Fuzzing Android Native System Libraries via Dynamic Data Dependency Graph

Xiaogang Zhu, *Member, IEEE*, Siyu Zhang, Chaoran Li, Sheng Wen, *Senior Member, IEEE*, and Yang Xiang, *Fellow, IEEE*

***Abstract*— Google suggests using only the APIs documented in Android SDK. However, many app developers still choose Java Native Interface (JNI) to access system libraries because of the flexibility and freedom that non-SDK methods provide in implementing complex functions. However, using JNI may have unexpected consequences, including low-level bug-driven crashes. The bugs in system libraries can propagate to Android apps, and further cost much time and energy for developers to debug them. We develop a fuzzing tool, called JDYNUZZ, that exposes the bugs in system JNI to mitigate the aftermath of direct invocation of JNI. To fuzz a system library, one needs to not only prepare appropriate inputs, but also deal with the challenge of maintaining a correct sequence of API calls, both syntactically and semantically. To solve the challenge, the crux of JDYNUZZ is the dynamic refinement of a data dependency graph, which gradually resolves the problem of syntactic and semantic incorrectness when constructing API sequences. JDYNUZZ achieves the dynamic refinement based on the feature of Java reflection, which enables us to dynamically modify API sequences and test different code regions. We evaluate JDYNUZZ on the most recent version of Android Open Source Project (AOSP), *i.e.*, version android-12.0.0_r31. In our experiments, JDYNUZZ discovers 34 new bugs in system JNI libraries, all confirmed by Google.**

***Index Terms*— Android native library, fuzzing, dynamic data dependency.**

I. INTRODUCTION

AS THE most popular mobile operating system, Android has more than 71% of the global smartphone market share [1]. As of mid-2022, close to 3.3 million apps were available on Google Play [2], which is the largest yet not the only app store in business. To maintain an acceptable level of app robustness in such a large market, Google suggests using only the officially documented classes in its Android SDK for app development. Ignoring this suggestion may have consequences, including software crashes that are triggered by the bugs existing in low-level system libraries. Recent investigation indicated that Android users can tolerate a daily crash rate of 0.25 percent for an app, but most of them will uninstall the app if the crash rate is doubled [3]. A common violation of Google's advice on SDK is calling system libraries directly through Java Native Interface (JNI), since such a non-SDK approach provides more flexibility in implementing complex functions. Our empirical study (Sec. IV-A) indicates that over 66.7% Android apps in Google Play access system libraries via JNIs, leaving their users to live with the consequences.

Direct unsafe calls to system libraries through JNI trigger the underlying bugs that may, for example, cause the aforementioned app crashes. The issues caused by such bugs may confuse developers since they are rooted in system libraries. Our research (reported in Sec. IV-A) indicates that 88,412 questions on Stack Overflow [4] are about crashes in native libraries, including system libraries. Native system libraries are usually compiled without the additional symbols and information required in debugging (*e.g.*, code lines). Even a knowledgeable developer has to spend a tremendous amount of energy and money in debugging and fixing such bugs. It is thus critical for developers to be aware of unsafe calls. Moreover, bugs, particularly those associated with memory in system libraries, can result in severe consequences such as denial of service or remote code execution.

In this paper, we propose to develop a novel fuzzing tool, called JDYNUZZ, to expose the bugs triggered by unsafe calls in Android system libraries. In addition to allowing Google to fix bugs before they result in severe consequences, the list of unsafe calls can also be shared with app developers, informing them of potential risks when invoking certain calls via JNI. The key to the success of JDYNUZZ is to solve the problem of data dependency [5]. Correct data dependency allows the fuzzer to test a code deep in the system library, which in turn enables the detection of bugs hidden in complex software execution paths. To the best of our knowledge, most existing works for analyzing Java programs focus on either statically constructing precise call graph [6], [7], [8], [9], [10], [11] or dynamically testing single interfaces of Android servers [12], [13]. These works ignore the data dependencies among different interfaces, making the test on interfaces insufficient. The most related work is a fuzzing tool, called FANS, which has considered the dependency of interfaces in the multiple level Android Interface Definition Language (AIDL) [14]. However, the idea of FANS cannot be borrowed/copied in JDYNUZZ's case because of the distinct features of AIDL. For example, FANS relies on Binder-related methods to infer the sequence of calls, but it does not suit for JDYNUZZ since those methods are missing in JNI.

Manuscript received 30 June 2023; revised 27 November 2023; accepted 6 February 2024. Date of publication 23 February 2024; date of current version 26 April 2024. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Chunyi Peng. *(Corresponding author: Chaoran Li.)*

The authors are with the School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, VIC 3122, Australia (e-mail: xiaogangzhu@swin.edu.au; siyuzhang@swin.edu.au; chaoranli@swin.edu.au; swen@swin.edu.au; yxiang@swin.edu.au).

Digital Object Identifier 10.1109/TIFS.2024.3369479

In JDYNUZZ, we propose to develop a dynamic data dependency graph (D3G), which is statically initialized and then dynamically refined based on Java reflection. To the best of our knowledge, we are the first to consider dynamic data dependency graph in fuzzing Android system [5]. Specifically, each node in the D3G contains the necessary information of generating an API or object. The D3G is initialized based on the fact that a variable must be initialized or assigned with a value before usage. As for the dynamic refinement of the D3G, JDYNUZZ utilizes Java reflection to dynamically insert, prune or modify the graph. To improve the efficiency, JDYNUZZ is designed with a server-client model, where the server maintains the D3G and the client executes commands based on Java reflection. Due to the nature of Java reflection, the client can invoke any call sequence, which is sent from the server, without recompiling the client. In this way, JDYNUZZ is able to dynamically refine the D3G with the feedback from the execution result of the client. APIs or objects that are successfully executed will be saved in the D3G. Meanwhile, failure of object creation or API execution increases the likelihood that nodes will be disregarded.

We implement a prototype of our approach to demonstrate that JDYNUZZ can be used to test the JNI libraries in the Android system. The fuzzing is performed by JDYNUZZ on Android Open Source Project (AOSP), version android-12.0.0_r31. We compile the whole system first and extract the framework JNI model through the compiling output, *e.g.*, compiling command and internal jars. There are more than 4,800 JNIs in Android system libraries and 193 unique type of objects. Finally, JDYNUZZ discovers 34 new bugs in the JNI libraries, and they are confirmed by the vendor.

The main contributions of this paper are as follows:

- We design and implement a prototype JDYNUZZ to facilitate Android native system libraries fuzzing.
- Our results indicate that the proposed D3G in JDYNUZZ can improve the efficiency of fuzzing.
- JDYNUZZ can identify 34 new unique bugs and unsafe calls on the most recent release AOSP (android-12.0.0_r31).
- We release the tool, JDYNUZZ, to benefit the community at https://github.com/SioYooo/JDYNUZZ.

This work can benefit OS designers, software engineering developers, and the research community. For OS designers, Google can improve the SDK design or enhance the management of non-SDK APIs. This can be achieved either by strengthening the restrictions on blacklisted items or by addressing bugs in native libraries. For developers and the research community, we provide a tool to test the JNI libraries, which can help them implement a more robust app.

II. BACKGROUND

This section provides background information of JNI and Java reflection in Android.

A. JNI

The JNI [15] is a mechanism that allows Java code to invoke or be invoked by native apps and libraries programmed in other programming languages, *e.g.*, C/C++. Java variables including objects and basic type variables are passed to native methods through JNI. The variables can also be read or written by native methods. JNI helps developers to handle the situations when an app cannot be coded by sole Java language, *e.g.*, functionalities that involve low-level code. Although JNI is convenient for developers, it may confuse developers between Java methods and native methods if they use the feature of reflection. One of the consequences is that developers may invoke APIs in system libraries via JNI without being aware of it. As a result, apps invoking system libraries may have unexpected errors that are rooted in system libraries. Another problem for the JNI mechanism is that analyzers need to be knowledgeable of at least two programming languages (due to the cross-language feature) in order to analyze the relationships among APIs in Jave-side. Since the relationships between APIs are critical when fuzzing JNI, the feature of cross-language increases the engineering works for fuzzing.

B. Java Reflection in Android

The Java reflection can manipulate internal properties outside the program itself. For example, in Android, an app can inspect or modify the members of the `.jar` files that are outside the app (*e.g.*, non-SDK APIs in `android.jar`). Specifically, Java reflection can obtain the class and the function inner the class through the strings (*e.g.*, class name, function name). Furthermore, functions can be invoked through the methods provided by Java reflection. Normally, Google prohibits app developers from calling non-SDK APIs via Java reflection. However, developers can still break the restrictions, such as compiling apps with `android.jar` that has non-SDK APIs. In the meanwhile, Google adapts a variety of ways to restrict non-SDK API invocation including API check in compilation, and API blacklist in run-time.

C. Fuzzing

Fuzzing generally consists of three components that are input generator, execution environment, and defect monitor [5], [16]. The input generator is responsible for generating numerous inputs. The input generator is also responsible for constructing fuzz drivers, which are the executable programs that call APIs (Application Programming Interfaces) [17]. Fuzzing tests a target program at runtime, which requires fuzzers build appropriate running environment. It is straightforward for fuzzing to test command-line programs [18]. Yet, it requires many engineering efforts to perform fuzzing on many applications, such as IoT devices [19] or Android systems.

D. Break Restriction on Non-SDK APIs

Google undoubtedly prohibits apps from calling non-SDK APIs. Thus, Google adapts a variety of methods to restrict Non-SDK API calls, including `android.jar`, blacklist policy, *etc*. However, programmers and hackers will always find a way to circumvent the restriction. The restriction breaking begins with SDK replacement, which replaces

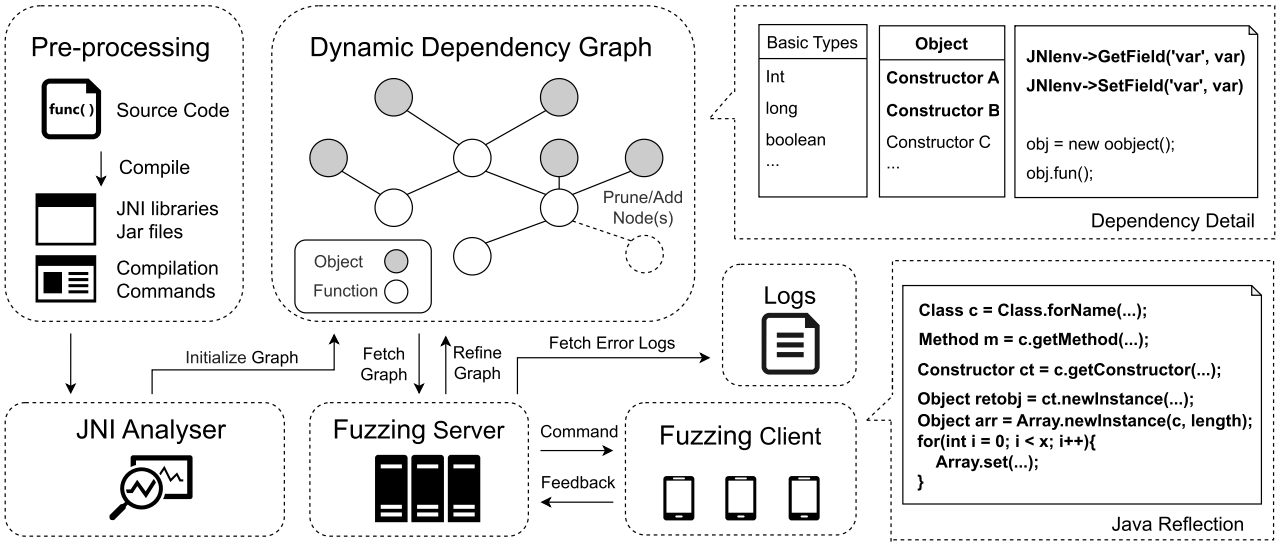


Fig. 1. A overview of the four-component JDYNUZZ: Pre-processing, JNI Analyzer, Fuzzing Server, and Fuzzing Client.

`android.jar` with a full API version during the development phase, thereby making APIs accessible to apps during the development and compilation phases. With the upgrading of restriction, the more advanced restriction breaking way is to invoke non-SDK APIs with Java and NDK reflection. When the information of the target API is known, such as the API class and API signature, a developer can locate and invoke the API using the provided hooking APIs in Java and NDK. As restrictions intensify, more methods of evasion emerge, such as double reflection and call stack breaking [20]

E. Motivation Example

The bugs in native system libraries not only pose threats for Android apps but also cost much for app developers to debug them. Android Studio (AS) [21] is the most popular Android development integrated development environment (IDE), which provides a debugger with both Java and native languages. However, if an app invokes non-SDK APIs, especially system JNI, the native debugger provided by AS is not available. This is because the debugger can only be attached on the native libraries built in the app project. For the debugging of code outside an app, Android officially provides a solution [22] that attaches and loads symbols with LLDB [23] or GDB [24]. However, the preparation of debug and the analysis of native libraries are quite complex for a Java Android developers. As for the developers, they need to download and compile AOSP following the building document [25]. Then, the document providing debug solutions tells developer to run the script `gdbclient.py`. However, the result of our attempt is that the script (in AOSP version android-12.0.0 r31) cannot successfully run. It costs us a long time to fix the bug inside the script. With the bug-free `gdbclient.py`, a developer can connect to LLDB or GDB server running in the Android device. When finishing the preparation, the developer need to set breakpoints and track the execution inside native libraries to figure out the root cause of a bug. The execution probably involves Android inter-process communication (IPC), such as binder and AIDL. Therefore, we propose JDYNUZZ to fuzz the unsafe calls targeted on Android native system libraries.

III. THE DESIGN OF JDYNUZZ

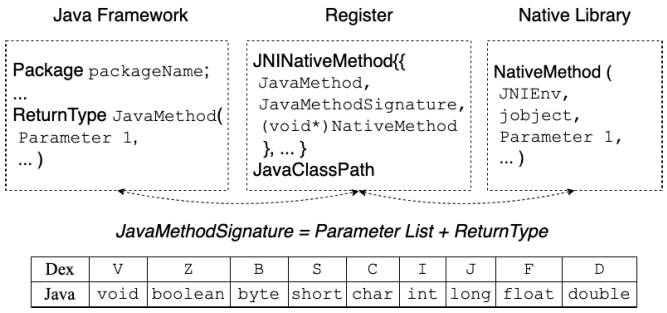
To fuzz Android native system libraries, one may consider directly invoking the native-side JNI functions, so as to reuse the tremendous tools designed for fuzzing the C/C++ program. However, to achieve this, the JVM environment needs to be constructed and passed into the native-side JNI function, which involves numerous engineering efforts, and more importantly, sacrifices the fuzzing efficiency. In addition, directly invoking the native-side JNI functions bypasses JNI, through which a real-world app invokes the native system libraries. Consequently, such fuzz testing may also miss bugs in the JNI.

To this end, we propose and implement a prototype tool, JDYNUZZ, to fuzz Android native system libraries via JNI on the Java-side rather than directly fuzzing the native-side. JDYNUZZ is designed as a distributed model, which can simultaneously support fuzzing for multiple devices. Figure 1 depicts the overall structure of JDYNUZZ. As illustrated, JDYNUZZ consists of four modules, which are *Pre-processing*, *JNI Analyser*, *Fuzzing Server*, and *Fuzzing Client*. The *Pre-processing* module prepares the JNI libraries and compilation commands for JNI Analyser to process the source code of the Android system. *JNI Analyser* extracts the information of JNI functions for fuzzing tests, *e.g.*, the lists of Java native methods. Moreover, the Analyser can infer the dependencies among JNIs and initialize the D3G with static analysis. *Fuzzing Server* acts as the role of manager in JDYNUZZ. It instructs the execution of *Fuzzing Client* and receives the feedback and error logs from the *Fuzzing Client*. More importantly, *Fuzzing Server* refines the D3G for a more precise and effective fuzz testing. The *Fuzzing Client* module receives and executes the commands from *Fuzzing Server* and sends the feedback to it. The commands include the sequence of JNI invocation, the construction of objects, and the basic variables. Moreover, the *Fuzzing Client* invokes Java methods via reflection so as to facilitate fuzzing without recompilation.

```
1  // com_hello.java
2  package com.hello;
3  public class HelloJNI {
4    static{
5      System.loadLibrary("native_side");
6    }
7    public native void staticLog();
8  }

9  // native_side.cpp
10 JNIEXPORT void JNICALL
11 Java_com_hello_HelloJNI_staticLog (JNIEnv *env, jobject obj) { ... }
```

Fig. 2. Static registration JNI code example.



Dex	V	Z	B	S	C	I	J	F	D
Java	void	boolean	byte	short	char	int	long	float	double

Fig. 3. Dynamic JNI communication model.

```
1  // com_android_internal_net_NetworkUtilsInternal.cpp
2  static void android_net_utils_setAllowNetworkingForProcess(
3    JNIEnv *env, jobject thiz, jboolean hasConnectivity) {
4    ...
5  }

6  static const JNINativeMethod gNetworkUtilMethods[] = {
7    {"setAllowNetworkingForProcess",
8     "(Z)V",
9       (void *)android_net_utils_setAllowNetworkingForProcess},
10   ...
11 };

12 int register_com_android_internal_net_NetworkUtilsInternal(JNIEnv
13 *env) {
14   return RegisterMethodsOrDie(env,
15 "com/android/internal/net/NetworkUtilsInternal",
16 gNetworkUtilMethods, NELEM(gNetworkUtilMethods));
17 }

18 // NetworkUtilsInternal.java
19 package com.android.internal.net;
20 public class NetworkUtilsInternal {
21   public static native
22   void setAllowNetworkingForProcess(boolean allowNetworking);
23 }
```

Fig. 4. Dynamic registration JNI code example.

We utilize SOOT [26] and CLANG [27] for the static analysis of D3G. SOOT is a well-known Java optimization framework for analyzing Java code, and it has been extensively used in numerous static analysis studies [28], [29], [30]. CLANG is a front-end of the LLVM compiler, which we use to analyze the native-side JNI methods and infer the Java-side JNI method call sequences. In the following subsections, we describe the design and implementation of each module in detail. To discover bugs in JNI libraries, we propose a generation-based fuzzing solution and detail its design in this section. Moreover, we use Java reflection to dynamically generate objects and invoke APIs in sequences.

A. Pre-Processing

We begin by compiling the source code of Android OS with the build toolchain (*e.g.*, CMake or Ninja) provided by AOSP to obtain the intermediate artifacts (*e.g.*, `.jar` files with complete java codes). We then build commands (*e.g.*, native library compilation commands), which are required in the subsequent steps to analyze JNI-related information in the system. We are unable to automate this module due to the heterogeneity of software compilation techniques and intermediary products. Note that the pre-processing module includes most engineering works, which are not considered as our technical contribution. However, it is essential to facilitate the proposed JNI analysis.

B. JNI Analyzer

As depicted in Figure 1, after manually compiling the AOSP source code, the following modules are performed in an automatic manner. In this subsection, we introduce the JNI Analyzer that is responsible for the D3G initialization, which includes JNI identification and dependency inference.

1) JNI Identification: This step identifies and pairs the JNI methods on both Java- and Native-sides, and it also serves as the prerequisite of the static dependency inference. JDynuzz identifies JNI through modeling the JNI registrations. Specifically, there are two types of registrations involved in the Android framework, including static and dynamic registration.

a) Static JNI registration: According to the naming convention of JNI methods, native-side functions are to be named as `Java_package_class_functionName`. For instance, as shown in Figure 2, functions in lines 7 and 11 are a pair of JNI methods. The name of the native-side JNI function (line 11) is composed of the Java package name (line 2), the Java class name (line 3), and the Java-side JNI function name (line 7). The Java Virtual Machine (JVM) registers the native library for linking the native-side JNI functions (line 5).

b) Dynamic JNI registration: Figure 3 explains the dynamic JNI registration mechanism. The Register includes a collection of JNI mappings of both Java- and native-sides, which indicates the linkage between Java- and native-side JNI functions. The mapping includes 1) the Java method name, 2) the Java method signature, which is composed of the parameter list and the return type (in the Dalvik executable format [31]), and 3) the pointer to the corresponding native function in the Native Library. In addition, the Java class path, which is used to locate the Java method in the Java-side framework, is also included in the Register.

We further explain the working process of JNI identification with an example given in Figure 4, which is derived from Android framework files `NetworkUtilsInternal.cpp` and `NetworkUtilsInternal.java`. The method `NetworkUtilsInternal` (line 12) registers the JNI

Algorithm 1 Dependency Inference

Input: JavaClass C.
Output: Dependency D.

```
1:  D ← ∅
2:  L ← ∅
3:  R ← ∅
4:  for each statement ∈ JavaClass do
5:      if typeOf(statement) is Assignment then
6:          for each variable ∈ statement do
7:              if position(variable) = left then
8:                  L ← L ∪ (variable, function)
9:              else if position(statement) = right then
10:                 R ← R ∪ (variable, function)
11:     else if typeOf(statement) is Call then
12:         for each variable ∈ statement do
13:             if setValue(variable) then
14:                 L ← L ∪ (variable, function)
15:             else if getValue(variable) then
16:                 R ← R ∪ (variable, function)
17: for each (V_L, F_L) ∈ L do
18:     for each (V_R, F_R) ∈ R do
19:         if V_L = V_R then
20:             D ← D ∪ (F_L, F_R)
```

methods. It includes the Java class path (line 15), which specifies the Java-side package name (line 19) and class name (line 20). Other parameters of the registration method include the method mappings (line 16), whose declaration can be found in line 6. The mapping contains the Java-side method name (line 7), the method signature (line 8), and the pointer to the corresponding native function (line 9). Therefore, the Java- and native-side JNI functions can be extracted precisely by leveraging static analysis. Specifically, the JNI methods and their corresponding native methods are located through searching the keywords `JNIEXPORT`, `RegisterMethodsOrDie`, `registerNativeMethods`, and `jniRegisterNativeMethods` across the `.cpp` files. Then, we analyze the files with CLANG to extract the Java class path and the JNI mappings.

2) Static Dependency Inference: The nodes in D3G include function nodes and object nodes, as depicted in Figure 1. Object nodes are the prerequisites of function nodes and are served as the input of functions. Function nodes can connect to each other to form a call sequence. One of the main innovations of our proposed approach is incorporating the D3G to improve the efficiency of fuzzing. The D3G is initialized by static inference and further refined by the feedback during dynamic fuzzing. While the dynamic refinement will be explained in section III-C, we elaborate on the static initialization of D3G in this subsection.

Normally, APIs are designed to be invoked in particular sequences (*e.g.*, the API responsible for initialization is called before other APIs). It may lead to program crash if violating such calling orders. While Android well documents the dependencies for SDK APIs, it does not provide such information for non-SDK APIs including the JNI functions of the native system library. Therefore, we infer the call sequences of the JNI functions. Algorithm 1 describes the process of inferring such dependency relationship. Specifically, the inference begins with traversing each class's statement (lines 4-16). The statement represents a minimum syntactic unit, *e.g.*, call and assignment. For assignment statements (line 5), the variable on the left of the assignment statement is added to the L set (lines 7-8), while the one on the right is stored in the R set (lines 9-10). For each call statement, JDYNUZZ verifies whether a call sets or gets a variable in the JNI environment (lines 13 and 15), *e.g.*, `JNIEnv.SetField` and `JNIEnv.GetField`. The functions that set variables are stored in the L set (line 14), while the functions that get variables are kept in the R set (line 16). Finally, the variables and functions in the L and R sets will be checked. The matched functions are saved in D, which become edges in D3G (lines 19-20).

C. Fuzzing

In this subsection, we explain how server and client are designed, and how D3G is refined.

1) Fuzzing Server: The Fuzzing Server is responsible for communicating with the Fuzzing Client (including sending the commands to and receiving the feedback from the client), analyzing the fuzzing results, and refining the D3G accordingly. The fuzzing server first fetches the initial D3G generated in the JNI Analyzer. It then extracts the information required for executing the test (*e.g.*, call sequences) and sends a string command containing these information to the client. After the client parses and runs the commands, the execution results, together with detailed error information, will be sent back to the Fuzzing Server for further analysis. The information is also used to update the D3G accordingly.

When a crash occurs in the native system libraries, Tombstone and HWASan are leveraged to catch the crash information for further analysis. The Tombstone contains specific details about the crashed process. Specifically, it includes stack traces for all threads involved in the crashed process (beyond just the thread that captures the crash signal), a complete memory map, and a list of all descriptors of open files. Note that, we ignore crashes with error code `SEGV_ACCERR` because it is thrown by the incorrect object generation, *i.e.*, it is caused by incorrectly construction of objects rather than triggering a real bug. In addition to Tombstones, JDYNUZZ also leverages HWASan, a sanitizer designed for detecting memory errors in arm64 Architecture, to provide additional memory information that Tombstones ignores. As the same error may be triggered more than once, JDYNUZZ removes duplicates based on the Top-3 method of stack trace (*i.e.*, the information of both error type and the first three crash stacks).

2) Fuzzing Client: The Fuzzing Client receives commands from the Fuzzing Server and prepares the context and variables for execution. After executing the commands, it returns the feedback to the server for further analysis. We first prepare the Java context for executing the commands. Specifically, the Java context is loaded from the system packages through the function `loadClassOrPackage`. Then, the Fuzzing Client

Algorithm 2 Variable Generation

Input: Array of Variable Type String S.
Output: Array of Variable V.

```
1:  function generateVariable(S)
2:      V ← ∅
3:      for each s ∈ S do
4:          if isArray(s) then
5:              size ← random()
6:              V_t ← (type(s))[size]
7:              for each v_t ∈ V_t do
8:                  v_t ← randomVarValByType(s)
9:              V ← V ∪ V_t
10:         else
11:             V_t ← randomVarValByType(s)
12:             V ← V ∪ V_t
13:     return V
14: function randomVarValByType(s)
15:     if typeOf(s) is PrimitiveType then
16:         ω ← randome(0, 3)
17:         switch ω do
18:             case 0
19:                 V_t ← randomLargeVal(s)
20:             case 1
21:                 V_t ← randomSmallVal(s)
22:             case 2
23:                 V_t ← randomMinOrMaxVal(s)
24:     else
25:         ω ← constructorNum(s)
26:         δ ← random(0, ω)
27:         S ← constructorParametersStringArray()
28:         V_t ← generateVariable(S)
29:     return V_t
```

decodes the command, and generates the inputs for fuzzing. The inputs include the call sequences and the arguments involved in the call sequences. In addition, for non-static method invocations, the caller objects are also generated. As detailed in Algorithm 2, for arrays, we first randomize their sizes and then randomize their corresponding values (lines 4-13). For primitive type arguments, we assign them a random value (lines 15-23). More specifically, for types of numbers (*i.e.*, `byte`, `short`, `int`, `long`, `float`, and `double`), we randomly assign either a small value (*i.e.*, the absolute value is less than or equal to 100), a large value (*i.e.*, the absolute value is greater than 100), or a minimum/maximum value (*i.e.*, smallest/largest number allowed in that type). The values generated in this way have better diversity than random. For `char` and `boolean` types, we randomly assign a legit value (*e.g.*, either *true* or *false* for a boolean variable). For objects, we randomly choose a constructor for initialization (lines 25-28). Any primitive and types involved in constructing an object follows the corresponding rules mentioned before.

3) Graph Mutation: With dynamic execution states, JDYNUZZ refines the D3G so that it can construct syntactically and semantically correct call sequences. D3G consists of nodes that are either Objects or API Calls, and the graph mutation includes adding/pruning object nodes or API nodes. Object nodes represent the object constructions and they are prerequisites of certain API calls. Based on static analysis, JDYNUZZ generates an initial D3G, which contains the information of call sequences of APIs. The call sequences will be refined based on the feedback from the client, by either adding nodes to or pruning nodes from the D3G. Note that adding a node to the D3G will increase the probability of executing it, while pruning a node decreases it.

JDYNUZZ includes the four dynamic refinement operations. The refinement is dynamic because JDYNUZZ automatically updates D3G based on run-time information of executing call sequences. By using Java reflection, the refinement does not need recompilation and can efficiently obtain required information. 1) *Adding object nodes.* An object will be added into the D3G when a call sequence is executed without error and it contains a new edge that contains the object. Meanwhile, the objects will be stored to be reused in later fuzzing campaigns. The reuse aims to ensure that the pre-processed APIs in the call sequences execute correctly, which improves the efficiency of JDYNUZZ. 2) *Pruning object nodes.* Object nodes are used less often if their constructions fail more frequently. Specifically, such failure may be caused by the infinite loops existing in constructors. For example, `String` object has construction that needs another `String` to created. This failure can be caught by feedback indicating that the object is incorrect, such as the dead object exception raised by Java or error code `SEGV_ACCERR` raised by the native code. In such cases, we reduce the probability of using such constructor(s). 3) *Adding call sequence nodes.* Due to the complexity of the code coupling, covering all possible call sequences through static analysis is not practical. With dynamic refining, JDYNUZZ attempts to add an API to the existing D3G, which improves the accuracy of static data dependency graph. We add an API to a sequence if the sequence executes successfully with the API but the API fails to execute individually. 4) *Pruning call sequence nodes.* The invocation of an API node is dynamically refined. The increase of failures to call an API raises the probability to skip using the API node in the call sequence.

IV. EVALUATION

The evaluation aims to answer the following research questions:

- **RQ1 (JNI Invocation Statistics)**: What is the present status of invoking JNI functions?
- **RQ2 (Unsafe Calls)**: How effective is JDYNUZZ in finding bugs in Android native system libraries?
- **RQ3 (Dynamic Data Dependency Graph)**: How effective is dynamic data dependency graph?

Experimental Setup: We implement the fuzzing server of JDYNUZZ on Ubuntu 18.04 with Intel (R) Core (TM) i9-9920X CPU @ 3.50GHz and 128 GB RAM. We test our JDYNUZZ on Android phone Pixel 3, with OS version AOSP SP1A.210812.016.C1, tag android-12.0.0_r31. We enable HWAsan while compiling AOSP to capture memory bugs.

TABLE I
STATISTICS OF APPS USING SYSTEM JNI FUNCTIONS

Target SDK version	# of apps	Apps using System JNI functions	% of apps
28 (Android 9)	540	415	76.85%
27 (Android 8.1)	9,577	7,822	81.67%
26 (Android 8)	9,223	6,834	74.10%
25 (Android 7.1)	6,388	4,528	70.88%
< 24 (Android 7)	17,811	9,424	52.92%
Total	43,539	29,023	66.66%

A. JNI Invocation Statistics

This section begins with examining the statistics of system JNI invocation in Android apps. The apps analyzed in this paper were collected in August 2019 from the Google Play store, during which time the most recent SDK release is version 28. Because Google has not modified its restrictions on invoking system JNI functions since then, we believe that the collected data demonstrates the widespread usage of system JNI functions in real world Android apps. We extract the invocation of system JNI functions using `veridex`, which is included in the AOSP releases. We further group the usage of system JNI functions based on the target SDK versions and demonstrate the proportion of apps in each API level that uses system JNI functions. The results in Table I show that 66.7% of apps uses the system JNI functions. Such prevalent usage of system JNI functions further motivates our proposed research.

Moreover, we analyze the number of crash-related questions about using JNI or Java reflection to access Android native system libraries on Stack Overflow [4], which is a question-and-answer website for programmers. We crawl 17 million questions till March 2022 from the Stack Overflow website. Each question contains information such as a title, a body, and the answer count. Within the collected questions, we first filter questions about JNI, Java reflection, and non-SDK API in Android through keyword search. We then manually check and further remove questions that are not related to crashes and bugs, such as questions about what JNI is. Finally, we find that 88,412 questions match our criteria. Within these questions, 71,975 ones have at least one answer, and 40,308 of them have an accepted answer. In other words, there are 16,437 questions remain unanswered, while 48,104 (54.4%) questions do not have a satisfactory answer. In addition, each question has on average 2,313 views, with the most popular question receives 923,742 views. The presented data, to some extent, indicates that developers are actively (mis-)using Android system JNIs and find it difficult to debug individually, thereby seeking help from Stack Overflow.

Answer to RQ1. 66.7% of real-world Android apps on Google Play Store utilize the system JNI functions. Developers are actively (mis-)using Android system JNIs and find it difficult to debug individually.

B. Unsafe Calls

JDYNUZZ ran on AOSP version android-12.0.0 r31 for 7 days, and discovered 34 new bugs in the native system libraries. Table II illustrates the detailed information about the bugs, including the involved library, the types of the bugs, and the number of bugs in each library. Regarding the types of bugs, the most frequent bug is heap-buffer-overflow (12 in total) discovered in 4 libraries. Heap-buffer-overflow is triggered when the index is out of the boundary of an array in heap. The bug allocation-size-too-big appears 9 times in 5 libraries. This bug is caused by a huge allocated size whose value is passed from Java side to native side. Other bugs include global-buffer-overflow (caused when the index is out of the boundary of a global array), stack-buffer-overflow (triggered when the index is out of the boundary of an array on stack), and memory-violation (triggered when the program tries to access an unallocated heap chunk). There are 6 bugs that cannot be identified by Tombstone or HWASan, and we mark them as *Unknown-type*. Interestingly, the library `libandroid_runtime.so`, where JNI functions on the native side are compiled into, contains the most number of bugs (13 out of 34).

We further reveal the corresponding Java-side methods involved in the discovered bugs. Table III displays these unsafe calls on the Java-side. Due to the limit of space, we aggregate the results in the class level. The complete list of function-level unsafe calls are available in our open-source repository. We believe that the list of unsafe calls can help developers create a more robust app when involving native system libraries. We also submit the detected bugs to Google and all 34 bugs are confirmed by Google.

Figure 5 illustrates an example of a bug discovered by JDYNUZZ in `JetPlayer`, a build-in music player in Android. In Figure 5, the method in line 2 initializes the number of soundtracks. The method in line 1 mutes specific tracks using a boolean array named `muteArray`. These two methods are located in the Android framework (Java) and can be invoked by apps using the restriction-breaking techniques [32]. Java- and native-side JNI functions are

```
1  native_queueJetSegmentMuteArray(..., boolean[] muteArray,
   ...);
2  native_setup(..., int maxTracks, ...);
-------------------------------------------------------------
3  android_media_JetPlayer_setup(... jint maxTracks, ...) { ...
4    JetPlayer* lpJet = new JetPlayer(..., maxTracks, ...);
5    ...
6    env->SetLongField(..., (jlong)lpJet);
7    ... }

8  android_media_JetPlayer_queueSegmentMuteArray(... ,
   jbooleanArray muteArray, ...) {
9    JetPlayer *lpJet = (JetPlayer *)env->GetLongField(...);
10   ...
11   jboolean *muteTracks = NULL;
12   muteTracks = env->GetBooleanArrayElements(muteArray,
13 NULL);
14   ...
15   int maxTracks = lpJet->getMaxTracks();
16   for (jint trackIndex=0; trackIndex<maxTracks; trackIndex++){
17     if (muteTracks[maxTracks-1-trackIndex]==JNI_TRUE)
18   ... }
```

JAVA NATIVE

Fig. 5. An example of a bug discovered by JDYNUZZ.

TABLE II
BUGS FOUND BY JDYNUZZ (ROOT CAUSE LIBRARIES)

Bug File	Bug Types	# Bugs
libandroid_runtime.so	Heap-buffer-overflow * 6, Unknown-type * 3, Allocation-size-too-big * 4	13
libandroidfw.so	Global-buffer-overflow * 1	1
libart.so	Global-buffer-overflow * 1, Stack-buffer-overflow * 1, Allocation-size-too-big * 1	3
libgraphicsenv.so	Allocation-size-too-big * 1	1
libicu_jni.so	Unknown-type * 1	1
libjavacrypto.so	Allocation-size-too-big * 1	1
libmedia_jni.so	Heap-buffer-overflow * 2, Memory-violation * 1	3
libcrypto.so	Memory-violation * 1, Heap-buffer-overflow * 1	2
libhwui.so	Heap-buffer-overflow * 3, Allocation-size-too-big * 2	5
libjavacore.so	Stack-buffer-overflow * 1, Unknown-type * 2	3
libutils.so	Memory-violation * 1	1
Total		34

TABLE III
UNSAFE CALLS FOUND BY JDYNUZZ

Class of Unsafe calls	# of Unsafe calls
android.os.HwBlob	4
android.view.SurfaceControl	4
android.util.CharsetUtils	2
android.util.PathParser	2
android.os.Process	2
android.hardware.SerialPort	2
android.media.JetPlayer	2
android.opengl.EGL14	1
android.graphics.drawable.VectorDrawable	1
android.graphics.Interpolator	1
android.opengl.GLES30	1
android.animation.PropertyValuesHolder	1
android.os.GraphicsEnvironment	1
android.graphics.BitmapRegionDecoder	1
android.app.backup.FileBackupHelperBase	1
android.graphics.Path	1
android.os.HwParcel	1
com.android.org.conscrypt.NativeCrypto	3
libcore.io.Memory	3
com.android.i18n.timezone.internal.Memory	1
dalvik.system.VMDebug	1
Total	34

registered through dynamic JNI registration. The native methods are implemented respectively in lines 3 and 8. The setup method creates a `JetPlayer` object with the `maxTracks` integer (line 4). The object is saved in the JNI environment, which can be loaded from any location (line 6). The `IpJet` object then assigns the value of `maxTracks` (line 15). Additionally, `muteArray` is transformed into a pointer `muteTracks` (line 12). In the *for* loop (lines 16-18), the mutation status of the tracks are checked through traversing the boolean array. The heap-buffer-overflow bug occurs when the index value is larger than the size of the `muteTracks` array. To avoid such errors, the JNI functions need to verify the array's access validity before accessing array elements.

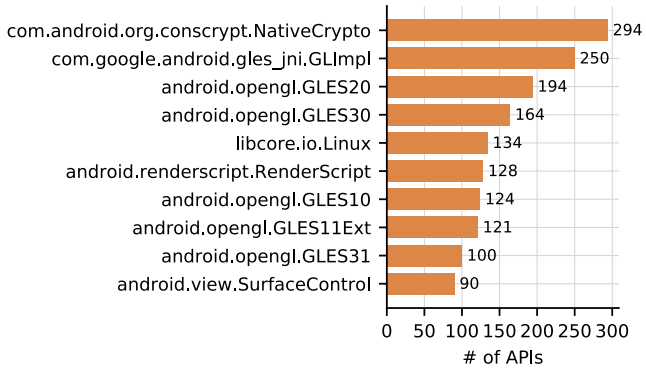


Fig. 6. # of the APIs in classes (Top 10).

Answer to RQ2. JDYNUZZ successfully exposes 34 new bugs in Android native system libraries, which are all confirmed by Google. We list the vulnerable libraries and the corresponding Java-side unsafe method calls.

C. Dynamic Data Dependency Graph

In this section, we provide the statistics of function and object nodes in JNI. We also demonstrate the efficiency of D3G when fuzzing JNI.

1) Statistics of D3G Nodes: JDYNUZZ fuzzes Android native system libraries via JNI and uses D3G that consists of functions and objects.

a) Function statistics: In total, 4,802 JNI functions are extracted from 320 Java classes, with an average of 15 functions per class. 70.8% of the JNI functions are static, while the remainder 29.2% are non-static. It takes approximately 30 minutes to extract the JNI functions of Android native system libraries. Figure 6 shows the top 10 classes with the highest number of functions in each class. The encryption class, `NativeCrypto`, contains the most JNI functions among all classes. Four classes from the `opengl` package, which is responsible for graph-related operations, are also among the top ranked classes.

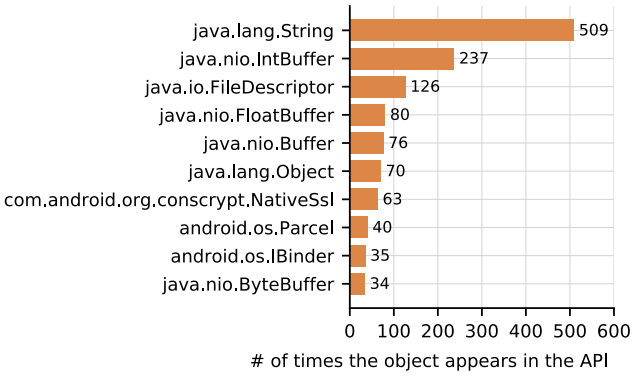


Fig. 7. # of times the object appears in the API (Top 10).

b) Object statistics: There are 193 unique objects involved in the JNI function calls. Given this significant number, manually coding the construction of these objects, as done in [14], demands considerable human effort. In JDYNUZZ, these objects are automatically generated randomly with one of their constructors. Each object contains 1.7 constructors on average. 49.24% of constructors do not require object as input parameters. Figure 7 depicts the frequency that objects appear in JNI functions. `String` objects appear the most frequently (509 times). Four types of buffer objects also appear frequently, 238, 80, 76, and 34 times, respectively. Note that the `object` type appears in top 10. The `object` type is the super class of all Java objects.

2) Efficiency of D3G: Due to the significant gap between the existing Android fuzzing research and JDYNUZZ, it is not practical to compare our work with them. Specifically, a number of works concentrate on fuzzing the binder in Android [12], [13], [14]. However, the JNI mechanism is vastly distinct from the binder; hence, these works cannot be applied to fuzzing JNI. Another related work is a JVM platform fuzzing [33]. However, it does not support ARM64 environment of the Android devices. Therefore, we perform ablation experiments (*i.e.*, excluding certain functionalities of JDYNUZZ) to demonstrate the effectiveness of JDYNUZZ.

Callable JNI functions are the JNIs that can be tested by fuzzing. If JNI functions cannot be accessed, the fuzzing cannot execute the code of them. More callable JNI functions indicate higher chance to discover bugs in system libraries. The metric of *D3G details* are the edges and depths of the graph. A node in D3G is an JNI function or an object, and two nodes form an edge. The edges in D3G can form an API call sequence, which is tested by fuzzing. The depth of the graph is the length of the branch. The D3G is a tree-like graph, and the depth demonstrates the complexity of the dependency. A long API sequence can test deep code and discover bugs hidden in complex execution paths.

To demonstrate the effectiveness of JDYNUZZ, we conduct comparative experiments. In the experiments, we remove some fuzzing components to demonstrate the effectiveness of JDYNUZZ's certain strategies. JDYNUZZ's fuzzing strategies include JNI extraction, dynamic API invocation, static data dependency graph generation, and dynamic refinement of data dependency graph.

*a) Fuzzing without any data dependency graph (*JDYNUZZ*-blank):* In this experiment, we fuzz Android native system library without any data dependency graph. The data dependency graph includes automatic object generation, dynamic function invocation, static data dependency graph generation, and dynamic refinement of data dependency graph. As the results shown in Table IV, I,484 JNI functions have objects as parameters and 1,400 JNI functions are member methods. After de-duplicating, there are 2,447 object-related JNI functions that are not callable, indicating that 2,355 JNI functions are callable. Only 15 bugs are found due to the small number of callable functions. The fuzzing only costs 4 days because of the reduced number of the callable APIs.

*b) Fuzzing with static data dependency graph (*JDYNUZZ*-static):* In this experiment, we fuzz Android native system library without dynamic API invocation and dynamic refinement of dependency graph. Without dynamic refinement, the data dependency graph is static. As the results shown in Table IV, the graph is the initialized D3G which has 1,838 edges. There is no edge about objects. The D3G consists of 91 small standalone graphs due to the separation of the JNI classes. The average length and depth of graphs is 20.2 and 1.7. The maximum depth of graph is 4. Without dynamic API invocation, every test case of call sequence needs to recompile the fuzzing tool. Manual code modification costs the most time, which is more than 1 minute. The average compilation time of fuzzing is 12 second. The installation phase also cost more than 1 minute. These manipulations extremely increase the fuzzing time and this experiment costs about one month, but JDYNUZZ-static only finds 32 bugs.

*c) Fuzzing with static data dependency graph and dynamic API invocation (*JDYNUZZ*-static-invoke):* In this experiment, the fuzzing tool is equipped with static Data Dependency Graph and dynamic API invocation. The dynamic API invocation uses Java reflection to execute API sequences, without the requirement of recompilation and manual efforts. However, since JDYNUZZ-static-invoke does not update D3G, the graph has only 1,838 edges with the maximum depth of 4. Therefore, JDYNUZZ-static-invoke discovers the same number of bugs as JDYNUZZ-static, which is 32 bugs. Because JDYNUZZ-static-invoke automatically executes call sequences without the requirement of recompilation, this experiment takes about 7 days, which is less than JDYNUZZ-static. Therefore, the dynamic API invocation increases the speed of fuzzing.

*d) Fuzzing with D3G (*JDYNUZZ*):* With full fuzzing strategies, JDYNUZZ can generate all the JNI inputs and invoke APIs dynamically and flexibly without recompilation. As the results shown in Table IV, there are 2,011 edges of call sequence in 91 small graphs. The average length and depth of graphs are 22.1 and 1.8. The maximum depth of graph is 4. Additionally, 13.5% of APIs and 37.3% of object constructors frequently cause errors, and their probability of being invoked falls below 50%. Moreover, 2165 successfully constructed objects are added into the D3G. Because the refinement increases edges in D3G, more code regions of native system libraries are tested. Therefore, JDYNUZZ identifies 34 bugs

TABLE IV
EFFICIENCY OF D3G

Fuzzers	# Callable JNI Functions	# Edges of D3G	Max Depth of D3G	# Bug Discovery
JDYNUZZ-blank	2,355	N/A	N/A	15
JDYNUZZ-static	4,802	1,838	4	32
JDYNUZZ-static-invoke	4,802	1,838	4	32
JDYNUZZ	4,802	2,011	4	34

demonstrated in Table II, where two bugs are never discovered by other comparative fuzzers. As for the fuzzing time, this experiment costs about 7 days.

Answer to RQ3. JDYNUZZ can successfully refine the data dependency graph constructed by static analysis. The refinement can further help fuzzing discover more bugs.

V. DISCUSSION

In this section, we discuss the implication and threats to validity of our work.

A. Implication

As we expect, developers of third-party JNI cross-language libraries, particularly third-party library suppliers, can utilize JDYNUZZ to thoroughly test libraries. Even if the library users can fuzz the libraries on binary, it is difficult for them to read or edit the binary library. They can only attempt to avoid these potentially harmful inputs prior to calling. Thus, fuzzing based on open source code is reasonable. With saying that, JDYNUZZ is fully capable of testing binary native library. Actually, our approach can support binary native library with reasonable amount of extra engineering work. The prototype of JDYNUZZ is implemented with Clang as an open-source native library analysis tool because prototype based on open-source needs less engineering work and can demonstrate the effectiveness of our tool. However, our approach does not rely on any open-source-related features. The proposed prototype, JDYNUZZ, now only supports open-source project (*e.g.* AOSP) due to the restriction of our selected static analysis framework (*i.e.*, Clang only supports analyzing source code). Nevertheless, JDYNUZZ can be implemented with a static binary analysis framework, such as IDA [34], PIN [35], Angr [36], Capstone [37], and Dyninst [38], to support fuzzing on binary libraries. Moreover, machine learning can also support static analysis of binary code [39], [40], [41].

B. Threats to Validity

1) Fuzzing Target: In this paper, we fuzz the Android native system libraries through the JNI rather than the Android apps. Our approach ignores the complex app logic and generate input for JNI functions via Java reflection, which makes fuzzing more efficient. As a comparison, fuzzing the native system library via apps may waste significant amounts of resources on exploring from the app entry to the native system library calls. However, such approach may lose the context information when fuzzing via an Android app.

2) Inference of Fuzzing Input: JDYNUZZ performs fuzzing based on Android native system libraries through the extracted function model from static analysis. Due to the design of Google, some functions use the super class (*e.g.*, `java.lang.Object`) as an input parameter. In Java, `java.lang.Object` is the super class of all objects. The input can be any object when `java.lang.Object` is in the parameter list. This will make it difficult for JDYNUZZ to generate the data for required types. JDYNUZZ may generate tests with numerous incorrect objects, which may reduce the efficiency of fuzzing.

3) Object Generation: In our work, objects are generated based on their respective constructors. Due to the coding practices of programmers, a constructor may not fully initialize an object. An object may contain member variables that must be initialized by other functions. For instance, a student object's score may be assigned via the function named `exam`. In this case, JDYNUZZ will produce an insufficient object. It will directly lead to invalid tests or affect efficiency of fuzzing.

VI. RELATED WORK

A. Cross-Language

Numerous works have proposed solutions for analyzing cross-language code. Wei et al. [9] proposes an approach that can generate the cross-language dataflow to investigate the security issues on Android apps. Fourtounis et al. [11] analyze the binary libraries and cross-reference the data to locate the native code of Java callbacks. JUCIFY [42] provides a method of static analysis that extends Android app analysis to native code. However, these works all focus on the identification of the cross-language modeling based on static analysis. Our work focuses on the dynamic data dependency graph identification and refinement, such as the input generation (*i.e.*, basic variable and object generation) of the extracted interfaces and the investigation of call sequences.

B. Fuzzing on Android

In the research community, the researchers have already used fuzzing on Android. Mulliner and Miller [43] discover several vulnerabilities in the SMS functionality. Droidfuzzer [44] detects intent vulnerabilities in apps. Targets of the test are the activities that take MIME data. Sasnauskas and Regehr [45] provide an alternative intent fuzzing method that combines static analysis and random test case generation. Mahmood et al. [46] designs a white-box fuzzing technique that can automatically discover cloud-based apps with a huge number of test cases. Additionally, some works focus on the detection of app vulnerabilities and cannot be applied to the

fuzzing of system JNI functions [28], [47], [48], [49], [50], because the model characteristics of system JNI functions (*e.g.*, JNI functions discovery and dependency analysis) are absent from these works. Kai et al. [51] present a method for fuzzing the Binder mechanism within the source code. Binder is an Android Inter-Process Communication (IPC) component. The mutation algorithm suggested in the paper is utilized to create IPC data. Meanwhile, Chizpurfle [13] focuses on the vendor system's fuzzing, which does not require source code. The Android services are tested on the finalized Android system without modification. BinderCracker [12] further extends the fuzzing to the native side. It analyzes IPC communication across several apps and extracts transaction models to simulate test cases. FANS [14] extracts AIDL models and generates test cases for native system services with random input. All of the works concentrate on Binder-related fuzzing based on Binder characteristics. Binder and JNI are distinct communication systems with different features for extracting models and generating fuzzing test cases. GraphFuzz [52] proposes a fuzzing on C/C++ libraries. It generates the static data flow graphs and performs the mutations at the graphs level. In our work, we perform the D3G which is upgradable during fuzzing. Our graph is refined via the feedback of fuzzing.

VII. Conclusion

In this paper, we propose a novel method, JDynuzz, for fuzzing the JNI functions of systems. The key of JDynuzz is to construct the dynamic data dependency graph, which can refine the call sequences. The most recent stable version of Android is used to assess the performance of JDynuzz. The results show that JDynuzz can detect 34 new memory bugs at the native level and point out 34 unsafe JNI calls, allowing developers to design sophisticated apps with secure invocation to the system JNI functions.

References

[1] (May 2022). *Mobile Operating System Market Share Worldwide*. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

[2] (2022). *Number of Apps Available in Leading App Stores As of 2022*. [Online]. Available: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

[3] (2022). *New Crittercism Report Identifies Key Trends and Influences for M-Commerce Mobile App Adoption*. [Online]. Available: https://www.retailitinsights.com/doc/new-crittercism-report-identifies-key-trends-and-influences-for-m-commerce-mobile-app-adoption-0001

[4] (2022). *Stack Overflow—Where Developers Learn, Share, & Build Careers*. [Online]. Available: https://stackoverflow.com/

[5] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surveys*, vol. 54, no. 11s, pp. 1–36, Jan. 2022, doi: 10.1145/3512345.

[6] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, "DroidNative: Automating and optimizing detection of Android native code malware variants," *Comput. Secur.*, vol. 65, pp. 230–246, Mar. 2017.

[7] L. Xue et al., "NDroid: Toward tracking information flows across multiple Android contexts," *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 3, pp. 814–828, Mar. 2019.

[8] M. Sun, T. Wei, and J. C. Lui, "TaintART: A practical multi-level information-flow tracking system for Android runtime," in *Proc. 2016 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 331–342.

[9] 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 *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 1137–1150.

[10] J. Samhi et al., "Jucify: A step towards Android code unification for enhanced static analysis," in *Proc. ICSE*, 2022, pp. 1232–1244.

[11] G. Fourtounis, L. Triantafyllou, and Y. Smaragdakis, "Identifying Java calls in native code via binary scanning," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2020, pp. 388–400.

[12] H. Feng and K. G. Shin, "BinderCracker: Assessing the robustness of Android system services," 2016, *arXiv:1604.06964*.

[13] A. K. Iannillo, R. Natella, D. Cotroneo, and C. Nita-Rotaru, "Chizpurfle: A gray-box Android fuzzer for vendor service customizations," in *Proc. IEEE 28th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2017, pp. 1–11.

[14] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge, "FANS: Fuzzing Android native system services via automated interface analysis," in *Proc. 29th USENIX Secur. Symp. (USENIX Security)*. Berkeley, CA, USA: USENIX Association, Aug. 2020, pp. 307–323. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/liu

[15] (2022). *JNI APIs and Developer Guides*. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/

[16] X. Feng, X. Zhu, Q.-L. Han, W. Zhou, S. Wen, and Y. Xiang, "Detecting vulnerability on IoT device firmware: A survey," *IEEE/CAA J. Autom. Sinica*, vol. 10, no. 1, pp. 25–41, Jan. 2023.

[17] D. Babić et al., "FUDGE: Fuzz driver generation at scale," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Aug. 2019, pp. 975–985.

[18] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1032–1043.

[19] X. Feng et al., "Snipuzz: Black-box fuzzing of IoT firmware via message snippet inference," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2021, pp. 337–350.

[20] S. Yang, R. Li, J. Chen, W. Diao, and S. Guo, "Demystifying Android non-SDK APIs: Measurement and understanding," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, 2022, pp. 647–658.

[21] (2022). *Download Android Studio & App Tools—Android Developers*. [Online]. Available: https://developer.android.com/studio

[22] (2022). *Using Debuggers | Android Open Source Project*. [Online]. Available: https://source.android.com/docs/core/debug/gdb

[23] (2022). *Lldb Homepage—The Lldb Debugger*. [Online]. Available: https://lldb.llvm.org/

[24] (2022). *Gdb: The GNU Project Debugger*. [Online]. Available: https://www.sourceware.org/gdb/

[25] (2022). *Using Debuggers | Android Open Source Project*. [Online]. Available: https://source.android.com/docs/setup/build/building

[26] (1999). *Soot—Java Analysis Framework*. [Online]. Available: http://sable.github.io/soot/

[27] (2000). *Clang: A C Language Family Frontend for LLVM*. [Online]. Available: https://clang.llvm.org

[28] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proc. ACM Conf. Comput. Commun. Secur.*, Oct. 2012, pp. 217–228.

[29] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 636–654.

[30] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of Android malware through static analysis," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2014, pp. 576–587.

[31] (2022). *Dalvik Executable Format | Android Open Source Project*. [Online]. Available: https://source.android.com/docs/core/dalvik/dex-format

[32] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On demystifying the Android application framework: Revisiting Android permission specification analysis," in *Proc. 25th USENIX Security Symp. (USENIX Security)*, Austin, TX, USA, 2016, pp. 1101–1118. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/

[33] (2022). *Coverage-guided, In-process Fuzzing for the JVM*. [Online]. Available: https://github.com/CodeIntelligenceTesting/jazzer

[34] (2022). *Hex Rays—State-of-the-Art Binary Code Analysis Solutions*. [Online]. Available: https://hex-rays.com/ida-pro/

[35] (2022). *Pin—A Dynamic Binary Instrumentation Tool*. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html

[36] Y. Shoshitaishvili et al., "SoK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 138–157.
[37] (2022). *The Ultimate Disassembly Framework—Capstone—The Ultimate Disassembler*. [Online]. Available: http://www.capstone-engine.org/
[38] (2022). *Github—Dyninst/Dyninst: Dyninstapi: Tools for Binary Instrumentation, Analysis, and Modification*. [Online]. Available: https://github.com/dyninst/dyninst
[39] J. Zhang, L. Pan, Q.-L. Han, C. Chen, S. Wen, and Y. Xiang, "Deep learning based attack detection for cyber-physical system cybersecurity: A survey," *IEEE/CAA J. Autom. Sinica*, vol. 9, no. 3, pp. 377–391, Mar. 2022.
[40] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proc. IEEE*, vol. 108, no. 10, pp. 1825–1848, Oct. 2020.
[41] X. Chen et al., "Android HIV: A study of repackaging malware for evading machine-learning detection," *IEEE Trans. Inf. Forensics Security*, vol. 15, no. 1, pp. 987–1001, Jul. 2019.
[42] J. Samhi et al., "JuCify: A step towards Android code unification for enhanced static analysis," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, 2022, pp. 1–13.
[43] C. Mulliner and C. Miller, "Fuzzing the phone in your phone," in *Black Hat USA*, 2009.
[44] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "DroidFuzzer: Fuzzing the Android apps with intent-filter tag," in *Proc. Int. Conf. Adv. Mobile Comput. Multimedia MoMM*, 2013, pp. 68–74.
[45] R. Sasnauskas and J. Regehr, "Intent fuzzer: Crafting intents of death," in *Proc. Joint Int. Workshop Dyn. Anal. (WODA) Softw. Syst. Perform. Test., Debugging, Analytics (PERTEA)*, Jul. 2014, pp. 1–5.
[46] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, "A whitebox approach for automated security testing of Android applications on the cloud," in *Proc. 7th Int. Workshop Autom. Softw. Test (AST)*, Jun. 2012, pp. 22–28.
[47] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer, "An empirical study of the robustness of inter-component communication in android," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2012, pp. 1–12.
[48] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "IntentFuzzer: Detecting capability leaks of Android applications," in *Proc. 9th ACM Symp. Inf., Comput. Commun. Secur.*, Jun. 2014, pp. 531–536.
[49] Y. Hu and I. Neamtiu, "Fuzzy and cross-app replay for smartphone apps," in *Proc. 11th Int. Workshop Autom. Softw. Test*, 2016, pp. 50–56.
[50] A. Dawoud and S. Bugiel, "Bringing balance to the force: Dynamic analysis of the Android application framework," in *Proc. NDSS*, 2021, pp. 1–12.
[51] W. Kai, Z. Yuqing, L. Qixu, and F. Dan, "A fuzzing test for dynamic vulnerability detection on Android binder mechanism," in *Proc. IEEE Conf. Commun. Netw. Secur. (CNS)*, Sep. 2015, pp. 709–710.
[52] H. Green and T. Avgerinos, "GraphFuzz: Library API fuzzing with lifetime-aware dataflow graphs," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 1070–1081, doi: 10.1145/3510003.3510228.

Xiaogang Zhu (Member, IEEE) received the Ph.D. degree from the School of Computer Science and Engineering, Swinburne University of Technology, in 2021. He is currently a Research Fellow with the Swinburne University of Technology. He has published papers on top venues, such as the ACM SIGSAC Conference on Computer and Communications Security (CCS), the USENIX Security Symposium (USENIX), the Network and Distributed System Security Symposium (NDSS), the IEEE/ACM International Conference on Software Engineering (ICSE), IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, and *ACM Computing Surveys* (CSUR). His research interests include detecting vulnerabilities in software, the IoT devices, and Windows applications. He is interested in techniques such as fuzzing, machine learning, and symbolic execution.

Siyu Zhang received the bachelor's degree from Zhengzhou University and the Master of Network and Security degree from Monash University. He is currently pursuing the Ph.D. degree with the Swinburne University of Technology. His research interests include detecting vulnerabilities in software. He is interested in techniques, such as fuzzing and machine learning.

Chaoran Li received the Ph.D. degree from the Swinburne University of Technology, Australia. He has published many papers on top venues, such as the Proceedings of the 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, and IEEE TRANSACTIONS ON CYBERNETICS. His research interests include mobile software analysis, mobile security, and adversarial machine learning.

Sheng Wen (Senior Member, IEEE) received the Ph.D. degree from Deakin University, Melbourne, in October 2014. He has been working full-time as an Associate Professor with the Swinburne University of Technology. In the last six years, he has published more than 50 high-quality papers in the last six years, including 35 journal articles (25 ERA A/A* journal articles and 11 IEEE TRANSACTIONS journal articles) and 18 conference papers (top conferences like IEEE ICDCS) in the fields of information security, epidemic modeling, and source identification. His representative research outcomes have been mainly published on top journals, such as IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, IEEE TRANSACTIONS ON INFORMATION SECURITY AND FORENSICS, and IEEE COMMUNICATIONS SURVEYS AND TUTORIALS.

Yang Xiang (Fellow, IEEE) received the Ph.D. degree in computer science from Deakin University, Australia. He is currently a Full Professor and the Dean of the Digital Research and Innovation Capability Platform, Swinburne University of Technology, Australia. He is also leading the blockchain initiatives at Swinburne. In the past 20 years, he has been working in the broad area of cyber security, which covers network and system security, AI, data analytics, and networking. He has published more than 300 research papers in many international journals and conferences. His research interests include cyber security, which covers network and system security, data analytics, distributed systems, and networking. He is the Coordinator of Asia for IEEE Computer Society Technical Committee on Distributed Processing (TCDP). He is the Editor-in-Chief of the Springer Briefs on Cyber Security Systems and Networks. He serves as an Associate Editor for IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, IEEE INTERNET OF THINGS JOURNAL, and *ACM Computing Surveys*. He served as an Associate Editor for IEEE TRANSACTIONS ON COMPUTERS and IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS.