01-2024. [Online]. Available: http://giflib.sourceforge.net/

[27] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim, "FLEXDROID: enforcing in-app privilege separation in Android," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2017/09/flexdroid-enforcing-in-app-privilege-separation-android.pdf

[28] M. Barbareschi, A. Cilardo, and A. Mazzeo, "Partial FPGA bitstream encryption enabling hardware DRM in mobile environments," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 443–448. [Online]. Available: https://doi.org/10.1145/2903150.2911711

[29] National Institute of Standards and Technology, "National vulnerability database - CVE-2019-11931 detail," 2019, accessed 27-01-2024. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-11931

[30] ——, "National vulnerability database - CVE-2019-11932 detail," 2019, accessed 27-01-2024. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-11932

[31] ——, "National vulnerability database - CVE-2019-11933 detail," 2019, accessed 27-01-2024. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-11933

[32] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "Libafl: A framework to build modular and reusable fuzzers," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1051–1065. [Online]. Available: https://doi.org/10.1145/3548606.3560602

[33] Z. Zhou, "Redroid—remote android," 2024, accessed 06-08-2024. [Online]. Available: https://github.com/remote-android

[34] S. Anand and L. Clapp, "ELLA: A tool for binary instrumentation of android apps," 2016, accessed 27-01-2024. [Online]. Available: https://github.com/saswatanand/ella

[35] Y. Zhauniarovich, A. Philippov, O. Gadyatskaya, B. Crispo, and F. Massacci, "Towards Black Box Testing of Android Apps," in *10th International Conference on Availability, Reliability and Security*, ser. ARES, 2015, pp. 501–510.

[36] J. Liu, T. Wu, X. Deng, J. Yan, and J. Zhang, "Insdal: A safe and extensible instrumentation tool on dalvik byte-code for android applications," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 502–506.

[37] C.-C. Yeh and S.-K. Huang, "Covdroid: A black-box testing coverage system for android," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 3, 2015, pp. 447–452.

[38] A. Developers, "Enable multidex for apps with over 64k methods," 2023, accessed 27-01-2024. [Online]. Available: https://developer.android.com/build/multidex

[39] habreil, "dex2jar issue #333: Is there any plan to support android q version?" 2019, accessed 27-01-2024. [Online]. Available: https://github.com/pxb1988/dex2jar/issues/333

[40] F-Droid Contributors, "F-droid—free and open source android app repository," 2010, accessed 04-08-2024. [Online]. Available: https://f-droid.org/

[41] H. Guo, T. Su, X. Liu, S. Gu, and J. Sun, "Effectively finding ICC-related bugs in android apps via reinforcement learning," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, 2023, pp. 403–414.

[42] 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.

[43] R. Sasnauskas and J. Regehr, "Intent fuzzer: crafting intents of death," in *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, 2014, pp. 1–5.

[44] 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
[45] B. F. Demissie, D. Ghio, M. Ceccato, and A. Avancini, "Identifying android inter app communication vulnerabilities using static and dynamic analysis," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 255–266. [Online]. Available: https://doi.org/10.1145/2897073.2897082
[46] Fuzion24, "AndroidIntentFuzzer: Android Null Intent Fuzzer," 2015, accessed 17-04-2024. [Online]. Available: https://github.com/Fuzion24/AndroidIntentFuzzer
[47] MindMac, "IntentFuzzer," 2017, accessed 17-04-2024. [Online]. Available: https://github.com/MindMac/IntentFuzzer
[48] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi
[49] sunxiaobiu, "Timemachine issue #15: How can i instrument closed-sourced apk?" 2022, accessed 27-01-2024. [Online]. Available: https://github.com/DroidTest/TimeMachine/issues/15
[50] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection - literature review and empirical study," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 507–518.
[51] D. Pizzolotto and M. Ceccato, "[research paper] obfuscating java programs by translating selected portions of bytecode to native libraries," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2018, pp. 40–49.
[52] M. Alhanahnah, Q. Yan, H. Bagheri, H. Zhou, Y. Tsutano, W. Srisa-An, and X. Luo, "Dina: Detecting hidden android inter-app communication in dynamic loaded code," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 2782–2797, 2020.
[53] 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
[54] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 641–660. [Online]. Available: https://doi.org/10.1145/2509136.2509549
[55] 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
[56] 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
[57] 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*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 245–256. [Online]. Available: https://doi.org/10.1145/3106237.3106298
[58] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical GUI testing of Android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 269–280.
[59] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," *SIGPLAN Not.*, vol. 48, no. 10, p. 623–640, Oct. 2013. [Online]. Available: https://doi.org/10.1145/2544173.2509552
[60] 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.
[61] W. Enck, D. Octeau, P. D. McDaniel, and S. Chaudhuri, "A study of android application security." in *USENIX security symposium*, vol. 2, no. 2, 2011.
[62] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "Covert: Compositional analysis of android inter-app permission leakage," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 866–886, 2015.
[63] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 229–240. [Online]. Available: https://doi.org/10.1145/2382196.2382223
[64] J. Zhong, J. Huang, and B. Liang, "Android permission re-delegation detection and test case generation," in *2012 International Conference on Computer Science and Service System*, 2012, pp. 871–874.
[65] B. F. Demissie and M. Ceccato, "Security testing of second order permission re-delegation vulnerabilities in android apps," in *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–11. [Online]. Available: https://doi.org/10.1145/3387905.3388592
[66] MindMac, "Intentfuzzer," 2013, accessed 27-01-2024. [Online]. Available: https://github.com/MindMac/IntentFuzzer
[67] Fuzion24, "Android null intent fuzzer," 2014, accessed 27-01-2024. [Online]. Available: https://github.com/Fuzion24/AndroidIntentFuzzer
[68] T. Wu and Y. Yang, "Crafting intents to detect icc vulnerabilities of android apps," in *2016 12th International Conference on Computational Intelligence and Security (CIS)*, 2016, pp. 557–560.
[69] K. Choi, M. Ko, and B.-M. Chang, "A practical intent fuzzing tool for robustness of inter-component communication in android apps," *KSII Trans. Internet Inf. Syst.*, vol. 12, pp. 4248–4270, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:53112930
[70] G. Kondoh and T. Onodera, "Finding bugs in java native interface programs," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 109–118. [Online]. Available: https://doi.org/10.1145/1390630.1390645
[71] S. Li and G. Tan, "Finding bugs in exceptional situations of JNI programs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 442–452. [Online]. Available: https://doi.org/10.1145/1653662.1653716
[72] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang, "Safe java native interface," in *Proceedings of IEEE International Symposium on Secure Software Engineering*, vol. 97. Citeseer, 2006, p. 106.
[73] M. Furr and J. S. Foster, "Polymorphic type inference for the JNI," in *Programming Languages and Systems*, P. Sestoft, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 309–324.
[74] E. L. Guevel, "Android greybox fuzzing with AFL++ frida mode," *Quarkslab's blog*, 2023, accessed 06-08-2024. [Online]. Available: https://blog.quarkslab.com/android-greybox-fuzzing-with-afl-frida-mode.html