Hercules Droidot and the murder on the JNI Express

Luca Di Bartolomeo* *EPFL*
Philipp Mao* *EPFL*
Yu-Jye Tung *UC Irvine*
Jessy Ayala *UC Irvine*
Samuele Doria *University of Padua*
Paolo Celada *EPFL*
Marcel Busch *EPFL*
Joshua Garcia *UC Irvine*
Eleonora Losiouk *University of Padua*
Mathias Payer *EPFL*

Abstract

Android developers rely on native libraries to improve app performance, often overlooking the increased security risk. Executed in the same process as the app Dalvik bytecode, vulnerable libraries expose the app to low-level security threats such as access to the app's private data. Vulnerability discovery in this environment exposes several key challenges: (i) coping with complex cross-language interactions between the app running on a high-level runtime environment and the low-level code of native libraries, (ii) inference of a precise interaction model between the app and the library, and (iii) scaling to the breadth of the Android ecosystem.

Automatic harness generation for libraries is challenging, especially in mixed language environments such as Android. Existing work either slices snippets of program code, ignoring the cross-language challenges of bringing up the Android runtime environment or require heavy manual efforts on a limited selection of applications. The current best practice to discover vulnerabilities in native libraries on Android is to task a human analyst to reverse engineer both the app and the library along with manually writing a test harness.

Our solution, named POIROT, *automatically synthesizes fuzzing harnesses* for Android native libraries without source code or manual effort. POIROT supports bidirectional JNI (Java Native Interface) interactions, mimics the app's usage of a native API, and scales to the largest apps on the Google Play Store. We evaluated POIROT on the 3,967 most popular Android apps that use native libraries and report 4,282 unique crashes affecting 934 apps. We triaged 200 crashes and identified 25 bugs affecting 16 native libraries included in 34 high-impact apps such as WeChat (with 3 CVEs assigned). All the bugs have been responsibly disclosed to the respective vendors.

*These authors contributed equally to this work

1 Introduction

Native libraries are common in Android apps: while primarily used for performance reasons, they also introduce security risks. The DVM (Dalvik Virtual Machine) provides an extensive runtime environment for memory-safe Dalvik bytecode, but it does not guarantee safety against native code, which runs in the same process with access to all app data, exposing a typical cross-language attack scenario [39]. Bugs in native code can compromise the app and can serve as a stepping stone to compromise the system [32]. A surprisingly high number of apps ship native libraries containing vulnerabilities [7]. This issue may stem from developers using outdated libraries, or from the usage of custom proprietary libraries, which are less commonly available and thus harder to test for security bugs. Yet, automatically exploring the security impact of native Android libraries remains under-explored.

Related work explored automatic harness synthesis for library fuzzing [21, 29–31, 57, 59]. Those approaches focused on analyzing multiple consumers of a given library. By building and mutating a generic consumer, they try to maximize coverage over the entire library's code while fuzzing. Their focus was to detect bugs in the internals of the library itself, irrespective of the existence of a consumer program that could trigger those bugs. Only a few works [12, 54] focus on vulnerabilities that manifest through a specific consumer's usage of a library. However, they either focus on extracting single-function code snippets (side-stepping the challenges of mixed-language environments [12]) or require large amounts of manual effort for even a small number of carefully picked targets [54].

In this work, we focus on automatically detecting vulnerabilities in Android native libraries that a specific consumer app could trigger, by mimicking the app's usage of the library while fuzzing. This problem poses three major challenges. (i) *App-specific JNI-interaction*: the JNI (Java Native Interface) abstraction layer supports bidirectional interaction between Java and native code. Java functions can invoke native ones, which can also interact with Java objects or invoke Java functions. Supporting those interactions together with the rest of the Android ecosystem in a fuzzing environment is challenging. (ii) *App mimicry*: the generated harnesses should invoke the same sequences of native APIs that the app could potentially perform during normal execution. Furthermore, native calls may pass elaborate data types through the JNI. Those data types could be complex Java objects (e.g., `Bitmap`) or might have constraints (e.g., a string representing a file path). There might also be semantic relationships between parameters (for instance, a `ByteArray` is passed as the first argument, and its size as the second). (iii) *Scalability*: To demonstrate real-world applicability, our system needs to work on a large, non-filtered dataset of apps. However, state-of-the-art approaches in Android static analysis fail to scale to the size and complexity of the majority of modern Android apps [45].

We achieve our goal by designing POIROT, the first effective solution capable of generating fuzzing harnesses to automate the process of vulnerability detection in Android native libraries. To solve the above challenges, the design of POIROT necessitated several features. POIROT leverages the behavior of a specific native library consumer (the app) and benefits from statically analyzing the type-rich Dalvik bytecode of this consumer. POIROT relies on two analysis passes to emulate the app's usage of the library: *call-sequence analysis* to address API call sequences and *argument-value analysis* to constrain the input space. The analysis includes custom optimizations and is tailored to scale to modern large apps. Then, the extracted app-specific interaction patterns are automatically translated into a fuzzing harness. Those harnesses offer a *customized Android environment* that handles the intricate interactions across language contexts. Finally, we engineered an orchestrator framework allowing fuzzing campaigns to scale horizontally on Android virtual devices.

To demonstrate the effectiveness of POIROT, we analyze the 3,967 most popular APKs from the Google Play Store that contain native libraries. We uncover 4,282 crashes in the bundled native libraries. After triaging 200 of them, we identify 34 vulnerabilities across 34 apps.

Our core contributions are as follows:

- We present the first approach for automated and scalable Android native library fuzzing. Our analysis passes extract API call sequences of a library and constrains the input space of parameters by looking at a single consumer app. POIROT exploits type-rich Java information to augment the analysis passes. POIROT can analyze the largest APKs available on the Google Play Store.
- We introduce a way to support arbitrary context switching between Java and native code during fuzzing through the JNI interface.
- We present novel forkserver optimizations to mitigate the very expensive cost of the `fork` syscall in Android.
- We perform an extensive analysis of popular Android apps and responsibly disclose 34 vulnerabilities to app and library developers. So far, 3 CVEs have been assigned.

Following best practices on fuzzing research [46] we strive for full reproducibility. POIROT is available at `https://doi.org/10.5281/zenodo.15586318`.

2 Challenges

JNI-Interaction Awareness (C1) The JNI abstraction layer enables cross-language, bidirectional communication between Java and native code embedded in the same Android app. Native code may create, destroy or invoke Java objects at any time. While the execution initially starts on the Java side, native code may call back into the DVM at any time. In addition, the native side interacts with Android-specific mechanisms (e.g., Activities), and the additional features offered by a mobile device (e.g., sensors). Simply running a fuzzer inside an Android enviroment is not enough. The fuzzing speed is crippled by the large amount of virtual memory mapped in a typical application, significantly slowing down the fork syscall.

App Mimicry (C2) Native libraries encompass a multitude of API functions that can be invoked in various sequences. Those native functions may contain bugs in their implementation (e.g., because apps ship outdated libraries). Android apps usually employ only a subset of these APIs. Consequently, the app should be considered vulnerable only if the buggy function is included in its API usage. Furthermore, the app may call these APIs in a sequence that the library developer did not anticipate, potentially exposing vulnerabilities due to its incorrect usage of the library. Previous approaches [12,21,29–31,57,59] in automatic harness synthesis focused on finding bugs in the library itself, ignoring incorrect usage by the consumer. For example, if an app invokes the decode function of a video codec libray without calling first the relevant initialization functions, then the corresponding crash is due to a bug in the consumer app and not the library.

It is thus necessary to replicate the app's behavior when testing relevant parts of the library. This includes awareness of the parameter types and any eventual parameter constraints (e.g., a `String` parameter could be treated as a file path by the native code, or an `Integer` represents the length of another parameter). However, this requires recovering the app-specific API usage. Currently, identifying such API usage requires a human analyst, who manually reverse engineers and encodes them in a fuzz driver. This process is time-consuming, demanding considerable experience and engineering effort.

Scalability (C3) Unfortunately, there are no datasets with ground truth for vulnerable libraries in Android apps. Even worse, Google Play stopped publishing lists with the most downloaded apps that were used in previous work [7]. The

```java
package example;
public class MyActivity extends Activity {

  public native int nativeOpen(String p);
  public native int nativeOpen2(String p);
  public native String nativeRead();

  public static void onCreate() {
    // this will call JNI_OnLoad
    System.loadLibrary("mylib.so");

    // read user-controlled input
    SharedPreferences prefs =
      this.getSharedPreferences(...);
    File file = new File(prefs.getString(
     "/my/file/path"));

    // pass File Java Object to native side
    int fd = getFd(file);

    // another call to native function
    // that requires native file descriptor
C2  String msg = nativeRead(fd);

    System.out.println(msg);
  }

  public int getFd(File f) {
    // must pass a valid path to file
C2  int fd = nativeOpen(f.getAbsolutePath());
    return fd;
  }

}

}
```

```c
#include <stdio.h>
#include <jni.h>

JNIEXPORT jint nativeOpen(JNIEnv *env, ..., jstring path) {
  // Java style string to C
  char *real_path = env->GetStringUTFChars(path, NULL);
  fd = open(real_path, O_CREAT);
C1 env->ReleaseStringUTFChars(path, real_path);
  return fd;
}

JNIEXPORT jstring nativeRead(JNIEnv *env, ..., int fd) {
  char msg[256];
C2 if (fd == -1) return NULL; // must be valid file descriptor
  read(fd, msg, 256);
  // C style string to Java String
C1 jstring result = env->NewStringUTF(msg);
  return result;
}

JNIEXPORT jint JNI_OnLoad(JavaVM* vm, ...) {
  JNIEnv* env;
  vm->GetEnv(env, ...);
  // find class to which to add native methods
  jclass class = env->FindClass("example.MyActivity");
  static const JNINativeMethod methods[] = {
     {"nativeOpen", "(Ljava/lang/String)I", nativeOpen},
     {"nativeRead", "()Ljava/lang/String", nativeRead}
  };
C1 int rc = env->RegisterNatives(class, methods, 2);
}
```

Figure 1: Example of challenges for fuzzing native libraries. On the left, the Java source code of an application. On the right, the C++ source code of the relevant native library `mylib.so`. Underlined methods highlight every time the execution switches context between language environments. The circles reference the challenges outlined in Section 2.

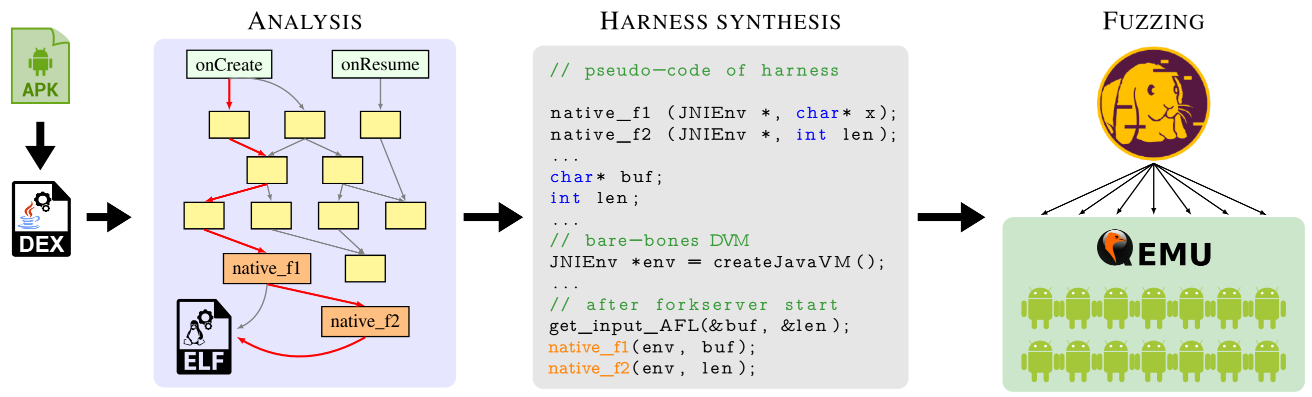
only viable alternative to demonstrate feasibility and scalability is to conduct a large-space analysis, mitigating potential biases through cherry-picked small datasets. The growing complexity of the Android ecosystem, including its apps, is challenging the capabilities of existing static analysis tools [8, 11, 15, 24, 25, 44, 53] to analyze current apps effectively. A recent study [45] proves how tools focused on call graph generation frequently show limitations when analyzing modern apps. Our experimental evaluation in Section 5.3 confirms this and highlights how common are timeouts or empty results.

Example Figure 1 demonstrates an Android app that uses a native library to access a file, highlighting the above challenges. The `JNI_OnLoad` method is a standard method included in all JNI native libraries that takes care of registering the exported native functions to the Java side. The DVM automatically calls it as soon as the library is loaded. The native code invokes the `RegisterNatives` JNI method to dynamically expose functions declared in the Java code (**C1**). The Java code will then call `nativeOpen` with a string. The string must be a valid path to a file (**C2**). The native function `nativeOpen` uses JNI callbacks (**C1**) to allocate and free a buffer containing the bytes of the Java string. Subsequently, `nativeRead` will be called, where a check that a valid file descriptor was opened before will be performed (**C2**).

3 Design

Poirot is a fully automated, harness synthesis framework for large-scale, app-specific native library fuzzing. Poirot discovers bugs through a three-stage fuzzing pipeline (see Figure 2).

First, Poirot extracts the Dalvik bytecode (DEX) and the native libraries from the APK. Poirot analyzes the type-rich Java-based call sites of native library APIs found in the Dalvik bytecode. Poirot employs a set of scalable and efficient static analysis passes to recover relevant information for harness synthesis. For the *call sequence*, we perform the analyses on top of a CFG without relying on expensive static analysis such as symbolic execution or data-flow analysis. For the *argument analysis*, although we rely on data-flow analysis, we avoid scalability issues by performing the data-flow analysis strictly intra-procedurally. The results of the static analysis



```
// pseudo-code of harness

native_f1 (JNIEnv *, char* x);
native_f2 (JNIEnv *, int len);
...
char* buf;
int len;
...
// bare-bones DVM
JNIEnv *env = createJavaVM();
...
// after forkserver start
get_input_AFL(&buf, &len);
native_f1(env, buf);
native_f2(env, len);
```

Figure 2: POIROT finds bugs in Android native libraries in three stages: analysis, harness synthesis, and fuzzing.

Algorithm 1: *doCallSequenceAnalysis*(c, ps)

```
   Input: function call $c$, method $m$'s program paths $ps$ extracted at $c$
   Output: method $m$'s summary store $\Sigma_m$
1  foreach program path $p \in ps$ do
2      $rs \leftarrow \{\ \}$;
3      foreach program statement $s \in p$ do
4          if statement $s$ is a native function call then
5              if $rs$ is empty then
6                  $rs \leftarrow rs \cup [s]$;
7              else
8                  foreach call sequence $cs \in rs$ do
9                      $cs \leftarrow cs \frown [s]$;
10         else if statement $s$ is a function call with function $f$ then
11             $\Sigma_f \leftarrow$ summary store corresponding to function $f$;
12             foreach call sequence $cs \in rs.snapshot$ do
13                 foreach function call $d \in NotPostDom_{calls}(f)$ do
14                     foreach call sequence $cs_f \in \Sigma_f(d)$ do
15                         $rs \leftarrow rs \cup cs \frown cs_f$;
16                         $rs \leftarrow rs \setminus cs$;
17     foreach per path result $r \in rs$ do
18         $\Sigma_m(c) \leftarrow \Sigma_m(c) \cup r$;
```

allows POIROT to synthesize effective consumer-specific harnesses for JNI native libraries.

The second stage automatically synthesizes harnesses based on the analysis results. Those include semantic information of native call parameters (e.g., file paths, array length) and call-sequence information (e.g., `malloc` before `free`). The harness initializes a bare-bones DVM with Android callback support to mimic a real-world environment, enabling complex bidirectional JNI interactions.

Lastly, the harness and the target library are deployed on a farm of virtualized Android emulators on an aarch64 server. An orchestrator script then submits fuzzing jobs to a queue system, from which the emulators get the required data to run the fuzzing campaign.

3.1 Call-sequence Analysis

The key goal of our call-sequence analysis is to identify API call sequences required to properly mimic native library usage by an app. POIROT achieves this by: (i) recovering the app Dalvik call graph; (ii) extracting the intra-procedural paths; and (iii) extracting the inter-procedural paths. To avoid the scalability issues that plague existing Android static analysis approaches [45], the analysis is custom and includes optimizations that mitigate the problem of state explosion allowing it to scale to large apps.

As a first step, POIROT relies on Class Hierarchy Analysis [22] to build the call graph, as suggested by prior works [9, 13, 17, 47]. Second, POIROT performs backward paths extraction from each function call until its method entry using a depth-first search. Third, POIROT uses Algorithm 1 to expand intra-procedural paths into inter-procedural ones.

Algorithm 1 takes as input a target function call and the set of intra-procedural paths from the method entry to the target function call. Based on the example from Figure 1, we describe how the algorithm creates call sequences for the Java methods `onCreate` and `getFd`. The algorithm takes as input the following intra-procedural paths extracted during the second step (denoted by a sequence of program line numbers) for `onCreate`: (10,13,15,19) and (10,13,15,19,23); and the following intra-procedural path for `getFd`: (30). The target function call is the last element in the path.

While traversing a program path, if a program statement is a native function call, it is added to that target's call sequences *rs* (lines 5-9 of Algorithm 1). A new call sequence is added to *rs* if *rs* is empty (line 6 of Algorithm 1). Otherwise, the new call sequence is concatenated to each call sequence in *rs* (lines 8-9 of Algorithm 1). For Figure 1, native function call `nativeRead` is added to the *rs* for `onCreate`'s path (10,13,15,19,23), and native function call `nativeOpen` is added to *rs* for `getFd`'s path (30).

When analyzing a non-native function call f, we extend the call sequence inter-procedurally by iterating over the non-native function call f's summary store (lines 13-14 of Algorithm 1) using $NotPostDom_{calls}(f)$, which returns the function calls in function f that are not post-dominated by other function calls in f (line 13). Program statement s_b post-dominates program statement s_a if every program path from s_a to the function's exit contains s_b. $NotPostDom_{calls}(f)$ allows Algorithm 1 to compute complete call sequences in f and eliminate partial call sequences in f (lines 14-16 of Algorithm 1). For Figure 1, *rs* for `onCreate`'s path (10,13,15,19,23) will contain the following call sequences after traversal: {[`nativeOpen`, `nativeRead`]}. The call sequences are saved to their respective method's summary store—a list of call sequences inside a method—after the program path is traversed (lines 17-18 of Algorithm 1).

Optimizations We design specific pruning optimizations that exploit our specific scope (analyzing only native calls) and enable us to analyze large apps by mitigating the problem of state explosion. During the first step (Dalvik call-graph construction), we prune the call graph by excluding standard library methods (e.g., `"android.*"`, `"com.google.android.*"`, `"com.android.*"`). We exclude those packages as the AOSP standard library has already been fuzzed before and POIROT focuses on bugs in libraries shipped by individual apps. Additionally, during the second step (intra-procedural path extraction), instead of using the classic approach of retaining every instruction along the program path, we only consider native and non-library function calls (i.e., if a basic block does not contain any function call, our analysis will ignore it).

3.2 Argument Analysis

The argument analysis pass collects the semantics of a function's parameters from the context of the call site of the native function. As shown in UTopia [30], awareness of argument values such as array length (one of the parameters is the length of another array-typed argument) greatly reduces false positive crashes.

To collect argument values for a native function, we perform a backward data-flow analysis originating from the actual parameters of the native function. The data-flow analysis is strictly intra-procedural. While this limits its precision, it ensures scalability. The actual parameter is populated with:

1. A constant primitive type,
2. The length of another array-like parameter,
3. A filepath,
4. An empty array,
5. Otherwise, a fuzzer-provided input.

If the data-flow analysis encounters an `array-length` instruction and the addressed object is also passed to the native function as a parameter, a corresponding constraint reflecting this length-value relationship is saved. Furthermore, encountering an `invoke` of a `File` object retrieving its path results in the actual parameter being considered a filepath (e.g., in Figure 1, this pass can extract the value of the `path` variable passed to the `nativeOpen` function). Finally, if the analysis encounters a `new-array` instruction and the resulting empty `Array` object is passed into the native function without modification, we infer that the native function will likely populate the array with some results to be then reused later. If the analysis pass cannot establish any knowledge about the actual parameters, the fuzzer-provided input is used to populate it.

3.3 Harness

We now use the information collected to synthesize a harness. The synthesized harness must accurately reflect the information obtained from the analysis step. Each library harness consists of multiple *fuzz drivers*, each designed to target a specific native function. These fuzz drivers take fuzzer-generated input, split it, and provide the chunks as arguments for function calls.

Android Runtime To support cross-language interactions for JNI fuzzing, mocking can provide the necessary runtime environment to abstract the DVM functions a native library can call [43]. This has many shortcomings, mainly due to the complexity and large surface of the DVM functionality (see Appendix Section A.2 for more details). Instead, we provide a functional but stripped Android environment [23] when calling a native method. The ART (Android RunTime, implemented in `libart.so`) exposes functions to create a bare-bones Java VM along with a corresponding `JNIEnv` structure. JNI native functions always take a pointer to this structure as their first parameter. The `JNIEnv` structure contains a list of callbacks that native libraries use to interact with the DVM (e.g., `GetStringUTFChars`). Furthermore, we load into our bare-bones DVM all the classes that the app under test implements. This allows us to closely replicate the execution environment in which an app typically invokes the native libraries. This approach enables POIROT to support complex bidirectional JNI interactions.

Leveraging Analysis Results We now describe how the results of the analysis passes are encoded in a fuzz driver. A synthesized fuzz driver calls every function of the call sequence in the same order. Since the call sequence reflects the app's usage of the native library, the fuzz driver effectively mimics the app. The call sequence analysis pass might report multiple different call sequences for a given native method. This is common when some native calls are behind certain conditions (e.g., a native initialization method that must be called). In this case, the harness uses the longest callsequence, as it is the one that is most likely to include all state-buliding native methods and mimick the app better. The fuzz driver then employs the results of the argument value analysis pass to constrain the input space of the function arguments. For example, when dealing with an array-length constraint, the argument is set to the length of the fuzzer-generated input. Similarly, for an empty array, the argument is set to a newly-created array. In the case of a filepath, the argument is set to a temporary filepath, and the fuzzer input is written to that file. In all other cases, the raw fuzzer input is parsed into the corresponding Java type (e.g., for a String argument, fuzzer bytes are converted to a Java String object through a call to `NewStringUTF`)

Input/Output Matching Functions in a given call sequence might have data dependencies between them, as the example in Figure 1 illustrates. To avoid expensive computational analysis, we opt for a simple input/output matching based on the argument and return type. For a given function, we check if the parameters' types match the return type of any of the previous functions in the call sequence. In case there is a match, the harness will forward the output of the relevant previous function to the target native call.

Supported Java Types Our fuzz drivers support fuzzing all Java primitive types as well as Java strings, byte arrays, and byte buffers. In our dataset we find that by supporting these primitive types, we support 64% of native functions; see Figure 3. No intrinsic limitation prevents this approach from being extended to other types: only engineering effort is required to transform the fuzzer input into the desired object type. Section 5.7 in our evaluation shows the amount of human effort necessary to add support to an additional type.

4 Implementation

For the sake of reproducibility, we include technical details of the process of replicating a lightweight Android runtime and the signature extraction of functions that native libraries expose. In total, our framework consists of around 7,000 lines of code. POIROT extends Phenomenon [24, 25] to build the Dalvik call graph. Phenomenon relies on Soot [49] to model standard Android framework callbacks. To instrument and fuzz the libraries we leverage dynamic binary instrumentation. We use Frida [2] with its AFL++ integration (version 4.22) [28].

Android Runtime The Android Runtime library `libart.so` provides access to a method called `JNI_CreateJavaVM` that creates a minimal Java VM instance. Additionally, the library `libandroid_runtime.so` exposes the function `registerFrameworkNatives`, which, when supplied with a pointer to an instance of a DVM, calls the `registerNatives` method to detect dynamically registered native methods, and initializes a bare-bones Android environment. Our fuzz driver then loads the native

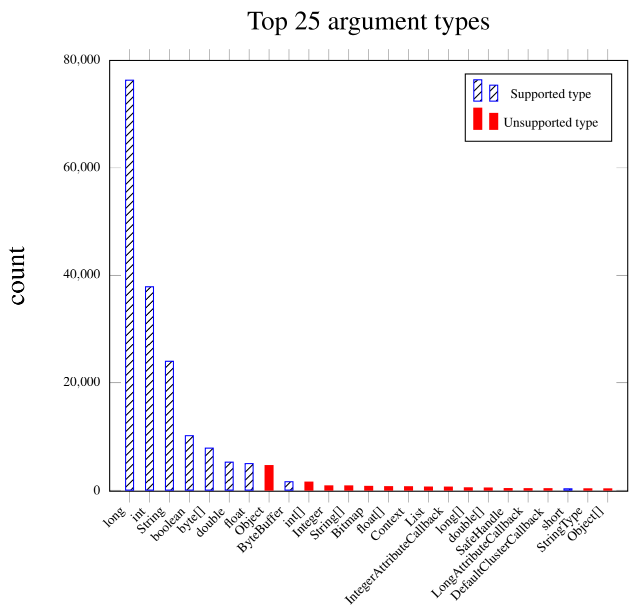


Figure 3: The number of occurences for the top 25 most used argument types in native functions in our dataset. By supporting primtive types, java strings, byte arrays and bytebuffers (13 types out of 7161 types overall), we support 64% of native functions in our dataset.

library through `dlopen` and then calls its `JNI_OnLoad` method to properly initialize the library. The same method used to create a minimal DVM (`JNI_CreateJavaVM`) returns a pointer to the `JNIEnv` structure, which is then passed to the native functions during fuzzing.

Native Function Signature Extraction An app may load multiple native libraries. POIROT must infer which of the loaded native libraries implements an observed native function. Many libraries shipped with APKs are stripped and *do not* export the symbols of the library interface. The library interface is dynamically registered with the Dalvik runtime using the `JNI_Onload` function, which is the only function that must be exposed. POIROT employs the mocking approach described by Rizzo [43] to intercept calls to the `registerNatives` function and populate the array of function pointers that the target library exposes to the Java code.

Path limit To mitigate path explosion during call sequence analysis, we cap the number of paths extracted for each native method. This upper bound prevents the analysis from getting stuck on a relatively small subset of complex methods. This limit is configurable by the user. The authors of Phenomenon report that using a path limit of 100 incurs in a loss of 7.6% of the total paths [25]. To be conservative, we use a default value of 1,000. As our evaluation suggests (Section 5.2), this value is a reasonable choice in the context of our dataset.

Device	W/o PT Opt (execs/s)	W/ PT Opt (execs/s)	W/o ART (execs/s)
Samsung Galaxy A40	80	280	600
Samsung Galaxy S10	180	660	1020
Google Pixel 4	300	850	3000
Android Emulator	420	1670	3800

Table 1: The executions per seconds reported by AFL when running our driver on our test devices. The driver exits after forking and is configured with and without the page-table size optimization (PT Opt). The rightmost column shows the maximum possible executions per second, running a driver that does not load the ART.

Runtime Optimizations Our harness is hindered by the overhead from the `fork` syscall, heavily utilized by AFL++'s forkserver. On Android, the large number of mapped libraries for each app process (243 by the Android Runtime, plus app-specific ones) results in a large page table (592kB), making forking expensive [61]. Figure 4 shows our measurements. The fork time scales up with the page-table size. However, it turns out that most of those libraries are typically not required during the execution of a native library as they are only used for initialization routines by the DVM.

To reduce `fork` overhead, POIROT replaces the memory mappings performed by the dynamic loader with a file-based `mmap` of the same library at the same address. This approach provides two key advantages. Firstly, a file-based `mmap` is *lazy*, resulting in pages being instantiated only after being accessed. Second, the standard dynamic loader uses at least three page-table entries per library (e.g., one for the read-only segments, one for the executable ones, and one for the writable ones). Using a file-based `mmap` with permissions `rwx`, it is possible to use a single page-table entry per library, thus reducing the overall cost associated with memory mappings. Further efficiency is achieved by concatenating multiple consecutive libraries for a unified `mmap`.

Table 1 shows the executions per second, with and without our page-table size optimization. On average, page-table optimization results in roughly a three-fold speedup in executions per second during fuzzing.

5 Evaluation

The evaluation answers the following high-level research questions about POIROT:

- **RQ1**: Can POIROT scale to large real-world Android apps? We compare POIROT with other state-of-the-art tools on the most popular 100 APKs in our dataset.
- **RQ2**: Do our analysis passes improve the fuzzing process? We perform an ablation study on POIROT on the top 1,000 APKs to determine the usefulness of its analysis passes.
- **RQ3**: How many native vulnerabilities can POIROT find in large, real-world apps? We perform a fuzzing campaign on our full dataset (3,967 APKs) and triage the results.

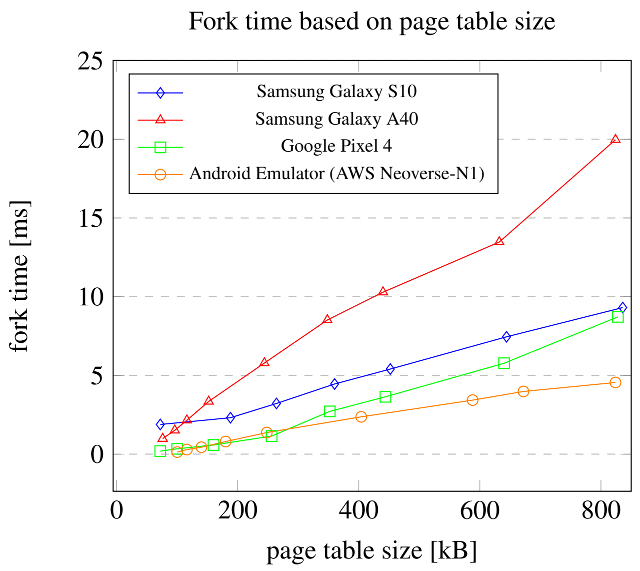


Figure 4: The time to fork, based on the page table size. The time to fork was measured in the parent by exiting directly in the child after the fork. The page table size (vmPTE) was read from /proc/self/status. The size of the page table is increased by mapping memory and touching every byte in the mapping.

Academic best practices for the evaluation of fuzzers [46] strongly recommend statistical tests to determine the significance of the difference between two models. In our case, the only related approach to fuzz Android native libraries is ATLAS (discussed in Section 5.8). However, no source code is available. As no comparable alternative exists, we therefore omit the statistical test. To enable future research to reproduce our results, we release all source code and our dataset openly.

5.1 Dataset

To create a representative dataset of popular Android apps, we scraped the "top app lists" from the Google Play store, downloading the 100 most popular apps per category in 16 countries. After eliminating duplicate apps, we downloaded a total of 20,970 apps. Of these, we excluded those without native libraries, reducing the total number of APKs in our dataset to 3,967. The least popular app is `com.njiuko.dbb` (with 108 thousand downloads), while the most popular is `com.android.chrome` (with 13.5 billion downloads). To avoid rate limiting, we downloaded the latest versions of these apps from AndroZoo [6]. In our dataset, each app includes on average 12 native libraries, with some apps reaching up to 733 (e.g., `com.facebook.creatorstudio`). The average size of a native library in our dataset is 2.9 MB, with applications typically utilizing 18 exposed JNI functions.

5.2 Setup

For running the static analysis passes, we employ a 16-core AMD EPYC 7302P x86_64 machine with 64 GB of RAM. For the fuzzing campaigns, we use a 64-core Neoverse-N1 AWS Aarch64 machine (c6g.metal instance) with 128 GB of RAM where we distribute the fuzzing jobs across 30 virtualized Android emulators. We configured POIROT to use a path limit of 1,000. In our evaluation, less than 1% of methods hit this path limit threshold.

5.3 RQ1: Scalability of the Analysis Passes to Large APKs

We compare the runtime of both our call-sequence pass and argument-analysis pass against state-of-the-art static analysis tools for Dalvik-native reachability on Android: FlowDroid [8], DroidReach [15] and JuCify [44]. For this experiment, we consider the top 100 most popular APKs in our dataset. We used both a 30-minute timeout and a 24-hour timeout with a 32-GB limit on the available memory for each tool. Table 2 shows the results. While we can run JuCify on the provided benchmark APKs from the authors' repository, we unfortunately do not get any output from any of the apps in our dataset. With FlowDroid and DroidReach, we were able to analyze, respectively, 41 and 21 of the 100 apps we considered. DroidReach does not profit from an increased timeout. From the 26 apps that previously resulted in a timeout four run out of memory and 22 return empty results. FlowDroid analyzed 1.5x more apps with a 48x longer timeout, indicating diminishing returns from further timeout increases. Assuming 49% of apps take at least 24 hours, FlowDroid would require 127.8 core years to analyze the apps in our dataset. While POIROT's call-sequence analysis pass struggles with some apps (analysis completed for 81 of the apps), the argument-analysis pass scales successfully to all apps (analysis completed for all apps). POIROT cannot complete the analysis for 19 apps. Eight of them use too much memory and five exceed the time limit, indicating that POIROT struggled to scale to the complexity of those apps. Two test cases crash while Soot tries to build the call graph of the app. For the remaining four cases POIROT returned no call sequences. This could be due to an imprecision in our static analysis, potentially one of those outlined in Section 7. In summary, we reach the same conclusion of another recent evaluation by Zhang et al. [58] and by J. Samhi et al. [45]: modern inter-procedural data-flow based tools are not yet up to the challenge of analyzing complex apps such as `com.facebook.orca`.

	#Completed	#Crashed	#Out of Memory	#Timeout	#Empty Result
	30-Minute Timeout				
POIROT Argument-analysis	100	0	0	0	0
POIROT Call-sequence	72	1	4	10	13
FlowDroid	26	2	0	64	8
DroidReach	21	1	0	26	52
JuCify	0	0	0	0	100
	24-Hour Timeout				
POIROT Argument-analysis	100	0	0	0	0
POIROT Call-sequence	81	2	8	5	4
FlowDroid	41	2	0	49	8
DroidReach	21	12	4	0	63
JuCify	0	0	0	0	100

Table 2: Comparison of scalability of static analysis tools on the top 100 most popular APKs, with a 30-minute and a 24-hour time limit, and a 32-GB memory limit. Note: the "# Completed" column does not necessarily indicate a correct result, but simply that the tool returned a non-empty result.

5.4 RQ2: Evaluation of the analysis passes

For this experiment, we consider the top 1,000 most popular APKs (roughly the top 25% of our dataset).

Out of all the 841,190 native functions, we excluded all functions that are not called by the app. leaving us with 18,907 remaining functions (this is not unexpected, as many apps only use a tiny subset of the exposed native functions). We perform four fuzzing runs across our dataset with the following four configurations:

- A "*naive*" fuzzing run without performing a static analysis in advance, to use as baseline.
- A "*call-sequence*" fuzzing run with only the call-sequence analysis pass enabled.
- An "*argument-analysis*" fuzzing run with only the argument analysis pass enabled.
- A "*complete*" fuzzing run with both passes enabled.

Following existing harness generation works [12], we allocate 4 minutes of fuzzing time for each fuzz driver. In total, this amounts to about a month of compute time per run. To measure the contribution of the analysis passes in terms of coverage increase during fuzzing, we collect the total coverage of a library harness by merging the individual coverage maps of each fuzz driver targeting that library. To avoid giving an unfair advantage to the "call-sequence" run, we instrument the library in such a way that coverage starts getting collected only on the last method of the call sequence. Furthermore, to perform a meaningful comparison, we exclude functions that are non-reachable.

We count the number of libraries that showed coverage increase after fuzzing each relevant fuzz driver for 4 minutes. If, after 4 minutes, the harness did not collect any additional coverage, we discard it. Otherwise, we mark the harness as *effective*. The "naive" run generated effective fuzzing harnesses for 304 libraries. The "call-sequence" run generated instead effective fuzzing harnesses for 612 libraries (101% more than the naive run). Instead, the "argument-analysis" run generated effective fuzzing harnesses for 331 libraries (9% more than the naive run). When both analyses are enabled, POIROT generated effective fuzzing harnesses for 617 libraries (103% more than the naive run). In this experiment, the "naive" configuration found 813 unique crashes. The "call-sequence" one found an additional 441 unique crashes. The "argument analysis" found 19 additional unique crashes. Finally, the "complete" one gathered 40 more crashes. We will now compare the total collected coverage between the different configurations.

Benefits of the analyses over the "naive" configuration Figure 5 shows the difference in total coverage (per library) of different configurations against the "naive" run. To emphasize the distinction, Figure 5 includes only data points where the difference is non-zero.

We observe that enabling an analysis pass will, on average, increase the coverage collected for a given library. In particular, the call-sequence run displays a 148% average coverage increase compared to "naive"; the argument analysis run shows a 16% average coverage increase, and finally the complete run displays a 156% average coverage increase. We also observe that the argument-analysis pass has a lower impact than the call-sequence pass and shows a coverage difference on fewer libraries.

Figure 5 shows how the difference in coverage can be sometimes negative. In those cases, the "naive" run collects more coverage compared to the other runs. This indicates that a minority of the harnesses exhibit a disadvantage when incorporating the analysis passes, potentially due to the imprecision of our static analyses (Section 7). Take, for instance, the library `libnavigator.so`. Our call-sequence analysis erroneously recommends providing the output of the function `jniUpdateAction` to the function `jniLoadNewRectLocations`. However, these two functions are unrelated, leading to less coverage being collected. Furthermore, if the harness crashes while calling the methods of the call sequence before the target method, the collected coverage will be zero. This outcome is attributed to the experiments's design, wherein coverage data gathered prior to the invocation of the target native method is disregarded. This mechanism explains cases where the difference in coverage is a large negative number.

Study of the individual contributions of each pass In contrast to before, we now compare the coverage difference of having both passes enabled (the "complete" configuration) against having a single pass enabled. Figure 6 shows the result of this experiment. Overall, the coverage collected by the "complete" run is 3% more than the "call-sequence" and 120% more than "argument-analysis".

When comparing the "complete“ configuration against the "argument-analysis" one, we notice that there are libraries for which the coverage improvement of the "complete" configuration is greater than 60,000. It should be noted that the maximum feasible value is 65,536, which corresponds to the size of the AFL coverage map. This indicates that those libraries are complex, requiring multiple API calls to access their full functionality. Without the insights from the call-sequence analysis, it becomes challenging to explore them effectively. When comparing instead the "complete" configuration against the "call-sequence" one, the difference is more subtle. Although the average increase in coverage is marginal (3%), the discovery of an additional 40 unique crashes in the "complete" configuration implies the existence of libraries that cannot be effectively fuzzed without the necessary argument constraints. An example of a library that demonstrates the necessity of both analysis passes is `libimostream.so` used by app `com.imo.android.imoimhd`. Some functions of this library (e.g., `getOggFileSampleRate`) depend on the function `startReadOggFile` being called before (thus requiring a call sequence). Moreover, the function `startReadOggFile` takes a string as an argument used by the library as a file path (thus requiring an argument constraint). Without both passes, the fuzzer is unable to effectively explore this library.

5.5 RQ3: Finding Vulnerabilities in Real-World Apps

To address RQ3, we conduct a long-running campaign (60 days of total compute time) considering the complete dataset (the top 3,967 most popular APKs with native libraries). In this campaign, a total of 39,992 fuzz drivers were generated, of which 19,812 were marked as effective (Section 5.4) and then fuzzed. Out of those 19,812 harnesses, 6,120 (30.8%) had associated call sequences.

In general, 69,347 crashes were found. After deduplication, we are left with 4,282 crashes affecting 3,368 native functions. We assigned three expert analysts to triage crashes for a period of one month each, using the following triaging process:

1. *Reproduce the Crash*: Analysts first attempted to reproduce each crash on a physical test device. Our provided scripts facilitate quick replication of the fuzzing environment, the specific crashing input, and the harness for interacting with the target library, thereby confirming reproducibility. This step often required decompiling the native library using Ghidra [5] to analyze and understand the context of the crash.

2. *Debug and Categorize*: A separate script established a debugging environment attached to the problematic library. Analysts then examined backtraces, faulting instructions, and other runtime information to categorize the crash (e.g., stack overflow, heap overflow, null pointer dereference, or double free).

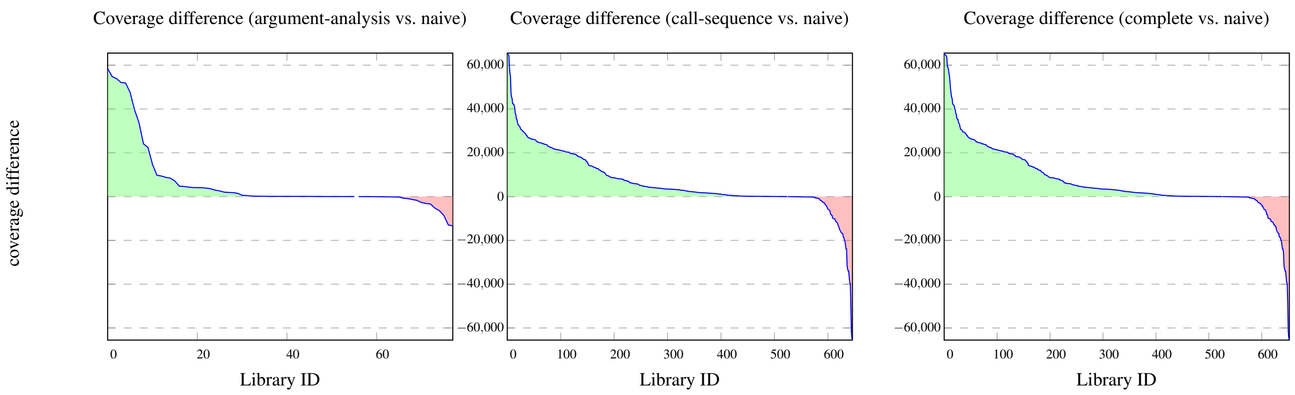


Figure 5: Coverage difference, per library, between the "argument-analysis", "call-sequence", and "complete" runs against the "naive" one. A positive difference (in green) represents more edges explored compared to the "naive" approach. A negative value (in red) represents libraries where the "naive" approach collected more coverage instead. Points where the difference is 0 are omitted. Note: plot 2 and 3 are very similar, but not equal.

3. *Identify Native Callsite*: Using JADX [3], analysts decompiled the application to pinpoint the callsite of the native function causing the crash.

4. *Determine Reachability*: Analysts reverse-engineered relevant Java code, analyzing application logic and tracing cross-references, to ascertain if attacker-controlled data could reach the vulnerable native function.

5. *Assess Exploitability*: Finally, if all the previous steps were successful, analysts further reverse-engineered the app to determine if the crash was genuinely triggerable. This involved understanding the conditions an attacker would need to satisfy, such as specific application states (e.g., required library initialization) and data constraints (e.g., printable characters only).

Section A.1 shows a practical example of the triaging process. The triaging process was significantly complicated by the closed-source nature of the libraries and applications under analysis, requiring substantial reverse engineering efforts for both native code and Dalvik bytecode. With each crash demanding 2 to 6 hours of expert analysis, it is infeasible to triage the entirety of the 4,282 unique crashes. Consequently, we adopted a prioritization strategy, instructing analysts to prioritize crashes that generated the highest edge coverage. This heuristic prioritizes crashes that drive interaction in the target library.

In total, our analysts were able to triage 200 crashes. 52 of those were discarded as they were relevant to functions that did not accept attacker-controlled input. Of the remaining 148 crashes, 25 crashes turned out to be true positives (unique vulnerable functions). The remaining 123 crashes were determined to be false positive crashes.

- 80 false positive crashes were due to our argument analysis pass missing constraints. For example, fuzzing an argument to a function that the app always calls with a constant value. The main reason behind those is twofold: first, the constant value might originate from a different function (the argument analysis pass is strictly intra-procedural). Secondly, the value might not be in our set of supported argument types listed in Section 3.2.

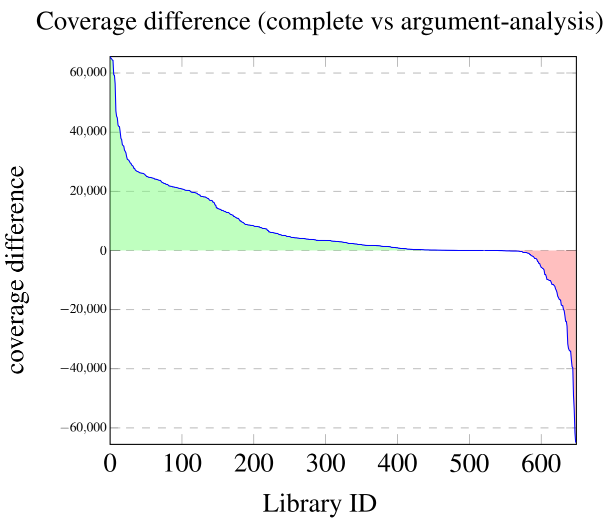
- 43 crashes were due to calling a function with missing state, denoting a false negative result of the call sequence analysis pass. In every case we analyzed, this is due to a missing initialization function that POIROT failed to identify. This failure occurred for one of these two possible reasons: the function was too complex to analyze (hitting the path limit threshold) or there were imprecisions in the ICFG (i.e., reflections, callbacks).

The 25 true positive crashes were relative to 25 different native functions affecting 16 unique native libraries. We manually analyzed all the applications that used those 25 vulnerable functions and identified a total of 34 vulnerable apps. We have responsibly disclosed all discovered vulnerabilities to the respective vendors.

It is important to acknowledge a potential bias in our results stemming from the methodology used to select which crashes to triage. Specifically, we focused on the 200 crashes that exhibited the highest coverage. Despite this selection process, we did not notice any significant relationship between the level of edge coverage collected by the harness and the probability of a crash being a true positive. This observation suggests that the subset of crashes we examined can be considered a representative sample of the whole collection.

In total, 25 out of the 200 crashes (12.5%) are true positives. We note that the 12.5% ratio of true positives may initially appear modest.

However, this performance is in line with other state-of-the-art library fuzzing efforts. For instance, UTopia [30] reported



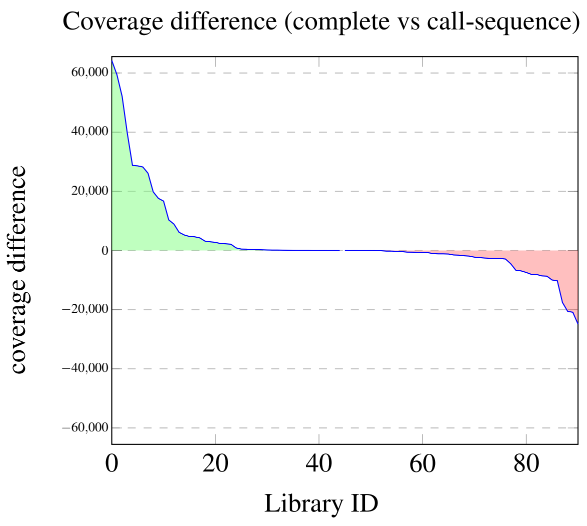


Figure 6: Coverage difference, per library, between the complete run and the call-sequence and argument-analysis runs. Libraries where the difference is 0 are omitted.

a 9% true positive rate. We suspect that while UTopia has the advantage of having source code and unit tests to aid its harness synthesis process, the relatively lower bug detection rate could be due to their focus on well-known, open-source libraries. These libraries are more widely used and better tested than the proprietary and potentially outdated ones included in Android apps.

In our triage experiment we included all target functions to present comprehensive, reproducible results. Prefiltering the target set (e.g., ATLAS [54] only considers media parsing libraries) could however significantly increase the true positive rate of crashes. In Appendix Section B, we show how two simple prefiltering strategies would have doubled the ratio of true positives in our triage experiment.

5.6 Case Study tpCamera (CVE-2023-30273)

To demonstrate the real-world bug-finding capabilities of our approach, we present a case study on a use-after-free vulnerability in the MP4Encoder native library used by the tpCamera app. This vulnerability was discovered during our large scale fuzzing campaign described in Section 5.5. The fuzz driver that discovered this vulnerability uses a specific call sequence to trigger use-after-free. Additionally, it also uses array-length constraints for two arguments.

The tpCamera app is used to manage the tp-link cameras and live stream the video feed. The vulnerability allows an attacker in the same network as the phone or who has compromised a tp-link camera to execute code on the victim's phone once the user presses the record button in the app. The MP4Encoder library is used by the app to merge h264 or jpeg files into an MP4 file and is used for the recording feature of the app.

The call sequence that discovered this vulnerability is the following:

1. **iniPacker:** Sets up the necessary state (a global on the heap) and opens an mp4 file for writing.

2. **packVideo:** Called with an invalid jpeg as input. As a result, the mp4 file is closed and the global is freed.

3. **packVideo:** Called with an invalid jpeg as input. The jpeg must be the same size as the `iniPacker_global` struct to trigger the use-after-free. The input is malloced into the freed global and the file pointer passed to `fclose` is the first qword of the input.

See Listing 1 for the details of the vulnerability in `packVideo`. The vulnerability gives the attacker control over the pointer passed to `fclose`. An attacker is then able to call any function with controlled arguments. We developed a proof-of-concept exploit to confirm that, given a heap leak and a libc leak, an attacker gains arbitrary code execution. We also verified that the vulnerability can be triggered in the app by a MITM attacker. In fact, after pressing the recording button (which calls `iniPacker`), the camera continuously polls the camera over untrusted HTTP to download the video feed, and repeatedly calls the `packVideo` function without checking the return value. As a result, we were able to trigger the use-after-free remotely.

The vulnerability was reported to the developers on February 10th, 2023. CVE-2023-30273 was assigned to this vulnerability on April 27th, and a fix was shipped on July 19th 2023.

```
jint packVideo(JNIEnv *env, ..., jbyteArray input,
               jint input_len, jlong timestamp) {
    char* input_bytes = env->GetByteArrayElements(input);
    if (iniPacker_global == 0){ return -1; }
    FILE* stream = *iniPacker_global //corrupted with UAF
    ... // check input bytes, involves input_len
    if(video_corrupted){// if check on input bytes fails
      fclose(stream); //vulnerable fclose
      free(iniPacker_global) //not set to 0 after free
      return -1;
    }
    ...
}
```

Listing 1: The parts of the code in the `packVideo` function relevant to the vulnerability. The `iniPacker_global` variable is allocated in the `iniPacker` function. The `GetByteArrayElements` function allocates the extracted bytes on the heap. If the length of the input byte array is in the same chunk range as the size of the `iniPacker_global` struct, the input overwrites the freed global struct (in the second call to `packVideo`).

5.7 Cost of adding support to more types

As denoted in Section 3.3, POIROT supports only a limited set of Java types as fuzz parameters. However, adding support to a new Java type requires just engineering effort. To quantify this effort, we task an independent Android app developer to extend POIROT with support for the `int[]` type. The developer had some fuzzing experience writing harnesses but was not familiar with our codebase. In total, the task took about 15 hours, including the implementation and testing. The changes in the codebase amount to 81 additional lines of code.

This shows that adding new types to POIROT is straightforward, comes at a reasonable cost, and does not require any design changes.

5.8 Comparison against ATLAS

ATLAS [54] (which was developed concurrently) automates fuzzing harness generation for Android native libraries using a variety of cross-language analysis steps. Specifically, it uses symbolic execution to model the API interaction between the Java side and the native side to better model the fuzz driver. ATLAS also uses static analysis and taint analysis to derive call sequences.

Multiple aspects of the ATLAS design introduce limitations. First, symbolic execution and taint analysis are very expensive and notoriously hard to scale to even medium-sized binaries [19]. Secondly, ATLAS requires porting DVM code to the JVM to be able to fuzz the targets outside Android. One disadvantage of this is the exclusion of targets that interact with Android-specific mechanisms (e.g., Binder). The most limiting factor however is that this process requires a human in the loop to address dependency and harness generation issues, which somewhat contradicts the claim for automation.

App name	Library	BB (POIROT)	BB (ATLAS)
SamsungGallery2018	libagifencoder.quram.so	837	2303
Gallery_T_CN	libMiuiGalleryNSGIF.so	529	387
OplusEngineerCamera	lib_rectify.so	1228	794

Table 3: Basic blocks explored over 24 hours of fuzzing.

In fact, the authors explain that the manual effort required to manually fix each harness is so expensive they limited their dataset to 17 apps. Furthermore, the dataset is not generic, as all libraries targeted are related to media codec parsing.

Due to the insufficient dataset and the fact that ATLAS is not open-source, we cannot directly compare it against POIROT. Nonetheless, we perform a small indirect coverage comparison on a subset of their apps. ATLAS reports the number of basic blocks covered after 24 hours of fuzzing for 8 of the 17 apps in their dataset. All of those apps are proprietary, not publicly available but rather shipped by vendors as part of their firmware. We searched available datasets with vendor firmware dumps and recovered 3 of the 8 apps. We compare the basic block obtained by POIROT after fuzzing for 24 hours. Results are shown in Table 3. We observe how POIROT outperforms ATLAS except for the first app ("SamsungGallery2018"). This app exhibits the pattern ATLAS was designed for (JNI interaction based on a native pointer "mHandle" stored in a class field).

In conclusion, we are unable to consider ATLAS a viable alternative to POIROT due to its reliance on expensive analysis passes, requirement of a human in the loop, a fuzzing environment outside of Android, low basic-block coverage, and most importantly the lack of source code.

We open-source POIROT and all its artifacts.

6 Related Work

6.1 Automatic Harness Synthesis For Libraries

Recent research indicates a growing interest in automatic harness synthesis. However, existing research focuses on an arbitrary library consumer. The main goal is the discovery of bugs *in the library*, rather than in the application using the library. This allows approaches like the one presented by Hopper [21], which iteratively refines call sequences by using coverage as a feedback mechanism. The authors identify correct API sequences that build state by detecting increases in edge coverage. In our case instead, we narrow down the API usage to strictly mimic what the unique consumer app is doing to identify vulnerabilities in its specific usage of the API.

Fudge [12] generates fuzz drivers for open-source libraries by extracting snippets from a single consumer, but requires intervention by a human analyst. FuzzGen [29] avoids this by creating an *Abstract API Dependence Graph* (A^2DG), that improves the automatic analysis. UTopia [30] analyzes human-authored unit tests for harness synthesis, while GraphFuzz [27] introduces mutation strategies through API graphs of functions with data dependencies.

APICraft [57], DAISY [59], and Winnie [31] use header files and execution traces to refine harnesses for the Apple OSX SDK and Windows apps, respectively. This approach, however, relies on a human analyst operating the app to generate the execution traces.

6.2 Android/JNI vulnerability detection

Many Android app analysis systems overlook native libraries [4]. Ndroid [55] explores Java-native interfaces for information leaks, using QEMU. However, it struggles with complex user interactions (e.g., login screens). JN-SAF [53] is a data-flow analysis tool for Java and native code to detect information leaks. As other works already failed running JN-SAF [7,44], we tried running it and ultimately omitted it from our evaluation after a quick check.

JuCify [44] approximates native code to a Jimple [50] intermediate representation augmenting existing Soot data flow analyzers. DroidReach [15] studies Android native function reachability uses a combination of Angr [51] and Ghidra [5].

There are only three prior works on JNI fuzzing. The first by Zhao et al. [60] is limited to Windows XP JNI programs and lacks a conclusive evaluation. JNIFuzzer [43] is a proof of concept implementation for fuzzing Android libraries that does not consider implications of the Java side. We discuss JNIFuzzer in Appendix Section A.2. ATLAS [54] is the most relevant work in Android native library fuzzing and we discuss it in detail in Section 5.8.

6.3 Android Fuzzing

Some previous works [10,34,38,56] study how to fuzz Intent objects to be sent looking for crashes in a target app. Fuzzing has also been applied to target Trusted Execution Environments [18], Android's native system services [33] and Binder interactions [33]. Moreover, fuzzing has proved useful for automatically interacting with apps and triggering execution of certain APIs [9] or hidden malicious behaviour [41,52]. However, vulnerabilities in Android native libraries remain unexplored, lacking a proposed methodology to address them.

Several related works have the potential to improve the fuzzing process for Android native libraries. Avatar^2 [40] is a dynamic analysis orchestrator framework that collects and unifies information from many different systems. KARONTE [42] explores the security impacts of data flowing through different binaries in embedded firmware. QBDI [26] is a dynamic instrumentation framework that supports coverage information for Android native libraries, and reports a lower overhead than Frida.

7 Discussion

7.1 Generalizability to other Systems

The challenges we tackle in this work are not unique to Android. In fact, generic Java programs that bundle native libraries are affected by the same challenges. Other languages also allow for interoperability with native code. For example, popular languages such as Rust, Python, and Ruby all have support for the Foreign Function Interface (FFI) that enables interoperability with C code. Ruby and Rust, like Java's JNI, have a specific location where C methods can be identified; for Ruby, a C method is registered with `attach_function`. For Rust, a C method signature is declared inside `extern "C"`. Although Python's foreign function library `ctypes` does not have a specific location where C methods are identified, native libraries are registered with the `CDLL` function and C methods are called on the return value of `CDLL`. Finally, several studies already exposed the challenges of cross-language fuzzing. Namely, POLYFuzz [36] is a fuzzer for multi-language systems composed of C, Python, and Java, although the prototype focuses on the coverage collection and fuzzing technique, and does not automatically synthesize harnesses.

7.2 Limitations

Imprecision of static analysis We prioritize scalability in our static analysis, albeit at the cost of some precision. We favor the simpler Class Hierarchy Analysis (CHA) for resolving polymorphic calls, as opposed to SPARK [35], which employs points-to analysis for this purpose. Our static analysis does not employ data-flow. This can result in fuzzing native functions that never accept attacker-controlled data, leading to spurious crashes. Our input-output matching only considers data types of parameters and does not use data-flow, potentially generating non-existent data dependencies. Finally, our approach is not immune to the typical challenges encountered in Java static analysis, such as Java reflections [37,48], imprecise callback analysis [16,20], and obfuscated Java code [14]. Other forms of obfuscation (e.g., relocating the entire app logic to native libraries) are not handled either.

We believe that adding inter-procedural data-flow analysis to our approach is possible, but would require a set of optimizations tailored to our use-case, which we thus leave for future work. An additional way to improve would be performing flow analysis on the native libraries themselves, which could expose more semantic constraints between parameters passed to them.

DVM Threads Due to inherent limitations of the `fork` syscall, only one thread will be present in the child process. Hence, if the app spawns a thread in the `JNI_OnLoad` initialization, incorrect behavior may be observed during fuzzing.

DVM State A native function may expect a certain state to be built from the Java code. For example, an app may initialize a static variable of a given class inside its `onCreate` method. A native library may then use `env->FindClass` and `env->GetFieldID` to retrieve the value of the static variable and check for its state. POIROT does not currently address this.

Android Framework Interactions Our bare-bones DVM setup described in Section 3.3 does not include Android components such as GUI (e.g., Drawables), hardware peripherals (e.g., camera, GPS), or common system services (e.g., Gallery) that a native library might try to interact with, leading to potential limitations in testing the full range of attack vectors.

Arbitrary Java Arguments Certain native functions have arguments of type `Object` (i.e., an unspecified type). Further analysis in the Java code or the native context would be required to provide additional insight about the `Object` and enable fuzzing of relevant methods.

8 Conclusions

Traditionally, harness synthesis for Android apps has required a human analyst to reverse engineer both the application and the often closed-source native libraries. The analyst then had to carefully encode the API call sequences and constraints in manually written fuzz drivers. Existing automatic harness synthesis approaches are focused on open-source libraries or require a human in the loop. POIROT introduces an efficient automatic harness synthesizer targeting proprietary Android native libraries, mimicking the behavior of a specific consumer app. We solve key challenges in closed-source library fuzzing like JNI interaction, API call sequences, and argument constraints, by exploiting the type-rich Java information.

POIROT discovers bugs in large, real-world COTS APKs. We analyzed a total of 3,967 APKs and found 4,282 crashes, identifying 34 vulnerable apps. Our study demonstrates the effectiveness of incorporating static analysis passes in the fuzzing process, as it leads to an increase in coverage and the discovery of otherwise undetected bugs. All vulnerabilities have been responsibly disclosed to the relevant vendors.

9 Ethics Considerations

Our Android native vulnerability finding research carefully considers the ethical implications of vulnerability discovery and disclosure. We follow a responsible disclosure process by immediately reporting all discovered vulnerabilities to the vendors (which assigned CVEs when relevant). This approach ensures that vulnerabilities are addressed and patched before public disclosure, minimizing potential harm to Android users worldwide. The only vulnerability we specify the details of in Section 5.6 has been reported and patched by the vendor. We manually verified that the patch is effective. The research methodology was designed with multiple stakeholder perspectives in mind, weighing both the benefits and potential risks. While our vulnerability finding techniques could theoretically be used by malicious actors, we determined that the defensive benefits significantly outweigh the risks, as our work enables Android platform moderators to proactively identify and fix security issues before they can be exploited. Additionally, all our testing was conducted in isolated lab environments to prevent any accidental impact on production systems or end users.

10 Open science

We release POIROT as open source. The code, dataset and artifacts can be found at https://doi.org/10.5281/zenodo.15586318.

Acknowledgements

We thank the anonymous reviewers for their detailed feedback. This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868) and SNSF PCEGP2 18697 and the project "SEcurity and RIghts In the CyberSpace - SERICS" (PE00000014 - CUP H73C2200089001) under the National Recovery and Resilience Plan (NRRP) funded by the European Union - NextGenerationEU.

References

[1] Fresco webp. https://github.com/facebook/fresco/blob/b7778fce28158b91c5eba304056e0d96f33a828f/static-webp/src/main/jni/static-webp/webp.cpp. Accessed: 03-02-23.

[2] Frida. https://github.com/frida/frida. Accessed: 04-02-23.

[3] jadx - dex to java decompiler. https://github.com/skylot/jadx. Accessed: 13-06-2023.

[4] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick McDaniel, and Matthew Smith. Sok: Lessons learned from android security research for appified software platforms. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 433–451. IEEE, 2016.

[5] National Security Agency. Ghidra software reverse engineering suite. https://ghidra-sre.org/. Accessed: 30-01-23.

[6] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.

[7] Sumaya Almanee, Arda Ünal, Mathias Payer, and Joshua Garcia. Too quiet in the library: An empirical study of security updates in android apps' native code. In *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion*, 2021.

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

[9] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 217–228, New York, NY, USA, 2012. Association for Computing Machinery.

[10] Michael Auer, Andreas Stahlbauer, and Gordon Fraser. Android fuzzing: Balancing user-inputs and intents. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 37–48, 2023.

[11] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 426–436, 2015.

[12] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. FUDGE: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 975–985. ACM.

[13] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. On demystifying the android application framework:{Re-Visiting} android permission specification analysis. In *25th USENIX security symposium (USENIX security 16)*, pages 1101–1118, 2016.

[14] Michael Batchelder and Laurie Hendren. Obfuscating java: The most pain for the least gain. In *International Conference on Compiler Construction*, pages 96–110. Springer, 2007.

[15] Luca Borzacchiello, Emilio Coppa, Davide Maiorca, Andrea Columbu, Camil Demetrescu, and Giorgio Giacinto. Reach me if you can: On native vulnerability reachability in android apps. In *Computer Security–ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part III*, pages 701–722. Springer, 2022.

[16] Priyanka Bose, Dipanjan Das, Saastha Vasan, Sebastiano Mariani, Ilya Grishchenko, Andrea Continella, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Columbus: Android app testing through systematic callback exploration. *arXiv preprint arXiv:2302.09116*, 2023.

[17] Bobby R Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. Jshrink: In-depth investigation into debloating modern java applications. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 135–146, 2020.

[18] Marcel Busch, Aravind Machiry, Chad Spensky, Giovanni Vigna, Christopher Kruegel, and Mathias Payer. Teezz: Fuzzing trusted applications on cots android devices. *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1204–1219, 2023.

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

[20] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

[21] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. Hopper: Interpretative fuzzing for libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1600–1614, 2023.

[22] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995 9*, pages 77–101. Springer, 1995.

[23] Caleb Fenton. Creating a java vm from android native code. https://calebfenton.github.io/2017/04/05/creating_java_vm_from_android_native_code/. Accessed: 24-01-23.

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

[25] Joshua Garcia and Sam Malek. Path-sensitive analysis of message-controlled communication for android apps. (UCI-ISR-16-4), September 2016.

[26] Google. Quarksalab qbdi. https://qbdi.quarkslab.com/. Accessed: 27-07-23.

[27] Harrison Green and Thanassis Avgerinos. GraphFuzz: library API fuzzing with lifetime-aware dataflow graphs. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1070–1081. ACM.

[28] Eric Le Guevel. Android greybox fuzzing with afl++ frida mode. https://blog.quarkslab.com/android-greybox-fuzzing-with-afl-frida-mode.html. Accessed: 20-07-2023.

[29] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. {FuzzGen}: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2271–2287, 2020.

[30] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. Utopia: Automatic generation of fuzz driver using unit tests. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2676–2692. IEEE, 2023.

[31] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. WINNIE : Fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings 2021 Network and Distributed System Security Symposium*. Internet Society.

[32] Mateusz Jurczyk. Google project zero: Mms exploit part 1, introduction to the samsung qmage codec and remote attack surface. https://googleprojectzero.blogspot.com/2020/07/mms-exploit-part-1-introduction-to-qmage.html. Accessed: 30-01-23.

[33] Wang Kai, Zhang Yuqing, Liu Qixu, and Fan Dan. A fuzzing test for dynamic vulnerability detection on android binder mechanism. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 709–710, 2015.

[34] Anatoli Kalysch, Mark Deutel, and Tilo Müller. Template-based android inter process communication fuzzing. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ARES '20, New York, NY, USA, 2020. Association for Computing Machinery.

[35] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using s park. In *Compiler Construction: 12th International Conference, CC 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings 12*, pages 153–169. Springer, 2003.

[36] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. Polyfuzz: Holistic greybox fuzzing of multi-language systems. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[37] Yue Li, Tian Tan, and Jingling Xue. Understanding and analyzing java reflection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(2):1–50, 2019.

[38] Amiya K. Maji, Fahad A. Arshad, Saurabh Bagchi, and Jan S. Rellermeyer. An empirical study of the robustness of inter-component communication in android. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, 2012.

[39] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-language attacks. In *NDSS*, 2022.

[40] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar 2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, volume 18, pages 1–11, 2018.

[41] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. Making malory behave maliciously: Targeted fuzzing of android execution environments. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, page 300–311. IEEE Press, 2017.

[42] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1544–1561. IEEE, 2020.

[43] Claudio Rizzo. *Static Flow Analysis for Hybrid and Native Android Applications*. PhD thesis, Ph. D. Dissertation. Royal Holloway–University of London, 2020.

[44] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. Jucify: a step towards

android code unification for enhanced static analysis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1232–1244, 2022.

[45] Jordan Samhi, René Just, Tegawendé F. Bissyandé, Michael D. Ernst, and Jacques Klein. Call graph soundness in android static analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 945–957, New York, NY, USA, 2024. Association for Computing Machinery.

[46] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. Sok: Prudent evaluation practices for fuzzing. *arXiv preprint arXiv:2405.10220*, 2024.

[47] Prashast Srivastava, Flavio Toffalini, Kostyantyn Vorobyov, François Gauthier, Antonio Bianchi, and Mathias Payer. Crystallizer: A hybrid path analysis framework to aid in uncovering deserialization vulnerabilities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1586–1597, 2023.

[48] Xiaoyu Sun, Li Li, Tegawendé F Bissyandé, Jacques Klein, Damien Octeau, and John Grundy. Taming reflection: An essential step toward whole-program analysis of android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–36, 2021.

[49] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[50] Raja Vallee-Rai and Laurie J Hendren. Jimple: Simplifying java bytecode for analyses and transformations. 1998.

[51] Fish Wang and Yan Shoshitaishvili. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.

[52] Xiaolei Wang, Yuexiang Yang, and Sencun Zhu. Automated hybrid analysis of android malware through augmenting fuzzing with forced execution. *IEEE Transactions on Mobile Computing*, 18(12):2768–2782, 2019.

[53] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1137–1150, 2018.

[54] Hao Xiong, Qinming Dai, Rui Chang, Mingran Qiu, Renxiang Wang, Wenbo Shen, and Yajin Zhou. Atlas: Automating cross-language fuzzing on android closed-source libraries. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 350–362, 2024.

[55] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin T.S. Chan. NDroid: Toward tracking information flows across multiple android contexts. 14(3):814–828.

[56] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, pages 68–74, 2013.

[57] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. Apicraft: Fuzz driver generation for closed-source sdk libraries. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2811–2828, 2021.

[58] Junbin Zhang, Yingying Wang, Lina Qiu, and Julia Rubin. Analyzing android taint analysis tools: Flowdroid, amandroid, and droidsafe. *IEEE Transactions on Software Engineering*, 48(10):4014–4040, 2021.

[59] Mingrui Zhang, Chijin Zhou, Jianzhong Liu, Mingzhe Wang, Jie Liang, Juan Zhu, and Yu Jiang. Daisy: Effective fuzz driver synthesis with object usage sequence analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 87–98. IEEE, 2023.

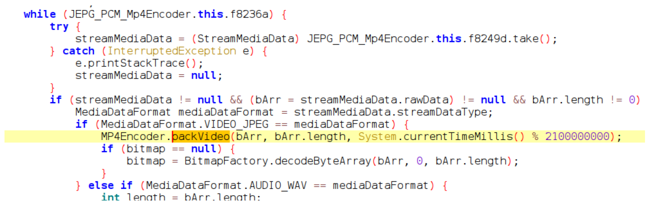
[60] Jinjing Zhao, Yan Wen, Xiang Li, Ling Pang, Xiaohui Kuang, and Dongxia Wang. A heuristic fuzz test generator for java native interface. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on Software Qualities and Their Dependencies*, pages 1–7, 2019.

[61] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. On-demand-fork: a microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 540–555, 2021.

A Appendix

A.1 Triage Example

To illustrate our triage process, we detail a real-world crash discovered in the tpCamera application (Section 5.6), specifically within its libTPMp4Encoder.so library. This example demonstrates the triage process:



```
while (JEPG_PCM_Mp4Encoder.this.f8236a) {
    try {
        streamMediaData = (StreamMediaData) JEPG_PCM_Mp4Encoder.this.f8249d.take();
    } catch (InterruptedException e) {
        e.printStackTrace();
        streamMediaData = null;
    }
    if (streamMediaData != null && (bArr = streamMediaData.rawData) != null && bArr.length != 0)
        MediaDataFormat mediaDataFormat = streamMediaData.streamDataType;
        if (MediaDataFormat.VIDEO_JPEG == mediaDataFormat) {
            MP4Encoder.packVideo(bArr, bArr.length, System.currentTimeMillis() % 2100000000);
            if (bitmap == null) {
                bitmap = BitmapFactory.decodeByteArray(bArr, 0, bArr.length);
            }
        } else if (MediaDataFormat.AUDIO_WAV == mediaDataFormat) {
            int length = bArr.length;
```

Figure 7: Screenshot of the JADX decompiler showing the vulnerable `MP4Encoder_packVideo` callsite

1. *Reproduce the Crash*: Initially, the analyst reproduced the crash by executing the provided script to send the identified crashing input to the vulnerable libTPMp4Encoder.so library: `python3 fuzzing/triage.py -target com.tplink.skylight.apk -target_function MP4Encoder_packVideo -device <device_id> -c -r` This script executed the crashing input against the target library on the test device, successfully reproducing the crash and yielding the following backtrace:

```
#0  0x712da17e4c abort @ libc.so
#1  0x712da0c914 scudo::die @ libc.so
#2  0x712da0cf8c scudo::~ScopedErrorReport @ libc.so
#3  0x712da0d1c0 scudo::reportInvalidChunkState @ libc.so
#4  0x712da0e5cc scudo::Allocator::deallocate @ libc.so
#5  0x6e802a305c mp4_write_one_jpeg @ libTPMp4Encoder.so
#6  0x6e802a6608 MP4Encoder_packVideo @ libTPMp4Encoder.so
#7  0x61b2c8ee2c fuzz_one_input at harness_debug.cpp:76
```

   The call to scudo::die immediately suggested a memory corruption issue.

2. *Debug and Categorize*: Subsequently, the analyst attached GDB to the harness to inspect the crash in detail. The backtrace (Frame #1) confirmed the involvement of the Scudo memory allocator. Root cause analysis necessitated decompiling `libTPMp4Encoder.so`. Decompilation revealed that the Scudo exception was triggered by a free operation (`scudo::Allocator::deallocate` at Frame #4) on a global variable within `mp4_write_one_jpeg` (Frame #5). Using GDB to trace this call, the analyst determined that the pointer being deallocated had already been freed. Thus, the crash was categorized as a double-free, stemming from a use-after-free vulnerability involving the identified global variable.

3. *Identify the Target Native Callsite*: Using JADX [3], the analyst decompiled the tpCamera application's Dalvik bytecode. JADX's search functionality was then employed to locate the Java callsite invoking the vulnerable native function, `MP4Encoder_packVideo`. The relevant decompiled code snippet is presented in Figure 7.

4. *Determine Reachability*: Leveraging JADX's cross-reference feature, the analyst traced the origin of the `bArr` parameter supplied to the vulnerable function (see Figure 7). This analysis revealed that `bArr` originates from a queue populated with JPEG images received from the network. Specifically, an IoT camera streams these images to the application over HTTP without authentication to provide a live feed. This confirmed that the vulnerable `MP4Encoder_packVideo` function was reachable via unauthenticated network input.

5. *Determine Vulnerability (Assess Exploitability)*: The final step involved assessing exploitability. First, the analyst verified that the harness's call sequence accurately mirrored the application's logic. The harness correctly initialized the library via `initVideo` and then invoked `packVideo` twice, mimicking the application, which also calls `initVideo` followed by `packVideo` in a loop during stream recording (Figure 7). Second, argument constraints were examined. The harness correctly encoded the array-length constraint for the second argument. The third argument, used only for logging, did not influence the crash. Crucially, the application performed no sanitization checks on the attacker-controlled `bArr` parameter itself.

Metric	JNIFuzzer	POIROT
APKs Targeted	340	[c]
APKs Succesfully Fuzzed	22	22
Functions Fuzzed	40	94[a]
Crashes Found	0	457
Deduplicated Crashes	0	50
Crashes (Prefiltered)	0	4
True Positives (Triaged)	0	1
False Positives (Triaged)	0	3
Reasons for TPs	*N/A*	1 Stack overflow[b]
Reasons for FPs	*N/A*	1 Unsat. Constraints 2 Unreachable

[a] "Fuzzable" functions where POIROT detected increase in coverage.
[b] Stack buffer overflow from controlled data in SharedPreferences; confirmed fixed in the newest app version.
[c] POIROT only considered the APKs JNIFuzzer was able to fuzz.

Table 4: Comparative Fuzzing Performance of JNIFuzzer and POIROT on the Common Subset of 340 ARM64 APKs from JNIFuzzer's Dataset.

Based on these findings, the analyst confirmed the crash as a true positive vulnerability, specifically a network-reachable memory corruption, and subsequently reported it to the vendor.

A.2 Comparison against JNIFuzzer

JNIFuzzer's analysis is restricted to function signatures, heavily limiting the analysis scope. The generated fuzz drivers only call a single native function. Furthermore, the current

JNIFuzzer implementation can fuzz only armv7a (32bit) native libraries. JNIFuzzer does not employ coverage information, limiting itself to black-box fuzzing. Finally, JNIFuzzer supports fewer Java types (i.e., only numeric primitives and partial support for strings).

For a direct comparison, we evaluated POIROT on the same 2017 dataset used by JNIFuzzer (4,171 APKs – we recovered the list of targeted APKs from the JNIFuzzer evaluation and open-source prototype). Due to its age, only 340 APKs contained ARM64 native libraries suitable for comparison. On this subset of 340 APKs JNIFuzzer successfully fuzzed 22 APKs, covering 40 functions, but found no crashes and provides no coverage information. POIROT, in contrast, identified 94 fuzzable functions (with increasing code coverage), and discovered 50 crashes. After prefiltering POIROT's 50 crashes, 4 unique ones remained. Manual triage revealed one true positive——a stack buffer overflow from controlled SharedPreferences data (confirmed fixed in the newest app version)——and three false positives (one unsatisfiable constraint, two unreachable code paths).

A.2.1 Limitations of mocking JNI

JNIFuzzer employs a mock `JNIEnv` structure to implement JNI interface methods as specified in `jni.h`. Yet, as Rizzo [43] notes, fully replicating the JNI interface is infeasible. Listing 2, from Facebook's `fresco` [1] library, shows a concrete example of the limitations of the mocking approach. This function parses the input bytes and returns a WebPImage object, created from the native context by invoking `env->NewObject`, which in turn calls the constructor of the WebPImage class. Mocking such complex Java/native interactions would require a comprehensive reimplementation of the Java class model itself. Instead, POIROT supports JNI interactions by using a genuine `JNIEnv` linked to a bare-bones DVM, outlined in Section 3.3.

To show how important the need to support such interactions is, we employ Ghidra [5] to study the prevalence of hard-to-mock methods (e.g., FindClass) in native libraries. We decompile each exported native function and collect which JNI callbacks are used. Table 5 shows the result of this analysis. While the most frequent JNI callbacks are string operations, other popular ones rely on an app's specific imported or defined classes (e.g., GetFieldID) which highlights the importance of having a real DVM handling those.

```
jobject CreateFromDirectByteBuffer(JNIEnv *env, ...) {
   ... // Create the WebPImage with the native context.
   jobject ret = env->NewObject(
      sClazzWebPImage,
      sWebPImageConstructor,
      (jlong) spNativeContext.get());
   return ret;
}
```

Listing 2: Native method that creates a `WebImpage` object.

`JNIEnv*` callback	Instances (Freq.)
`env->NewByteArray`	1,642 (1.4%)
`env->SetByteArrayRegion`	1,758 (1.5%)
`env->NewGlobalRef`	1,995 (1.8%)
`env->ReleaseByteArrayElements`	3,816 (3.4%)
`env->GetByteArrayElements`	3,820 (3.4%)
`env->GetArrayLength`	3,958 (3.5%)
`env->GetLongField`	4,151 (3.7%)
`env->ExceptionClear`	4,271 (3.8%)
`env->ThrowNew`	4,351 (3.9%)
`env->GetObjectClass`	4,501 (4.0%)
`env->GetMethodID`	4,588 (4.1%)
`env->DeleteLocalRef`	5,654 (5.0%)
`env->NewStringUTF`	5,991 (5.3%)
`env->GetFieldID`	6,612 (5.8%)
`env->FindClass`	6,964 (6.1%)
`env->ReleaseStringUTFChars`	9,197 (8.1%)
`env->GetStringUTFChars`	10,081 (8.9%)
Total	112,331

Table 5: Most common JNI callbacks found in native methods to switch execution to the Java side (excluding callbacks with less than 1% frequency)

B Prefiltering strategies

We explore the possibility of using prefiltering strategies to reduce the number of spurious crashes, in line with prior work [54]. We analyze two different approaches: a signature-based method, and a heuristical functionality filtering based on library symbol names.

In the first one we identify native functions that expect a raw pointer as one of the arguments. This pattern very often results in false positives as the fuzzer might pass random data instead of a valid pointer. This first approach would prevent 68 false positives (and 1 true positive), increasing the true positive rate to 18.32%.

The second approach is based on the observation that some native functions provide auxiliary functionality and are unlikely to process unconstrained attacker controlled input. We apply a lexical filter targeting function names indicative of common, generally robust, programming idioms or specific functional domains. For example, functions clearly denoting logging functionality (i.e., `.*_Log_.*`) are unlikely to be exploitable. This second filter would exclude 51 false positives and increase the true positive rate to 16.78%.

The two approaches combined would exclude 103 false positives (and 1 true positive), increasing the true positive rate to 25%. While we did not include prefiltering in our triaging experiment, we believe that for practical applications of POIROT, prefiltering effective in reducing the triaging effort.