Atlas: Automating Cross-Language Fuzzing on Android Closed-Source Libraries

Hao Xiong*†
College of Computer Science and Technology, Zhejiang University
Hangzhou, China
mart1n@zju.edu.cn

Qinming Dai*†
College of Computer Science and Technology, Zhejiang University
Hangzhou, China
qinm_dai@zju.edu.cn

Rui Chang‡†
College of Computer Science and Technology, Zhejiang University
Hangzhou, China
crix1021@zju.edu.cn

Mingran Qiu†
College of Computer Science and Technology, Zhejiang University
Hangzhou, China
qiumingran@zju.edu.cn

Renxiang Wang†
College of Computer Science and Technology, Zhejiang University
Hangzhou, China
renxiang.wang@zju.edu.cn

Wenbo Shen†
College of Computer Science and Technology, Zhejiang University
Hangzhou, China
shenwenbo@zju.edu.cn

Yajin Zhou†
College of Computer Science and Technology, Zhejiang University
Hangzhou, China
yajin_zhou@zju.edu.cn

Abstract

Fuzzing is an effective method for detecting security bugs in software, and there have been quite a few effective works on fuzzing Android. Researchers have developed methods for fuzzing open-source native APIs and Java interfaces on actual Android devices. However, the realm of automatically fuzzing Android closed-source native libraries, particularly on emulators, remains insufficiently explored. There are two key challenges: firstly, the multi-language programming model inherent to Android; and secondly, the absence of a Java runtime environment within the emulator.

To address these challenges, we propose Atlas, a practical automated fuzz framework for Android closed-source native libraries. Atlas consists of an automatic harness generator and a fuzzer containing the necessary runtime environment. The generator uses static analysis techniques to deduce the correct calling sequences and parameters of the native API according to the information from the "native world" and the "Java world". To maximize the practicality of the generated harness, Atlas heuristically optimizes the generated harness. The Fuzzer provides the essential Java runtime environment in the emulator, making it possible to fuzz the Android closed-source native libraries on a multi-core server. We have tested Atlas on 17 pre-installed apps from four Android vendors. Atlas generates 820 harnesses containing 767 native APIs, of which 78% is practical. Meanwhile, Atlas has discovered 74 new security bugs with 16 CVEs assigned. The experiments show that Atlas can efficiently generate high-quality harnesses and find security bugs.

CCS Concepts

• **Security and privacy** → **Software and application security**.

Keywords

Android; fuzzing; static analysis; vulnerability

ACM Reference Format:
Hao Xiong, Qinming Dai, Rui Chang, Mingran Qiu, Renxiang Wang, Wenbo Shen, and Yajin Zhou. 2024. *Atlas*: Automating Cross-Language Fuzzing on Android Closed-Source Libraries. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3652133

1 Introduction

Android OS is currently the most popular mobile operating system, accounting for more than 70% mobile market share. A significant portion of Android applications, including both pre-installed apps from vendors and popular applications from app markets, utilize abundant closed-source native libraries. Programmed in C/C++, these libraries are prone to memory corruption vulnerabilities. Such vulnerabilities can be exploited for malicious activities, including privilege escalation. Recently, there has been a notable increase in the frequency of attacks targeting these closed-source repositories. For example, a researcher from Google, Mateusz, has discovered

*Both authors contributed equally to this research.
†Also with ZJU-Hangzhou Global Scientific and TechnologicalInnovation Center, Hangzhou, China.
‡Corresponding author.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3652133

multiple memory corruption vulnerabilities in Samsung's closed-source image parsing library and implemented a 0-click RCE attack on Samsung MMS [7]. Therefore, it becomes imperative to detecting vulnerabilities in Android closed-source native libraries.

Fuzzing has been proven an efficient method for detecting security issues in complex programs [16, 20, 21, 26, 28, 44, 48, 50, 52, 54, 57]. For fuzzing libraries, creating a fuzzing harness is a crucial component of the fuzzing process and typically requires substantial manual effort, particularly when dealing with complex and numerous libraries. Thus, researchers have proposed various works focusing on automated harness generation [14, 32–34, 60], which have shown effective results on platforms like MacOS, Windows, and others. Similarly, the Android platform encounters this problem as well. However, the methodologies that have proven effective on other platforms are not directly applicable to Android closed-source native libraries. The cross-language programming model and the complex runtime environment cause existing fuzzing solutions to fail.

The first challenge arises from cross-language programming model. State-of-the-art works[34, 60] on automatic fuzzing single-language programs collect information through dynamic tracing. However, this process demands extensive manual effort and is not suitable for testing large-scale Android native libraries. Researches [42, 43] on automated fuzzing for multi-language programs focuses on extracting type and dependency information from source code. However, this approach is impractical for closed-source programs. Therefore it is essential to propose a solution for automatically generating fuzz harness for Android libraries. ❶ **These libraries present a key challenge as they are multi-language programs and often lack accessible source code.**

The second challenge stems from by the complex runtime environment. Fuzzing on real devices or emulators has represented two opposing camps in the field over the years[29, 43, 48, 49, 52]. Researches on fuzzing Android [19, 35, 37, 45, 51, 53, 55, 59] are primarily conducted using real devices, which provide an integrated runtime environment. However, these approaches present several disadvantages: (**I**) The CPU in mobile devices, prone to overheating, is unsuitable for fuzzing, which demands high-load operation and a long-term process. (**II**) Mobile phones are expensive and can not be expanded like multi-core workstations. A practical solution is to conduct fuzz testing within an emulator. However, the existing fuzzing emulators do not provide a runtime environment with JVM (Java Virtual Machine). Thus, researchers [7, 10] must reverse and re-write the target APIs to avoid calling JNI (Java Native Interface) functions, which is time-consuming and error-prone. ❷ **Thus, a further substantial challenge is to establish a fuzzing environment that excels not only in terms of compatibility and performance, but also facilitates the harnesses construction.**

To address the above challenges, we propose Atlas, a comprehensive framework to fuzz Android closed-source native libraries. Atlas contains two main techniques: **cross-language harness generation** and **enhanced emulation**. ❶ **To address the first challenge effectively,** the crucial task lies in accurately extracting API sequences and determining the appropriate parameters, integrating insights from both the Java and native domains. In harness generation, Atlas employs a cross-language static analysis approach to gather information on the dependency relationships of APIs, as well as the constraints and attributes of their parameters. ❷ **To overcome the second challenge,** the critical factor is managing the JVM dependency within the runtime environment. To achieve enhanced emulation, Atlas offers a practical fuzzing environment implemented in a user-land emulator, complete with all necessary dependent components. Consequently, Atlas not only enables more extensive fuzzing of Android closed-source native libraries but also streamlines the process of harness construction.

In summary, we make the following contributions:

- Our work represents the first systematic study that identifies and addresses the major challenges associated with fuzzing closed-source Android native libraries via emulation.
- We have proposed and developed Atlas, the pioneering fuzzing framework designed for Android's closed-source native libraries. Atlas is capable of automatically generating practical harnesses and providing an effective fuzzing environment for Android native libraries.
- We have evaluated Atlas on top Android vendors, including Samsung, Xiaomi, Vivo, and OPPO, and found many security bugs. Till now, we have found 74 unique security bugs and got 16 CVEs. We have responsibly disclosed them and helped the vendor fix them.

2 Background

This section introduces the technical background of automatic fuzz harness generation and the Android JNI mechanism.

2.1 Automated Fuzz Harness Generation

Fuzzing is a software testing technique by invoking APIs with random input. The fuzzing harness highly affects the fuzzing results. A high-quality fuzz harness includes the correct calling sequences and parameters of the target API, which requires much manual work. Thus, the automatic harness generation technology has been proposed and significantly improves the efficiency of testing abundant APIs. State-of-the-art automated harness generation technologies such as Winnie[34] and APICraft[60] have demonstrated the effectiveness of their solutions on Windows and MacOS. The basic idea of their solution is to collect information about API through dynamic execution and generate harnesses based on it. There are some automated harness generation works on the Android platform. For example, FANS[43] generates an effective fuzz harness through information from the source code of the Android native service. However, there are still no effective solutions for closed-source libraries on the Android platform.

2.2 Java Native Interface(JNI) in Android

Android JNI (Java Native Interface) is a mechanism for interacting between Java and native code (typically C or C++) in Android applications. It provides JNI functions for native code to access variables or invoke functions of Java. As shown in Figure 3a and 3b, we use the JNI function `setLongField` to set the value of the Java variable member `handle` in class P. As you can see, `Java_com_example_P_nativeInitHandle` is the native implementation of the `nativeInitHandle`, but how can the program find the implementation of the native methods declared in Java. As

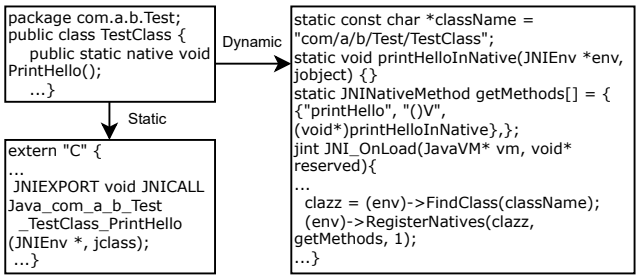

Figure 1: An example presenting static and dynamic methods of native function registration.

shown in Figure 1, Android JNI provides two different methods to register JNI methods.

- **Static Registration** In static registration, a programmer declares the native methods in the application layer. Then he/she uses javac/javap to generate the corresponding C/C++ header files. An obvious feature of static registration is that the names of native functions generated follow a certain rule. These function names are the package path and the class name, separated by the "_" symbol.
- **Dynamic Registration** In dynamic registration, a programmer needs to override the `JNI_Onload` function, which is called when the library is loaded and is responsible for binding native function pointers to the JVM. He/She can use the `registerNatives` function to set the mapping relations between native methods in the application layer and native functions in the native layer. There are no specific naming rules to obey in this progress.

3 Atlas Design

Figure 2 shows the high-level design of Atlas, our automatic fuzzing framework for closed-source Android libraries. To solve the challenges brought by limited information from closed-source libraries and the complex runtime environment of Android libraries, Atlas contains two main parts: (**I**) generating fuzzing harnesses for closed-source native libraries via cross-language analysis; (described in Section 3.1 and Section 3.2) (**II**) constructing an enhanced runtime environment for fuzzing. (described in Section 3.3)

System Overview. As shown in Figure 2, Atlas takes vendor libraries and Apks as its input. The whole pipeline can be divided into three main parts. Before entering the analysis, Atlas collects the mapping relationships between native functions (also as library APIs) and native methods in the Java layer according to the two registration modes of JNI functions. Then it will enter (**I**) the **cross-language analysis phase**. Native Code analyzer ❷ obtains variable dependency meta-information through taint analysis. Based on it, Java Bytecode Analyzer ❸ performs bottom-up static analysis to get the call sequences of target APIs and parameters and then outputs intermediate harness. After that, Atlas enters (**II**) the **harness optimization phase** to improve the qualities of the intermediate harness, and this phase contains three stages. In the deduplication stage, Atlas ❹ filters out harnesses with high similarity to avoid repeated testing of the same target APIs. In the completion stage, Atlas ❺ checks whether the parameters in the harness are all prepared. If not, it will heuristically complete the harness based on variable dependency meta information. In the injection stage, Atlas ❻ locates fuzzable parameters according to their attributes and generates the final harness. In the end, the output harness enters (**III**) the **fuzzing phase**. Atlas ❼ first sets the runtime environment with the essential Java environment and executes the harness within it to fuzz.

Pre-Process. Before harness generating, Atlas needs to find the corresponding relationships between the native functions and their declarations in the Java layer. Atlas finds the relationships through the parameters of JNI registration(e.g. `registerNatives`) and the static naming rules (mentioned in Section 2.2).

3.1 Cross-language Analysis

In this section, we discuss how to correctly infer API sequences as well as attributes and constraints of the API parameters, which are crucial to synthesizing high-quality fuzzing drivers. Since there is a multi-language programming model in Android native libraries, we comprehensively analyze the information from the native layer and the Java layer.

(1) Native Code Analyzer. Context switching results in the loss of API dependency information, which challenges harness generation. More specifically, when performing a bottom-up analysis on a target native API, the Java analyzer cannot deduce which native APIs on the execution path need to be retained for lack of information from the native world. Figure 3 shows explicit and implicit API dependencies derived from the real case encountered. Figure 3a is the Java implementation of `P` Class, which has a variable member named `handle`. This variable member can be initialized through the constructor `P(String str)` (line 2 of Figure 3c) or the `javaInitHandle` method (line 3 of Figure 3d). Figure 3c and Figure 3d are the consumer programs of `javaDecode`, showing explicit and implicit API dependencies, respectively. When the Java analyzer analyzes the API dependencies for `javaDecode` in the case of Figure 3c, it will keep lines 2-3 because an object must be initialized before calling the non-static method `javaDecode`. But in the case of Figure 3d, the Java Analyzer will not keep line 3 because of the lack of information from the native world. Therefore, it is necessary to analyze the native functions to obtain relevant information before Java Analyzer starts.

The Native Code Analyzer is designed to perform taint analysis on cross-layer variables, which are the basis for deducing implicit API dependencies. More specifically, it tracks the reading and writing operations performed on API parameters. To achieve this goal, we first divide the JNI functions into two groups. One is the reading functions(e.g., `GetStringUTFChars`), and the other is the writing functions(e.g., `SetLongField`). Then we symbolically execute the target native APIs and use a fake `JNIEnv` struct to redirect the JNI methods call to our analyzing functions. In parallel with symbolic execution, we trace the propagation of API parameters to improve the accuracy of the analysis. We achieve this goal by modeling the JNI methods. For example, in `c_str = (*env)->GetStringUTFChars(env, j_str, &isCopy)`, there is a taint propagation from `j_str` to `c_str`. We use the following

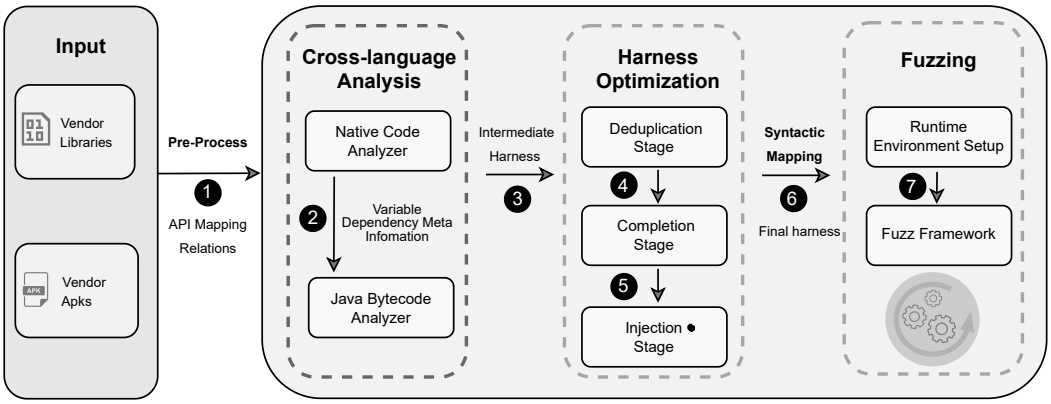

Figure 2: The Overview of Atlas.

```
1  package com.example;
2  public class P{
3   protected long handle;
4   native int nativeDecode(long j, Bitmap bitmap);
5   native void nativeInitHandle(P obj, String str);
6   public P(String str) {
7     this.mHandle = 0L;
8     if (str != null) nativeInitHandle(this, str);
9   }
10 public int javaDecode(Bitmap bitmap) {
11   if (bitmap == null) return 0;
12   return nativeDecode(this, bitmap);
13  }
14  public void javaInitHandle(String str) {
15    if (str == null) return 0;
16    return nativeInitHandle(this, bitmap);
17  }
```

(a) Java implementations of class P.

```
1  JNIEXPORT jint JNICALL Java_com_example_P_nativeDecode
2  (JNIEnv *env , jobject x, jlong cur_obj, jobject bm){
3    jclass jclazz = env->FindClass("com/example/P");
4    if(jclazz == NULL) return -1;
5    jfieldID jfid = env->GetFieldID(jclazz, "handle", "J");
6    if (jfid == NULL) return -1;
7    void * initial_ptr = env->GetLongField(cur_obj, jfid);
8    return nativeDecode(initial_ptr, bm);
9  }
10 JNIEXPORT void JNICALL Java_com_example_P_nativeInitHandle
11 (JNIEnv * env, jobject x, jobject obj, jstring s){
12   jclass jclazz = env->FindClass("com/example/P");
13   if(jclazz == NULL) return ;
14   const char *filePath = env->GetStringUTFChars(s, nullptr);
15   void * initial_ptr = nativeInitial(filePath);
16   jfieldID jfid = env->GetFieldID(jclazz, "handle", "J");
17   if (jfid == NULL) return ;
18   env->SetLongField(obj, jfid, (jlong)initial_ptr);
19   return ;
20 }
```

(b) Native Methods implementations of class P.

```
1 public static long wrapperFunc(File file){
2   P Decoder = new P(file.getPath());
3   Bitmap createBitmap = BitmapUtils.createBitmap(...);
4   Decoder.javaDecode(createBitmap);
5   ...
6 }
```

(c) The consumer program of class P containing explicit API dependency.

```
1 public static long wrapperFunc(File file){
2   P Decoder = new P();
3   Decoder.javaInitHandle(file.getPath());
4   Bitmap createBitmap = BitmapUtils.createBitmap(...);
5   Decoder.javaDecode(createBitmap);
6   ...
7 }
```

(d) The consumer program of class P containing implicit API dependency.

Figure 3: Explicit and Implicit API Dependency Examples.

format to record the read/write operations and the data source (from which parameter and its type signature).

$$(op, tarObj, nos, sig_1, sig_2, ..., sig_{nos})$$
$$op \rightarrow R/W$$
$$tarObj \rightarrow readSrc\ if\ op == R\ else\ writeDes$$
$$nos \rightarrow num\ of\ signature\ layers$$
$$sig_x \rightarrow the\ x_{th}\ layer\ of\ signature$$

The op indicates the operation type(read or write). The $tarObj$ represents the source of reading operations if op is reading(R) and refers to the destination of the writing operation if op is writing(W). The nos indicates the signature level of the operated object (because it may be a variable member within a nested structure). The last nos signatures(sig_n) recursively reveal the structure in which the operated object resides. Let's take the JNI methods in Figure3b for example. The R/W information of `Java_com_example_P_nativeInitHandle` is

$$(W, p2, 1, \{\text{“}com/example/P\text{”}, \text{“}handle\text{”}, \text{“}J\text{”}\})$$

which means `Java_example_P_nativeInitHandle` writes the 2^{nd} parameter and the signature of the writing destination is `{''com/example/P'', ''handle'', ''J''}`.

(2) Java Bytecode Analyzer With the help of native information, we can infer correct API calling sequences and parameters from the Java world. This part introduces how Java Bytecode Analyzer generates IH (Intermediate Harness) based on the R/W information of JNI methods from Native Analyzer. Java Bytecode

Analyzer uses a bottom-up multi-path search algorithm to analyze Java methods. In the beginning, we locate the Java methods that invoke the native method. Then we find the multiple execution paths containing the target Java methods. After that, Java Bytecode Analyzer performs dependency analysis. For each path, the analyzer performs an intra-method backward taint propagation analysis on the bytecodes. Specifically, it reserves the bytecodes related to the taint and filters out the rest. Meanwhile, the analyzer collects the constraints and attributes of variables from bytecodes. When the analyzer finishes the intra-method analysis, it recursively continues to analyze the callers. The detailed algorithm is shown in Algorithm 1 and Algorithm 2.

Algorithm 1 shows the workflow for generating IH from the beginning. The input is *JM*(all Java bytecodes of input APKs) and *RW*(native R/W information). To begin with, we find *jm* containing native method call for each *jm* in *JM*.(lines 2-3) For each *jm*, we record all calling locations in *nativeMethodLocs*. (line 4) For each *nmLoc*, we initialize a new object *ih* which records bytecodes, dependencies, constraints, and attributes of variables. For the bytecodes calling native methods, we have five steps to analyze each of them. (lines 8-12) (**I**) We store the bytecode. (line 8) (**II**) We analyze instructions such as assign and call to infer the dependency from the Java layer and update the dependencies of *ih*. (line 9) (**III**) We analyze the dependency from the native layer and again update it. If we find a variable that is read in the native layer, we add it to the unresolved dependency. If we find a variable is written, we remove the corresponding one as it is resolved. (line 10) (**IV**) For constraint analysis, we focus on the compare instructions and extract the constant values. (line 11) (**V**) For attribute analysis, we concentrate on the variable types and infer the possible usage of the variable. (line 12) We pay attention to some specific Java methods, e.g., `java/io/File->getPath()` and `android/text/TextUtils->isEmpty(java/lang/CharSequence)`, of which arguments or the return value can provide more precise attribute information. After these steps, we move on to a bottom-up recursive analysis, which is detailed in Algorithm 2.

Algorithm 2 describes the workflow for generating IH for a native method call. We find all the intra-method paths (Java bytecodes) that can reach the *sloc*. (line 2) For each *path* in *paths*, we firstly copy the previous *ih* as the entry point for the analysis. After that, we analyze the whole path from the end to the start. (lines 6-23) For each location, we update the dependency state of *ih* from the Java layer and the native layer. If the state changes, we store the *bc* and update the constraints and attributes of the variables. (lines 7-10) If the dependencies are still not satisfied after we have visited all *loc*s, we continue to analyze the callers if they exist. (line 22) In other words, the analysis terminates when all the dependencies are resolved or there are no callers. Finally, we add the *ihCopy* into *IH* and complete it in the next stage. (lines 13, 19)

Figure 5 shows an example used to illustrate how Algorithm 1 and 2 work. Figure 4 shows the corresponding source code. Assume that before analyzing the bytecode, the Java analyzer has known that the constructor function of pDecoder writes variable `mHandle` and `nativeDecodeFrame` reads `mHandle` in the native layer (obtained by Native Analyzer). The workflow of generating intermediate harness for `nativeDecodeFrame` is as follows: (**I**) The analyzer first locates `decodeFrame`, which invokes `nativeDecodeFrame`. Then it performs bottom-to-up tiant analysis. Specifically, it records instructions related to the parameters (the red and blue code). After that, the analyzer adds variable `mHandle`, `arr`, and `idx` to the dependency items since the initialization of them is not found within `nativeDecodeFrame`. (**II**) The analyzer then locates `start` to resolve the dependency items. It performs a similar analysis within `start`. The analyzer skips the green code as it doesn't associated with the taint data flow. The initialization of `arr` and `idx` is found directly within `start` (offset 24-28, 2c-2e, 38-3c). Based on the information from the native world, the analyzer also finds the initialization of `mHandle` (offset 0-4). After that, we will continue to analyze until we find that the attribute of `filePath` is the file path. Finally, the red and blue code is used to generate an intermediate harness of `nativeDecodeFrame()`.

Algorithm 1 Intermediate Harness Generation Workflow.

Input : All Java method bytecodes of APK/Jar JM, All native R/W Infomation RW
Output : A set of Intermediate Harnesses(Java bytecode sequence) IH

```
1:  IH ← ∅;
2:  for jm in JM do
3:      if jm.containNativeMethodCall() then
4:          nativeMethodLocs ← jm.getNativeMethodLocs();
5:          for nmLoc in nativeMethodLocs do
6:              ih ← new_intermediate_harness();
7:              bc ← jm.getBytecode(nmLoc);
8:              ih.add_bytecode(bc);
9:              ih.update_java_dep(bc);
10:             ih.update_native_dep(getNativeReads(RW, bc));
11:             ih.update_constraint(bc);
12:             ih.update_attribute(bc);
13:             IH ← IH ∪ AnalyzeJavaMethod(RW, jm, nmLoc, ih);
14:         end for
15:     end if
16: end for
17: return IH
```

Algorithm 2 Java Method Analyzing Workflow.

Input : All native R/W information RW, Java method byteCodes jm, start point of analysis $sloc$, intermediate harness(Java bytecode sequence) ih
Output : Set of Intermediate Harnesses(Java bytecode sequence) IH

```
1:  IH ← ∅;
2:  paths ← jm.getAllPathToLoc(sloc);
3:  for path in paths do
4:      ihCopy ← copy(ih);
5:      for loc in path.reverse() do
6:          bc ← jm.getByteCode(loc);
7:          if ihCopy.update_java_dep(bc)
            or ihCopy.update_native_dep(getNativeRW(RW, bc)) then
8:              ihCopy.add_bytecode(bc);
9:              ihCopy.update_constraint(bc);
10:             ihCopy.update_attribute(bc);
11:         end if
12:         if ihCopy.all_dep_satisfied() then
13:             IH.add(ihCopy);
14:             break;
15:         end if
16:     end for
17:     if !ihCopy.all_dep_satisfied() then
18:         if isEmptySet(jm.getCallers()) then
19:             IH.add(ihCopy);
20:         else
21:             for jmCaller, loc in jm.getCallers() do
22:                 IH ← IH∪ AnalyzeJavaMethod(RW, jmCaller, loc, ihCopy);
23:             end for
24:         end if
25:     end if
26: end for
27: return IH
```

```
1. public class pDecoder {
2.     protected long mHandle;
3.     protected native void nativeInitHandle(pDecoder pD, String str);
4.     protected native int nativeDecodeFrame(long j, byte[] arr, int i);
5.     protected native int nativeGetFrameNum(long j);
6.     public pDecoder(String str) {
7.         nativeInitHandle(this, str);
8.     }
9.     public int getFrameNum() {
10.        return nativeGetFrameNum(this.mHandle);
11.    }
12.    public int decodeFrame(byte[] arr, int idx) {
13.        if (arr == null) { return 0; }
14.        return nativeDecodeFrame(this.mHandle, arr, idx);
15.    }
16. }
17. public class GifCodec {
18.    public static boolean start(String filePath) {
19.        pDecoder pD = new pDecoder(filePath);
20.        int frameNum = pD.getFrameNum();
21.        pEncoder pE = new pEncoder();
22.        pE.setDispose(2);
23.        byte buf = new byte[0x10000];
24.        for (int i = 0; i < frameNum; i++) {
25.            pD.decodeFrame(buf, i);
26.        }
27.        return true;
28.    }
29. }
```

Figure 4: The source code of the example used for explaining Algorithm 1 and Algorithm 2.

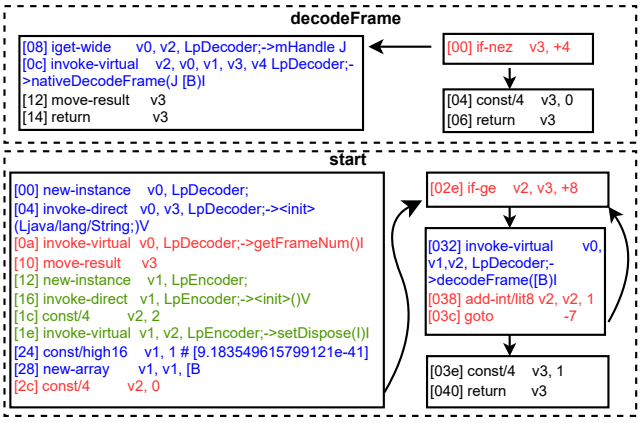

Figure 5: The bytecode and CFG of the example used for explaining Algorithm 1 and Algorithm 2.

3.2 Harness Optimization

We found that there are still 3 main problems within the intermediate harness: (**I**) duplicate harness with high similarity due to similar paths; (**II**) incompleteness in harness sequences due to the lack of path or the limits of static analysis techonology; (**III**) uncertainty of input data injection due to unclear semantics of the harness parameters. To improve the quality of the harnesses, we designed three stages to solve the corresponding problems: the **deduplication stage**, the **completion stage**, and the **injection stage**.

The **deduplication stage** is used to reduce duplicate harnesses. According to our observations, independent harnesses have the following two characteristics: (**I**) They have different native API call sequences; (**II**) Even if the calling sequences are the same, their parameters are different (The execution paths are sometimes determined by specific constant values.). Based on these observations, we propose a workflow to complete the deduplication task, as shown in Figure 6. The workflow of the deduplication stage contains three steps: (**I**) We cluster the input IH according to the native API calling sequence into IH_1 to IH_n. (**II**) For each IH_x, we perform another clustering within it according to the set of the constant parameters (3) For each subclass IH_{mCn}, we choose one harness as the representative and output it.

The **completion stage** judges whether the parameters of IHs are complete and complete them if necessary. There are two main reasons for the incomplete IH. One is that the limitation of the static analysis of the Java layer makes it hard to find the callers(line 21 in Algorithm 2). The second is that the variable dependency information can not be found due to the limitations of the symbolic execution on the native layer. To address the issue of insufficient context information, we utilize a heuristic insertion approach to maximize the completion of IH based on the available data. We first judge whether all variables and parameters have been initialized. If some variables are missing, we add them into the harness according to the type of the required parameter. If the missing parameter is a common class, such as *jstring*, we use a common template to complete its initialization. If it is some specific class, we will find the initialization process in existing IHs. Specifically, if the class X lacks its initialization in IH_A and we find it is initialized in IH_B, we will adapt the initialization code from IH_B to IH_A.

The **injection stage** locates the input point of fuzz data in IH. There are two types of JNI methods input, one is buffer type, and the other is file type. We first analyze the buffer or string that needs to be set manually in IH, i.e., non-constant. We set the buffer type input as the injection point and obtain its length, type, and other information to guide the fuzz input generation. If it is a string, we will decide whether it is fuzzable according to whether it is called by a file type API (e.g., File API in Java, fopen in C/C++).

After these three stages, Atlas generates the **final harness** according to syntax mapping rules, which will be handed over to the next stage for fuzzing. Figure 7 shows the harness generated for `getFrameNum` in Figure 4.

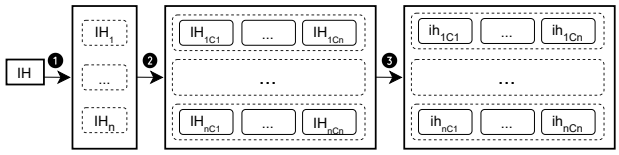

Figure 6: Depuplication stage workflow.

3.3 Enhanced Runtime Environment

We propose an enhanced runtime environment, which streamlines harness generation and execution. We discuss the existing solutions and introduce our novel enhanced fuzzing runtime environment.

One existing solution [7, 10] of fuzzing Android closed-source native libraries is to execute harness on a user-land emulator. However, for lack of a Java runtime environment, the execution process will crash when encountered with JNI function calls. To address this issue, state-of-the-art works utilize patching native APIs on a binary level or re-writing the harness to directly call pure native functions. However, these approaches require much manual effort in reversing and are error-prone. **Another solution** is to

```
1. int main()
2. {
3.     JavaVMOption options[1];
4.     JavaVM *jvm; JNIEnv *env;
5.     JavaVMInitArgs vm_args;
6.     long status; jint ret = -1;
7.     ...
8.     options[0].optionString = "-
   Djava.class.path=/path/to/the/packagefile";
9.     vm_args.options = options;
10.    status = JNI_CreateJavaVM(&jvm, (void **)&env, &vm_args);
11.    ...
12.    jclass class_D = env->FindClass("pDecoder");
13.    jmethodID methodId_init = env->GetMethodID(class_D, "
   <init>", "(Ljava/lang/String;)V");
14.    jstring fuzz_target0 = charTojstring(env, argv[1]);

15.    jobject ret_init = env->NewObject(class_D, methodId_init,
   fuzz_target0);
16.    jmethodID methodId_getNumOfFrame = env-
   >GetMethodID(class_D, "getNumOfFrame()", "()I");
17.    jboolean ret_getNumOfFrame = env->CallIntMethod(ret_init,
   methodId_getNumOfFrame, 1);
18.    ...
19. }
```

Figure 7: Harness code for example in Figure 4.

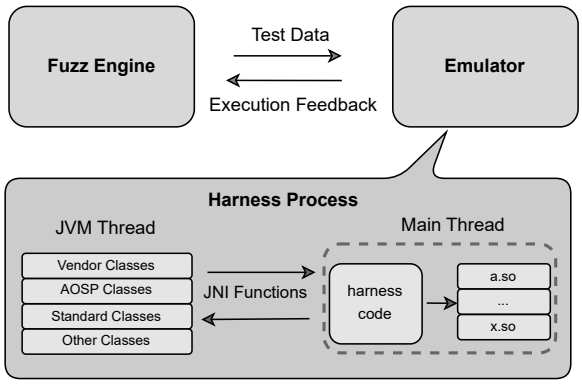

Figure 8: Atlas Fuzz Framework.

execute the harness within an OS-level emulator, which runs a whole Android OS and provides the essential Java runtime environment. Nevertheless, to the best of our knowledge, there is no existing open-source emulator that can run the Android OS from vendors with high compatibility. Besides, the OS-level emulation for fuzzing user-mode libraries results in low efficiency, let alone adding coverage collection or sanitizers into it. Iannillo et al.[31] have confirmed that utilizing ptrace to trace the coverage results in a performance slowdown of 11.97x on average. On the contrary, the user-land emulator provides a lightweight runtime environment with coverage feedback and address sanitizer. Thus, there is an urgent demand for an enhanced runtime environment with Java runtime environment in the user-mode emulator.

Our proposed fuzzing framework makes it easier to write and execute harnesses. As shown in Figure 8, the harness process running inside of the emulator receives test data from the fuzz engine. The fuzzing engine gets feedback from the emulator, such as code coverage and crash reports, to guide the mutation of test data. The harness process consists of two threads, *Main Thread* and *JVM Thread*. *Main Thread* loads the harness and native libraries and interacts with *JVM Thread* through JNI functions.*JVM thread* handles JNI function calls from *Main Thread*. Apart from supporting the general JNI functions like `GetStringUTFChars`, it should support accessing or invoking classes or methods defined in vendor-specific Java packages. Thus it needs to load essential Java packages including vendor classes, AOSP classes, standard classes, and other classes.

Algorithm 3 The Workflow of Fuzz Runtime Environment Setup for a Harness.

Input : A Final Harness fh
1: $load\_packages \leftarrow \emptyset$;
2: $load\_lib \leftarrow \emptyset$;
3: $Jmethods \leftarrow get\_all\_Java\_mthods(fh)$;
4: $Nmethods \leftarrow get\_all\_native\_mthods(fh)$;
5: **for** jm in $Jmethods$ **do**
6: $load\_packages \leftarrow load\_packages \cup find\_all\_packages(jm)$;
7: **end for**
8: **for** nm in $Nmethods$ **do**
9: $load\_lib \leftarrow load\_lib \cup find\_all\_alibs(nm)$;
10: **end for**
11: **if** isError($EnvLoad(load\_packages, load\_lib)$) **then**
12: $manual\_fix()$;
13: **end if**

Our workflow of fuzz runtime environment setup is described in Algorithm 3. To execute the harness, we need to find the essential native libraries and Java packages. We use the Java methods and native methods called in *ih* as the analysis entry. (lines 3-4) For *Jmethods*, we find the import packages from the file header of its class and recursively find all the dependent packages. (lines 5-7) For *nativemethods*, we locate the library where the method is implemented and recursively find all the dependent libraries of that. (lines 8-10) Then we load packages (*load_packages*) and libraries (*load_lib*) into the harness process. (line 11) We modify the initialization parameters of JVM to load Java packages. We load native libraries by linking or *dlopen*. During the loading process, there may be problems like Java package incompatibility, and the lack of dependencies on the packages or libraries. We manually investigate and fix the problems if we encounter any loading failure. (line 12)

The fuzz engine we choose is AFL, and the emulator is qemu-aarch64. We port OpenJDK[9] to the emulator to run *JVM Thread* for handling JNI function calls. Since the supported bytecode formats are different in OpenJDK and ART(Java used in), we use the tools dex2jar[1] to transfer the bytecodes and load them into *JVM Thread*.

4 Implementation

Atlas is implemented as a system with four modules, the pre-processing module (661 lines of Python), the cross-language analysis module (4,588 lines of Python), the optimization module (1873 lines of Python), and the Fuzzing environment module.

In the pre-processing module, Atlas automatically unpacks all the Apk files in the firmware to extract the dynamic link libraries. Then Atlas applies jadx[2] tool to decompile the Apk/Jar files to locate the native method declarations, and uses the IDAPython[3] script and the JN-SAF[56] tool to find the native functions in the dynamic link libraries.

The cross-language analysis module can be divided into the Native Code Analyzer submodule and the Java ByteCode analyzer

submodule. The Native Code Analyzer leverages and improves the tool JN-SAF[56] (we add support for 64-bit libraries and fix many minor bugs) to analyze the libraries. The Java ByteCode Analyzer uses the framework Androguard[5] to analyze the Apk and Jar files to extract method-level information, including the control flow graph and parameter information. The optimization module performs a translation from intermediate harnesses (consisting of Java bytecode) to final harnesses (C/C++ code) in the last, which relies on the syntax mapping rules written manually.

For the fuzzing environment module, we use afl-qemu[25] as our fuzz engine and use QASAN[24] as the sanitizer. We port OpenJDK [9] to our environment as the JVM engine and use it to load all the Java packages needed.

5 Evaluation

We evaluated Atlas on real-world Android applications to answer the following questions:

- How scalable and accurate are the fuzzing harnesses created by Atlas? (Section 5.1)
- How does each component contribute to the harness generation? (Section 5.2)
- How necessary is the complete fuzzing runtime environment and how efficient is it? (Section 5.3)
- Can Atlas discover new bugs in real-world applications? (Section 5.4)

Hardware Configuration We evaluate within Ubuntu 22.04 with an Intel Xeon Gold 6230R CPU (26 cores and 52 threads at 2.10GHz) and 128 GB RAM. In the throughput evaluation, we choose M1 Pro as the ARM64 platform and Pixel 6 as the real device.

Dataset Configuration To demonstrate the ability of Atlas of **fuzz harnesses generation**, we carefully select four/five complex apps from each of the 4 Android vendors. We choose these apps from the firmware with the March 22, 2023 security patch. Note that these 17 apps are all pre-installed applications by the manufacturer. These 17 apps perform different functionalities, covering image processing, audio decoding, font parsing, etc. These functionalities heavily use native libraries to complete complicated but error-prone functions, which are suitable for fuzzing. To demonstrate the ability of Atlas of **finding new bugs**, we extend our dataset to more pre-installed applications in the four vendors. In the throughput evaluation, we choose 8 programs from oss-fuzz[4] as our benchmark programs.

We believe our dataset of 17 apps, 767 native APIs, and 820 generated harnesses is reasonably large for our purposes. On the one hand, the apps come from four different Android vendors and cover different functionalities. On the other hand, the dataset is already larger than existing datasets used by previous works for harness generation. For example, Winnie [34] generates and evaluates 59 harnesses for Windows applications, Intelligen [61] selects 18 open-source projects and evaluates around 100 harnesses, and FuzzGen [32] selects 7 open-source libraries and covers 292 APIs.

Fuzz Configuration We assign 2 CPUs to each generated harness and fuzz it for 24 hours. We select from some public corpus[6] and the built-in files of mobile phones and use afl-cmin to remove redundant seeds. We use QASAN[8] as address sanitizer. We choose the default afl mutator.

5.1 Fuzz Harness Generation

Scalability of Atlas. We apply Atlas to 17 vendors' pre-installed apps to test the **scalability of harness generation**. Table 1 lists the apps we choose and the `FHG` column shows the harness generated by Atlas. We manually investigate 820 generated harnesses involving 767 native APIs. Specifically, we compile the harnesses and execute them in our fuzz environment. This phrase often involves time-consuming issues like linking the harness with essential dynamic libraries and loading indispensable Java packages. These require much manual effort since we have to find the missing libraries and fix incompatibility problems on loading Java packages into the environment, which complicates our experiment and that's why we choose only 17 apps. To validate the accuracy of the generated harness, we counted the harnesses that can be run without any modification (CH column), the harnesses that need manual modification (FM column), and the failed harnesses(F column).

The result shows that 51% of the harnesses can be used directly after being generated, while 27% require minor manual fixes, and 22% fail to work. Furthermore, we calculate the average ratio of the number of lines of code that need to be modified to complete the harness within the 27% fixed harnesses. It shows that only 5.6% of the code is required to be modified or added, indicating that it costs little manual effort. In total, 78% of the harnesses generated by Atlas are practical w/o a few manual modifications, illustrating the accuracy of Atlas. We count two reasons for the remaining 22% of harnesses' failure. Firstly, 29% of the harness failures are due to the incompleteness of the harness itself since the analyzer cannot find the caller. Secondly, 71% of harnesses are due to the dependency of the tested libraries on system-level or hardware resources, which cannot be provided in our user-level emulator. Besides, the time cost of CLA ranges from 2 minutes to 212 minutes, which heavily depends on the API amount and the size of input APKs. We also record the time costs of the OP-stage, which are all less than 2 minutes.

Accuracy of Atlas. To evaluate the accuracy of Atlas, we compare Atlas against human experts. To conduct this experiment, we recruited 4 participants in total. All the participants are professional researchers with over 3 years of Android security and fuzzing. We choose 2 APKs from each vendor and compare the code coverage between the auto-generated and manually-written harnesses. Table 2 shows the basic blocks of each harness within 24 hours. Overall, Atlas outperforms all the programs and discovered 1.37x more basic blocks than human experts on average. We calculate the p-values[46] and all of them are less than 0.05, meaning that the accuracy of Atlas is statistically significant($p < 0.05$).

Results. In this section, the statistics confirm that Atlas achieves both accuracy and scalability in harness generation.

5.2 Contributions of Each Component

We evaluate the contribution of the native analyzer and each stage of the optimization phase in generating the fuzz harness in this part.

Contribution of the native analyzer. In the cross-language analysis part, Atlas uses the native analyzer to collect variable dependency meta information in the native layer. To evaluate its impact, we keep track of the instances when the Java analyzer

Table 1: An overview of the harness generation of Atlas and effectiveness of each component.

Vendor	APK	FHG								CLA			OP-Stage			
		FH	CH	FM	F	F.Pct	F-1	F-2	API	IH	NN	T(min)	Dup	Comp	NoComp	Inj
Samsung	SamsungGallery2018	60	33(55%)	20(33%)	7(12%)	7.1%	2	5	78	233	51(22%)	9.7	125(54%)	21(17%)	79(63%)	60(48%)
	framework	59	28(47%)	10(17%)	21(36%)	9.6%	8	13	50	295	63(21%)	144.5	88(30%)	37(42%)	28(32%)	59(67%)
	SamsungCamera	49	27(55%)	6(12%)	16(33%)	4.9%	5	11	55	416	131(31%)	8.3	154(37%)	40(26%)	39(25%)	49(32%)
	Notes40	89	53(60%)	14(16%)	22(24%)	5.0%	4	18	74	2841	612(22%)	211.8	1867(66%)	125(7%)	132(7%)	89(5%)
Xiaomi	Gallery_T_CN	98	53(54%)	17(17%)	28(29%)	6.0%	16	12	82	1390	301(22%)	16.6	242(17%)	39(16%)	109(45%)	98(40%)
	MIMediaEditor	65	34(52%)	10(15%)	21(33%)	5.6%	13	8	51	1541	273(18%)	24.9	144(9%)	78(54%)	26(18%)	65(45%)
	SmartHome	59	38(64%)	7(12%)	14(24%)	5.7%	2	12	55	826	122(15%)	19.6	192(23%)	42(22%)	43(22%)	59(31%)
	MIUIMusicT	44	22(50%)	9(20%)	13(30%)	5.0%	2	11	39	846	206(24%)	24.8	103(12%)	40(39%)	37(36%)	44(43%)
OPPO	OplusEngineerCamera	44	16(36%)	22(50%)	6(14%)	3.6%	1	5	47	511	89(17%)	3.8	352(69%)	3(1%)	62(18%)	44(13%)
	WallpaperChooser	15	9(60%)	4(27%)	2(13%)	3.9%	1	1	17	163	41(25%)	2.6	39(24%)	3(8%)	22(56%)	15(38%)
	OplusEngineerMode	6	2(33%)	2(33%)	2(33%)	4.8%	0	2	10	204	18(9%)	4.6	82(40%)	18(22%)	28(34%)	6(7%)
	OplusLauncher	3	0	3(100%)	0	4.0%	0	0	3	27	3(11%)	5.3	13(48%)	0	13(100%)	3(23%)
	OplusLocationService	20	10(50%)	4(20%)	6(30%)	8.5%	1	5	14	76	21(28%)	2.0	21(28%)	16(76%)	5(24%)	20(95%)
Vivo	VivoBrowser	47	26(55%)	9(19%)	12(26%)	5.1%	2	10	45	720	172(24%)	47.3	152(21%)	34(22%)	35(23%)	47(31%)
	VideoPlayer	27	16(59%)	8(29%)	3(12%)	4.9%	1	2	25	331	47(14%)	135.7	75(23%)	15(20%)	26(35%)	27(36%)
	VivoCamera	80	52(65%)	14(18)	14(17%)	5.6%	2	12	72	1163	95(8%)	17.8	243(21%)	31(13%)	80(33%)	80(33%)
	VivoGallery	55	27(49%)	16(29%)	12(22%)	6.3%	6	6	50	500	64(13%)	43.8	126(25%)	19(15%)	63(50%)	55(44%)
Total		820	446(51%)	175(27%)	199(22%)	5.60%	29%	71%	767	12083	2309(19%)	42.5	4018(32%)	561(23%)	827(21%)	820(37%)

* FHG: Final harness generation. FH: Final harness. CH: Correct harness. FM: Fixed manually. F: Failed harness. AFLocs.Pct: Percentage of average manually fixed lines of code. F-1: Failed harness for incompleteness. F-2: Failed harness for lack of a runtime environment. API: Total APIs invoked in final harnesses. CLA: Cross-language Analysis stage. IH: Total intermediate harnesses. NN: Native analyzer needed harnesses. T: Time cost of cross-layer analysis.

Table 2: Comparison of basic block coverage.

APK	Library	BBs		p-value
		BH	MH	
SamsungGallery2018.apk	libagifencoder.quram.so	2303	2010	2.78E-03
ARDrawing.apk	libimagecodec.quram.so	2995	2350	3.12E-03
OplusEngineerCamera.apk	lib_rectify.so	794	540	3.45E-03
OplusLauncher.apk	liboplus_image_process.so	1131	735	1.52E-03
VideoPlayer.apk	libvad_check.so	350	241	5.63E-03
iReader.apk	libZYAccDecoder.so	3983	2720	1.43E-03
MIUIMusicT.apk	libmp4encode.so	6304	5700	4.32E-03
Gallery_T_CN.apk	libMiuiGalleryNSGIF.so	387	237	1.64E-03

* BBs: the amount of babic blocks. BH: the harnesses generated by Atlas. MH: the manually-written harnesses.
* Each harness is executed for 10 loops and each loop is 24 hours. The p-value is calculated according to Mann-Whitney U test[46].

utilizes the meta information on variable dependencies provided by the native analyzer. The results depicted in Table 1 indicate that out of the 12083 intermediate harnesses produced by Atlas, 2309 harnesses utilized data from the native analyzer for generating, representing 19%. This highlights the crucial role that the native analyzer plays in the generation process of harnesses.

Contribution of each stage of the optimization phase. As shown in Table 1, after completing the cross-language analysis, Atlas generates 12083 harnesses in total, which is a huge number for manual verification. We record the number of harnesses affected after the three optimization stages. In the deduplication stage, Atlas clusters harness with high similarity into groups. After this stage, the total harness is reduced to 4018 groups, and the average deduplication ratio is 68%. Atlas then randomly selects one from each group as the output of the current group and optimizes it in the next stage. In the completion stage, Atlas will complete the harness for missing variables and dependencies. In this stage, Atlas completes 561 harnesses, and the average completion ratio was 23%. In the injection stage, Atlas judges whether the current harness has proper input parameters suitable for fuzzing. After the injection phase, Atlas injects 820 harnesses within 4018 harnesses. On average, 37% of the harnesses are injectable, meaning that only a small part of the harnesses are fuzzable. After going through three optimization stages, Atlas reduces 12083 harnesses to 820 high-quality harnesses, which means it observably increases the usability of the generated harness without much manual effort.

Results. This section demonstrates the effectiveness and necessity of the native analyzer and the optimization stage, which verifies the rationality of our design.

Table 3: Executable Harnesses.

Vendor	Samsung	Xiaomi	OPPO	Vivo
M1	8/20	14/20	13/20	10/20
M2	14/20	19/20	16/20	18/20

* M1: Executable harnesses without Java runtime environment; M2: Executable harnesses with Atlas(equipped with Java runtime environment).

5.3 Necessity and Performance of Enhanced Fuzz Framework

The Necessity of Enhanced Fuzz Framework. We conduct a harness programming experiment to prove the necessity of our fuzzing framework. The experts are the same as those in Sec 5.1. We assign each expert 20 random native APIs and ask them to construct the harness within 4 days. Each expert is responsible for one vendor to ease familiarity with the characteristics of different vendors' libraries. We record the amount of executable harnesses in two different environments. As shown in Table 3, when experts use Atlas to write test frameworks, they can successfully write harnesses for 84% of the APIs with the help of the Java runtime environment in Atlas. However, they can only finish 56% without the environment. To summarize why experts can not finish some harnesses, we investigate the APIs and interview the researchers. In the end, we boil it down to two main reasons: (**I**) The native API needs to be rewritten is too complicated, and the rewriting process is prone to errors. (**II**) It's hard to provide some substitute functions and variables for complex JNI functions and Java variables within the native APIs. The results show that our fuzz framework provides the necessary runtime environment and streamlines the process of writing harnesses. Meanwhile, rewriting APIs or patching libraries

at the binary level requires a lot of reversing work and domain knowledge, which is error-prone and time-consuming. Compared with manually writing harnesses without a JVM environment, Atlas can provide a runtime environment for more APIs and streamlines the process of harness generation.

Throughput of Enhanced Fuzz Framework. We compare our fuzzing framework against those on Android emulators and real devices. The data in Table 4 is calculated using formula 1.

$$relative\_speed = \frac{speed}{speed_{QU\_RD\_A}} \quad (1)$$

Table 4: Comparison of performance.

Program \ Platform	QU-X86-L	D-ARM64-AE	QU-ARM64-AE	D-RD-A	QU-RD-A	D-ARM64-L	QU-ARM64-L
cJson	0.24	3.83	1.28	3.83	1.0	0.44	2.3
libjpeg-turbo	0.13	0.49	0.18	2.02	1.0	0.13	0.47
harfbuzz	0.55	2.96	1.33	7.03	1.0	1.07	2.92
c-ares	0.45	0.32	0.73	0.24	1.0	0.89	0.11
lz4	0.33	6.29	1.42	3.14	1.0	0.63	3.14
http-parser	0.23	6.33	1.36	4.75	1.0	0.43	2.38
libopus	0.59	1.63	0.8	1.38	1.0	0.89	1.1
zlib	0.39	3.61	1.31	3.66	1.0	1.3	2.31

* The platform consists of three parts, running mode, the architecture of the host, and the system information. QU: running within qemu-aarch64; D: running directly within the host; RD: Real Device(Pixel 6); L: Linux; AE: Android 12 on Android Emulator; A: Android12.
* Each harness is executed on a single CPU core for 10000 loops. We record the executing time of the target APIs, regardless of the initialization of the runtime environment since it is a one-time effort. The data we used for calculating is the executing time per second of each program on each platform.

According to Table 4, we can find the speed of our framework on x86 devices (QU-X86-L) and on arm64 devices (QU-ARM64-L) are on average 37% and 73% of the speed on real devices respectively. It indicates that our enhanced fuzzing framework has a relatively good performance overhead while providing an essential running environment (JVM) and necessary fuzzing capabilities (e.g. code coverage collection, and memory corruption detection), which makes it possible to fuzz at scale.

Surprisingly, the speed on QU-ARM64-AE outperforms real devices on 5 programs, which means it can provide both high performance and a complete runtime environment. However, the memory consumption of an Android Emulator is about 3GB, which is much higher than Atlas(in the range of dozens to hundreds of MB) and makes it impractical to fuzz at scale.

Results. This section indicates that our enhanced fuzz framework is necessary for harness generation and performs well.

5.4 New Bug Findings

In this section, we verify whether Atlas can detect bugs on real apps. In this part, we expand the scope of our samples to more applications with complex native functions and high privileges. We use Atlas to automatically generate fuzz harnesses and choose the harness involving complicated native functions. For each harness, we run the fuzz on 2 CPU cores for 24 hours.

New Bug Findings. As shown in Table 5 and Table 6, Atlas finds 74 unique bugs and gets 16 CVEs in 13 applications from 4 Android vendors. Note that the bugs in Xiaomi, OPPO, and Vivo have not been fixed yet. Thus we do not show the concrete libraries here. These bugs cover five different types, including but not limited to heap out-of-bound read, heap out-of-bound write, use after free, and null pointer deference. At the time of writing, we reported all these bugs to the vendors and collaborated with them on the fix. We find that the amount of bugs is higher in Samsung. Here are two reasons : (**I**) Our work starts from Samsung, and we spend more time fuzzing Samsung's native libraries. (**II**) The firmware of Samsung contains more vendor-customized native libraries to perform specific functions instead of using open-source libraries.

Table 5: Unique bugs found by Atlas.

Vendor	APK	Library	HR	HW	UAF	NPD	SO	Sum
Samsung	Sam***.apk	libagi***.so	1	2	0	0	0	3
	frame***.jar	libBar***.so	1	3	0	1	1	6
		libsmst.so	11	11	0	0	1	23
		libves***.so	1	1	0	0	0	2
	Sam***.apk	libFac***.so	1	0	0	1	0	2
		libsec***.so	2	0	0	0	0	2
	Notes40.apk	libSD***.so	11	0	1	0	0	12
	Apex***.apk	libape***.so	2	1	0	1	0	4
	ARD***.apk	libViz***.so	0	0	0	2	0	2
	Phot***.apk	libten***.so	0	0	0	1	0	1
	Vide***.apk	libsav***.so	2	0	0	0	0	2
Xiaomi	Galle***.apk	-	2	2	0	0	0	4
	MIUI***.apk	-	1	0	0	0	0	1
OPPO	Oplus***.apk	-	2	0	0	0	0	2
	Heyt***.apk	-	0	0	0	6	2	8
Vivo	iReader.apk	-	0	0	0	6	2	8
Total			37	20	1	12	4	74

* HR: Heap out of bounds read. HW: Heap out of bounds write. UAF: Use after free. NPD: Null pointer deference. SO: Stack overflow. Since some bugs are still in the process of being fixed, we do not disclose the information about the library information here.

Table 6: The CVEs found by Atlas.

Vendor	APK	Library	CVE	Bug-Type
Samsung	SamsungGallery2018.apk	libagifencoder.quram.so	CVE-2022-26092	HOOB-W
			CVE-2022-27821	HOOB-R
	Notes40.apk	libSDKRecognition Text.spensdk.samsung.so	CVE-2022-39852	HOOB-W
			CVE-2022-36845	HOOB-R
			CVE-2022-36841	HOOB-R
			CVE-2022-36844	HOOB-R
			CVE-2022-36843	HOOB-R
			CVE-2022-36860	HOOB-R
			CVE-2022-36863	HOOB-R
			CVE-2022-36862	HOOB-R
			CVE-2022-36842	HOOB-R
			CVE-2022-36846	HOOB-R
			CVE-2022-36858	HOOB-R
	framwork.jar	libsmat.so	CVE-2022-39882	HOOB-W
	ApexService.apk	libapexjni.media.samsung.so	CVE-2022-36854	HOOB-R
	VideoEditorLite_Dream_N.apk	libsavsaudio.so	CVE-2022-39891	HOOB-R

New Insights. In our fuzzing journey, we find some other interesting insights. (**I**) The vulnerabilities are discovered recently while the vulnerable components were introduced several years ago. It shows the lack of sufficient security testing when these components were introduced. We think **continuous security testing is required to keep APIs as secure as possible**. (**II**) Our bug reports conflict with the internal reports of the SRC(Security Response Center) several times, which indicates that **our research raises the attention of the SRC and shows great posture that they are conducting their internal security testing.** (**III**) In our fuzzing process, we find security bugs in the library developed by other manufacturers (non-Android manufacturers), indicating that **vendors should pay more attention to testing all of the imported APIs.**

Results. In this section, We demonstrate that Atlas can find new security bugs and discuss some insights about bug findings.

6 Related Work

Android Fuzzing. There have been a number of longitudinal studies on Android fuzzing [13, 19, 20, 35, 37, 45, 51, 53, 55, 59]. To test vendors' Java services, Chizpurfle [31] utilizes dynamic binary instrumentation for coverage-feedback fuzzing. FANS [43] achieves better results in Android native system services by generation-based fuzzing. However, interface models require source-code-level information and can not be extracted from vendor-implemented Java services. Aafer et al. [11] propose a log-based fuzzing technique aimed at Android SmartTVs, which successfully found both memory corruption vulnerabilities and physical anomalies on IOT devices. Nevertheless, these works [13, 19, 20, 31, 35, 37, 45, 51, 53, 55, 59] focus on the Java code and ignore the information from native code. Besides, all the harness execution environments of the works above are within the physical devices instead of in the simulator, which is unsuitable for conducting extensive security testing.

Fuzzing Harness Generation. As a key component in API testing, fuzzing harnesses are programs that invoke library APIs and greatly affect fuzz testing results. Quality harnesses require domain-specific knowledge, requiring expert experience and much manual effort. Many methods have been proposed for automatically generating fuzzing harnesses with high quality [23, 27, 36, 47, 63, 64]. Works by source-level static analyzing [14, 32, 61] achieve good results. However, these strategies will fail when producing fuzzing harnesses for binary without source code or debugging information. Apart from these works, WINNIE and APICraft are proposed to generate fuzz harnesses for closed-source binary. WINNIE filters out graphical functions and implements a fork mechanism to raise the fuzzing efficiency. APICraft pays more attention to getting more accurate semantic relations and mutating the invoking sequences of closed-source Mac OS SDK APIs more reasonably. Unlike these two works, Atlas mainly performs cross-layer static analysis to get the minimal correct API sequences for harness generation instead of accelerating the fuzzing process (like WINNIE) or recombining API sequences for SDK testing (like APICraft).

Cross-language Analysis. While providing many benefits in software development, using multi-language architecture introduces quite a few severe security bugs. The cross-language analysis is increasingly vital in bug detection as more and more real-world software systems are constructed with multiple program languages. Existing works aim at different program languages, e.g., Java ✕ C [12, 39, 40, 56, 58], Java ✕ Javascript [15, 18, 38], and Javascript ✕ C/C++ [17, 22]. The aforementioned works have shown their effectiveness in detecting bugs but lack universality. To solve the trouble brought by language heterogeneity, PolyCruise [41] performs a method called DIFA (dynamic information flow analysis) on the pure application level. Recently, POLYfuzz [42] offers better fuzzing harnesses by generating the inputs that are needed to trigger vulnerabilities for general multi-language software. Atlas draws inspiration from these creative works, but some differences remain. Atlas performs cross-language analysis on closed-source binaries, which means it does not need additional information except bytecode and assembly code. Apart from that, the pre-built runtime environment and harness are not required in the static analysis process of Atlas. Besides, to get the valid API call sequences from cross-layer variable-dependent inference, the static analysis adopted by Atlas is from the bottom (C/C++ layer) to up (Java layer).

7 Discussion

Static Analysis. Java Bytecode Analyzer is a static analysis module, so it also encounters difficulties in common static analysis tools. The inheritance of classes in Java makes it hard to determine or find some Java method callers, resulting in false negatives and positives. At the same time, for the loop structure in the control flow graph, Java Bytecode Analyzer can recognize some apparent patterns, and for the unrecognized loop structure, the functionalities of the generated harnesses may be broken. Besides, indirect jumps in Android applications can lead to inaccuracies in static analysis. This is because indirect jumps can have multiple targets, making them difficult to predict. Atlas also suffers from indirect jumps. For example, the inaccurate CFG resulting from indirect jumps may lead to incomplete harnesses. Atlas concentrates on generating high-quality harnesses with existing tools. The ability of Atlas will get improved if these common challengs in static analysis are tackled in the future.

Fuzzing Input. Existing researches [30, 65] show that providing a high-quality corpus for fuzzing tests can achieve better results. Atlas does not perform efficient selection or construction in this regard. For convenience, Atlas provides a mixed corpus for all harnesses, that is, put some well-known file types in this corpus, including images, audio, videos, JSON, XML, and so on. In addition, Atlas may find multiple injectable targets (parameters) within a harness during the injection stage. To handle this case, inspired by DAISY [62], Atlas uses a dispatcher to pass different types of input files to a harness.

Binary Analysis. Atlas adopts binary analysis and there're some limititions. For lack of source code, it's hard to obtain type information. In our scene, this problem is partly resolved by modeling JNI functions, which contain type information in their parameters. Besides, binary analysis may encounter path/state explosion problems as well as loss of semantic information in symbolic execution. Thus we think it's better to combine pure binary analysis and dynamic testing methods.

8 Conclusion

In this paper, we propose Atlas, a practical automated fuzz framework for Android closed-source native libraries. Atlas utilizes information from the "native world" and the "Java world" to automatically generate harnesses for native APIs. Also, it provides a fuzzing environment with the essential Java environment. We have tested Atlas on 17 apps to show its capability to generate high-quality harnesses. We also apply it to many pre-installed apps from Android vendors for new vulnerabilities. Till now, Atlas has discovered 74 new security bugs with 16 CVEs assigned.

Acknowledgments

We thank the reviewers for their insightful comments and suggestions. This work is supported by the Key RD Program of Zhejiang Province(No.2022C01165). The findings herein reflect the work and are solely the responsibility of the authors.

References

[1] 2010. dex2jar. https://github.com/pxb1988/dex2jar.
[2] 2013. jadx. https://github.com/skylot/jadx.
[3] 2015. IDAPython. https://github.com/idapython/src.
[4] 2016. oss-fuzz. https://github.com/google/oss-fuzz.
[5] 2017. Androguard. https://github.com/androguard/androguard.
[6] 2019. fuzzing-corpus. https://github.com/strongcourage/fuzzing-corpus.
[7] 2020. MMS Exploit. https://googleprojectzero.blogspot.com/2020/07/mms-exploit-part-1-introduction-to-qmage.html.
[8] 2020. qasan. https://github.com/andreafioraldi/qasan.
[9] 2021. OpenJDK. https://github.com/PojavLauncherTeam/android-openjdk-build-multiarch.
[10] 2022. The Hidden RCE Surfaces That Control the Droids. https://www.blackhat.com/asia-22/briefings/schedule/index.html#the-hidden-rce-surfaces-that-control-the-droids-26083.
[11] Yousra Aafer, Wei You, Yi Sun, Yu Shi, Xiangyu Zhang, and Heng Yin. 2021. Android SmartTVs Vulnerability Discovery via Log-Guided Fuzzing.. In *USENIX Security Symposium*. 2759–2776. https://doi.org/10.1145/3579856.3582834
[12] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, Giovanni Vigna, et al. 2016. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *The Network and Distributed System Security Symposium 2016*. 1–15. https://doi.org/10.14722/ndss.2016.23384
[13] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 217–228. https://doi.org/10.1145/2382196.2382222
[14] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. 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*. 975–985. https://doi.org/10.1145/3338906.3340456
[15] Sora Bae, Sungho Lee, and Sukyoung Ryu. 2019. Towards understanding and reasoning about android interoperations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 223–233. https://doi.org/10.1109/ICSE.2019.00038
[16] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043. https://doi.org/10.1145/2976749.2978428
[17] Fraser Brown, Shravan Narayan, Riad S Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. 2017. Finding and preventing bugs in javascript bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 559–578. https://doi.org/10.1109/SP.2017.68
[18] Achim D Brucker and Michael Herzberg. 2016. On the static analysis of hybrid mobile apps: A report on the state of apache cordova nation. In *Engineering Secure Software and Systems: 8th International Symposium, ESSoS 2016, London, UK, April 6–8, 2016. Proceedings 8*. Springer, 72–88.
[19] Marcel Busch, Aravind Machiry, Chad Spensky, Giovanni Vigna, Christopher Kruegel, and Mathias Payer. 2022. TEEzz: Fuzzing Trusted Applications on COTS Android Devices. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 220–235. https://doi.org/10.1109/SP46215.2023.10179302
[20] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing.. In *NDSS*. https://doi.org/10.14722/ndss.2018.23159
[21] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138. https://doi.org/10.1145/3133956.3134069
[22] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. 2021. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases.. In *NDSS*. https://doi.org/10.14722/ndss.2021.24224
[23] Sebastian Elbaum, Hui Nee Chin, Matthew B Dwyer, and Jonathan Dokulil. 2006. Carving differential unit test cases from system test cases. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. 253–264. https://doi.org/10.1145/1181775.1181806
[24] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. 2020. Fuzzing Binaries for Memory Safety Errors with QASan. In *2020 IEEE Secure Development Conference (SecDev)*. 23–30. https://doi.org/10.1109/SecDev45635.2020.00019
[25] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://doi.org/10.5555/3488877.3488887
[26] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696. https://doi.org/10.1109/SP.2018.00040
[27] Patrice Godefroid. 2014. Micro execution. In *Proceedings of the 36th International Conference on Software Engineering*. 539–549. https://doi.org/10.1145/2568225.2568273
[28] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. 2018. Enhancing memory error detection for large-scale applications and fuzz testing. In *Network and Distributed Systems Security (NDSS) Symposium 2018*. https://doi.org/10.14722/ndss.2018.23312
[29] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. 2020. {PARTEMU}: Enabling Dynamic Analysis of {Real-World}{TrustZone} Software Using Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. 789–806. https://doi.org/10.5555/3489212.3489257
[30] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 230–243. https://doi.org/10.1145/3460319.3464795
[31] Antonio Ken Iannillo, Roberto Natella, Domenico Cotroneo, and Cristina Nita-Rotaru. 2017. Chizpurfle: A gray-box android fuzzer for vendor service customizations. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 1–11. https://doi.org/10.1109/ISSRE.2017.16
[32] Kyriakos K Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. Fuzzgen: Automatic fuzzer generation. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 2271–2287. https://doi.org/10.5555/3489212.3489340
[33] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. 2023. UTopia: Automatic Generation of Fuzz Driver using Unit Tests. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2676–2692. https://doi.org/10.1109/SP46215.2023.10179394
[34] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. 2021. WINNIE: fuzzing Windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*. https://doi.org/10.14722/ndss.2021.24334
[35] Anatoli Kalysch, Mark Deutel, and Tilo Müller. 2020. Template-based Android inter process communication fuzzing. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*. 1–6. https://doi.org/10.1145/3407023.3407052
[36] Alexander Kampmann and Andreas Zeller. 2019. Carving parameterized unit tests. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 248–249. https://doi.org/10.1109/ICSE-Companion.2019.00098
[37] Stephen Kyle, Hugh Leather, Björn Franke, Dave Butcher, and Stuart Monteith. 2015. Application of domain-aware binary fuzzing to aid Android virtual machine testing. *ACM SIGPLAN Notices* 50, 7 (2015), 121–132. https://doi.org/10.1145/2817817.2731198
[38] Sungho Lee, Julian Dolby, and Sukyoung Ryu. 2016. HybriDroid: static analysis framework for Android hybrid applications. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 250–261. https://doi.org/10.1145/2970276.2970368
[39] Siliang Li and Gang Tan. 2009. Finding bugs in exceptional situations of JNI programs. In *Proceedings of the 16th ACM conference on Computer and communications security*. 442–452. https://doi.org/10.1145/1653662.1653716
[40] Wen Li, Haipeng Cai, Yulei Sui, and David Manz. 2020. PCA: memory leak detection using partial call-path analysis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1621–1625. https://doi.org/10.1145/3368089.3417923
[41] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. 2022. {PolyCruise}: A {Cross-Language} Dynamic Information Flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. 2513–2530.
[42] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. 2023. POLYFUZZ: Holistic Greybox Fuzzing of Multi-Language Systems. (2023), 1379–1396. https://doi.org/10.5555/3620237.3620315
[43] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan, and Jianwei Zhuge. 2020. FANS: Fuzzing Android Native System Services via Automated Interface Analysis.. In *USENIX Security Symposium*. 307–323.
[44] Qiang Liu, Flavio Toffalini, Yajin Zhou, and Mathias Payer. 2023. VIDEZZO: Dependency-aware Virtual Device Fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 3228–3245. https://doi.org/10.1109/SP46215.2023.10179354
[45] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. 2012. A whitebox approach for automated security testing of Android applications on the cloud. In *2012 7th International Workshop on Automation of Software Test (AST)*. IEEE, 22–28. https://doi.org/10.5555/2663608.2663613
[46] Nadim Nachar et al. 2008. The Mann-Whitney U: A test for assessing whether two independent samples come from the same distribution. *Tutorials in quantitative*

Methods for Psychology 4, 1 (2008), 13–20.

[47] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 75–84. https://doi.org/10.1109/ICSE.2007.37

[48] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. 2021. V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2197–2213. https://doi.org/10.1145/3460120.3484811

[49] Hui Peng and Mathias Payer. 2020. {USBFuzz}: A Framework for Fuzzing {USB} Drivers by Device Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. 2559–2575.

[50] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14. https://doi.org/10.14722/ndss.2017.23404

[51] Raimondas Sasnauskas and John Regehr. 2014. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*. 1–5. https://doi.org/10.1145/2632168.2632169

[52] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. {kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels. In *26th USENIX security symposium (USENIX Security 17)*. 167–182.

[53] Hossain Shahriar, Sarah North, and Edward Mawangi. 2014. Testing of memory leak in android applications. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*. IEEE, 176–183. https://doi.org/10.1109/HASE.2014.32

[54] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *2019 Network and Distributed Systems Security Symposium (NDSS)*. Internet Society, 1–15. https://doi.org/10.14722/ndss.2019.23176

[55] Xiaolei Wang, Yuexiang Yang, and Sencun Zhu. 2018. Automated hybrid analysis of android malware through augmenting fuzzing with forced execution. *IEEE Transactions on Mobile Computing* 18, 12 (2018), 2768–2782. https://doi.org/10.1109/TMC.2018.2886881

[56] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. 2018. 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*. 1137–1150. https://doi.org/10.1145/3243734.3243835

[57] Jinyan Xu, Yiyuan Liu, Sirui He, Haoran Lin, Yajin Zhou, and Cong Wang. 2023. MorFuzz: Fuzzing Processor via Runtime Instruction Morphing enhanced Synchronizable Co-simulation. (2023), 1307–1324. https://doi.org/10.5555/3620237.3620311

[58] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin TS Chan. 2018. NDroid: Toward tracking information flows across multiple Android contexts. *IEEE Transactions on Information Forensics and Security* 14, 3 (2018), 814–828. https://doi.org/10.1109/TIFS.2018.2866347

[59] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Haixin Duan. 2014. IntentFuzzer: detecting capability leaks of android applications. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. 531–536. https://doi.org/10.1145/2590296.2590316

[60] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. 2021. APICraft: Fuzz Driver Generation for Closed-source SDK Libraries.. In *USENIX Security Symposium*. 2811–2828.

[61] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. 2021. IntelliGen: Automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 318–327. https://doi.org/10.1109/ICSE-SEIP52600.2021.00041

[62] Mingrui Zhang, Chijin Zhou, Jianzhong Liu, Mingzhe Wang, Jie Liang, Juan Zhu, and Yu Jiang. 2023. 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)*. IEEE, 87–98. https://doi.org/10.1109/ICSE-SEIP58684.2023.00013

[63] Sai Zhang, David Saff, Yingyi Bu, and Michael D Ernst. 2011. Combined static and dynamic automated test generation. In *Proceedings of the 2011 international symposium on software testing and analysis*. 353–363. https://doi.org/10.1145/2001420.2001463

[64] Wujie Zheng, Qirun Zhang, Michael Lyu, and Tao Xie. 2010. Random unit-test generation with MUT-aware sequence recommendation. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 293–296. https://doi.org/10.1145/1858996.1859054

[65] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv.* 54, 11s, Article 230 (sep 2022), 36 pages. https://doi.org/10.1145/3512345

Received 16-DEC-2023; accepted 2024-03-02