JNFuzz-Droid: A Lightweight Fuzzing and Taint Analysis Framework for Android Native Code

1st Jianchao Cao
Jiangxi Normal University
NanChang, China
jc_cao@jxnu.edu.cn

2nd Fan Guo
Jiangxi Normal University
NanChang, China
fguo@jxnu.edu.cn

3rd Yanwen Qu
Jiangxi Normal University
NanChang, China
qu_yw@jxnu.edu.cn

***Abstract*—The need to account for native code in Android apps is becoming urgent as the usage of native code is growing with both benign and malicious apps. However, most current state-of-the-art analysis tools cannot effectively analyze the data-flow behavior of native code. On the one hand, existing native dynamic analysis tools are primarily based on test input generation tools to analyze Android apps and are therefore unable to locate native code quickly. On the other hand, existing native static analysis tools are based on symbolic execution to analyze native code and are therefore limited by the path and state explosion issues. In order to effectively analyze the behavior of sensitive data in the native world, we first proposed *JNFuzz*, a fuzzing module for Android native libraries based on Client/Server architecture. Then, we propose *JNFuzz-Droid*, a lightweight automated fuzzing and taint analysis framework for Android native code, based on this. *JNFuzz-Droid* first locates the Android native code to which sensitive data is passed and then uses *JNFuzz* to perform fuzzing the native code to improve code coverage while analyzing the data flow in native code with a dynamic binary tool. Experimental results on benchmarks and real-world apps show that *JNFuzz-Droid* can effectively detect the leakage or transfer of sensitive data in app native code and outperforms the state-of-the-art native analysis tools.**

***Index Terms*—Android Taint Analysis, Dynamic Analysis, Fuzzing Testing, Android Native Code, Static Analysis**

I. INTRODUCTION

The Android operating system has continuously dominated the smartphone market for a decade, with a market share exceeding 72% as of February 2023, according to Statcounter [1]. At the same time, developers often use native code in their apps for improved performance in processor-intensive tasks and compatibility with legacy code. A recent study [2] showed that about 40% of benign apps use native code for implementing data communication or achieving better performance. However, native code is also vulnerable to malware exploitation. Recent studies [2]–[8] have revealed the utilization of native code in apps for various purposes, such as performing anti-analysis (i.e., Emulator Detection behaviors and Timing Check [9] behaviors), hiding malicious operations (e.g., private data leak) and evading antivirus detection programs for malicious apps, and so on.

Among these purposes, leaking sensitive data is one of the main threats of using native code in these malicious attacks, so determining the data flow behavior of native code is a key step to identify such threats. Our statistics (Section V-A) on malware apps that have emerged in recent years shows that up to 79.7% of malware apps contain native code. However, native code of Android app is scarcely considered in app security vetting [2], [7]. In the majority of static [10]–[16], dynamic [17], [18] and hybrid techniques [19], [20], native code is overlooked since it presents several challenges.

The increasing concern over vulnerabilities in native code of Android apps has driven both academia and industry to develop analysis tools. While dynamic analysis approaches [3]–[6], [21]–[24] provide the advantage of accurately evaluating the execution of native code during each round of analysis. However, they are often limited by their reliance on test input generation and the lack of efficient path exploration techniques, which makes it challenging to effectively locate and analyze native code.

On the other hand, static analysis offers a complete evaluation of the entire execution state space, making it more comprehensive. The state-of-the-art static solutions for detecting data leaks in apps with native code, such as *JN-SAF* [2] and *JuCify* [7] utilize angr [25] symbolic execution to analyze native code. Nevertheless, the utilization of symbolic execution for path exploration may result in path explosion, making these tools unsuitable for large-scale testing. Additionally, they do not take into account Linux system calls functions (e.g., sendto()) in the native code.

In addition, researchers also try to use fuzzing techniques to analyze Android native code, such as LibFuzzer [26], Frida mode [27] and QBDI mode [28] in AFL++ [29] and so on. However, due to a lack of consideration for the Java VM environment (on which JNI functions depend) and limited adaptation to ARM architecture, there is no publicly available fuzzing tool with common applicability for Android native libraries. Furthermore, fuzzing is primarily used for software defect detection, rather than taint analysis.

As far as we know, there is currently no effective technique for detecting sensitive information leaks in Android native code. Based on the above discussion, our approach involves the following steps: ❶ We identify the native functions that receive sensitive data in the global view provided by static analysis; ❷ We execute these native functions and utilize dynamic analysis techniques to monitor, analyze, and evaluate their behavior; ❸ We implement a fuzzing module for the Android native function, which is used to improve the coverage of the native function. By successfully addressing these challenges, we can

2640-7574/24/$31.00 ©2024 IEEE
DOI 10.1109/SANER60148.2024.00033

effectively analyze the behavior of sensitive data in the native code.

In this paper, we propose *JNFuzz*, an efficient black-box fuzzing module based on C/S architecture for fuzzing the Android native libraries on real devices, and design the *JNFuzz-Droid* based on it. *JNFuzz-Droid* first uses a state-of-the-art static analysis tool to obtain the sensitive information flow from Java code to native method. Next, it resolves the native functions corresponding to the native methods. Then, *JNFuzz* module is used for fuzzing native functions to improve code coverage, and a lightweight dynamic binary instrumentation tool is also used to trace the data flow of the native code. Finally, the data flow in the native code is analyzed to determine if sensitive information leaks or transfers.

The main contributions of this paper are as follows.

- We propose *JNFuzz*, an efficient black-box fuzzing module for Android native libraries based on C/S architecture.
- Based on *JNFuzz*, we design *JNFuzz-Droid*, a lightweight automated fuzzing and taint analysis framework for native code of Android apps. It can quickly locate the Android native code to which sensitive data is passed and automatically analyze and discover data leakage or transfer issues in native code.
- We design *CCBench*, a set of benchmark apps, with the objective of analyzing the information leakage of the native world in common scenarios. It also reveals the weakness of existing analysis tools.
- We evaluate *JNFuzz-Droid* on a set of real-world apps. The experimental results show that *JNFuzz-Droid* outperforms the state-of-the-art native analysis tools *JN-SAF* and *JuCify*.
- We release the source code of *JNFuzz-Droid* and *CCBench* at https://github.com/cjc-github/JNFuzz-Droid

The remainder of this article is organized as follows. We first introduce background notions and motivate our work in Section II. In Section III, we discuss challenges and our solutions, and Section IV describes *JNFuzz-Droid* architecture in detail. The experimental evaluations are reported in Section V, and the limitations of *JNFuzz-Droid* are discussed in Section VI. Finally, we overview the related work in Section VII and conclude in Section VIII.

II. BACKGROUND AND EXAMPLE

We provide the necessary background information to understand how Android native world works. In addition, we also provide a motivating example to discuss the challenges of sensitive data leakage in Android native world.

A. Native Development Kit

The Android Native Development Kit (NDK) [30] enables developers to write native code in the C and C++ programming languages and integrate it into an APK through the utilization of Gradle [31]. The NDK employs the Java Native Interface (JNI) [32] to bridge communication between Java and other programming languages. It is mainly used in cases such as improving program performance, reusing existing third-party libraries, increasing complexity in decompilation, and so on.

B. Java Native Interface

The Java Native Interface (JNI) [32] is a native programming interface between Host JVM and native libraries. It allows Java code inside a Java Virtual Machine (VM) to interoperate with libraries written in C or C++ languages. When using JNI in the NDK, developers should use the keyword "*native*" to declare the methods implemented as native code in Java code. The *System.loadLibrary()* function is then invoked to load the native library containing the native code into the virtual machine.

C. JNI Data Structure

There are two key data structures in JNI, *JNIInvokeInterface* [33] and *JNINativeInterface* [34]. As Fig. 1 illustrated, both of them contains a list of function pointers. *JavaVM** and *JNIEnv** are the pointers which points to the head of each table.

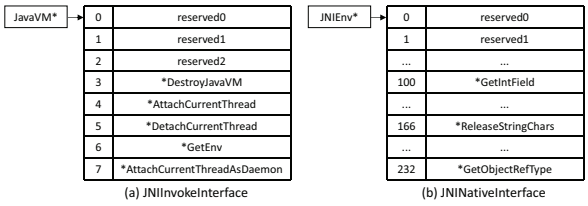

Fig. 1. JNIInvokeInterface and JNINativeInterface Structures

(1) JNIInvokeInterface: It provides JNI functions to create-/destroy Java VM and allocate/discover *JNIEnv*.

(2) JNINativeInterface: It provides JNI functions to create-/check/update Java objects, invoke Java methods, load classes and load class information, etc.

D. A Motivating Example

A malicious app developer can rely on native code to implement some key sensitive, or even malicious, parts of the app behavior. Fig. 2 illustrates an example app (named "Native-leaking"). It consists of two worlds. 1) Java world: It contains two Activity components: *MainActivity* and *utilActivity*. Where the *utilActivity* loads library "*native-lib*" and imports two native methods *fun1()* and *fun2()*. 2) Native world: It declares four functions: *leaksensitive()*, *Java_com_test_-example_utilActivity_fun1()*, *native_fun()* and *JNI_OnLoad()*. Among them, the *leaksensitive()* function writes sensitive data to the *LOG()* by invoking the *sqrt()* function to determine the length of its second argument.

The following sequence of events (as labeled in Fig. 2) can happen in reality:

(1). Once the *MainActivity* component execution requirements are satisfied, the app will jump to the *utilActivity* component. In the *utilActivity* component, two native methods *fun1()* (J27) and *fun2()* (J28) implemented in the native library "*native-lib*" (loaded at J13) will be invoked when the user clicks the button "*register*" (J22).

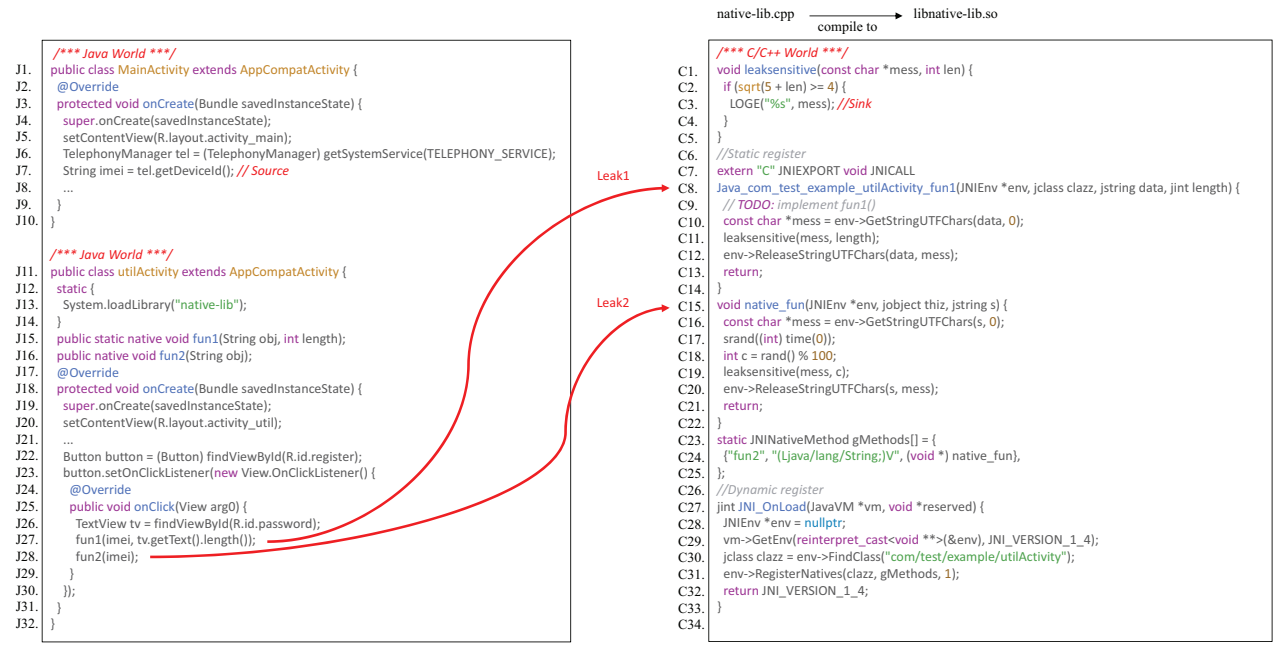

Fig. 2. The Native-leaking App: The arrowed lines among the app components highlight some of the inter-language communication. (Methods and codes are simplified for convenience)

(2). Next, as shown in Leak1 and Leak2 below, these two native methods will leak sensitive data in different ways. The native functions reachable from the Java code follow the JNI naming convention [35] (Leak1) or are registered through the *RegisterNatives()* callback (Leak2).

Leak1: a). The corresponding native function for native method *fun1()* is *Java_com_test_example_utilActivity_fun1()* (C8) via applying JNI naming convention. The method has two arguments: The first argument is the sensitive data *imei*, and the second argument is the length of the text in the TextView "*password*". b). Then, the *leaksensitive()* function with the text length as the second argument was invoked to leak the sensitive data *mess*. Leak1 is a way to trigger the leak of sensitive data via user interaction.

Leak2: a). The *JNI_OnLoad()* function binds native method *fun2()* to native function *native_fun()* at runtime by invoking JNI function *RegisterNatives()* (C31). The native method *fun2()* has only one argument, sensitive data *imei*. b). The *native_fun()* function first generates a random integer *c* between 0 and 100 (lines C17-18). Then, the *leaksensitive()* function with integer *c* as the second argument was invoked to leak the sensitive data *mess*. Leak2 is a way to trigger the leak of sensitive data via random generation.

Limitation of current techniques: Existing native dynamic analysis techniques use test input generation tools (e.g.,Monkey [36]) to explore enough behavior to analyze Android apps. It may be difficult to automatically identify and satisfy the execution path of native methods (e.g., from *MainActivity* component to the *fun1()* function). In addition, whether sensitive data will be leaked in the *native_fun()* function depends on the return value of the *rand()* function, which means it is difficult to trigger the leakage of sensitive data in native code all at once using existing dynamic analysis techniques.

Existing native static analysis techniques (e.g., *JN-SAF* [2]) employ symbolic execution to explore native code paths. However, this approach presents limitations, as it cannot parse and store floating-point values (such as the return value of *sqrt()* function), which leads to analysis interruptions and thus fails to detect sensitive data leakage in native code.

Existing fuzzing techniques (e.g., AFL [37]) can fuzz *leaksensitive()* function, but cannot fuzz native functions (e.g., *native_fun()*). Specifically, existing fuzzing techniques lack consideration for the Java VM (i.e., how to properly create *JNIEnv** pointers for executing JNI functions), hindering the fuzzing of native functions.

Approach: The discussion of the limitations faced by existing techniques inspired us to construct a processing path that consists of three steps: static analysis, fuzzing native functions, and data flow analysis.

On the one hand, we note that it is difficult to quickly obtain the execution path from the Source API (in *MainActivity* component) to the native methods (in *utilActivity* component) using dynamic techniques. To this end, we can first use static analysis techniques to quickly obtain the native methods (i.e., *fun1()* and *fun2()*) that may have leaks.

On the other hand, using static analysis techniques to analyze apps with native code can easily lead to path explosion. One idea is to execute native functions dynamically. However, as we know from Leak2, dynamic analysis techniques may require a large number of executions of native code to trigger sensitive information leakage. Therefore, we decided to

perform fuzz on native functions.

In order to bridge the gap between static analysis and fuzzing, it is necessary to obtain the native function corresponding to the native method. e.g., the *native_fun()* function corresponding to the *fun2()* method.

Furthermore, We also need to monitor the behavior in the native function to determine whether sensitive data leakage is triggered. For example, by tracking propagation of the sensitive data, when the text length of TextView "*password*" is not less than 11 (in Leak1), we can conclude that the variable *mess* to be written into the *Log()* is sensitive.

III. Core Challenges and Our Solutions

There are several challenges in building the **approach** described in Section II-D. Challenge 1: How to fuzz native functions. We need to implement an efficient and practical fuzzing module for the Android native libraries. Challenge 2: How to locate target native method. We need to quickly locate target native methods and improve testing efficiency. Challenge 3: How to resolve native method calls. We need to resolve native functions corresponding to native methods. Challenge 4: How to analyze data flow. We need to identify, trace, and analyze the data flow information in native code.

A. Challenge 1: Fuzzing Native Functions

In order to fuzz native functions, we first describe the process of how Android app loads and invokes native methods. Then we analyze the feasibility of implementing fuzzing of native code. Finally, we introduce *JNFuzz*, an efficient fuzzing module for Android native libraries based on the Client/Server architecture.

1) Process of Android app loading native code

When loading native code, the Android app invokes the native library based on the *dlopen()* and *dlsym()* functions. Specifically, the Android app can load native code by using the dynamic linker (*dlopen*) to open the native library and the *dlsym* function to obtain the address of the native function or the *JNI_OnLoad* function. Finally, Java code invokes the native function by passing arguments to it.

The first argument of the native function is a JNI interface pointer of type *JNIEnv**, which is used to access the JNI environment. The second argument of the native function can be either a *jobject* or a *jclass*, depending on whether the native function is invoked as an instance method or a static method in Java world, respectively. The remaining arguments correspond to regular native method arguments.

2) Feasibility analysis of implementing fuzzing

Through the above process analysis, we found that once all the argument values (JNI type) of the native function are obtained, we can fuzz the native code. Therefore, obtaining argument values is the key to implementing fuzzing. Next, we will introduce: ❶ how to get argument values of *JNIEnv** pointer type; ❷ how to get argument values of other JNI types.

Get JNIEnv* pointer type: JNI provides APIs to load Java VM into native code and interact with it. One of these APIs, *JNI_CreateJavaVM*, loads and initializes a Java VM and returns a pointer to the *JNIInvokeInterface* structure, *JNIEnv*. *JNI_CreateJavaVM* has three arguments, the first two are pointers *JavaVM** and *JNIEnv**, which point to the *JNIInvokeInterface* (Fig. 1a) and *JNINativeInterface* (Fig. 1b) structures. The third argument is a structure called *JavaVMInitArgs*, which provides the VM startup options where we can specify the APK file and the native libraries that compose the app.

In order to create a JVM, we need to find the address of *JNI_CreateJavaVM* function in `libdvm.so` or `libart.so` file. Before invoking it, we also need to register some framework native methods used by the app. We can achieve this by invoking the corresponding function (e.g., *registerFrameworkNatives()*) in `libandroid_runtime.so` file. Using this approach, we are able to execute native code in different versions (e.g., DVM or ART) and architectures (e.g., ARM).

Get other JNI types: JNI defines a set of C/C++ types that correspond to the primitive and reference types in the Java programming language. This set of types are divided into two categories: primitive types and reference types. The primitive types are *typedefs* of C/C++ primitive types and can be used directly in C/C++ code. The reference types cannot be used directly and needs to be converted by the JNI function.

3) Client/Server architecture

The traditional method of fuzzing Android native libraries (Harness) has three drawbacks (as shown in Fig. 3a): ❶ Creating and initialization of the Java VM is time-consuming for each run of the Harness process. ❷ Determining the address of the native function via *dlopen()* and *dlsym()* is also time-consuming for each run of the Harness process. ❸ Dynamic instrumentation for subsequent taint analysis reduces efficiency and is time-consuming and unstable, leading to reduced fuzzing efficiency.

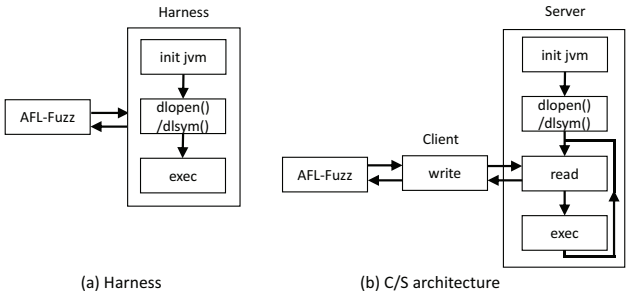

Fig. 3. Client/Server Architecture

To address above issues, we proposes JNFuzz, an innovative black-box fuzzing module for Android native libraries based on Client/Server (C/S) architecture (as shown in Fig. 3b). In this module, the Client acts as the target program for AFL++, saving the seeds generated by AFL++ to a shared memory based ring buffer. When everything is ready (i.e., creating and initialization the Java VM and obtaining the address of native function), the Server will sequentially extract the seeds that the Client has saved from the buffer, then proceed to process and execute the native function.

The JNFuzz module allows for efficient processing as the Server only needs to initialize the Java VM and determine the native function address once (similar to storing a VM snapshot). Furthermore, dynamic instrumentation can be performed just once for countless times data flow analyzes.

We fuzz a representative native function called *bubbleSort* with 50 inputs for comparison. The experiment was carried out 30 times, each round lasting 2 hours. Results show that the average number of executions per round of *JNFuzz* is 197 times higher than the traditional method.

B. Challenge 2: Get Target Native Methods

In this section, we first pick out static analysis tools to obtain data flow information and then improve the efficiency of subsequent fuzzing by obtaining constant parameters values of native methods.

1) How to quickly locate target native methods?

With the help of existing studies [38]–[41] and a comparative experiment scenario specifically designed for this paper, we have chosen the static analysis tools. The reasons for selection are as follows. ❶ CHEX [11] is mainly used to detect component hijacking problems in Android; ❷ DroidSafe [10] cannot modify the Source and Sink lists, and supports the latest API level (i.e., targetSdkVersion) up to 19 [39]. ❸ We applied Amandroid [15], [16] and FlowDroid [13], [14] on NativeFlowBench [42] benchmarks using the native method of the Java code as Sink. The results show that Amandroid detects 14 issues while FlowDroid only detects 9.

Finally, *JNFuzz-Droid* chose *AmanDroid* as our static analysis tool and *FlowDroid* as the alternative static analysis tool.

2) How to improve the fuzzing efficiency of native code?

Precise string and complex types analysis is expensive and non-trivial as mentioned in prior research [43]–[45] and to improve the efficiency of fuzzing, we obtain the constant parameter values of the native method from the perspective of reducing fuzzing parameters.

For *AmanDroid*, we present a new approach to determine constant parameter values on top of the Reverse Search (RS) Algorithm through the following steps: First, we obtain the *Inter-procedural Data Dependence Graph* (IDDG), which is constructed by the Intermediate Representation (IR) language *pilar* [15], [16], generated when analyzing the APK with *AmanDroid*. Next, we identify the invoke statement for the native method parameters on IDDG. Then, starting from this invoke statement, we traverse the graph backward to locate the assignment statement for the parameters. Finally, by analyzing the assignment statement, we determine the values of the parameters.

As an example, using the RS algorithm, it can be determined that the second argument value of the *foo()* function is 2, as shown in Fig. 4.

For *FlowDroid*, it uses the *Jimple* [46] IR for analysis and retains the known constant parameter values of the apps by default. Therefore, we can directly obtain the constant parameter values of the native method from the sink (i.e., native method) statement obtained from the analysis of *FlowDroid*.

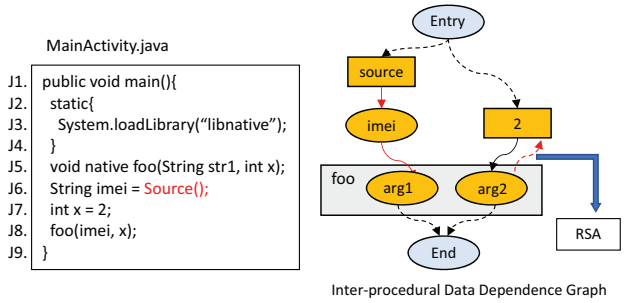

Fig. 4. Reverse Search Algorithm

C. Challenge3: Resolve Native Method Calls

In order to build a unified platform for static analysis and fuzzing, we need to resolve the binary where the native method resides and the corresponding native function.

JNI allows two ways to resolve a native method call to a native function:

(1). **Static register:** This mode follows the naming convention in the JNI specification [35] to generate the corresponding native function names.

(2). **Dynamic register:** JNI allows developers to register [47] native method signature to native function mapping dynamically.

In order to accurately locate the binaries and native functions corresponding to native methods, *JNFuzz-Droid* needs to get the native function names corresponding to native method signatures. We propose a *nativeMap* data structure. The *nativeMap* is a map where the key is the native method signature, and the value is the corresponding native function name and the containing *so* file.

Algorithm 1 shows the pseudo code for generating *nativeMap* of a given APK A. For a given native method signature *sig*, we generate its native function name *funName* according to the naming convention. Then we use the tools in the Binutils library to get the exported function names of each binary file *soFile* and analyze whether *funName* exists in it. If yes, we add it into the *nativeMap*. If not, we continue to check the dynamically registered functions list for *soFile* and check if the method signature is dynamically registered. If yes, we add it into the *nativeMap*.

Algorithm 1 Resolve Native Method Calls

Input: APK, A
Output: A'native method to so file map, nativeMap

```
1:  procedure GetNativeName(apk)
2:      nativeMap ← emptymap
3:      for all sig ∈ sinkSign(apk) do
4:          funName ← resolve(sig)
5:          for all soFile ∈ getAllELF(apk) do
6:              if funName ∈ Binutils(sig, soFile) then
7:                  nativeMap(sig) ← (funName, soFile)
8:              else
9:                  dynamicMap ← getDynamicRegisterFun(soFile)
10:                 if sig ∈ dynamicMap then
11:                     nativeMap(sig) ← (dynamicMap(sig), soFile)
12:     return nativeMap;
```

However, developers often use the *strip* command to remove and hide most dynamically exported symbols from the so

file for security, APK size reduction, etc. Therefore, getting the corresponding dynamically registered functions for native methods is complicated. We analyze each binary file in APK, *soFile* , as follows.

(1) We search *JNI_OnLoad* in the set of the exported symbol, if it is found, we proceed to the next step.

(2) Take Fig. 2 as an example. We utilize nativediscloser [48] to perform symbolic execution from *JNI_OnLoad()*. When the symbolic execution engine reaches *RegisterNatives*, we can get the memory address of the *gMethods* array. Because each element is accessible at a fixed offset through the *JNINativeMethod* structure, we can resolve each element value of the *gMethods* based on the address and the structure of *JNINativeMethod*. If *sig* is found, we proceed to next step.

(3) We execute the *soFile* and intercept the execution of *RegisterNatives*, locating the address of corresponding native function based on the *gMethods* array structure.

D. Challenge4: Data Flow Analysis

To analyze NDK/JNI-aware data flow in Android native code, we introduce an innovative annotation-based taint analysis on top of *Frida* [49], a function-level dynamic instrumentation framework, avoiding the overhead of taint propagation at the instruction-level or basic block-level. The whole procedure consists of three steps:

First, we identify and recognize the functions that are invoked during execution. They are divided into three categories: Native functions, JNI functions, and Linux system calls. ❶ Native functions: the entry points to native code. ❷ JNI functions: functions in the *JNINativeInterface* structure. ❸ Linux system calls: system library functions, such as the *memcpy()* functions in the `libc.so` file, etc.

Next, we hook the identified functions and write the corresponding annotations to implement an annotation-based data flow tracker for native code. We designed two custom annotations to implement the data flow tracker for the native world.

(1) TaintAnnotation: For identified functions that may have taint propagation, we model the taint propagation relationship between parameters and return values of them. By hooking these functions, *TaintAnnotation* is used to annotate the taint propagation relationship (e.g., value and address) between parameters and return values.

(2) SummaryAnnotation: To obtain specific behavior of native code, we analyze the semantics of the system library functions and JNI APIs we are interested in (e.g., *open()*, *FindClass()*, etc.). By hooking them, *SummaryAnnotation* is used to annotate interesting process behavior (e.g., opened files). Finally, a data flow (taint) analysis is performed based on the annotations.

The implementation details were described in Section IV-D.

IV. THE JNFUZZ-DROID FRAMEWORK

Fig. 5 illustrates the workflow of the *JNFuzz-Droid* framework, which consists of four major steps: 1) *TaintPath Acquisition*: obtains the sensitive data flow from Source API to native methods and native method parameters values; 2) *Fuzzing Environment Builder*: resolves native method calls and generating the environment for fuzzing; 3) *JNFuzz*: fuzzing the native function to improve code coverage; 4) *VA-Trace*: tracks the data flow of sensitive data in the native function.

Among them, *TaintPath Acquisition* and *Fuzzing Environment Builder* execute Java and native world on the PC, respectively. *JNFuzz* and *VA-Trace* are executed on real Android devices.

A. TaintPath Acquisition

JNFuzz-Droid first uses *apktool* [50] (V2.6.1) to collect the signatures of native methods as sinks, and then uses static analysis tools (e.g., *AmanDroid*) to detect TaintPaths from the default source APIs (e.g., getDeviceId()) to these sinks. Finally, using the innovative approach presented in Section III-B2, *JNFuzz-Droid* obtains constant parameter values of the sink methods of TaintPaths, and saves them to *JNIScript*.

B. Fuzzing Environment Builder

JNFuzz-Droid first search for ELF files or compressed packages containing ELF files in the non-standard (i.e., *assets*) folder. If so, these files will be extracted to the *lib* folder. Next, it determines the best-performing CPU architecture in the APK as the target architecture, and the judging criteria is: *arm64-v8a* $>$ *armeabi-v7a* $>$ *armeabi*. Then, the algorithm described in Section III-C is used to obtain the information corresponding to the native methods in the *JNIScript* script. Finally, it builds the Server code of *JNFuzz* from the information obtained above, and the implementation is described in Section IV-C.

C. JNFuzz

JNFuzz is an innovative fuzzer based on C/S architecture for Android native libraries.

Implementation of JNFuzz: The native function was executed by invoking the native function address with constructed JNI type parameters values. These parameters include: JNIEnv*, jclass/jobject, tainted parameters (provide default initial values according to Source API), constant parameters, and uncertain parameters. Among them, we obtain the *JNIEnv** via initialization of the Java VM, and utilize it to get the second parameters *jclass/jobject*. For uncertain parameters, *JNFuzz-Droid* cuts a segment of bytes from the seed according to the uncertain parameters type sequentially, and converts this segment of bytes to the corresponding JNI type value.

D. VA-Trace

In order to track and analyze the sensitive data flow in native code, for the first time, *JNFuzz-Droid* introduces a taint analysis module Value Address Trace (VA-Trace) for the native world in the **Server**, based on the Frida framework [49]. VA-Trace consists of the following step.

Identifier design: We identify and recognize functions in native code to intercept them. ❶ For native functions, we can identify them by exporting the function symbol or

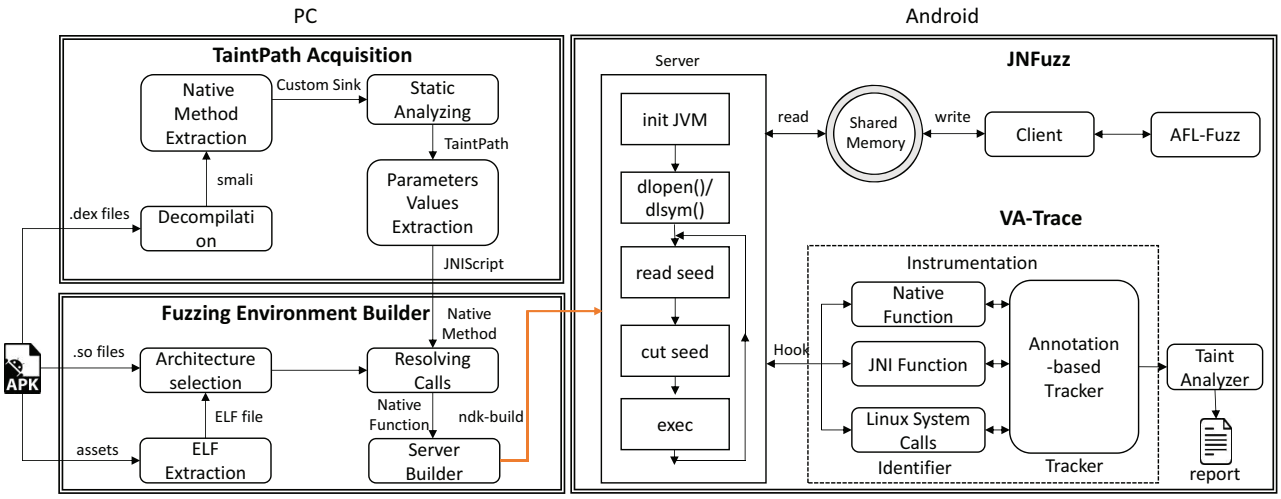

Fig. 5. Workflow of JNFuzz-Droid

the corresponding function address. ❷ For *JNINativeInterface* functions, we can identify them at a fixed offset using *JNIEnv**. ❸ For Linux system calls, we can identify them by exporting the function symbol.

Tracker design: We model the taint propagation semantics of the selected functions (as shown in Table I) in native code to enable tracking. Native sink functions have a * label.

TABLE I
List of Library Calls for Taint Propagation

Category	Library Calls
JNINativeInterface	Call<type>Method(v)*, FindClass, ...
Print & Log	__android_log_print*, (f|s|sn|vsn|vs|vf)printf*, ...
Network	gethostbyname, recv(from), connect, send(to)*, listen, socket, bind, accept, select, write*, ...
File Operation	(f)write*, (f)open, (f)read, fput(c|s)*, fdopen
Dynamic Loading	dl(m)open, dl(v)sym
Monitoring	inotify_init, inotify_add_watch, inotify_rm_watch
Execution	execcl(p|e)*, execv(p|e)*, system, popen
Process Management	kill, ptrace, fork, get(e)uid, get(e)gid, get(p)pid, ...
Time	clock_gettime, gettimeofday, ...
File Permission	(f|l)chmod, (f|l)chown, access
Other	strlen, str(n|case|ncase)cmp, str(n)cpy, strstr, str(r)chr, strcat, strdup, strto(u)l, strtod, memcpy, malloc, memset, memcmp, memchr, memmove, atoi, atol, atoll

Using the function *GetStringUTFChars()* as an example, Fig. 6 illustrates how to model its taint propagation operation. *GetStringUTFChars* is the 170th element of *JNINativeInterface*. Therefore, its offset to *JNIEnv** is 169*4 = 674 = 0x2A4. We first identify this offset address, and then propagate the *TaintAnnotation* from the first parameter to the return value by value and address.

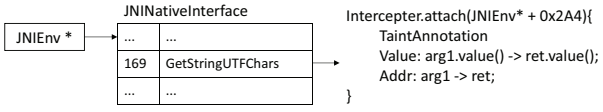

Fig. 6. GetStringUTFChars Function Model

In addition, *JNFuzz-Droid* may need to modify the parameters and/or the return values of system calls to explore native code paths. For example, the C&C server may be shut down while executing native code. In this case, *JNFuzz-Droid* replaces some return values during network connections to trigger malicious payloads. Similarly, we also modified some *Monitoring* and *Process Management* APIs to explore the native code paths.

Analyzer design: We analyze the *TaintAnnotation* during each execution of native code by *JNFuzz*. Based on the values and address information of the functions in the *TaintAnnotation*, the taint propagation paths are generated to determine if sensitive data leaked or passed in the native functions.

V. EVALUATION

We perform experiments to answer the following research questions (RQ):

RQ1: What is the current state of native library usage in real-world apps?

RQ2: How does *JNFuzz-Droid* perform on benchmark apps?

RQ3: How does *JNFuzz-Droid* perform in real-world apps?

All experiments were performed on a notebook with Ubuntu 20.04, AMD Ryzen 7 4800H@2.90GHz and 16G RAM, and a OnePlus 5T terminal with Android 8.1.0.

A. RQ1: What is the current state of native library usage in real-world apps?

This section presents general statistics about native library usage in both benign and malicious apps in recent years.

Dataset: We selected ❶ the 3682 Android apps from *F-Droid* [51] (the latest versions as of May 2022) as a sample of benign apps and ❷ the 3549 Android apps from *AndroZoo* [52] (from Jan 2020 to Sep 2022) as a sample of malicious apps (where we consider an app as malicious when at least 20 Antivirus engines in *VirusTotal* [53] have flagged it).

Results: Table II shows the investigation results. They indicate that 808 (i.e., 21.9%) of benign apps contain native method declarations and 799 (i.e., 21.7%) of benign apps

TABLE II
Native Library Usage

	F-Droid	AndroZoo		F-Droid	AndroZoo
Total App	3682	3549			
Has Native Method	808	2829	/Total App	21.9%	79.7%
Has .so File	799	826	/Total App	21.7%	23.3%
Has ELF in asset	39	769	/Total App	1.1%	21.7%
Has Encrypted zip	0	19	/Total App	-	0.5%
Has Native Activity	3	0	/Total App	0.1%	-
Total Native Method	127300	330477	/Has Native Method	157.5	116.8

contain at least one .so file. Regarding malware, 2829 (i.e., 79.7%) of apps contain native method declarations, and 826 (i.e., 23.3%) of apps contain at least one .so file. This means the .so file is probably downloaded at runtime or hidden under non-standard (i.e., *assets*) folder. We also analyzed these folder and found that 769 malware apps and 39 benign apps hide .so files in the *assets* folder by modifying the extension or compressing them. In addition, only a few apps used Native Activity components, so *JNFuzz-Droid* did not consider this situation.

RQ1 answer: The native method declarations are substantially more common in malware (79.7% vs. 21.9%). These results indicate the need to gain insight into the native code and understand its behavior.

B. RQ2: How does JNFuzz-Droid *perform on benchmark apps?*

This section presents performance of *JNFuzz-Droid* compared with *JN-SAF*, and *JuCify* on common benchmarks and hand-crafted dataset *CCBench*. Since all benchmark apps were very small, the data leakage paths could be found within a very short duration, so we set the fuzzing time budget to 1 minute.

1) RQ2.a: Results on Benchmarks

We have collected benchmarks that evaluate the inter-language data flow analysis capability of Android static analysis tools. Based on these benchmark, we utilize *AmanDroid* to pick out apps that may have a data leakage path from Source API to native methods. As a result, 14 apps were selected from *NativeFlowBench* [42], 4 apps from *BenchApps* [54] and 3 apps from *DroidBench3.0* [55]. On these 21 apps, we compare the effectiveness of *JN-SAF* [2], *JuCify* [7] , and *JNFuzz-Droid*. We run each tool against all the benchmark apps to check if the tool could correctly report the potential data leakage and corresponding data paths. Results of analysis are presented in Table III.

TABLE III
Results on Benchmarks

TP = True Positive, FP = False Positive, FN = False Negative

Benchmark (Num)	JN-SAF			JuCify			JNFuzz-Droid		
	TP	FP	FN	TP	FP	FN	TP	FP	FN
NativeFlowBench (14)	12	0	1	1	0	12	7	0	6
BenchApps (3)	0	0	3	3	1	0	3	0	0
DroidBench3.0 (3)	0	0	3	1	0	2	3	0	0
Precision	100%			83.3%			100%		
Recall	63.2%			26.3%			68.4%		
F_1-score	77.4%			40.0%			81.2%		

Results: *JN-SAF* performed best on *NativeFlowBench* while detecting nothing on other ones. *JuCify* did well on *BenchApps* but only found one leak on other ones. Due to our current inability to handle native function parameters with custom types, *JNFuzz-Droid* had 6 false negatives, while we detected all leaks on other ones.

RQ2.a answer: The above benchmarks are presented to detect specific tasks and may not satisfy other analysis tools. Overall, the recall and F1 score rates of *JNFuzz-Droid* are higher than the state-of-the-art static analysis tools.

2) RQ2.b: Results on CCBench

The above benchmarks are all designed for inter-language data flow analysis capabilities and do not analyze data leakage of the native world in many common scenarios. For evaluation purposes, the *CCBench* dataset was designed to test native world data leakage in apps under common scenarios. *CCBench* contains 23 apps categorized into four parts: Part A: inter-language control flow analysis challenges: conditional control and loop control [2]. Part B: Linux system library calls modeling [3]. Part C: ARM architecture and threading issues [56], [57]. Part D: native method overloading and others [58], [59]. To compare with existing static analysis tools, we used *__android_log_print* as a sink method to evaluate *JN-SAF* and used native callbacks to invoke the *Log* function as a sink method to evaluate *JuCify*. The analysis results of the three tools on *CCBench* are shown in Table IV.

Results: *JNFuzz-Droid* correctly analyzed all apps. Due to space constraints, we have only partially described *CCBench*.

JN-SAF uses Angr [25] to recover precise CFG (called *CFGAccurate*) for native code. *CFGAccurate* does not perform constraint solving when encountering branches. To ensure the integrity of CFG, *CFGAccurate* introduces enforced execution, which enforces the execution each edge. It will result in false positive (e.g., smt, switch) and false negative (e.g., explosion_path, while) outcomes.

JuCify is a framework that combines Android bytecode and native code into a unified model (i.e., global call graph) to detect data leaks. The native code analysis is built on top of *Angr*, while the Java code analysis and unified model rely on the *Soot* [60] framework. However, *Soot* framework, designed for the Java code, lacks support for long long type in the native code, causing the analysis of app *atoll* to fail.

In summary, neither *JN-SAF* nor *JuCify* could handle Linux system calls functions (e.g., tcp_client), model complex expressions (e.g., math_library), and take sufficiently into consideration native method overloading.

RQ2.b answer: In the above scenarios, existing native static analysis techniques expose some currently unsolved problems, while *JNFuzz-Droid* can successfully analyze all apps based on fuzzing.

C. RQ3: How does JNFuzz-Droid perform in real-world apps?

This section presents the performance of *JNFuzz-Droid* compared with *JN-SAF* and *JuCify* in real-world apps. *JN-SAF* and *JuCify* used the default configuration, and the fuzzing time

TABLE IV
Results on CCBench

O = True Positive, * = False Positive, X = False Negative

App Name	JN-SAF	JuCify	JNFuzz-Droid
Part A: Inter-language Control Flow			
explosion_path	X	O	O
smt	*		
switch	*		
while	X	O	O
math_library	X	X	O
condition	X	O	O
weak	X	O	O
Part B: Linux System Call			
atoll	X	X	O
strcpy	X	O	O
strcpy1	X	X	O
tcp_client	XX	OX	OO
udp_client	XX	OX	OO
Part C: Architecture and Thread			
armeabi	O	O	O
armeabi-v7a	O	O	O
arm64-v8a	X	O	O
thread_leak	O	O	O
thread_in_leak	O	O	O
thread_noleak			
Part D: Overloading and Misc			
native_method_overloading	O	X	O
native_method_overloading1	X	O	O
global_imei	O	O	O
interrupt_cfg	X	O	O
log_noleak		*	
Sum, Precision and Recall			
O, higher is better	6	15	21
*, lower is better	2	1	0
X, lower is better	15	6	0
Precision p = O/(O + *)	75.0%	93.8%	100%
Recall r = O/(O + X)	28.6%	71.4%	100%
F_1-score = 2pr/(p + r)	41.4%	81.1%	100%

budget was set to 10 minutes (about 90000 times) for *JNFuzz-Droid*, since *JNFuzz-Droid* did not fuzz the whole app but a single native function. For the result, we only counted the data flow associated with native code.

Dataset: We utilized dataset which included ❶ randomly selected 2500 malware apps (1596 of which contain native code) from *AndroZoo* and ❷ all malware apps (1929 apps, 591 of which contain native code) from Android Botnets [61] as supplementary samples, to encompass as many scenario and situations as possible.

Results: *JN-SAF* found 4 apps with issues, *JuCify* did not find any issues. The major reason for this result is that the complexity and scale of native code in real apps make it impossible for them to complete the analysis in the default time budget. For example, *JN-SAF* always fails to generate the corresponding summary within 5 minutes.

JNFuzz-Droid found 56 apps with security issues. Among them, 30 apps used classic encryption (in Section V-C1), 12 apps used uninstall (in Section V-C2), and 12 apps used hard coding (in Section V-C3). Furthermore, one app used invoke `Log()` function to print `IMEI`, and another [1] used invoke `execlp` command (i.e., `am start -a android.intent.action.VIEW -d IMEI`) in native code to leak sensitive data.

In the following, we discuss three case studies on how *JNFuzz-Droid* revealed sensitive data behaviors in native code.

1) Case Study1: Inter-language Data Transmission

Some malware [2] obfuscates or encrypts the sensitive data passed from Java code to native code to evade detection. *JNFuzz-Droid* revealed that Java code passes `IMSI` to native method `encrStringd()` (the corresponding native code is stored in the *assets* folder), which encrypts `IMSI` and returns the encrypted value. In fact, the encrypted `IMSI` is then sent via `SMS` to 10658423 (as shown in Fig. 7).

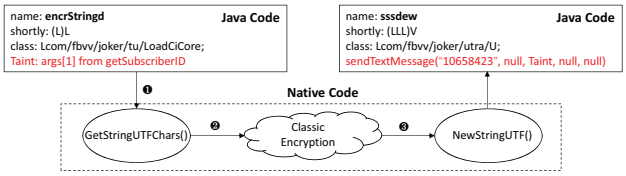

Fig. 7. Inter-language Data Transmission

Note: *FlowDroid* detects this leak with many false positives, while *JNFuzz-Droid* can dynamically validate these false positives.

2) Case Study2: Native Monitoring

Some malware [3] monitors its installation directory and leaks sensitive data, such as phone number, when it is uninstalled. *JNFuzz-Droid* revealed (as shown in Fig. 8) that Java code first invokes native method `startwork()` that passes tainted data related to the phone number to its native code. Then, the native code creates a child process and utilizes `inotify` to detect the app's uninstallation. Once uninstalled, the malware sends out the phone number via a network connection.

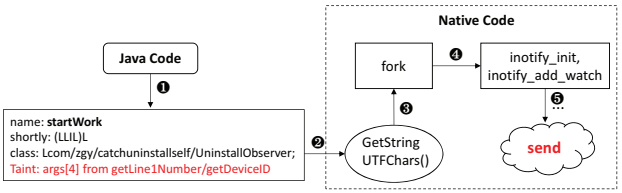

Fig. 8. Inter-language Data Leakage

3) Case Study3: Native Hard-code

Some malware [4] uses hard-code to hide the receiver of sensitive data into native code. *JNFuzz-Droid* found (as shown in Fig. 9) the function begin fuzzed always returns a sensitive string such as the email address. We consider it suspicious because of the intuition that these strings are related to private data leakage. In fact, this malware leaks user's private data (i.e., phone number) via these email addresses or URLs.

[1] MD5:2397094dba5801544b3147a85b36bd19
[2] MD5:bc0feca701548fecde8d7d6f0d3af8f2
[3] MD5:ea2e8b3aee42c6398e825aba0707da29
[4] MD5:1ddf1e0ffae94902e4a2a83cf4ef5fff

```
C1.  jstring Java_com_qwe_MainAct_stringIPBank(){
C2.    return env->NewStringUTF("http://eruktfrhk.iego.net/appHome/");
C3.  }
C4.  jstring Java_com_qwe_MainActiv_stringUser(jstring str){
C5.    const char* mess = env->GetStringUTFChars(str, 0);
C6.    return env->NewStringUTF("uhg@vip.126.com");
C7.  }
C8.  jstring Java_com_qwe_MainActiv_Password(jstring str){
C9.    const char* mess = env->GetStringUTFChars(str, 0);
C10.   return env->NewStringUTF("67rg5r4hre545");
C11. }
```

Fig. 9. Native Hard-code

RQ3 answer: Experiments indicate *JNFuzz-Droid* can highlight the data flow in native code more effectively than existing tools and discover sensitive data leaks or transfers in real-world apps.

VI. LIMITATIONS

JNFuzz-Droid is a step forward in the field of research to detect sensitive data leakage and delivery in Android native code, and presents a few limitations despite the promising performance. First, *JNFuzz-Droid* relies on existing tools to find potential data leak paths from Source API to native methods. Limitations of them are carried over to *JNFuzz-Droid*. These limitations include Java reflection, dynamic loading [62], and so on.

Second, the data flow analysis of *JNFuzz-Droid* is based on Frida framework. This, however, may pose a challenge as malware may detect the existence of Frida and halt the execution of malicious payloads. For example, malware can execute the shell command *netstat* or *ss* to check whether the TCP port 27042 is occupied. To address this challenge, we could modify the port or `/proc/pid/maps` mapping file. Nevertheless, this is an arm race between analysis tools and anti-analysis techniques.

Third, our framework currently employs individualized fuzzing, which is influenced by the state of the Java world. Additionally, current parameter type modeling only considers primitive type, primitive array and some interactive types (e.g., android.content.Intent), which leads to possible failures when analyzing native functions with custom type or other complex type. These factors combined contribute to the less efficient exploration of fuzzing in the *JNFuzz-Droid* framework. In future work, we will enhance static analysis, determine the specific values of parameters, and incorporate binary rewriting techniques [63] or feedback mechanisms.

VII. RELATED WORK

Static Native Analysis. Due to the complexity of native code, there are only a few static analysis-based approaches. *JN-SAF* [2] is an inter-language static analysis framework to detect sensitive data leaks in Android apps. CTAN [64] and SANT [57] implement distinct extensions to *JN-SAF* to improve its performance. *JuCify* [7] is a framework that combines Android bytecode and native code into a unified model to detect data leaks. However, the above tools have limitations in real-world apps due to issues such as path explosion and constraint solving. In addition, there are static techniques for analyzing native code. LibDroid [65] presents a new approach conduct to native data-flow analysis. George et al. [66] propose an approach to recover JNI callbacks in the native code. Luca et al. [56] propose DroidReach, a novel static approach to assess the reachability of native function calls in Android apps.

Dynamic & Hybrid Native Analysis. NDroid [3] is a novel dynamic taint propagation tool based on *QEMU* [67], which tracks JNI and system library functions in Java and native code. Harvester [19] is a hybrid analysis tool that combines static backward slicing to identify interesting code with the execution of the code for extracting runtime values. Malton [24] runs on real mobile devices, and it is a dynamic analysis platform built on *Valgrind* [68] for malware detection based on flow tracking of Java and JNI code. Going Native [5] proposed a new method to generate a native code sandboxing policy automatically.

Binary Code Analysis. Angr [25] is a binary analysis platform combining various program analysis techniques, including static analysis, symbolic execution, and binary lifting. Frida [49] is a dynamic instrumentation toolkit for reverse engineering and manipulating Android and iOS apps, enabling JavaScript code interaction for internal app functions. *JNFuzz-Droid* is based on Frida to track the flow of sensitive information in native code. Qiling [69] is a binary emulation and analysis framework offering advanced instrumentation for complex apps. However, it does not currently emulate JNI functions. QBDI [70] is a dynamic binary instrumentation framework providing low-level control over binary code execution, supporting multiple OS and architectures but lacking ARM analysis.

VIII. CONCLUSION

We have investigated existing Android native analysis tools and summarised their advantages and disadvantages. Based on these insights, we proposed and implemented *JNFuzz-Droid*, a lightweight automated fuzzing and taint analysis framework for analyzing the behavior of sensitive data in the native world. Experimental results on benchmarks and a large number of real-world apps illustrated that *JNFuzz-Droid* could effectively detect the leakage or transfer of sensitive data in app native code and outperforms the existing tools *JN-SAF* and *JuCify*. Overall, the *JNFuzz-Droid* framework is designed to help quickly and accurately locate data leaks in Android native code, providing a valuable tool for security researchers and others working in the Android security space.

IX. ACKNOWLEDGMENT

We would like to express our gratitude to all those who provided assistance during the course of this research.

REFERENCES

[1] Mobile Operating System Market Share Worldwide, "https://gs.statcounter.com/os-market-share/mobile/worldwide/," 2023.

[2] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "JN-SAF: precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS*. Toronto, ON, Canada: ACM, 2018, pp. 1137–1150.

[3] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. S. Chan, "Ndroid: Toward tracking information flows across multiple android contexts," *IEEE Trans. Inf. Forensics Secur.*, vol. 14, no. 3, pp. 814–828, 2019.

[4] L. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21th USENIX Security Symposium*. Bellevue, WA, USA: USENIX Association, 2012, pp. 569–584.

[5] V. M. Afonso, P. L. de Geus, A. Bianchi, Y. Fratantonio, C. Kruegel, G. Vigna, A. Doupé, and M. Polino, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in *23rd Annual Network and Distributed System Security Symposium, NDSS*. San Diego, California, USA: The Internet Society, 2016.

[6] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in *22nd Annual Network and Distributed System Security Symposium, NDSS*. San Diego, California, USA: The Internet Society, 2015, pp. 1–15.

[7] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein, "Jucify: A step towards android code unification for enhanced static analysis," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1232–1244.

[8] H. Zhou, S. Wu, X. Luo, T. Wang, Y. Zhou, C. Zhang, and H. Cai, "Ncscope: hardware-assisted analyzer for native code in android apps," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. South Korea: ACM, 2022, pp. 629–641.

[9] Timer-checks at Android Anti-Reversing Defenses, "https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05j-testing-resiliency-against-reverse-engineering#timer-checks," 2023.

[10] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe," in *22nd Annual Network and Distributed System Security Symposium, NDSS*. San Diego, California, USA: The Internet Society, 2015, pp. 4014–4040.

[11] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: statically vetting android apps for component hijacking vulnerabilities," in *the ACM Conference on Computer and Communications Security, CCS*. Raleigh, NC, USA: ACM, 2012, pp. 229–240.

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

[13] 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," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–269.

[14] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, no. 6, p. 259–269, jun 2014.

[15] 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," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1329–1341.

[16] 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, no. 3, apr 2018.

[17] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 5:1–5:29, 2014.

[18] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber, "Artist: The android runtime instrumentation and security toolkit," in *2017 IEEE European Symposium on Security and Privacy, EuroS&P*. Paris, France: IEEE, 2017, pp. 481–495.

[19] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques," in *Network and Distributed System Security Symposium (NDSS)*. San Diego, California, USA: The Internet Society, Feb. 2016, pp. 1–15.

[20] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J. Lalande, and V. V. T. Tong, "Groddroid: a gorilla for triggering malicious behaviors," in *10th International Conference on Malicious and Unwanted Software, MALWARE*, IEEE. Fajardo, PR, USA: IEEE Computer Society, 2015, pp. 119–127.

[21] T. Kim, H. Ha, S. Choi, J. Jung, and B. Chun, "Breaking ad-hoc runtime integrity protection mechanisms in android financial apps," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS*. Abu Dhabi, United Arab Emirates: ACM, 2017, pp. 179–192.

[22] M. Sun, T. Wei, and J. C. S. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna, Austria: ACM, 2016, pp. 331–342.

[23] L. Xue, C. Qian, and X. Luo, "Androidperf: A cross-layer profiling system for android applications," in *23rd IEEE International Symposium on Quality of Service, IWQoS*. Portland, OR, USA: IEEE, 2015, pp. 115–124.

[24] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards on-device non-invasive mobile malware analysis for ART," in *26th USENIX Security Symposium, USENIX Security*. Vancouver, BC, Canada: USENIX Association, 2017, pp. 289–306.

[25] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society, 2016, pp. 138–157.

[26] LibFuzzer, "https://llvm.org/docs/LibFuzzer.html," 2023.

[27] Frida_mode at AFLplusplus, "https://github.com/AFLplusplus/AFLplusplus/tree/stable/frida_mode," 2023.

[28] QBDI_mode at AFLplusplus, "https://github.com/AFLplusplus/AFLplusplus/tree/stable/utils/qbdi_mode," 2023.

[29] The AFL++ fuzzing framework, "https://aflplus.plus/," 2023.

[30] Android NDK, "https://developer.android.google.cn/ndk/," 2023.

[31] Gradle Build Tool, "https://gradle.org/," 2023.

[32] JNI, "https://docs.oracle.com/javase/8/docs/technotes/guides/jni/," 2023.

[33] The Invocation API, "https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/invocation.html," 2023.

[34] JNI Functions, "https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/functions.html," 2023.

[35] Resolving Native Method Names, "https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/design.html#resolving_native_method_names," 2023.

[36] Monkey, "https://developer.android.com/studio/test/monkey," 2023.

[37] american fuzzy lop, "https://lcamtuf.coredump.cx/afl/," 2023.

[38] F. Pauck, E. Bodden, and H. Wehrheim, "Do android taint analysis tools keep their promises?" in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*. Lake Buena Vista, FL, USA: ACM, 2018, pp. 331–341.

[39] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe," in *Proceedings of ISSTA 2018*. Amsterdam, The Netherlands: ACM, 2018, pp. 176–186.

[40] S. A. Rodriguez and E. van der Kouwe, "Meizodon: Security benchmarking framework for static android malware detectors," in *Proceedings of the Third Central European Cybersecurity Conference, CECC*. Munich, Germany: ACM, 2019, pp. 8:1–8:7.

[41] J. Zhang, Y. Wang, L. Qiu, and J. Rubin, "Analyzing android taint analysis tools: Flowdroid, amandroid, and droidsafe," *IEEE Trans. Software Eng.*, vol. 48, no. 10, pp. 4014–4040, 2022.

[42] NativeFlowBench, "https://github.com/arguslab/NativeFlowBench," 2023.

[43] D. Li, Y. Lyu, M. Wan, and W. G. J. Halfond, "String analysis for java and android applications," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*. Bergamo, Italy: ACM, 2015, pp. 661–672.

[44] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Static Analysis, 10th International Symposium, SAS*, ser. Lecture Notes in Computer Science, vol. 2694. San Diego, CA, USA: Springer, 2003, pp. 1–18.

[45] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 77–88.

[46] R. Vallee-rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," 1998.

[47] RegisterNatives, "https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/functions.html#RegisterNatives," 2023.

[48] nativediscloser, "https://github.com/gaojun0816/nativediscloser," 2023.

[49] Frida, "https://frida.re/," 2023.

[50] Apktool, "https://ibotpeaches.github.io/Apktool/," 2023.

[51] F-Droid, "https://www.f-droid.org/," 2023.

[52] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471.

[53] VirusTotal, "https://www.virustotal.com/," 2023.

[54] JuCify_benchApps at master, "https://github.com/JordanSamhi/JuCify/tree/master/benchApps," 2023.

[55] DroidBench3.0, "https://github.com/secure-software-engineering/DroidBench/tree/ddbd50c68dde18e1f6b75bfa13c617f986ba9e46," 2023.

[56] L. Borzacchiello, E. Coppa, D. Maiorca, A. Columbu, C. Demetrescu, and G. Giacinto, "Reach me if you can: On native vulnerability reachability in android apps," in *Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security*, ser. Lecture Notes in Computer Science, vol. 13556. Copenhagen, Denmark: Springer, 2022, pp. 701–722.

[57] S. B. Andarzian and B. T. Ladani, "SANT: static analysis of native threads for security vetting of android applications," *ISC Int. J. Inf. Secur.*, vol. 14, no. 1, pp. 13–25, 2022.

[58] Design Overview - Resolving Native Method Names, "https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/design.html#wp615," 2023.

[59] JNI Functions - Global and Local References, "https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#global_local," 2023.

[60] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, p. 13.

[61] A. F. A. Kadir, N. Stakhanova, and A. A. Ghorbani, "Android botnets: What urls are telling us," in *Network and System Security - 9th International Conference, NSS 2015*, ser. Lecture Notes in Computer Science, vol. 9408. New York, NY, USA: Springer, 2015, pp. 78–91.

[62] Y. Xue, G. Meng, Y. Liu, T. H. Tan, H. Chen, J. Sun, and J. Zhang, "Auditing anti-malware tools by evolving android malware and dynamic loading technique," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 7, pp. 1529–1544, 2017.

[63] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy, SP*. San Francisco, CA, USA: IEEE, 2020, pp. 1497–1511.

[64] S. B. Andarzian and B. T. Ladani, "Compositional taint analysis of native codes for security vetting of android applications," in *2020 10th International Conference on Computer and Knowledge Engineering (ICCKE)*. Mashhad, Iran: IEEE Computer Society, 2020, pp. 567–572.

[65] C. Shi, C. C. Cheng, and Y. Guan, "Libdroid: Summarizing information flow of android native libraries via static analysis," *Digit. Investig.*, vol. 42, no. Supplement, p. 301405, 2022.

[66] G. Fourtounis, L. Triantafyllou, and Y. Smaragdakis, "Identifying java calls in native code via binary scanning," in *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual Event, USA: ACM, 2020, pp. 388–400.

[67] QEMU, "https://www.qemu.org/," 2023.

[68] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*. San Diego, California, USA: ACM, 2007, pp. 89–100.

[69] Qiling Framework, "https://qiling.io/," 2023.

[70] QBDI - QuarkslaB Dynamic binary Instrumentation, "https://qbdi.quarkslab.com/," 2023.