POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

Android Native Library Fuzzing

Relatori
prof. Antonio Lioy
prof. Mathias Payer

Paolo Celada

Academic year 2021-2022

Summary

Android applications can have part of their components developed in a native language, such as C or C++. Developers, using the Native Development Kit, pack inside each application a shared library holding the native implementation of a subset of its methods. The Java Native Interface (JNI) allows each native method to interact directly with the rest, by providing a means to create or update Java objects, call Java methods, and several other operations. Two fundamental reasons lead to its integration: native programs have better performance, a key factor given Android's limited hardware, and offer the possibility to reuse tested and optimized native libraries. Unfortunately, any security guarantees provided by Java are invalidated when using native code. Native code does not provide temporal or memory safety and is susceptible to format string vulnerabilities and type confusion, which can all lead to critical consequences, including but not limited to code execution, privileges escalation, and control flow hijacking.

Security is therefore critical, and yet no public tools testing native components dynamically exist. Existing tools either perform data-flow static analysis or ignore any side-effect correlated to them while analyzing the overall application. When fuzzing native libraries in Android, the fuzzing engine should take into consideration that native code interacts with the rest of the application using the JNI, and therefore the Android Runtime (ART). If it is not capable of reproducing the ART behavior, the results generated by the fuzzer, if any, are not valid and reproducible by a stand-alone Android application.

We propose a framework to dynamically test native components in Android applications. First, we present the steps required to port a common fuzzing engine, AFL++, on an Android device, with the necessary patches. Then, we describe the design of a fuzzing harness crafted specifically to work with Android native components, which loads the ART to fulfill JNI requests, fetch the native target function address and fork its state at every execution to have the performance benefits of the fork server. It uses AFL++ as a black-box fuzzer. Considering the scarce performance when using it on a single Android device, we developed a framework to use such harness on a phone cluster, parallelizing per device each fuzzing campaign. The framework works together with a native method's extractor and is capable of fuzzing each method of a set of Android applications per name or signature. The results when using the framework on closed-source Android applications show that it is capable of both reproducing known CVEs in Android native components, and discovering new bugs. For any bug found, the library is manually analyzed using common debugging and reversing tools to perform root cause analysis.

Acknowledgements

First of all, I would like to thank Prof. Payer and the whole HexHive group for welcoming me and giving me the opportunity to work on a challenging but fascinating project. In particular, my thanks go to Luca, who patiently helped and guided me from day one. Then, I am grateful to all my friends, both the ones I knew since elementary school and the ones I met along my university journey and my basketball career, which all helped me during stressful times. Among them, I would like to especially thank my flatmates with whom I shared a considerable amount of time this past year. Then, my heartfelt thanks go to my girlfriend Giulia, for always being there and supporting me in this year of exchange. Finally, I would like to express my greatest gratitude to my family, which supported me in the educational journey that brought me to this achievement and encouraged me to always give it my best.

Contents

Chapter 1

Android Native Libraries

1.1 Background

Android is today the most popular operating system for mobile devices on the market. With its user-base of over 2.8 billion users, it is used in over 4000+ different devices all around the world, categorized as smartphones, tablets, and IoT devices [1]. It is an open-source operating system based on a modified version of the Linux kernel, developed by the *Open Handset Alliance* and other companies, with Google as the main sponsor. Currently, there are more than 18 billion different Android apps delivered as APK via different markets (as shown in more detail in Table 1.1) [2]. Because of its popularity and use, the security of the OS and the running applications is critical and subject to high interest in the scientific community.

App Market	*Number of APKs*
play.google.com	16,081,405
PlayDrone	1,446,564
appchina	1,127,350
anzhi	1,113,535
VirusShare	182,408
mi.com	113,583
1mobile	57,530
angeeks	55,818
slideme	52,467
fdroid	41,558
unknown	29,161
praguard	10,186
torrents	5,294
freewarelovers	4,145
proandroid	3,683
hiapk	2,512
genome	1,247
apk-bang	363

Table 1.1. Number of application's APK for several Android markets

Android applications are commonly developed in Java, a high-level language born with the slogan *"Write Once Run Anywhere"*, being platform-independent. Any machine equipped with a *Java Virtual Machine (JVM)* is able to compile its code, developed using the *Java Software Development Kit (SDK)*, to an intermediate interpretation called *Java Bytecode* (`.class`). Then, the standard bytecode is expected to run on every of such machines, no matter which one compiled it. The act of running requires first to be interpreted by the JVM. Interpreting is a live translation to the corresponding architecture-specific machine code. As this translation is performed on the

fly, it introduces a non-negligible performance overhead compared to languages directly compiled to machine-level code, such as C and C++.

Java is designed with particular attention to its security. Among the security features Java offers, we highlight the following:

- *JVM usage*: Java code is compiled and executed in a virtual machine (JVM), and as such it is possible to perform severe byte-code checks to prevent unauthorized/unwanted control flow modification and object accesses (for example bounds on checks when dealing with arrays);
- *pointer removal*: to prevent common security vulnerabilities such as dangling pointers, indirect pointer overwriting and use-after-free, Java does not support pointers explicitly and lets only the JVM use them implicitly;
- *memory safety*: Java is capable of handling memory automatically, through the use of the garbage collection mechanism. It mainly helps prevent memory leaks and memory corruption;
- *exception handling*: helps preserve the regular flow of the program, by specifying a fixed approach to handle errors.

Similarly for Android. It adopted Java as the main language for app development from the early stages. Code is now developed using the *Android SDK*, it is compiled to *JVM Bytecode* and then translated into *Dalvik Bytecode*(`.dex`). The *Dalvik Bytecode* is then run by the corresponding *Dalvik VM*, integrated with the JIT compiler, and now substituted by the *Android Runtime (ART)* to integrate even more sophisticated optimization techniques. The Dalvik VM is designed by Google to run on systems with limited hardware resources such as phones, which are delivered with limited processor speed and memory space. In terms of security and performance, applications written in Java (and only Java) reflect exactly what is listed in the previous paragraph about Java programs. They show modest performance, but worst compared to Native code, and inherit high security from the Java language.

1.1.1 Android Applications

Considering the design of the Android OS and the management of apps [3], each app runs as an individual Linux process, with its own unique memory space. This was first designed for security reasons, applying the concept of *sandboxing*. After booting an Android device, the first real process started once the Android runtime and the corresponding VM (either ART or Dalvik depending on the Android version) are set up is the *Zygote* process. The *Zygote* process is a special Android OS process providing code sharing across VM and fast and efficient applications startup. It preloads all classes and resources that an app will potentially need in its own address space and it starts listening on a socket for incoming requests from other processes (e.g. the *ActivityManagerService* responsible to start new applications). Whenever the user wants to start a new application, hence a new process, the *Zygote* handles the request by forking its state, providing to the new application an already initialized VM. It acts as a parent process for all applications. The function responsible for this is `Zygote.forkAndSpecialize()`. The VM that it provides to the new application can be either Dalvik VM or the Android runtime (ART). The Dalvik VM is the equivalent of the JVM, it was used until Android 4.4 KitKat. The ART instead is its successor. The core difference is the principle they follow to interpret Java code. The first follows JIT (*Just-In-Time*) compilation achieving low memory consumption, while the latter AOT (*Ahead-Of-Time*) compilation, improving application's overall performance. Focusing on ART, it is composed of a compiler (*dex2oat*) and a runtime (*libart.so*), simply a shared library loaded in the address space of the process.

1.2 Native Development Kit

In June 2009, Android released the first version of the Android Native Development Kit (NDK), a tool-set that allows app developers to implement part of the application using a native language.

The native languages integrated are C, C++, and assembly. Native code is compiled using the *Clang* compiler (*GCC* was provided as an alternative until NDK r17) into a shared library, packaged into the applications APK and integrated with the interpreted side using the *Java Native Interface*, in short JNI (explained in details in section 1.4).

For the scope of the project, it is important to list the reasons, with its advantages and drawbacks, on why the NDK was first introduced.

Advantages

First of all, it often improves the application performance, key factor given Android device's limited hardware. It allows code optimization at the assembly level, and it gives developers the possibility to use processor features that are not available through the SDK. A research conducted in 2001 by Wentworth and Langan [4] states that the original interpreted Java is around 11 to 20 times slower than its C++ counterpart, while the integration of JIT brought it down to be only 1.45 to 2.91 times slower (study conducted using Java 1.3 and 4 famous sorting algorithms, precisely bubble sort, insertion sort, quick sort, and heap sort). JIT is an optimization technique introduced subsequently in the JVM, allowing translation from Java Bytecode to machine-level instructions only the first time a method is called. It often happens, but not always, as the integration directly translates to work for the Dalvik VM, responsible to handle this "context switch" between compiled and interpreted code. In addition, being compiled prevents the VM to apply run-time optimizations.
Secondly, it enables reuse of already existing native libraries. It speeds up application development with libraries already optimized and tested, removing the need to rewrite them completely. Various fields benefit from this, mainly:

- *graphics operations*;
- *game engines*;
- *physical simulations*;
- *networking operations*;
- *signal processing*;
- *cryptographic operations*.

To give some examples, the messaging application *WhatsApp* benefits from it by reusing the native version of the Open Whisper System's open-source protocol for cryptographic operations, while apps like *Twitter* or *Facebook* use a library called *Fresco* for image loading and processing tasks.

Drawbacks

The most important and related one is that compiled code is highly insecure. It does not provide spatial and temporal memory safety, nor type safety, and can lead to multiple software vulnerabilities. An empirical study from Gang Tan and Jason Croft (2008) [5] on the native code present in the implementation of the Sun's JDK reports common patterns of bugs present. All are mapped to criticalities in the interaction with native code in Android applications. Recurrent patterns are:

- *incorrect exceptions handling*, leading to undefined control flow;
- *race conditions in file accesses*;
- *type confusions*, between Java and native types;
- *buffer overflows*;

- *insufficient error checking.*

Their common weakness enumeration is *CWE-111*(*Direct Use of Unsafe JNI*). In addition, native code is vulnerable to all common compiled language vulnerabilities, that could be either present in any third-parties libraries used or introduced by the developer itself. Commons are buffer and arithmetic overflow/underflow, temporal safety violations, format string vulnerabilities, type confusions [6]. The consequences are code execution, information leaks, privileges escalations, DOS, control flow hijacking, etc.
Following comes the fact that it highly increases the application's overall complexity and its integration prevents the application to be portable to any architecture equipped with a VM. Each application must be delivered to the end-user with a copy of all used native binaries for each of the target's platform. Common ones are *aarch64*, *arm*, *x86* and *x86-64*.

1.3 Native Libraries in Android Applications

Given its advantages, an increasing number of companies and their developers decided to opt for the integration of native components in Android applications. They are present in large and complex apps as *WhatsApp* and *Instagram*, or in small apps used for video conversions, encryption, image handling and so on. Their presence does not depend on the category of the application, but more on the requirements and specifications of the software. A study conducted in 2016 by Afonso et al. [7], even if with a different goal in mind, statically analyzed and filtered a total of 1.2 million Android apps to extract a subset of 446 thousands apps using native code in some extent. The percentage of native apps is therefore high, around 37.2%. The common usage pattern found by just looking at the source code of apps available as open-source (e.g. from the F-droid market [8]) is that developers mostly write multiple native functions only to use them as entry points to some third-party library functionality. In a single entry function all data handling and conversions happen, applications states are checked for presence and correctness and then finally there is the interaction with the native library, via a well defined API. Alternatively, the programmer can directly use a Java class which underneath has some native methods. In this case, the library is composed of classes and methods which help the Java programmer interface its application to a native library (e.g the GIF parser library *Android-Gif-Drawable* [9]).

1.3.1 Analysis and Testing

Over the years, Google has invested a lot of time and resources to limit the possible security vulnerabilities in each Android application. From running them in a restricted sandbox (similar to the one on which Java runs) to restricting inter-app communication to a well-defined API and using implicitly all security measured provided by leveraging the Linux kernel as foundation. In addition, considering the original *Google Play* Android market, all applications are vetted after being uploaded but before being made public. This allows the discovery of possible bugs and vulnerabilities early in time. Each application is tested using different static and dynamic techniques, which mainly target the Java side of the application, while there are no public tools to target the native side of the application, specifically considering dynamic tools. The latter is of high importance given the large number of security vulnerabilities introduced. Several main causes make it difficult, mainly:

- *closed-source*: most Android applications are closed-source, and only the compiled binary of the native component for each target architecture is available for security analysis. Analyzing the top-18 markets in Table 1.1, only 0.21% of the apps are guaranteed to be open-source. While Dalvik or Java Bytecode can be decompiled retaining precision, the same can not be applied to native compiled libraries;

- *JNI usage*: all native methods use in different amounts the JNI, through the `JNIEnv` interface pointer. When testing those methods, if the first parameter is not a valid `JNIEnv` pointer the execution terminates in a segmentation fault as soon as the target tries to dereference it at a

certain offset. This makes it complicated to analyze it for example using a trivial black-box fuzzing approach. AFL++ requires that at least one execution terminates successfully with the provided seeds, and with this set of targets it would not even start;

- *analysis tool portability on Android*: even if Android is based on the Linux kernel, certain libraries available are different with respect to their Linux counterpart (e.g *libc* version for Android is *bionic*), and the linker might be unable to resolve certain symbols (e.g. NON-POSIX extension's functions);
- *stateful analysis*: whenever an Android application calls a native method, its execution depends on a possible state built running the application in a normal environment. This state is checked or used by the native method itself, and its absence may prevent any valid analysis of the target. As an example, consider an application using a native library to perform image processing. In order to apply a sepia filter to an image (with native function *Java_com_pkg_image_applySepia*), the application first loads the image using another native function (for example *Java_com_pkg_image_loadImage*). In the case the filter function does not perform any safety checks and it is invoked without first loading the image, it always results in a crash;
- *guided analysis*: considering fuzzing as a dynamic testing technique, black-box only fuzzing may not be sufficient. To achieve grey-box fuzzing, meaning with coverage instrumentation guiding the fuzzer to reach deeper into the program, we should statically or dynamically add instrumentation to the target binary. This is achieved by integrating known tools into the fuzzing engine.

1.4 Java Native Interface

To enable operational compatibility between the Java and one of the interpreted languages previously listed, every sort of communication must go through the Java Native Interface.
The Java Native Interface (in short JNI), is a *foreign function interface programming framework* that allows developers to utilize the NDK, more specifically libraries developed in C, C++, or assembly, in an Android application. It makes it possible for Java running in the usual JVM (or ART in Android) to call and interact directly with any native programs and vice-versa (layout presented in Figure 1.1). It is composed of several functions and types definitions, all found inside the `jni.h` header file, while their implementation in the shared library `libart.so`. Following a detailed explanation of its design, the main components, and some usage examples.

1.4.1 Design

The JNI is composed of a set of functions, types, and constants. JNI functions are both defined as regular functions or through an interface pointer, head of a chain of pointers to a function interface. The JNI interface pointer points to a per-thread data structure pointer, which itself points to an array of pointers. Each of these pointers is defined at a fixed offset from the beginning of the array and points to an interface function. This interface function will then contain the actual implementation of the JNI function. The overall structure is similar to a C++ virtual method table or a Microsoft COM interface. A visual description of the pointer's chain is reported in Figure 1.2.

It is designed to satisfy the need to change type of language for both Java or Native applications, and it can be used in two opposite ways.
The first one is to use the JNI to integrate into any Java application components written in C, C++, and assembly. The native side interacts with the Java side exclusively through a series of JNI function calls. Looking at the Java side instead, the only addition is a step to load the library and one to declare methods as native. To load a specific native library into the application, it is required to call the `System.loadLibrary()` method by passing as the only argument the name of the shared library inside a *static* class initializer in the class using the library. The name of the library respects the normal pattern, without the prefix *lib* and the file extension, and it is

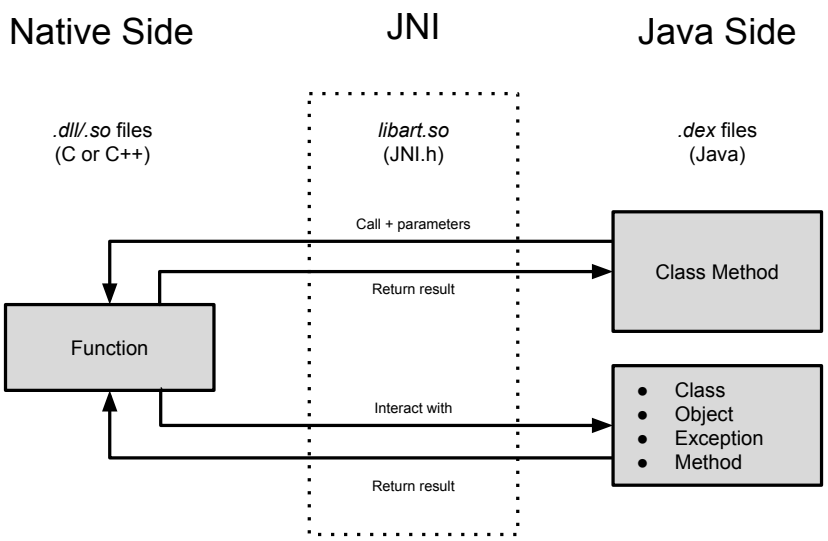

Figure 1.1. Interaction between Java and native side.

the JNI itself in charge to understand which extension to append (`.so` for Linux systems or `.dll` for windows ones). Then, native methods are given a meaningful name, declared using the *native* keyword, and are usually private. This tells the Java compiler that the method is provided in a different language. The methods name influences its corresponding name on the native side, as seen in subsection 1.4.3. In Figure 1.3 an example showing JNI usage on the Java side.
Another way to use the JNI is to allow any native applications to embed a Java virtual machine implementation and interact with it. The JVM must be loaded through a function called `JNI_CreateJavaVM` and the methods needed by the JVM framework linked using the corresponding function interface `registerFrameworkNatives`. From this moment on, the native application can use the invocation interface to execute any software components written in Java. The project strongly depends on both usage cases.

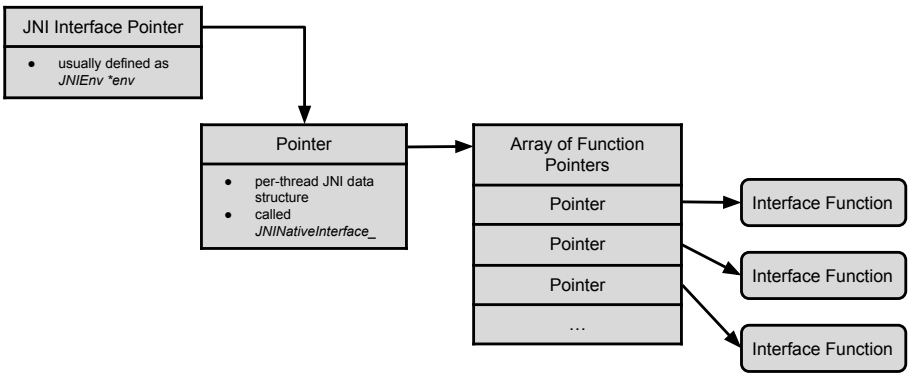

Figure 1.2. JNI Interface pointer structure (source: Oracle).

```
1         package com.name.jni.package
2         public class JNI{
3                 /* Load native library */
4                 static {
5                         System.loadLibrary("nativeLibrary");
6                 }
7
8                 /* Call native method */
9                 public static void main(String[] args) {
10                        new HelloWorldJNI().nativeMethod();
11                }
12
13                /* Native method declaration */
14                private native void nativeMethod();
15        }
```

Figure 1.3. Library loading and native method definition example.

1.4.2 JNI Functions and Primitive Types

Interface pointers and their implementations are key for the communication between the two contexts, and there are a total of two:

- *JNINativeInterface*: called `JNIEnv`, it points to an array of interface functions that deal directly with parameters transfer, methods calls, objects interaction, exception handling, and many more operations between the two sides. There are a total of 229 of these functions;
- *JNIInvokeInterface*: called `JavaVM`, it similarly points to an array of interface functions used to handle VM-related operations, such as fetching the corresponding JVM environment, handling threads, and destroying the VM;

In Table 1.2, some of the most used functions are reported for both interface pointers, with self-explicates names.

Data types instead are mapped depending on the type in the Java programming language. *Primitive types*, which include `int`, `float` and `char`, directly map into the respective JNI version, respectively `jint`, `jfloat` and `jchar`. *Reference types* instead, mainly array, classes, and instances of a class, are passed by reference to the native code and should be handled only through appropriate JNI functions. Signatures of each type are represented in the JNI using *smali*, the Java VM type representation [10].

JNINativeInterface	*JNIInvokeInterface*
NewObject	DestroyJavaVM
FindClass	AttachCurrentThread
GetMethodID	DetachCurrentThread
CallCharMethod	GetEnv
CallVoidMethod	AttachCurrentThreadAsDaemon
NewStringUTF	
GetStringUTFChars	
ReleaseStringUTFChars	
...	

Table 1.2. Most used functions for each interface pointer.

1.4.3 Naming Conventions

The JNI helps the runtime finds and links the implementation of native methods. Developers can decide to follow each one of the two possible naming conventions. The first follows a defined pattern name generated by the JNI itself, while the second one is a dynamic mapping of the preferred name and the function name of the Java implementation.

Defined pattern

A native method is defined inside the application's package and the corresponding Java class. When the building toolkit used (for example *Gradle* for Android Studio) invokes the Java compiler *javac*, a header file is automatically generated. It defines the name, parameters, and return type of any native methods defined in the application. The name of the function is composed of:

- A fixed *Java* prefix;
- the *package name*;
- the *class name*;
- the *method name*, as defined in the Java class;
- the *mangled argument signature* preceded by two underscore symbols, for overloaded native methods.

In-between all fields the underscore symbol, escaped with `_1` in case it is present in one of the names composing it. (Figure 1.4). All symbols are exported from the library, and the JVM look them up dynamically with `dlsym`.

```
1  #include <jni.h>
2
3  /* Defined pattern native method implementation */
4  JNIEXPORT void JNICALL Java_com_name_jni_package_JNI_nativeMethod
5    (JNIEnv *env, jobject thisObject) {
6        printf("Hello␣World!");
7  }
```

Figure 1.4. Defined pattern methods implementation.

Dynamic linking

This convention is more involved. It requires slightly more code by the programmer and it is error-prone, but might be preferred in some cases. The developer is in charge of registering them explicitly. Native methods can now take any preferred name and their implementation can be updated at runtime, relieving some of the VM workloads. In addition, it is useful when a native application having loaded a Java VM needs to link a Java method with a function defined in the native application itself. The VM would not be able to link against this function as it would search only in native libraries, not in the native application itself. The programmer is required to write inside the native library the implementation of a function called `JNI_OnLoad`. Its function interface is defined inside the `jni.h` header file, and it is the first function searched and called, if present, by the JavaVM after loading the library with `System.loadLibrary`. It is therefore common behavior to use it to set up any state needed by the native library itself. The implementation must then contain a call to `RegisterNatives`, interface function part of the `JNIEnv` array of functions. It takes as parameters the class in which the methods are being registered, a `JNINativeMethod`

structure containing information about the methods, and the number of methods. The structure `JNINativeMethod` is composed of: the method's name, as defined on the Java side, the *smali* signature specifying the method's parameters and return type, and the address of the methods implemented on the native side. A trivial example is shown in Figure 1.5, together with the *JNINativeMethod* structure definition. When following the dynamic linking convention only the `JNI_OnLoad` is exported, producing faster and smaller code, and avoiding the risk of collision with symbols of other libraries.

It is important to notice that for both conventions all methods always include as the first two parameters the JNI environmental pointer and the Java object that the method is attached to, respectively `JNIEnv *` and `jobject`.

```
1  /* Structure used by RegisterNatives to declare native methods via
       dynamic linking */
2  typedef struct {
3          char *name;
4          char *signature;
5          void *fnPtr;
6  } JNINativeMethod;
```

```
1  #include <jni.h>
2
3  /* Native method implementation */
4  JNIEXPORT void JNICALL newNativeMethod
5    (JNIEnv *env, jobject object) {
6          printf("Hello␣World!");
7  }
8
9  /* Structure used by RegisterNatives */
10 static JNINativeMethod methods[] = {
11         {"nativeMethod", "()", (void *) &newNativeMethod}
12 };
13
14 /* Function called when loading the native library */
15 JNIEXPORT jint JNI_OnLoad(JavaVM *vm, void *reserved) {
16         JNIEnv *env;
17         jclass clazz;
18
19         // Retrieve JNIEnv * and native method's class
20         (*vm)->GetEnv(vm, (void **)&env, JNI_VERSION_X_X);
21         clazz = (*env)->FindClass(env, className);
22         ...
23         // Register native methods via dynamic linking
24         (*env)->RegisterNatives(clazz, methods, sizeof(methods) /
               sizeof(methods[0]));
25         ...
26 }
```

Figure 1.5. JNINativeMethod structure and dynamic linking of methods implementation.

1.5 Fuzz Testing

Fuzz testing is a dynamic software testing technique, fully automated, in which a software component is tested by invoking it with unexpected, random, or invalid inputs. The final goal is to be able to detect as many crashes or hangs as possible from the program under test (or PUT), by following a quasi brute-force approach. It is a probabilistic testing technique based on multiple single concrete executions, and full data-flow coverage could be achieved only if run for an infinite amount of time (considering non-trivial code bases under test). Data-flow coverage is the process of considering all possible cases in a program execution based on how a variable can be defined and used. The overall process is considered to be *sound*, *scalable*, but *incomplete*. It is *sound* because, whenever a bug is found, it directly generates a proof of concept (POC). It, therefore, does not produce any false positives, contrary to the behavior of static testing techniques such as symbolic execution. It is *scalable* as it can be easily applied to large code bases. Lastly, it is *incomplete* because it is impossible to expect that the fuzzer will cover all possible inputs, and so control flows, due to the random nature of the input generation. Due to its high efficiency, simplicity, and soundness, it is subject to multiple research papers improving its effectiveness. One of the challenges remaining is the ability to port it to different domains, from kernels to web applications to Android applications. Common fuzzing engines include coverage-guided fuzzers like *AFL* [11], fuzzer developed by M. Zalewski and now substituted with its community-maintained fork *AFL++* [12], but also *honggfuzz* [13], *libFuzzer* [14], and others. Each has its unique characteristics, but the main idea behind remains the same.

1.5.1 Design

The key idea behind a fuzzer is to intentionally produce and provide malformed inputs into a system to generate and record a crash. It is composed of different components smoothly interacting with each other, carefully set up to achieve efficient target execution. Following the detailed description, with a visual representation in Figure 1.6. As fuzzing can be ported to several different domains, we focus on native code fuzzing.

Target Definition

The target program must be an executable binary that accepts some form of input from the user, e.g. from `stdin`. It then performs operations using such input, possibly triggering a crash. It is important to notice that a bug does not always trigger a crash, so an abnormal program termination. To avoid limiting the ability to find crashes to architectural violations (page faults, division by zero, etc.), assertion failures, or mitigations, often the target code is compiled with some form of sanitization, for example *ASAN* [15]. Another improvement given by the target binary is if it has been compiled with coverage instrumentation, or it has been statically rewritten to add it, meaning to transform the target executable to add instrumentation instructions while maintaining the original functionality. This helps the fuzzing engine achieve better code coverage, mutating the input using coverage feedback from the program itself. Both have their pros and cons, in terms of techniques complexity, performance, and ability to find new bugs.

Input Generation

An initial seed set needs to be provided to the fuzzer so that new inputs generated will be based on it. Having a diverse and complete set of initial seeds is key for efficient fuzzing. A single input can be generated following a pre-defined model, or it can be *mutation-based* if it includes deterministic and *havoc* phases, such as bits flipping, randomization, slicing, merging, and replacing inputs. Moreover, inputs most of the time must conform to API specifications that the target program must follow to avoid stopping as soon as those are checked. This is retrieved either by the program API when available, by source code analysis, or by reversing the program binary.

Target Environment

When fuzzing non-stand-alone programs, the execution environment in which the fuzzer runs the program must conform with the one expected by the program itself. In particular, when fuzzing libraries API, very often those must be called together with some environment state that is checked or used by the function itself and created by other parts of the program before the target function is called in a regular execution. To create a reliable and valid state, the fuzzing engine can leverage symbolic execution as an initial phase. Symbolic execution allows generalizing the execution of a program, by considering all possible inputs at once during an abstract execution. Inputs do not hold concrete values but symbolic ones, and the program execution is translated into a mathematical set of constraints. By solving them, it is possible to generate a concrete execution. Fuzzing benefits from it in particular when fuzzing libraries. In this use case, symbolic execution can abstract the program execution until the library call, solve the generated constraints, and provide the fuzzer a valid state so that the library function is used as expected.

Fork Server

Component introduced later in *AFL* and other fuzz engines as a form of optimization technique to increase throughput (executions per second). The fork server is injected as instrumentation into the target, and it is controlled through an IPC mechanism. It reduces the performance bottleneck generated by the `execve` syscall by stopping the execution at the beginning of the main function (the driver) and forking a child which inherits the program state generated during program and AFL initialization. The fork is executed in an infinite loop, and the instrumentation code is placed either before the actual target operations, or wherever deemed necessary when fuzzing libraries with the use of a fuzzing harness, a function in the middle between the fuzzer and the target responsible for calling it with the input generated by the fuzzer.

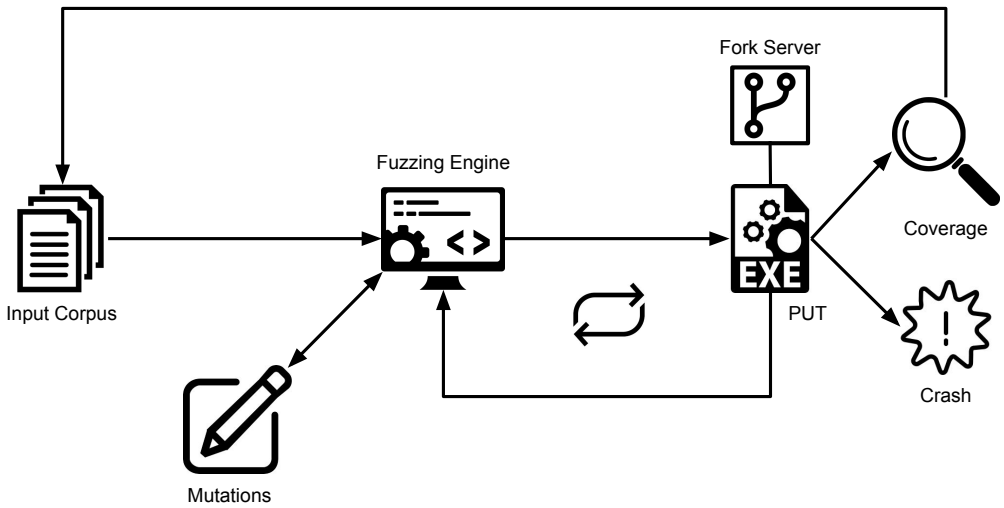

Figure 1.6. General design of a fuzzing framework.

1.5.2 Fuzzing libraries

Fuzzing libraries has a slightly different approach, due to the absence of a *main* function. Here the analyst must write a fuzzing harness (or stub) that reads the input generated by the fuzzer

and call the function following its API. It follows a *unit test* approach. The driver differs based on the target function, and it requires manual work from the analyst. Most library interfaces check parameters and states right at the beginning, therefore the driver must be carefully constructed to feed a valid input. In *libFuzzer* a function called `LLVMFuzzerTestOneInput` acts as the driver.

Following, we present a complete and highly extensible black-box fuzzing framework to target the native components of Android applications. It tries to address each of the proposed problems, by integrating already existing techniques into an Android device, together with newly added components. Then, we test the framework on some real-world open-source applications to understand its effectiveness and its limitations, and we finally move on to testing some public closed-source applications to discover new bugs.

Chapter 2

Previous Effort

2.1 Related Work

Static and dynamic software testing techniques have caught the attention of a high number of researchers and computer engineers. In vetting android applications, several different techniques have been developed over the years, each targeting a different component. While there is a great number of works that target the Java side of each application, either statically [16, 17, 18, 19, 20] or dynamically [21, 22, 23], few have been developed to consider the native components. None of the Java state-of-the-art analysis frameworks support cross-language analysis. When reaching a native function call, all tools apply a conservative model and ignore any side-effect correlated to it, leading to high imprecision in the analysis results. This result in *unsoundness*.

The works trying to address the problem perform mostly data-flow static analysis, considering that data can change "context" multiple times. For example, *StubDroid* [24] automatically infers library models for taint-analysis, using only their binary distributions, so that tools like *FlowDroid* do not have to re-analyze the library for each app. It is tested on the Android runtime library and results in a performance improvement of over 90% compared to using the static tool alone. Another interesting work is presented by Fourtounis et al [25]. Their approach manages to track and recover all calls to Java methods performed inside the native code through the JNI interface pointer, in large Android and Java applications. They do so via binary scanning, searching in all native libraries for constant strings matching Java methods name and Java VM types. Once constructed a set, a tool named *Native-Scanner* follows their propagation and identifies all entry points to Java code from native code without fully tracking all native calls (e.g. call-graph edges). The main drawbacks highlighted are that the tool is not specifically built for Android, so it misses all Android framework APIs calls, and has a hard time identifying obfuscated strings. To solve the first, Samhi et al. [26] improved it by implementing a framework called *NativeDiscloser* as part of a bigger tool, *JuCify*. Overall its goal is to move towards a unified static analysis tool able to perform precise cross-language analysis for all code in Android applications. It merges extracted call graphs from both sides of the application to construct a final model usable by common Android analysis frameworks. The de-obfuscation part instead has been addressed by Kan et al. [27]

2.2 JniFuzzer

Considering our main goal, which is constructing a framework capable of fuzzing Android native libraries starting from each native Java entry point, the most related previous effort comes from the work by Claudio Rizzo carried out at the University of London, with their framework JniFuzzer [28]. We present its design, its use in real-world applications, and the results generated, as a valid, although limited, alternative to our approach. To the best of our knowledge, this is the only other previous effort on the subject.

2.2.1 Design

When a programmer defines a native method on the Java side, it writes its implementation in the native language. After compilation, its symbols are exported and visible to the Android Runtime so that linking against it is possible dynamically, precisely when the runtime invokes `system.loadLibrary()`. Considering that each symbol is exported and accessible by any program loading the library, there is no need to simulate the Java user interface as it is possible to directly invoke the method. Invocation happens exactly as when using any shared library in C or C++. With this in mind, the framework consists of 3 components, as schematized in Figure 2.1.

Mock the JNI Environment

The native side of a real-word native application must both be able to handle and convert arguments to their native counterparts and interact directly with objects, classes, and methods of the Java side. As seen in subsection 1.4.2, this is managed by the *JNINativeInterface* interface pointer, with all function pointers populated at during initialization by the ART. If a native method is invoked by passing an invalid `JNIEnv` pointer, it crashes as soon as it encounters a portion of code requiring it. To solve this problem, the solution they propose is to mock a subset of the JNI functions with a custom implementation, so that their invocations will not crash the target. To avoid mocking the total of 229 of these, they analyzed a few of the target apps to determine the most used ones following a trial-and-error approach and limited their tool to target native functions using only this subset. To feed the target with each mocked implementation, a custom `JNIEnv` structure is required, with interface pointers pointing to the custom implementation. Similarly, for data types, a mocked version is provided to the target, and again only a subset of all available data types are implemented (in this case only `java.lang.String` and integer).

Function Pointer Extraction

As presented in subsection 1.4.3, there are 2 ways a developer can register native methods: either the VM is responsible for the creation of each function prototype on the native side, using a pre-defined pattern as a naming convention, or the developer implement a function called `JNI_OnLoad` to register them dynamically when loading the library. For the first case, the native symbol is extracted using the pattern-generated name as it is (e.g performing a symbol lookup operation with `dlsym`). For the second case instead, the only possible way is to provide a mock implementation of the `RegisterNatives` JNI function so that, when invoked by `JNI_OnLoad`, provides the pointer directly.

Target Execution

An executor component fetches the target function pointers from the extractor, reads the input generated by any fuzzer (e.g AFL), and passes it as input to the target function after conversion. Only one function must be implemented for each fuzzing campaign, and it is the one in charge of reading the input from the fuzzer. The inputs available are limited by the implemented mocked data types.

2.2.2 Evaluation Results

Using the apps from *Androzoo* as a test set, they were able to test the framework on 4171 APKs, each containing native methods, on different Android emulators. Considering that JniFuzzer can handle only API calls with a mocked implementation, this resulted in overall 69 native methods of which only one with both an integer and a string as a parameter, while the rest with integers only. The methods were contained in 32 different applications. The reports showed a total of 3 bugs, consisting of an arbitrary read, an arbitrary write, and a stack-based buffer overflow.

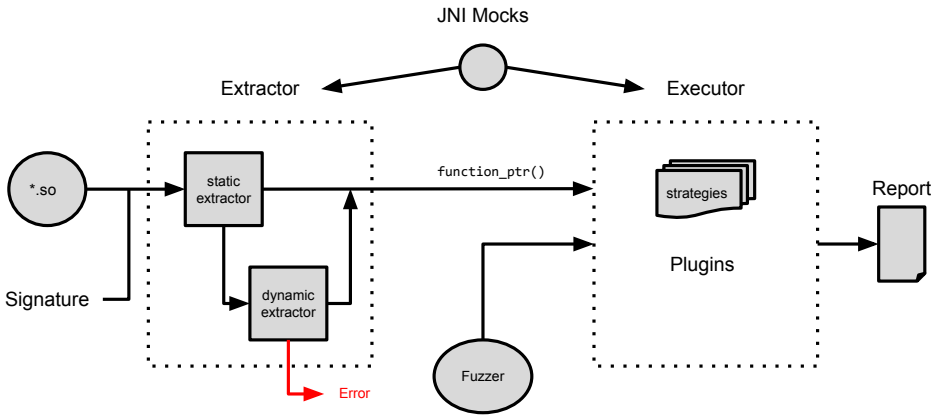

Figure 2.1. Design of JniFuzzer framework.

2.2.3 Limitations

As it is the first approach to fuzzing native components, it is subject to several limitations. We list a few of them:

- *Stateful fuzzing*: the native method is called without any build-up state, limiting the scalability of the framework to functions either performing some initialization or able to execute alone;
- *Triggerable bugs at user level*: even if there is a bug in native code, it might be not reachable from a user-controlled input in the Java application. This limits the importance of the discovered bug;
- *JNI environment mocks*: testable targets are limited by the JNI API used. To be able to fuzz any target, one would either need to implement the mock of each API function (229) or write an automatic stub generator;
- *Guided fuzzing*: fuzzing is performed in a black-box fashion, due to the lack of source code. For AFL, this translates to *dummy mode*, in which input is generated randomly without any coverage information, starting only from a set of seeds. Binary rewriting techniques or hardware feedback mechanisms (as AFL++ Coresight [29]) can solve this problem;

Chapter 3

Problem Evaluation and Design

3.1 JNI Functions Problem Evaluation

In fuzz testing, the binary under test should run without crashes when executed with valid input. Considering Android applications and their native methods, they all contain the `JNIEnv` interface pointer as the first argument. It allows the use of additional functions provided by the runtime environment to enable interoperability between the two worlds. To be able to execute such a target during fuzz testing, we must guarantee that each target function uses a valid interface pointer. When an invalid interface pointer is provided to a target function using it, the function terminates with a segmentation fault the first time it is dereferenced and used. To understand its usage frequency among Android applications, we statically analyzed a set of open-source applications provided by F-droid. The set comprehends a total of 3832 applications, not counting versions (∼250GB). The results of source code analysis using both python and bash are shown in Table 3.1. As we can see from this initial analysis, out of all F-droid applications only 340 of them contained native code and all used at least once the interface pointer. It is therefore impossible to consider fuzzing without first addressing this problem. The most recurrent ones are functions dealing with strings and Java methods. While it would be trivial to provide a custom implementation of a function like `GetStringUTFChars`, that simply returns an array of bytes representing a Java string, functions like `FindClass` or `CallVoidMethod` would be respectively difficult or impossible to mock retaining the original functionality, because the first one would require to mock a Java class, while the second to transfer execution to a Java method.

3.2 Considered solutions

The *JniFuzzer* framework addresses this problem by implementing a mocked version of the most used JNI functions, when possible. Even if we decided to go in the direction of building on such previous effort, this solution would still be both incomplete and imprecise. Mocking all JVM functionalities is practically impossible while retaining the original precision. We now present other possible solutions we considered for this problem, ranking them by feasibility.

3.2.1 ART Reuse

The core idea behind this solution is the need to have for each fuzzing execution a valid VM to provide as state, capable of executing such JNI calls. Considering what was introduced in subsection 1.1.1, during application start-up the *Zygote* process forks and provides to each application an initialized VM. With this in mind, the fuzzing harness would need to act somewhat like the *Zygote* process. It would need to start and load the VM, and then fork its state for every fuzzing execution. Considering that Android applications are highly multi-threaded, this does not come with ease. Each Android application is started with the main thread, the *UI thread*, responsible

JNI Method Name	*# Apps Using It*
GetStringUTFChars	238
FindClass	231
NewStringUTF	207
ReleaseStringUTFChars	201
GetMethodID	186
GetArrayLength	179
DeleteLocalRef	167
GetFieldID	145
NewGlobalRef	143
GetObjectClass	137
NewObject	127
GetStaticMethodID	123
NewByteArray	118
CallObjectMethod	116
DeleteGlobalRef	113
NewIntArray	113
CallVoidMethod	111
SetObjectArrayElement	110
NewObjectArray	109
Total Apps	340

Table 3.1. Usage frequency of JNI functions in Android applications

for everything happening on screen. To avoid using it for expensive or asynchronous tasks, it often delegates work to worker threads preventing the *Application Not Responding* dialog. The different threads classes available are:

- *AsyncTask*: helps get work on/off the UI thread;
- *HandlerThread*: deals with callbacks;
- *ThreadPoolExecutor*: manages the creation of groups of threads to efficiently deal with parallel tasks;
- *IntentService*: service intent requests.

The number of threads is not fixed and depends on the application, but it is always greater than 1. For a simple application such as *HelloJNI-Callback* [30] 11 or 12 threads start each run, depending on the number of binder threads. The fork system call when invoked in a program with multiple threads clones only the calling thread. The reason behind this design choice is to prevent problems derived from changes to the program's persistent state from threads not aware of each other.

To test this solution, we first tried to simply fork a toy application state (HelloJNI-callback) from its native side, then moved to fork a VM loaded manually by a native program. In the first case, the parent process proceeded without any issues with the regular control flow, while the child process was only able to execute native code. For instance, calls such as `CallVoidMethod` or return instructions, requiring the JIT compiler to compile Java code on the fly, were not able to execute as the JIT pool worker thread in charge of it was not present. For the second case instead, a VM loaded manually always starts 11 threads, but of which only 1 performs all tasks, the main thread. The other threads started but left sleeping are: *Jit thread pool*, 4 *runtime workers*, *signal catcher*, *HeapTaskDaemon*, *ReferenceQueueD*, *FinalizerDaemon* and *FinalizerWatchd*. Forking its state resulted in only the main thread propagating to the child process, as expected, but without losing any program functionality.

This first solution is the one fully developed and integrated with AFL++. Its design and components are presented in detail in section 3.5.

3.2.2 Mocks with caching strategy

The key insight of this second considered solution is the fact that fuzzers such as AFL++ apply only very small modifications to each input fed to the target. They apply bit-flips, slicing, randomization, and other transformations that end up modifying only a small portion of bytes. Therefore, we can suppose that between two runs a high number of calls to JNI methods have as input and output similar, or even equal, values. Storing such values in a cache DB improves performance, as it theoretically works without the constant need of a JVM running in the background fulfilling requests, after a first stabilization period.

We now present a possible design for such a solution. Each target native function is provided with a mocked JNI environment as the first argument. Each mocked JNI function first performs a cache lookup using the inputs arguments received with the JNI call. If there is a cache hit, it returns the output provided by the DB lookup. If rather, it results in a cache miss, each mocked method starts the JVM (as presented in subsection 3.5.1) and uses the execution's output of the real JNI method to create a new cache entry. Then, it returns such entry to the calling program. It is evident that for each fuzzing campaign on one or multiple targets the cache DB should be first populated with valid entries. It, therefore, needs an initial stabilization period showing poor performance. When fully populated instead we expect a high number of cache hits, drastically increasing performance. One problem that should be considered as well is the low number of cache hits when dealing with variable data.

This solution was considered as a backup solution, to be implemented only in case the ART reuse one did not work. As instead, we were able to fully implement the ART use, it can be seen as an optimization upgrade in the case the ART reuse solution shows scarce performance.

3.3 AFL++ In Android

Even if Android is built on top of the Linux kernel, it shares little in common with typical Linux distributions. Android uses libraries specifically built for it, it is not as customizable and requires a rooted device to acquire root privileges. The main problem when porting large software to Android, such as AFL++, is the presence of the *Bionic* standard C library instead of the classical *GNU C Library* (*glibc*). It was designed to best suit devices with less memory and processor power as Android ones, and it does not implement all of C11 and POSIX functions. About 70 are missing (as of *Android Oreo*) [31], causing the linker to be unable to find several symbols during compilation.

As a consequence, AFL++ does not compile out of the box in Android. To port it to Android, we had to apply a set of patches to the AFL++ source code. Each one of them, with the required prerequisites, is listed in subsection 3.7.1. AFL++ then works with *clang-13* only and instrument code in *CLASSIC* mode.

3.4 Native Methods Extractor

The first step to automatically test applications is to extract their native methods. This helps us filter our unwanted applications, manually inspect possible vulnerable functions (given their name), and inspect popular signatures of such functions to automate the testing process. As we showed before, an Android application does not always have a native component, and when it does the name of its functions can vary. Consequently, we can not inspect exclusively the native side. We integrated two existing tools to inspect the Java side of a large set of Android APK, to extract useful information. The design is reported in Figure 3.1.

It first extracts the `lib/` folder present in APKs with native components. It contains all shared libraries, and it is present only if the application uses them. Then, the extractor uses both Jadx [32] and QDox [33]. Jadx is a Dex to Java decompiler, capable of retrieving the entire app source code from an APK file while retaining precision. It is necessary to have the extractor works with QDox, a high-speed, small footprint parser capable of fully extracting class, interface,

and method definitions. It is provided as a Java library (*jar* file) and we use it to extract for each method declared as native its signature and all parameters needed to construct its pattern-based name: the package name, the class name, and the method name. Only the native declaration is extracted, not the method's use. As Jadx can not decompile 100% of the code, QDox occurs in some errors for certain decompiled applications. This is minor, as it rarely occurs and the probability of finding an error in the same Java file where a native method is declared is low (as shown in section 4.4). To deal with it, we ignore files with decompilation errors.

It produces as output both a list of native methods for each analyzed application together with its signature, and a sorted frequency list of signatures. The latter is used by the following tools to automate the fuzzing procedure.

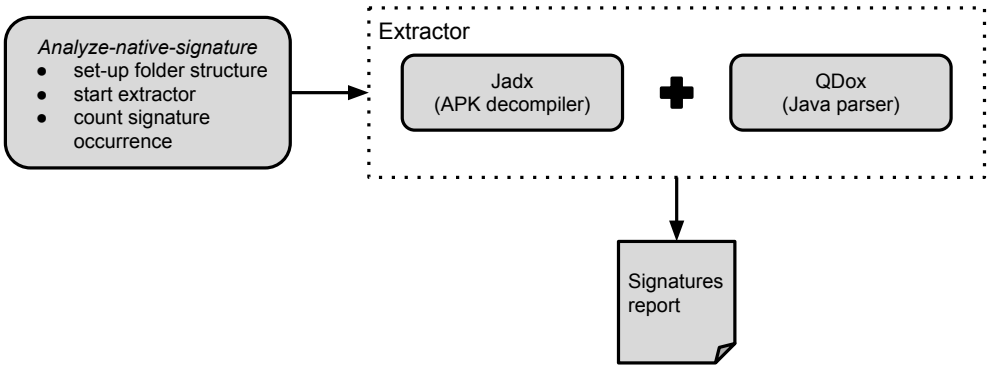

Figure 3.1. Native method extractor design and components.

3.5 Harness Design

As explained in subsection 1.5.2, to fuzz test libraries the analyst first prepares a fuzzing driver, or *harness*, built specifically for the target function. The harness is in charge of setting up a valid environment for the target function to run, reading inputs provided by the fuzzing engine, and moving execution to the target function with those as parameters. These steps are still valid when fuzzing Android native components. In this section, we present the design and components of a working harness stub, crafted specifically for our targets. In short, it:

- automatically loads the ART, to guarantee JNI functionalities;
- fetches the target function pointer by loading the native library, to call during fuzzing;
- sets up a deferred fork server.

From the point of view of the analyst instead, it is only required to add a small harness driver to define the targeted function signature, read the needed parameters from `stdin` or file and call the target function. The overall design is shown in detail in Figure 3.2.

3.5.1 Load ART

JNI provides a rich set of APIs, some of which allow native code to create and interact with a Java program, more precisely a JVM. This specific part of the JNI API is called *Invocation API*. The JNI function `JNI_CreateJavaVM` (contained in `libart.so`) loads and initializes a Java VM and returns a pointer to the JNI environment interface pointers, `JNIEnv`. It takes three parameters: the first two are pointers to such interface pointers, while the third one is a structure called `JavaVMInitArgs` that provides arbitrary VM start-up options. In it, we can specify where the VM will look for both the APK file and the native libraries that compose the application (in

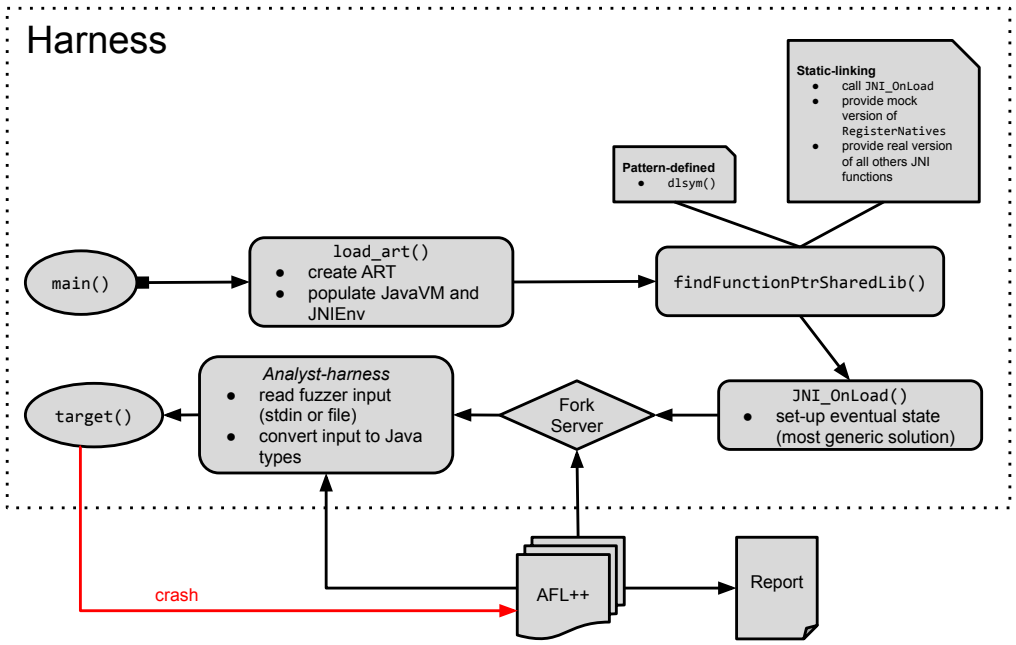

Figure 3.2. Harness design and components.

Figure 3.3 at lines 7 and 8). All other options are not important to the understanding of the design.

Then, we fetch its symbol with a library lookup using function `dlsym` and we call it. Before returning to the harness itself, the ART also needs to have registered some framework native methods used internally. We can do so by calling function `registerFrameworkNatives` inside `libandroid_runtime.so`. Inside there is a call to JNI function `RegisterNatives`, exactly as we would do when dynamically linking native methods in Java. The complete portion of C++ code, without error checking, is presented in Figure 3.3. In lines 4 and 5, *targetAppPath* is the path to the folder containing the application under test, composed of an APK file and the *lib* directory with all native libraries. In Figure 3.2, it is implemented by the component *load_art*.

3.5.2 Function Pointer Extraction

To deal with function pointer extraction, it is important to notice that all functions implementing Java native methods are exported by a shared library. Exporting has the advantage of preventing usage of undocumented API, such as symbols not exported, it reduces the risk of symbol collision with other libraries and decreases the program's startup time due to a lower number of symbols needed to be processed by the dynamic loader. Considering our objective, it allows fetching symbols through a simple library look-up when its full name is known. Two cases should be considered, depending on the convention followed to register such native methods (introduced in subsection 1.4.3):

- *Defined pattern*: when the native side function is named following a defined structure (*Java_com_package_class_method*), opening and calling `dlsym` on the target libraries provides us with a valid function pointer;
- *Dynamic linking*: when instead the native side follows a dynamic naming scheme, through `JNI_OnLoad`, `RegisterNatives` and the `JNINativeMethod` struct, we can not construct a name based on the Java components and perform the symbol look-up. Luckily, the structure `JNINativeMethod` contains as hard-coded strings Java side methods names, together with their native function pointers. To retrieve a single one, we provide to `JNI_OnLoad`, as first argument, a fake `JavaVM *`, with all original Java VM functions except the `GetEnv` one, of

```
1       static auto load_art() -> std::pair<JavaVM *, JNIEnv *>
2       {
3               // Set-up required arguments
4               std::string apk_path = "-Djava.class.path=" + targetAppPath +
                    "/base.apk";
5               std::string lib_path = "-Djava.library.path=" + targetAppPath
                    + "/lib/arm64-v8a";
6               JavaVMOption opt[] = {
7                       { apk_path.c_str(), nullptr},
8                       { lib_path.c_str(), nullptr}
9               };
10              JavaVMInitArgs args = {
11                      JNI_VERSION_1_6,
12                      std::size(opt),
13                      opt,
14                      JNI_FALSE
15              };
16
17              // Open shared libraries and fetch symbols
18              void * libart = dlopen("libart.so", RTLD_NOW);
19
20              void * libandroidruntime = dlopen("libandroid_runtime.so",
                    RTLD_NOW);
21
22              auto JNI_CreateJavaVM = (JNI_CreateJavaVM_t *)dlsym(libart,
                    "JNI_CreateJavaVM");
23
24              auto registerNatives = (registerNatives_t
                    *)dlsym(libandroidruntime, "registerFrameworkNatives");
25
26              // Create JVM
27              std::pair<JavaVM *, JNIEnv *> ret;
28              int res = JNI_CreateJavaVM(&ret.first, &ret.second, &args);
29              auto [vm_tmp, env_tmp] = ret;
30
31              // Register defaults native methods
32              jint res1 = registerNatives(env_tmp, 0);
33
34              return ret;
35      }
```

Figure 3.3. Native C++ code to create and load ART.

which we provide a mocked version, called `GetEnv_fake`. Inside it, we again build a fake JNIEnv structure, with all valid JNIEnv functions except the `RegisterNatives` one. This last function instead, when called by the real `JNI_OnLoad` only via the `JNIEnv` structure, intercepts the call to the real `RegisterNatives` and searches in the array of structures for a method with a name matching. When it finds one it saves the corresponding function pointer in a global variable for later usage when fuzzing. For example, if the harness needs to find a method with a defined pattern name as *Java_com_name_jni_package_JNI_nativeMethod*, and this exact string used in the `dlsym` look-up does not produce any result, then it must look for an entry of structure `JNINativeMethod` with *nativeMethod* as name when intercepting the `RegisterNatives` call.

In Figure 3.2, everything is implemented by the component *findFunctionPtrSharedLib*.

To have the most generic solution possible, without considering the potential state generated by the Java side, we call, if present, `JNI_OnLoad` from the target function shared library. This is useful as it happens that some native libraries generate an initial native state as the first thing when the library is loaded. We are fuzzing native methods that might depend on such state, so it is best to have the call to `JNI_OnLoad` to set it up.

3.5.3 Deferred Fork Server

Initializing the JavaVM and fetching the target function pointer is time-consuming. On a Google Pixel 4 device, it takes about 0.37 seconds to perform both tasks. To avoid repeating those steps for each fuzzing execution, we specify to AFL++ to set up a deferred fork server. It is accomplished by calling `AFL_INIT` at the right position in the harness, after the initialization phase but before any operation dealing with fuzzer inputs, so that the fork system call is called at this exact position. AFL++ then handles automatically the rest.

Using the fork server with the ART, we are following the solution introduced in subsection 3.2.1. When we create and load the ART for the first time, around 11 or 12 threads start. As soon as we fork the process after the first target execution, only the main thread remains alive. The main thread is in charge of performing all JNI-related work, and it is capable of executing Java calls from the native side. This is exactly what is needed to guarantee that the fuzzer will not simply crash when encountering a particular JNI call, which is the case when using partial mocks instead of the real JVM/ART.

3.5.4 Analyst Harness

Lastly, the analyst is required to manually add a small portion of code which depends on the target function signature and the strategy chosen for fuzzing it. It is, accordingly, easier to implement it manually, and it is divided into two parts:

- *function signature*: defines in the harness the signature of the function under test. This is done by declaring a global type, used later to save the fetched target function pointer;
- *AFL++ input*: implements a small portion of code that initializes, reads, and converts to JNI types the input generated by the fuzzer, after the deferred fork server. Then, such input is passed when calling the target function.

Considering that all native functions take as a second parameter a valid Java object, of the class defining the native method, we allocate it by using its class and the JNI. The name of the class is retrieved from the target function pattern defined name.

3.6 Framework Design

Launching a fuzzing campaign on a single native method requires following a set of steps. These include fetching the application and the library holding the target function implementation, setting up the fuzzing environment and its start-up. We automate every step as much as possible. Then, we address the problem of having low execution per second using the harness, by exploiting AFL++ in parallel mode. It allows running up to N concurrent fuzzing jobs with virtually no performance hit, with N being at most the number of cores available on the device. Finally, we port the automated solution to a phone cluster, with everything handled by a fuzzing manager. With the fuzzing manager, we can run N parallel fuzzing campaigns on each device, targeting in a single run multiple native functions with the same signature.

The design of the complete framework is reported in Figure 3.5. It is composed of 3 tools: a fuzzing driver based on target functions signature called *fuzzing_driver*, a fuzzing driver based on target function name called *fuzzing_one* and a *fuzzing_manager* handling both tools on a phone cluster.

Fuzzing_driver

It fuzzes for a given amount of time each native function contained in a target APK folder with a specific signature. It supports input from AFL++ either from `stdin` or file and can run parallel fuzzing campaigns on a specified number of cores. To provide to the harness the triplet Java function name, application's name, and native library name containing the function, it performs an extra step. The first two are given by the signature file generated by the Native Method Extractor, while the library name is retrieved either by looking at its symbols, for defined-pattern methods, or its strings, for dynamic-linking methods. When it launches AFL++ on multiple cores, it must detach as soon as a single process is started. For this, we use the trailing ampersand (`&`), which forks and runs the command asynchronously in a separate sub-shell. To terminate a process after an amount of time the `timeout` command. In line 14 instead, it forces the process running the script to wait for the termination of all started processes. In its absence, the script would not respect the number of cores provided, and it will try to start as many fuzzing campaigns as possible, often crashing the device. Its pseudo-code is reported in Figure 3.4.

```
1         Compile harness with afl-clang++
2         Extract methods with a certain signature
3         FOR all extracted methods:
4                 Derivate the corresponding app name and native library
5                 IF selected #cores > 1:
6                         FOR 1 to selected #cores:
7                                 IF read from file then
8                                         Launch AFL++ for an amount of time in
                                              master-slave mode (with @@)
9                                 ELSE
10                                        Launch AFL++ for an amount of time in
                                              master-slave mode
11                        Wait for all processes to finish
12                ELSE
13                        IF read from file then
14                                Launch AFL++ for an amount of time (with @@)
15                        ELSE
16                                Launch AFL++ for an amount of time
```

Figure 3.4. Fuzzing_driver pseudo-code.

Fuzzing_one

Similarly, it fuzzes for a given amount of time a single native method contained in a target APK folder given its pattern-based name. The pseudocode is as the one shown in Figure 3.4, with the only variation that it does not loop for all methods, and it searches for a single method in the same set of signature files generated by the *Native Methods Extractor*. Similarly, it retrieves the function's name, the application's name, and the library's name as in the *fuzzing_driver*.

Fuzzing_manager

This is a python script instead, managing all operations to port both previous tools on a phone cluster. It does so via ADB commands, and the use of a custom ADB python library. It has 4 functionalities implemented:

- *fuzz_signature*: it starts the *fuzzing_driver* script on each phone, after having transferred the entire framework on it and having set the CPU frequency scaling to `PERFORMANCE`. It

uses the `nohup` command to detach from the process on one phone and move to the next one. Once started on 1 core, there are 3 sets of processes running: one for the *fuzz_signature* script, another one for AFL++, and the last one for the harness itself;

- *fuzz_one*: similarly, it starts the *fuzzing_one* script on each phone, after setting the same initial conditions.
- *check*: it performs an intermediate or final check on all fuzzing campaigns by pulling from each device the entire output directory. It then extracts fuzzing statistics;
- *kill_fuzzer*: it terminates all processes related to a fuzzing campaign, for each device, using their PIDs.

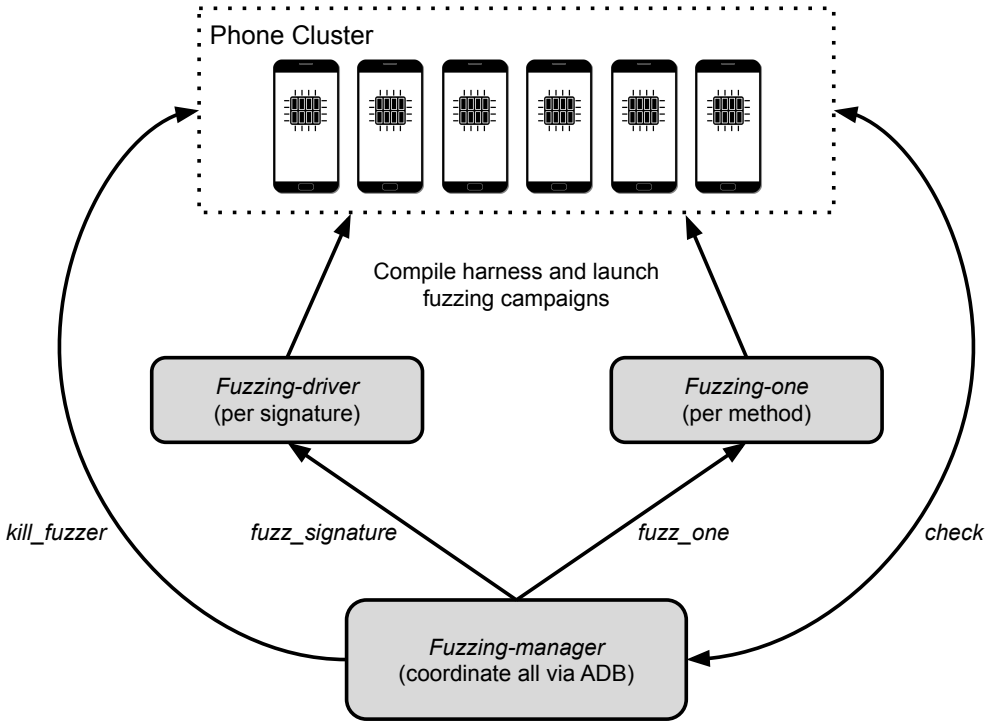

Figure 3.5. Fuzzing framework design and components.

.

3.7 Manual Set-Up and Usage

In the following section, we list the steps required to start fuzzing an android APK native method. The overall set of tools groups down to 4, mainly:

- *patched AFL++*: AFL++ with patches required to start on an Android device, because of multiple missing dependencies;
- *NativeExtractor*: the tool performs an initial APK analysis. It extracts and analyzes all Java method with a native implementation and count the occurrence of each method's signature to then fuzz them in an automated fashion;
- *Harness*: it is used together with AFL++ to fuzz a single native method. It provides the soundest result when fuzzing a method considering its API specifications;
- *AndroidNativeFuzzingFramework*: complete framework porting parallel fuzzing of Android applications native methods on multiple devices, based either on a single method's name or on multiple methods sharing the same signature.

3.7.1 AFL++ on Android

We now present the steps required to port AFL++ to Android, with the needed source-code patches. As our framework will not need to instrument any program except for a small harness, which we are not interested in testing, the AFL++ build presented instruments only in *CLASSIC* mode.

Prerequisites

Firstly, the Android device must be rooted. This is required to access and preload libraries present in system-level directories (e.g `apex/` or `system/`), and then later use all computational power of the device for short-lived processes as the ones fuzzing campaigns usually start. Secondly, a set of packages are needed by the compilation tool-chain. Termux application is not required, but it facilitates access to an Android shell and offers a patched and recompiled version of all common Linux tools for Android (e.g `grep`, `file`, `readelf` and many others). As an alternative everything can be done over *ADB*. The needed packages are *make*, *libandroid-shmem*, which emulates SYSV shared memory on top of *ashmem* shared memory system, and *ndk-sysroot* for unified headers. Lastly, *clang* as C compiler. At the moment of the project Termux only provided *clang-v13*, and we only focused on building AFL++ with it. Later on, Termux updated it to *clang-v14*, and given that Termux does not keep packages version history, *Clang-v13* should be installed from the corresponding `.deb` files, together with version 13 of *libcompiler-rt*, *libllvm*, *lld* and *llvm*. It is worth noticing that *GCC* is deprecated in Android NDK and, although few un-official versions exist, *clang* is preferred.

Patches

Patches should be applied to AFL++ source code, and are relative to the release 4.00c (of January 26th, 2022). The list of patches is as follows:

- *POSIX compliance issue*: in `src/afl-ld-lto.c:148`, deprecated POSIX function `index()` should be substituted with its counterpart `strchr()`, as it is missing from the *Bionic libc*.
- *LLVM symbols*: symbols present in `libLLVM-13.so` shared library are needed by `afl-cc` during execution. Without them, when testing the AFL++ compiler compilation stops due to it being unable to open `split-switches-pass.so`, `compare-transform-pass.so` and `split-compares-pass.so` because of missing symbols. Simply preloading the shared library, by setting the `LD_PRELOAD` environmental variable to its path, solves the problem. When clang is installed following the prerequisites, the library is located under `/data/data/com.termux/files/usr/lib/`;
- *afl-cc symbols*: AFL++ compiler `afl-cc` is unable to find symbols present in object file `afl-compiler-rt.o`, required to run. It is solved by manually adding the path to the file, contained in AFL++ main directory, as a parameter passed to the compiler invocation using `cc_params[]` inside its source file, `src/afl-cc.c`;
- *MMAP missing symbols*: `GNUmakefile:299` checks whether or not the target device can use *shm* or *mmap* to handle shared memory operations. It creates and executes a simple C program, calling both `shmget` and `shmctl`, and based on the result decides which one to use. The *Bionic libc* does not provide `shm_open` and `shm_unlink` symbols, used to create, open and remove/unlink POSIX shared memory objects. The program run in Android returns 0 even if symbols are missing from MMAP, causing *symbol's missing* exception during the linking phase. Changing the comparison from 1 to 0, resulting in *shm* usage, solves the problem. For Samsung A40 devices instead, we had to force *shm* usage also in `instrumentation/afl-compiler-rt.o.c`, by conditionally compiling only *shmat* operations when in Android. This is the most general solution to still guarantee compatibility in non-Android machines;

- *Compilation test*: as we said before, our patches make AFL++ target instrumentation works in CLASSIC-mode only. Even if not strictly required, to pass the instrumentation test automatically performed by AFL++ after compilation, we should add in `GNUmakefile.llvm:459` the compilation flag *–afl-CLASSIC*;

While running AFL++ on android together with ASAN, we discovered a bug in the implementation of AFL++ specifically for Android. It resulted in a heap-buffer overflow preventing the regular AFL++ startup, due to a wrong string length value passed to `memchr` in `src/afl-fuzz-stats.c:62`.

Then, by simply running `make` the compilation process terminates correctly.

3.7.2 NativeExtractor

Extracting and analyzing native methods and their signatures are the first steps towards fuzzing them. There are two requirements. The first is to have inside a folder `target_APK/` a folder for each app, containing a `base.apk` file. The second instead, is to have the Java decompiler *jadx* built from source (running `./gradlew dist` inside its source directory). Then to use, run:

```
$ ./analyze_native_signatures.sh qdox
```

It groups by and counts the number of methods per signature, and it generates for each app a file named `signatures_pattern.txt` containing a list of the native methods present inside each APK with their pattern-based name, with its extracted signature. It also creates a `/lib/arm64-v8a` folder with all native libraries inside. Both the file and the folder are needed by the *Android-NativeFuzzingFramework* when fuzzing a single method or per signature, therefore the extractor should be run once before performing any real fuzzing analysis on a set of target applications. As example, a portion of the file `signatures_pattern.txt` for an antivirus app is (with format `JavaName return_type:arg1_type,arg2_type,...`):

```
Java_com_trustlook_sdk_offlinescan_OfflineScanner_offlineScan
    int:String,String,
Java_com_tencent_mmkv_MMKV_reKey boolean:String,
Java_com_tencent_mmkv_MMKV_removeValuesForKeys void:String[],
Java_com_tencent_mmkv_MMKV_trim void:
```

3.7.3 Harness

The harness is used to fuzz a single native method. Before starting the fuzzing campaign, the analyst should write the fuzzing driver itself. It is in charge of reading the input generated by the fuzzer, converting it to Java types, and providing it to the target function, after having defined the target function type. In Figure 3.6 we report one of the harness used to test a GIF library in a messaging application. The target function, called *Java_pl_droidsonroids_gif_GifInfoHandle_openByteArray*, loads a GIF image from a Java byte-array. The harness reads the GIF file into a buffer and converts it into a Java byte-array. Then, it provides the Java byte-array `jbyteArray` to the native function.

After creating a valid harness, we can compile it and start fuzzing. As a prerequisite, the target application should have both a `base.apk` file and a library folder of the form `/lib/arm64-v8a`, often contained in the same directory, the harness as a c++ file, AFL++ compiled with the required patches for Android and `afl-clang++` location set in `PATH`. To compile the harness with *–afl-CLASSIC* instrumentation, we first preload symbols from LLVM-13 required by AFL++, then we compile. The sequence of commands is the following:

```
$ export LD_PRELOAD="/path/to/libLLVM-13.so"
$ afl-clang++ --afl-classic -Wall -std=c++17 -Wl,--export-dynamic harness.cpp
    -o harness
```

Then we launch the fuzzing campaign. The harness requires a few command line parameters to understand what to fuzz and where to locate the necessary files. It needs specified:

- the *target app path*;
- the *library name* containing the target function;
- the *target function name*, as it would be in the case it is defined following the defined pattern approach;
- an optional *file path* containing a valid input to feed to the target. This is optional as AFL++ can provide input through `stdin`. When instead it provides it through a file, this last parameter is substituted with @@, and AFL++ is in charge of replacing it with the correct file name.

We set `LD_LIBRARY_PATH` to the location of libraries required by AFL++, for example `libart.so` and the target library, and `LD_PRELOAD` to the C++ shared runtime. Finally, to launch the fuzzing campaign:

```
$ export LD_PRELOAD="/path/to/libc++_shared.so"
$ export LD_LIBRARY_PATH="/apex/com.android.art/lib64:/path/to/target_app/lib/
arm64-v8a:/system/lib64"

$ afl-fuzz -i <input_dir> -o <output_dir> -- ./harness <path/to/target_app>
    <lib_name> <target_name> [@@]
```

3.7.4 AndroidNativeFuzzingFramework

In section 3.6 we presented the design of the fuzzing framework. It supports parallel fuzzing campaigns using a single device based on either method's name or signature, or a phone cluster. The names of the scripts handling each case are respectively `fuzzing_one.sh`, `fuzzing_driver.sh` and `fuzzing_manager.py`. We go into usage details for each one of them.

Fuzzing_one.sh

It fuzzes a given native method for a certain amount of time on a specified number of cores in parallel. To invoke it, run:

```
$ ./fuzzing_one.sh <method-chosen> <time-to-fuzz> <input-dir> <output-dir>
    <read-from-file[0|1]> <AFL_DEBUG[0|1]> <parallel-fuzzing[0|N]>
```

The available parameters, with a similar description in the *help* manual, are:

- *method-chosen*: method chosen from `analyze_native_signatures.sh` script, as fuzzing target;
- *time-to-fuzz*: time to fuzz each method for, as float[s|m|h|d] (s=seconds, m=minutes, h=hours, d=days);
- *input-dir*: fuzzing input directory name, populated with meaningful seeds (same as required by regular AFL++);
- *output-dir*: fuzzing output directory name, it will be populated automatically by the framework depending on the parameters used;
- *read-from-file*: flag to specify if harness will read from file (1) or from stdin (0) the input generated by AFL++;
- *AFL_DEBUG*: set to use AFL++ in debug mode;
- *parallel-fuzzing*: specify the number N of cores to use for a parallel fuzzing campaign (if N is greater than the maximum number of available cores, then the maximum is used).

```
1         #include <jni.h>
2
3         /* Target function type definition */
4         typedef jlong function_t(JNIEnv *, jclass __unused, jbyteArray);
5
6         /* Harness main function */
7         int main(int argc, char *argv[]) {
8                 ...
9                 // ART startup (JNIEnv *, JavaVM *), function pointer
                      extraction and deferred fork server
10                ...
11
12                // Variable definitions
13                jbyteArray byteArray;
14                jclass MainActivityCls;
15
16                // Read bytes from GIF input file and save them into a buffer
17                std::ifstream fileStream(argv[2]);
18                fileStream.seekg(0, std::ios::end);
19                fileLen = fileStream.tellg();
20                fileBuf = new char[fileLen];
21                fileStream.seekg(0, std::ios::beg);
22                fileStream.read(fileBuf, fileLen);
23                fileStream.close();
24
25
26                // Convert into jbyteArray
27                byteArray = env->NewByteArray(fileLen);
28                env->SetByteArrayRegion(byteArray, 0, fileLen, (const jbyte *)
                      fileBuf);
29
30                // Call target function
31                info = targetFunctionPtr(env, MainActivityCls, byteArray);
32
33        }
```

Figure 3.6. Manual harness example for a GIF loading native method.

Fuzzing_driver.sh

Similarly, it takes as parameter a function signature as the ones generated by `analyze_native_signatures.sh`, it searches inside the `target_APK/` directory for methods having the same signature, and it fuzz for a certain amount of time each one of them, on a maximum of N cores (N being the maximum number of cores available on the device). To use it:

```
$ ./fuzzing_driver.sh <signature-chosen> <time-to-fuzz> <input-dir>
    <output-dir> <read-from-file[0|1]> <AFL_DEBUG[0|1]>
    <parallel-fuzzing[0|N]>
```

Fuzzing_manager.py

This python script instead is responsible of orchestrating the start-up, termination and intermediate check-ups of a fuzzing campaign on a phone cluster. It handles connections over ADB, so as a prerequisite it requires that all phones have ADB debugging enabled and are connected to

a central machine using either a TCP/IP connection or multiple USB ports. Like all previous scripts, each device must have a working AFL++ built for Android. Then, it is invoked as follows:

```
$ python fuzzing_manager.py [-h] --action {fuzz_signature,fuzz_one,check,
    kill_fuzzer} [--target TARGET] [--fuzz_time FUZZ_TIME] [--from_file
    FROM_FILE] [--parallel_fuzzing PARALLEL_FUZZING]
```

The options available, listed in the *help* manual, are:

- *action*: can be either *fuzz_signature*, *fuzz_one*, *check* and *kill_fuzzer*. It specifies the action to perform, with self-explanatory names;
- *target*: when *fuzzing_signature* or *fuzz_one*, indicates respectively the name of the target function and the signature;
- *fuzz_time*: time to fuzz each method for, as float[s|m|h|d] (s=seconds, m=minutes, h=hours, d=days);
- *from_file*: flag to specify if harness will read from file (1) or stdin (0) the input generated by AFL++, as before;
- *parallel_fuzzing*: specify the number N of cores to use for a parallel fuzzing campaign (if N is greater than the maximum number of available cores, then the maximum is used).

3.7.5 Debug POC

Once satisfied with the results, hopefully with some crashes detected by AFL++, the target function reporting a crash can be debugged using GDB. To do so, we should run the harness similarly as when fuzzing, but passing as input file, or `stdin`, the POC generated. Inside GDB, to focus only on the currently running thread, it is best to run `set scheduler-locking step`. This stops other threads from "seizing the prompt" by preempting the current thread while stepping. To launch GDB, the commands are as follows:

```
$ export LD_LIBRARY_PATH="/apex/com.android.art/lib64:/path/to/target_app/lib/
arm64-v8a:/system/lib64

# read input from file
$ gdb --args ./harness <path/to/target_app> <lib_name> <target_name> POC

# read input from stdin
$ gdb --args ./harness <path/to/target_app> <lib_name> <target_name> < POC
```

Together with GDB, we debug it with ASAN and reverse it with tools like Ghidra [34] and Radare2 [35]. A partial implementation of ASAN is available by preloading its library, such that it intercepts functions like `malloc`, `free` and others to apply safety checks, and then calls the original function. Before launching the harness, preload it by running:

```
$ export LD_PRELOAD=path/to/clang/13.0.1/lib/android/libclang_rt.asan-aarch64-
android.so:$LD_PRELOAD
```

Chapter 4

Results

4.1 Fuzzing Performance

In this section, we present the performance of the AFL++ fuzzer used in different contexts. We tested each use-case on two devices, a Google Pixel 4 and a Samsung A40. In Table 4.1 are reported the hardware specifications for each device.

	Google Pixel 4	*Samsung A40*
Chipset	Qualcomm SM8150 Snapdragon 855	SAMSUNG Exynos 7 Octa 7904
CPU	Octa-core	Octa-core
Core #0	1.78 GHz Kryo 485	1.6 GHz Cortex-A53
Core #1	1.78 GHz Kryo 485	1.6 GHz Cortex-A53
Core #2	1.78 GHz Kryo 485	1.6 GHz Cortex-A53
Core #3	1.78 GHz Kryo 485	1.6 GHz Cortex-A53
Core #4	2.42 GHz Kryo 485	1.6 GHz Cortex-A53
Core #5	2.42 GHz Kryo 485	1.6 GHz Cortex-A53
Core #6	2.42 GHz Kryo 485	1.8 GHz Cortex-A73
Core #7	2.84 GHz Kryo 485	1.8 GHz Cortex-A73

Table 4.1. Hardware specification of the Google Pixel 4 and the Samsung A 40 Android devices

Android Performance Test

First, to have a baseline comparison of our harness, we used the `test-performance.sh` script present in the AFL++ main folder to test AFL++ performance on an Android device, without considering the harness and Android applications in general. The script compiles and fuzzes for 30 seconds a simple c file, which reads input from either `stdin`, command line or file, and then returns. For both devices, the fuzzing campaign was run on the most performing core (2.84GHz for the Google Pixel 4, while 1.8 GHz for the Samsung A40), after setting each CPU's scaling governor algorithm to maximum performance, and achieved respectively 2900 and 1130 executions per second.

Harness Performance Test

Then, we tested the harness performance using a modified version of the HelloJNI-Callback [30] test app provided by Android as a NDK example. The native method under test is `Java_com_example_hellojnicallback_MainActivity_stringFromJNI`, patched to take as input a Java string, convert it into a native string, append the device's architecture to it and then return it as a Java string. With the same set-up as for the Android performance test, the fuzzing campaign

was stable at 119.8 executions per second for the Google Pixel 4, and 47.0 executions per second for the Samsung A40.

As we can see, when using the harness the fuzzing performance has a big drop (95.8%). The degradation is to be attributed to the use of the deferred fork server. We noticed that this behavior is present even in binaries not using our harness, only a deferred fork server initialization on an Android device. Although it reduces fuzzing performance for a normal binary, it does the opposite when used with a harness performing costly operations during initialization, like loading the ART and performing symbol's lookup. To prove this claim, we measure the fork server effectiveness by registering the performance degradation when removing it from our harness. For the Google Pixel 4 only, we have a 97.5% slowdown, which brings down executions per second from 119.8 to 3. It is therefore essential to have it.

Framework Performance Test

Lastly, we tested performance for both devices using the complete framework, so fuzzing on each device varying the number of cores. With the same PUT, we collected the data shown in Figure 4.1. The histogram on the left side reports the average executions per second on a single core, depending on the number of cores fuzzing in parallel. The histogram on the right side, instead, reports the total number of executions per second achievable on a single device when varying the number of cores used for fuzzing in parallel. As expected, the value on a single core decreases when increasing the number of cores used, achieving only virtually no performance degradation, while the total number of executions per second increases. The peak for the Google Pixel 4 is when 4 cores are used, because of the high difference in frequency with the remaining cores. For the Samsung A40 instead, the peak is reached when all 8 cores run in parallel, and it is almost equivalent to the Google Pixel 4.

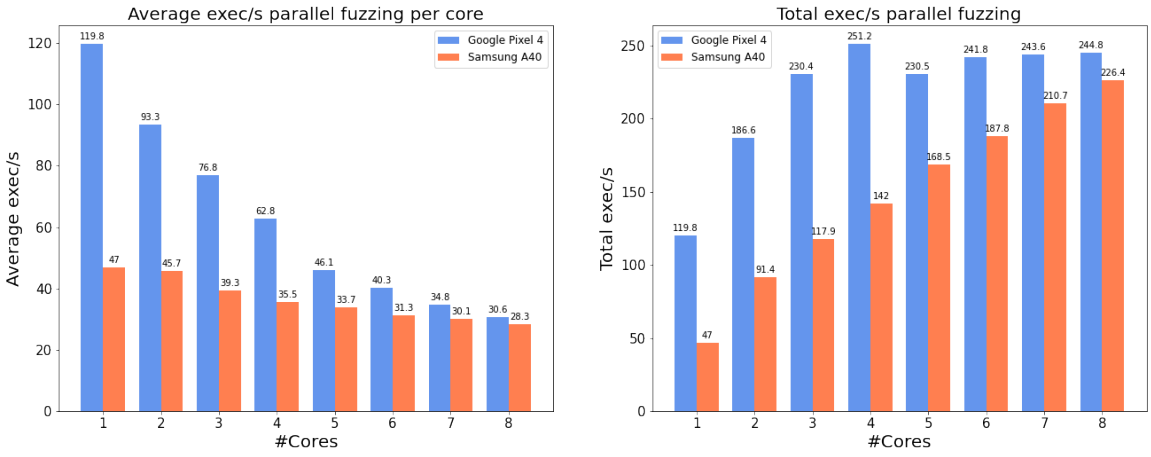

Figure 4.1. Framework performance varying the number of cores used.

4.2 Bug Reproducibility

The first step towards fuzzing closed source libraries is to have the guarantee that our harness is capable of detecting crashes and generating valid POCs. We tested it on two targets: a manually inserted stack buffer overflow in the HelloJNI-Callback sample application, and on a real-world CVE. In both cases, the harness was successful, proving that it is capable of finding bugs and can be applied to real-world applications.

Real World CVE

The real-world CVE taken into consideration was CVE-2019-11932, a double-free bug residing inside a GIF parsing library called *Android-Gif-Drawable* [9] allowing remote code execution. The library is used in several different applications (33000+), including WhatsApp, and it is open source. The bug is triggered using a corrupted GIF file when the following steps are executed:

- an attacker sends the corrupted GIF via WhatsApp message;
- the victim downloads it into the device's gallery;
- the victim then decides to send any file present in the gallery and clicks on the paper clip button to open it;
- the GIF file is loaded by the parsing library to show the user a preview of it, triggering the double-free bug;

The operation of loading the GIF file is performed using the function `DDGifSlurp` from `decoding.c`. It parses each frame individually, reallocating a buffer called `rasterBits` whenever certain conditions are met, briefly whenever the frame size changes. For reallocation it uses function `reallocarray`. When a frame has size 0, it performs a re-allocation of size 0, meaning a free operation. Accordingly, parsing two subsequent frames with one size of 0 (either height or width) means double freeing the same memory location. The bug is then exploited by controlling first the program counter, then dealing with ASLR and WˆX (both mitigations increasing the complexity to exploit the bug).

An example of a GIF causing the double free is reported in Figure 4.2. A GIF file is composed of 2 initial fixed length blocks specifying the version and the canvas size, and then several variable length fields. The field of our interest is the one starting with `0x2C`, representing a single image. Each image (or frame) starts with a header block of 10 bytes, which stores the image's dimensions. In the example, we have 4 frames, with the 2nd and 3rd being the ones causing the double free, with dimensions `0x00` and `0xf1c`.

```
47 49 46 38 39 61 18 00 0A 00 F2 00 00 66 CC CC
FF FF FF 00 00 00 33 99 66 99 FF CC 00 00 00 00
00 00 00 00 00 2C 00 00 00 00 08 00 15 00 00 08
9C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 F0 CE 57 2B 6F EE FF FF 2C 00 00
00 00 1C 0F 00 00 00 00 2C 00 00 00 00 1C 0F 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 2C 00 00 00 00
18 00 0A 00 0F 00 01 00 00 3B
```

Figure 4.2. Corrupted GIF causing CVE-2019-11932.

To reproduce the crash using our harness, we had to first analyze the source code of the library to locate how we could reach the vulnerable portion of code. The call stack generated to reach the bug is as follows:

- *load GIF file*: first we load the GIF file as a Java byte-array inside a native structure, and return a pointer to such structure. The native method responsible for loading the file is `Java_pl_droidsonroids_gif_GifInfoHandle_openByteArray`;
- *trigger double-free*: then, we use the loaded GIF file simulating the GIF preview operation, by calling `Java_pl_droidsonroids_gif_GifInfoHandle_renderFrame`. It takes as input both a Java bitmap and the pointer to the native GIF structure, and later calls `DDGifSlurp`.

The harness is in charge of loading into a Java byte-array the GIF file, extracting its dimensions, generating a valid bitmap using them, and calling both methods in the correct order. As we can see, the only way to reproduce this bug is to call both methods in sequence. Calling for example the second method only would have resulted in a state violation and a consequent invalid execution. This fact highlights the importance of building first a valid state for a certain native method and then using it for fuzzing. We let AFL++ run on a single device, the Google Pixel 4, for over 48 hours, and it was able to reproduce the crash generating the corrupted GIF file.

In addition, we were able to involuntarily reproduce and generate the POC of a heap buffer overflow bug caused by another portion of code, the `prepareCanvas` function, and generated by a corrupted GIF file when its image height is greater than its canvas height (the same is valid for its width, but exclusively). The bug was fixed with v1.2.20.

4.3 Dataset

For evaluation purposes, we used as a dataset the applications provided by the Androzoo collection. We collected all applications with the dexdate starting from 01/01/2021 and with a maximum size of 10MB. This resulted in a set of 25,988 applications. We then filtered them to keep only the ones using the NDK, based on the presence of a `/lib` folder in the APK, remaining with 3,743 native applications.

4.4 Native Methods Extractor Evaluation

We evaluated the success rate of the NativeExtractor using it on our dataset. This step is required by the inability to decompile 100% of the source code with Jadx, and by the fact that QDox originally works with the Java source code, not the decompiled one. Our strategy removes any Java file creating problems with QDox, mainly syntax errors. Analyzing 3743 Android applications (for a total of 17+ million Java files), QDox was unable to parse only 743 Java files (failure rate of 0.0042%). Those files contained 15 missed native methods, which is low considering 275171 native methods overall.

The NativeExtractor, after extracting all native methods, performs a signature frequency analysis on them. We used its output to set up an automated fuzzer, considering which kind of input could generate interesting vulnerabilities (for example strings or byte-arrays). In Table 4.2 we report the top 40 signatures considering all unique native methods for our dataset, with their frequency. For instance, the 5th most frequent, with signature `void:String,` handles a single string, which is an ideal source for buffer overflows. Another insight we noticed is the high number of libraries reused among applications. Most of the applications use the same libraries, generating a low number of unique native methods, in respect of the total number of native methods used.

4.5 Fuzzing Results

Using the output generated by the NativeExtractor, we chose to target native methods taking strings parameters only, either 1, 2, or 3 strings. We evaluated our framework to test the success rate of fuzzing in black-box fashion targets without considering their Java and native state, and to find unknown bugs in Android applications. In the following subsections, we present the results of both evaluations.

Frequency	*Signature*	*Frequency*	*Signature*
2487	`void:`	160	`void:a,`
1343	`int:`	142	`void:long,long,`
921	`String:`	138	`void:CharSequence,int,int,int,`
721	`void:int,`	138	`String:String,`
640	`void:String,`	132	`boolean:Object,`
583	`boolean:`	126	`void:RecyclerView.b0,int,`
523	`Dialog:Bundle,`	117	`void:long,int,`
509	`void:long,`	115	`boolean:int,`
437	`void:Bundle,`	112	`void:String,String,`
436	`void:View,`	111	`void:p,int,int,`
416	`void:Context,`	111	`String:Context,`
393	`void:boolean,`	109	`String:int,`
339	`int:long,`	105	`View:LayoutInflater,ViewGroup,Bundle,`
283	`int:int,`	103	`boolean:long,`
233	`void:Object,`	97	`void:long,float,`
212	`void:int,int,`	96	`void:DialogInterface,int,`
211	`long:`	94	`byte[]:`
206	`long:long,`	91	`boolean:MenuItem,`
179	`long:int,`	89	`void:CharSequence,`
172	`void:DialogInterface,`	84	`int:String,`

Table 4.2. Top 40 frequent signatures.

4.5.1 Stateless Fuzzing Success Rate

As we highlighted before, fuzzing native methods in a stand-alone way is often inconclusive. The reason resides in the way Android applications call native methods. Certain native functions should be always called after others, performing some form of initialization or loading, or after creating a Java state (creating objects, initializing variables, and other operations). In addition, often native methods do not perform safety checks on those state and simply assume that the application follows the correct call sequence. Unfortunately, retrieving the call sequence and the parameters constraints would require another full project, so we only calculated the percentage of methods that can be executed without the state by considering a method crashing for any input as non-testable. We report in Table 4.3 the percentage of testable methods among the ones taking strings parameters only, with the parameters used and the number of crashes found. We run the fuzzing framework on 1 Google Pixel 4 and 5 Samsung A40 devices. The results show that we can test 23.4% native methods on average. Opposite, to prove that the reason for 76.6% of non-testable methods is a state violation, we statically analyzed them picking some at random and analyzing both with GDB and Ghidra/Radare2. The results always reported a state violation. Most of them to be testable would need a harness somewhat similar to the one used in section 4.2 for CVE-2019-11932 initializing a native state, or a harness with a series of JNI calls initializing a Java state. In addition, none of the crashes was generated by a problem inside a JNI function, confirming the ability of our harness to reliably handle all JNI calls and solving the problem partially addressed by JNIFuzzer.

4.5.2 Discovered Bugs

Any crash discovered fuzzing the dataset resulted in a real bug in the native side of the application. Unfortunately, there is no guarantee that the bug discovered is triggerable from the Java side, thus a real security threat. The results show once again that the harness and the fuzzing framework work, and that it can be easily coupled with another tool dealing with the “state” aspect to find only bugs triggerable from the Android application, reducing false positives. In addition, one would need to run each fuzzing campaign for at least 24 to 48 hours to increase the number of crashes detected. We report a subset of the bugs discovered, and deemed significant.

Signature	*Testable Ratio*	*Testable Percentage*	*Fuzzing Duration[h]*	*Number Cores*	*Number Crashes*
`void:String,`	42/238	17.6%	2	4	3
`long:String,`	6/12	50.0%	2.5	6	0
`String:String,`	14/60	23.3%	2	6	1
`boolean:String,`	8/39	20.5%	2	8	0
`int:String,`	16/46	34.8%	2	8	4
`void:String,String,`	12/34	35.3%	2.5	8	2
`long:String,String,`	1/1	100.0%	2	8	0
`int:String,String,`	6/8	75.0%	2	8	0
`boolean:String,String,`	4/8	50.0%	2	8	0
`String:String,String,`	1/12	8.33%	2	8	0
`boolean:String,String,String,`	1/6	16.0%	2	8	0
`void:String,String,String,`	3/10	30.0%	2	8	0
Total	114/474	24.1%	-	-	10

Table 4.3. Results of multiple fuzzing campaigns per signature.

Bug #1

An off-by-one stack buffer overflow in the native method `Java_net_sourceforge_zbar_Image_-setFormat` from applications *Chimpa Bazaar*, *Onix Worker* and *Barcode And QR Code Generator*. The bug is classifiable using the Common Weakness Enumeration code as CWE-111 (direct use of unsafe JNI). The portion of code generating the buffer overflow is the initial call to the JNI function `GetStringUTFRegion`, responsible to convert a Java string into its native version. The buffer used as target is allocated with an invalid size considering all allowed input lengths, and an overflow by 1B occurs whenever the input string has size 4. As the native method has the stack canary mitigation activated, it aborts when it detects the overflow. This bug is not triggerable from the Java size: the string provided to the native method is hard-coded in the source code of the application.

Bug #2

A bug in the implementation of the JNI function `NewStringUTF`, used by the harness. The bug was discovered fuzzing the Android application *Guide for Scary Teacher 3D*, in particular its native method `Java_com_secrethq_store_PTStoreBridge_purchaseDidComplete`, and it is present in a total of 4 applications of our dataset. The function `NewStringUTF` creates a `java.lang.String` object from an array of characters (`char *`) in modified UTF-8 encoding. Its specification states that it returns NULL in case the string cannot be constructed (e.g. the input string is not UTF-8 or modified UTF-8 encoded). Instead, when provided with a non-UTF-8 array of characters (several `0xff` bytes) it frees it without invalidating its pointer and it returns a valid Java object (not NULL). A call to `malloc` after its use returns the same address as the original array of characters. This bug can lead to use after free, and it highlights a serious threat for all Android devices using this function in their native components, if it can be exploited.

Bug #3

A stack-based buffer overflow present in several methods of application *Live Microphone to Speaker* (100K+ downloads on Google Play Store), specifically:

- `Java_top_oply_opuslib_OpusTool_play`
- `Java_top_oply_opuslib_OpusTool_startRecording`
- `Java_top_oply_opuslib_OpusTool_openOpusFile`

- `Java_top_oply_opuslib_OpusTool_isOpusFile`

The bug can again be classified as CWE-111, considering that it is generated by improper use of the JNI function `GetStringUTFRegion`. The native function does not perform any input validation before interacting with the JNI, and any strings provided with a size greater than 256 overflows the target buffer allocated on the stack. As in bug #1, the stack canary detects the violation and promptly aborts the program. To understand if it is reproducible, we analyzed the Java side of the application. The string provided to all native methods corresponds to the file name of an opus audio file either registered through the application or received via any communication means (e.g WhatsApp). Although the file name must conform to the Linux limitation on file name's length (255 bytes), the application appends to it its file paths, generating the overflow for sufficiently long file names. The bug can be converted into arbitrary code execution, as long as the stack canary value is retrieved from a previous leak.

Bug #4

A buffer overflow in function `Java_Runtime_Native_init` from the Android game *PnuYozhika 3*. The overflow is prevented by the use of additional safety checks provided by the FORTIFY extension of the Bionic library, using function `__strcpy_chk`. As soon as the violation is detected, the program is aborted. The bug is not exploitable, but it emphasizes the ability of our fuzzing framework to find existing ones.

Bug #5

Stack buffer overflows in a total of 71 dictionary applications provided by *Dictionary Creator* and available in the Google Play Store. The most used one is a Thai dictionary, with 1M+ downloads. The bugs were discovered when attempting to perform a library-based analysis, instead of a signature one, on the dataset, which induced us to target a specific library called `libgetData.so`. The vulnerable native method is `Java_bestdict_common_code_BisObject_GetSound`, which takes the word chosen in the dictionary and other 3 strings as parameters and using the word constructs a query to Google translator, by concatenating it with an hard-coded string. The source of the bug depends on the application's version, and not all applications share the same version:

- *v.18*: the source of the overflow is a call to `sprintf` using as target a stack buffer of size 40960 bytes, without checking the input buffer length;
- *v.19+*: the source of the overflow is a call to `memcpy` using as source a hard-coded string with size 335 bytes and as destination a buffer which might be already full. A dictionary word with a size greater than 40625 but lower than 40960 generates the overflow.

Both bugs are exploitable in different ways by providing each application with a corrupted dictionary database.

Chapter 5

Conclusion

In this project, we showed how to port and use the fuzzing engine AFL++ on an Android device to fuzz Android application's native components. Our technique integrates existing tools, with the needed patches, to achieve automated black-box fuzzing, and primarily deals with the recreation and use of a valid state to fulfill JNI invocations. As it stands, it is capable of handling 100% of JNI calls. The core component is the harness and its interaction with the ART, while the framework orchestrates each fuzzing campaign on multiple devices. Its evaluation shows the ability to find new bugs in native methods, and reproduce them with the generated POC.

5.1 Future Work

In section 4.1, we reported how a single fuzzing campaign has low executions per second, even if testing the performance of the device proves the contrary. The main cause is having the ART loaded at all times, and using a deferred fork server initialization. To build up on our work, one could apply the caching mechanism presented in subsection 3.2.2 to the harness, to increase fuzzing throughput. Another improvement, simpler from our perspective, would be to understand why the deferred fork server initialization drastically reduces binary performance.

Then, we highlight how our framework mainly focuses on the complexity introduced by JNI calls. It is to be considered as a starting point to the incorporation of tools handling different aspects of application's fuzz testing, mainly stateful fuzzing and code instrumentation. The first one is required to both fuzz a wider range of native functions and has the guarantee that any POC generated corresponds to a crash triggerable from the Java side. The second instead is an optimization step. Following is a more detailed description.

5.1.1 Stateful fuzzing

All native functions crashing for any inputs provided resulted in a state assumption being violated. Most of the time, the correct call stack to reach the function under test requires first a call to one or multiple initialization native functions. Calling `JNI_OnLoad` only results sufficient in few cases. Symbolic execution on the Java side, following a source-to-sink approach, would help recreate a valid call stack of all native functions invoked. Here the source would be a user-generated input, while the sink is the target native method. In addition to a source-to-sink static analysis, solving all generated constraints on the user-provided input would create valid requirements to apply to the fuzzing input. Inputs would then assume values that the application alone could have generated, reducing the number of false positives. Tools performing similar tasks exist, for example Letterbomb [36] or FlowDroid [20], and both use the Soot [37] framework as symbolic execution engine.

5.1.2 Binary Instrumentation

Even if AFL++ performs "smart" black box fuzzing, coverage-guided fuzzing would help maximize coverage and generate more interesting inputs. Considering that most Android application's native components are closed source, the only techniques able to achieve it either use binary rewriting tools such as Retrowrite [38], or dynamic instrumentation tools like CoreSight [29]. A valid improvement of our work would require integrating one of those tools as a preprocessing step. In Appendix A, we report the steps we followed to integrate AFL++ CoreSight mode on Android and the reasons it did not succeed.

Bibliography

[1] A. Cranz, "There are over 3 billion active Android devices." https://www.theverge.com/2021/5/18/22440813/android-devices-active-number-smartphones-google-2021

[2] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting Millions of Android Apps for the Research Community", Proceedings of the 13th International Conference on Mining Software Repositories, New York, NY, USA, 2016, pp. 468–471, DOI 10.1145/2901739.2903508

[3] Google and Open Handset Alliance n.d. Android API Guide, https://developer.android.com/guide/

[4] S. Wentworth and D. D. Langan, "Performance Evaluation: Java vs C++", 39th Annual ACM Southeast Regional Conference, March 16-17, 2001, p. 2002

[5] G. Tan and J. Croft, "An Empirical Security Study of the Native Code in the JDK", Proceedings of the 17th Conference on Security Symposium, USA, 2008, pp. 365–377, DOI 10.5555/1496711.1496736

[6] K. Tsipenyuk, B. Chess, and G. McGraw, "Seven pernicious kingdoms: a taxonomy of software security errors", IEEE Security Privacy, vol. 3, November 2005, pp. 81–84, DOI 10.1109/MSP.2005.159

[7] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. De Geus, C. Kruegel, and G. Vigna, "Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy", Symposium on Network and Distributed System Security (NDSS), February 2016, DOI 10.14722/ndss.2016.23384

[8] F-Droid, "The f-droid repository." https://f-droid.org/

[9] Koral, "Android-gif-drawable." https://github.com/koral--/android-gif-drawable.git, 2013

[10] Oracle, "JNI 6.0 API Specification." https://docs.oracle.com/javase/7/docs/technotes/guides/jni/

[11] M. Zalewski, "American Fuzzy Lop." https://github.com/google/AFL, 2013

[12] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research", 14th USENIX Workshop on Offensive Technologies (WOOT 20), USA, August 2020

[13] GOOGLE, "HonggFuzz", 2016, https://google.github.io/honggfuzz/

[14] LLVM, "libFuzzer - a library for coverage-guided fuzz testing", 2017, http://llvm.org/docs/LibFuzzer.html

[15] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker", 2012 USENIX Annual Technical Conference (USENIX ATC 12), Boston, MA, June 2012, pp. 309–318

[16] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe", NDSS, 2015, p. 110

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

[18] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities", Proceedings of the 2012 ACM Conference on Computer and Communications Security, New York, NY, USA, 2012, p. 229?240, DOI 10.1145/2382196.2382223

[19] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps", ACM Trans. Priv. Secur., vol. 21, apr 2018, DOI 10.1145/3183575

[20] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps", Acm Sigplan Notices, vol. 49, no. 6, 2014, pp. 259–269, DOI 10.1145/2666356.2594299

[21] H. Shahriar, S. North, and E. Mawangi, "Testing of Memory Leak in Android Applications", 2014 IEEE 15th International Symposium on High-Assurance Systems Engineering, 2014, pp. 176–183, DOI 10.1109/HASE.2014.32

[22] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag", Proceedings of International Conference on Advances in Mobile Computing and Multimedia, New York, NY, USA, 2013, p. 68?74, DOI 10.1145/2536853.2536881

[23] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, and Z. Su, "Fully Automated Functional Fuzzing of Android Apps for Detecting Non-Crashing Logic Bugs", Proc. ACM Program. Lang., vol. 5, oct 2021, DOI 10.1145/3485533

[24] S. Arzt and E. Bodden, "StubDroid: Automatic Inference of Precise Data-Flow Summaries for the Android Framework", 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), May 2016, pp. 725–735, DOI 10.1145/2884781.2884816

[25] G. Fourtounis, L. Triantafyllou, and Y. Smaragdakis, "Identifying Java Calls in Native Code via Binary Scanning", Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, USA, 2020, p. 388?400, DOI 10.1145/3395363.3397368

[26] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyand'e, and J. Klein, "JuCify: A Step Towards Android Code Unification for Enhanced Static Analysis", 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), 2022, pp. 1232–1244, DOI 10.48550/ARXIV.2112.10469

[27] Z. Kan, H. Wang, L. Wu, Y. Guo, and D. X. Luo, "Automated Deobfuscation of Android Native Binary Code", 2019, DOI 10.48550/ARXIV.1907.06828

[28] C. Rizzo, "Static Flow Analysis for Hybrid and Native Android Applications". PhD thesis, Royal Holloway, University of London, 2020. (unpublished)

[29] A. Moroo and Y. Sugiyama, "ARMored CoreSight: Towards Efficient Binary-only Fuzzing", November 2021. https://ricercasecurity.blogspot.com/2021/11/armored-coresight-towards-efficient.html

[30] Google, "Hello JNI Callback." https://github.com/android/ndk-samples/tree/main/hello-jniCallback, 2015

[31] Google and Open Handset Alliance, "Android bionic status." https://android.googlesource.com/platform/bionic/+/master/docs/status.md

[32] Skylot, "jadx." https://github.com/skylot/jadx, 2019

[33] P. Hammant, "QDox." https://github.com/paul-hammant/qdox, 2016

[34] NSA, "Ghidra." https://github.com/NationalSecurityAgency/ghidra, 2019

[35] S. Alvarez, "Radare2." https://github.com/radareorg/radare2, 2006

[36] J. Garcia, M. Hammad, N. Ghorbani, and S. Malek, "Automatic Generation of Inter-Component Communication Exploits for Android Applications", Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, New York, NY, USA, 2017, p. 661?671, DOI 10.1145/3106237.3106286

[37] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java Bytecode Optimization Framework", Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, Mississauga, Ontario, Canada, 1999, p. 13

[38] S. Dinesh, N. Burow, D. Xu, and M. Payer, "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization", 2020 IEEE Symposium on Security and Privacy (SP), 05 2020, pp. 1497–1511, DOI 10.1109/SP40000.2020.00009

[39] E. Dolstra, "patchelf." https://github.com/NixOS/patchelf

Appendix A

AFL++ CoreSight in Android

Fuzzing engines such as AFL++, *libfuzzer* or *Honggfuzz* perform coverage-guided fuzzing using program instrumentation. To find new interesting program paths they benefit from run-time feedback. The feedback is generated by instrumentation inserted in the program under test either statically or dynamically. Static tools rewrite the target binary by inserting instrumentation. They do not influence fuzzing performance, while are complex to develop. For example, for ARM devices there is no public tool yet available, except for Retrowrite [38] which is still a work in progress, especially in Android. Dynamic tools instead insert code at run-time, resulting in obvious performance degradation. The de-facto standard tool to dynamically instrument without the source code is *QEMU* emulator mode. As an alternative, Ricerca Security developed ARMored CoreSight [29], a tool integrable with AFL++ and leveraging CoreSight ARM processor CPU feature to provide instrumentation. Coresight is for ARM CPU as Processor Trace is for Intel CPU, and it can capture branch executions at runtime. Compared to QEMU, it uses a native CPU feature instead of executing the PUT on a VM, outperforming it by nature (Figure A.1).

Our targets are closed-source Android application binaries, and as such we looked into CoreSight to have a working dynamic tool to instrument them. We spent quite some time trying to use it, so accordingly in this appendix we present AFL++ CoreSight mode, the steps we applied to build it, and the reasons why we were not able to accomplish it in Android. This section is to be considered as a first attempt and a starting point for future efforts.

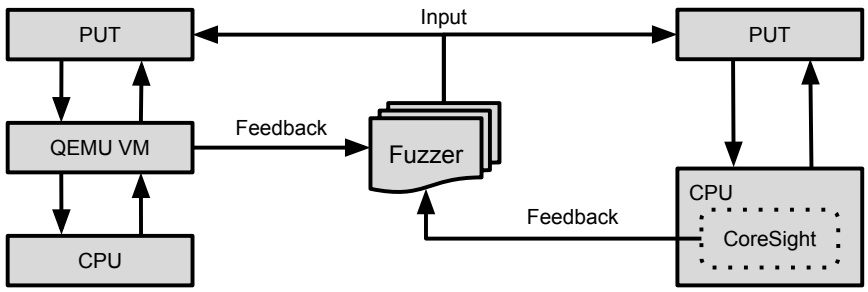

Figure A.1. QEMU and CoreSight modes in fuzzing (source: Ricerca Security).

Design

Modern ARM CPUs include in their design the hardware components for CoreSight. Three components called *trace source*, *trace sink* and *trace link* produce and provide to external entities trace data, such as information about branches useful when dealing with coverage fuzzing. AFL++

CoreSight is composed of *coresight-trace* and *coresight-decoder*, both running on user-space. The first handles and control CoreSight components and monitors the PUT. It is implemented as a derivation of the *afl-proxy* skeleton file provided by AFL++, and it works in fork server mode. The second instead uses the data trace output to generate meaningful coverage information, understandable by AFL++. Information is partial and is encoded in form of packets. Considering branches, for example, coresight-trace only stores the location of the branch instruction, not the destination address. To retrieve such information, the decoder disassembles the target binary.

Port on Android

As a prerequisite, AFL++ CoreSight requires only the Capstone disassembly library. The building process is divided into two steps: first, we must compile the coresight-trace and coresight-decoder components; then, the target binary must be linked against a patched version of the libc to provide the advantages of the fork server even in CoreSight mode. In our case, the fork server must be present considering that ART start-up time in the harness is non-negligible.

To compile both CoreSight components in Android, the only modification required to their source code is about function `pthread_setaffinity_np` used by CoreSight-trace. The function is used to set the CPU affinity mask of the given thread to a CPU set provided. It is a per-thread call, and it is part of the NON-POSIX extension of the libc. Bionic libc comprises only a subset of all NON-POSIX functions, and this one is not present. Instead, it can be substituted by `sched_setaffinity`, which does the same operation but per-process, providing a PID. To use it, we must slightly modify its implementation as follows:

1. fetch the preferred CPU before invoking `pthread_create`. It is later used inside the thread's routine;
2. make the thread routine `decoder_worker` call a modified version of `set_pthread_cpu_affinity` only if required (the preferred CPU value is non-negative);
3. patch function `set_pthread_cpu_affinity` to call `sched_affinitity` on the current thread, by passing 0 as PID.

Then, the target binary is linked against a modified version of the GNU C library (glibc) using the *Patchelf* [39] utility, containing the function used by AFL++ to start the fork-server. Problems arise when building the glibc patched implementation with LLVM/Clang. Due to its reliance on different GNU toolchain extensions or specific non-standard GCC behaviors, glibc requires to have a GNU GCC compiler with version 6.2 or higher. In Android, GCC is not available by default, and although few un-official releases exist none seems to be working as expected. The last available GCC compiler for Android was provided together with NDK r17, and it was version 4.9, too old to be used in our case. Using as an alternative to glibc *musl* also resulted in several missing symbols.

Even if the fuzzing advantage is noticeable, the final goal would be to use the fuzzing framework together with Retrowrite, and so we stopped any further attempts to build CoreSight.