The Dark Side of Native Code on Android

ANTONIO RUGGIA*, EURECOM, France
ANDREA POSSEMATO, EURECOM, France
SAVINO DAMBRA, GenDigital, France
ALESSIO MERLO, CASD - School of Advanced Defense Studies, Italy
SIMONE AONZO, EURECOM, France
DAVIDE BALZAROTTI, EURECOM, France

From a little research experiment to an essential component of military arsenals, malicious software has constantly been growing and evolving for more than three decades. On the other hand, from a negligible market share, the Android operating system is nowadays the most widely used mobile operating system, becoming a desirable target for large-scale malware distribution. While scientific literature has followed this trend, one aspect has been understudied: the role of native code in malicious Android apps. Android apps are written in high-level languages, but thanks to the Java Native Interface (JNI), Android also supports calling native (C/C++) library functions. While allowing native code in Android apps has a strong positive impact from a performance perspective, it dramatically complicates its analysis because bytecode and native code need different abstractions and analysis algorithms, and they thus pose different challenges and limitations. Consequently, these difficulties are often (ab)used to hide malicious payloads.

In this work, we propose a novel methodology to reverse engineering Android apps focusing on *suspicious* patterns related to native components, i.e., surreptitious code that requires further inspection. We implemented a static analysis tool based on such methodology, which can bridge the "Java" and the native worlds and perform an in-depth analysis of *tag* code blocks responsible for suspicious behavior. These tags benefit the human facing the reverse engineering task: they clearly indicate which part of the code to focus on to find malicious code.

Then, we performed a longitudinal analysis of Android malware over the past ten years and compared the recent malicious samples with actual top apps on the Google Play Store. Our work depicts typical behaviors of modern malware, its evolution, and how it abuses the native layer to complicate the analysis, especially with dynamic code loading and novel anti-analysis techniques. Finally, we show a use case for our suspicious tags: we trained and tested a machine learning algorithm for a binary classification task. Even if suspicious does not imply malicious, our classifier obtained a remarkable F1-score of 0.97, showing that our methodology can be helpful to both humans and machines.

CCS Concepts: • **Security and privacy** → *Software reverse engineering*; *Malware and its mitigation*; *Mobile platform security*.

Additional Key Words and Phrases: Android security, Android malware, Android JNI, Android native code, Android malware detection

1 Introduction

With more than 1.6 billion active users and a market share covering almost 75% of smartphone operating systems, Android is the world's most-used OS today. First introduced in 2010, Google's operating system has seen almost constant growth over the years and now covers the largest share of the market. This constant growth and the increasing number of users also attracted malware authors, and already in August 2010, the first malicious app for Android (`DroidSMS.A`) was detected by security companies. Over the following years, mobile malware has evolved by following trends in common with its more mature PC counterpart and exploring new directions intrinsically driven

Authors' Contact Information: Antonio Ruggia, antonio.ruggia@eurecom.fr, EURECOM, Sophia Antipolis, France; Andrea Possemato, EURECOM, Sophia Antipolis, France, andrea.possemato@eurecom.fr; Savino Dambra, GenDigital, Sophia Antipolis, France, savino.dambra@gendigital.com; Alessio Merlo, CASD - School of Advanced Defense Studies, Rome, Italy, alessio.merlo@unicasd.it; Simone Aonzo, EURECOM, Sophia Antipolis, France, simone.aonzo@eurecom.fr; Davide Balzarotti, EURECOM, Sophia Antipolis, France, davide.balzarotti@eurecom.fr.

by the evolution of the Android system. For instance, the evolution of the operating system and its support for numerous programming languages has resulted in apps written in various languages, from Java to .NET.

Since its very first version, Android has supported Java Native Interface (JNI), a mechanism to connect the Java language, which typically runs inside a virtual machine, and C/C++ languages, which are instead compiled into native code. While using JNI by benign apps has brought numerous advantages in terms of performance and resource consumption, it has also introduced numerous challenges when used by malicious software. In fact, while apps written in Java or Kotlin are not dependent on the device's architecture, the same cannot be said for the native components that use JNI. This, combined with the fact that today the Android system can run on many architectures, introduces numerous challenges for a malware analysis pipeline. Moreover, JNI serves a variety of different purposes. For instance, it can interact with the Android RunTime (ART) and instantiate new objects or modify their fields. It can also perform low-level operations or be used as a trampoline to jump back to the ART. This versatility and the additional complexity of its analysis have led malware authors to increasingly use the native layer to hide malicious code, perform suspicious operations, or complicate static and dynamic analysis [46, 58, 59].

At the time of writing, JN-SAF [59] and JuCify [49] represent state-of-the-art solutions to analyze apps with native code to detect data leaks statically. However, these tools only focus on detecting data leaks through Java and native code. In addition, they rely on Angr's symbolic execution [57], which often incurs in path explosion that prevents it from completing the analysis. However, despite the native code's growing popularity, no previous work has documented how Android malware uses and abuses the native layer. Furthermore, current Android anti-malware engines seem to pay little attention to the native components. For instance, we created and submitted to VirusTotal some malicious samples with well-known publicly available exploits in the native code: half of the samples went undetected, and the remaining were detected by just one engine.

To fill this gap, this paper presents a new approach for studying Android malware and provides a methodology to reverse engineer Android apps that use native code, taking into account all suspicious patterns that can be used for malicious purposes. More precisely, a malicious pattern clearly indicates a willingness to breach security, whereas a suspicious pattern requires further investigation to ascertain its purpose. Our methodology has been developed in collaboration with industry experts who manually reverse-engineer Android malicious samples daily. We implemented our methodology in *ANDani*, a framework to detect and *tag* suspicious native code usage in Android apps. These tags are very useful because they can be imported into reverse engineering frameworks and highlight portions of code that the analyst needs to focus on, as they may conceal malicious code.

We decided to follow a different approach w.r.t. the state-of-the-art, and we developed our analysis infrastructure by combining two components. The first is an extended version of the Soot [56] framework for bytecode analysis; we improved the entry points detection and managing concurrent execution. The second is an architecture-independent Ghidra plugin for the native code analysis that can handle JNI data structures and propagate inferred types from JNI methods' signatures through various functions.

In order to study the evolution of suspicious native code patterns over the years, compare the use of native functionalities in benign and malicious apps, and understand the underlying motivation behind possible discrepancies, we analyzed with *ANDani* a total of 113,476 APKs that include native code. Such APKs are divided into two different datasets: 97,829 malware from AndroZoo [7] spanning from 2010 to 2021, and 15,647 benign apps from the most downloaded apps of the Google Play Store.

Our measurements led to numerous insights into the use of native components. We found that malware is more likely to trigger the native component without user interaction, especially during the app startup and while reacting to Broadcast Receivers. For example, waiting for the mobile phone to be charged can be a good indicator that it is the right time for malware to perform intensive operations that would typically lead to excessive battery consumption and consequently raise the victim's suspicion. Native components are often employed for obfuscation by dynamically loading and executing both Java and other native code. In particular, the malware goes native to dynamically load and invoke methods of the Android framework that require dangerous permission to get the user's sensitive information in a twisted way. This creates leaps between these two different code "worlds," consequently making the analysis particularly difficult. Moreover, we also report an undocumented strategy: malware exploits JNI mechanisms to load malicious or harmless code at will, depending on environment checks. Finally, we found that malicious native libraries are often re-used among samples for several years without the authors even bothering to change their hash.

Our measurement study also highlighted several differences in how benign and malicious apps use native code. To prove the usefulness of *ANDani* and the reliability of the tags that can be extracted from its output, we used them to train and test a Random Forest algorithm able to classify goodware and malware. Our results are auspicious: the classifier can distinguish between the two classes with an average error of 0.02 and achieve an F1-score of 0.97. The output of the classification task allowed us to discriminate suspicious behaviors that are more correlated to malware and whose presence can hint at potential malicious patterns, thus providing valuable information to the malware analysts. For example, we found that the way native components are triggered through JNI contributes to nearly 50% percent of the accuracy.

In summary, this paper makes the following contributions:

- We defined a detailed methodology to assist analysts in reverse engineering native code by Android apps, which focuses on the disclosure of suspicious (potentially harmful) operations;
- We developed *ANDani*, a static analysis tool for Android apps to perform an in-depth behavior analysis of all aspects related to the native code;
- We performed the first longitudinal analysis on Android malware, specifically focusing on native code, over the past ten years and in current "top apps" on the Google Play Store to investigate the security impact of the native code and understand its behavior;
- We highlighted novel anti-analysis techniques, against both static and dynamic analysis, that we found in malware and goodware;
- We showed a concrete use case of suspicious tags: they obtained remarkable performances in a binary classification task as a sole feature.

The rest of the paper is organized as follows: Section 2 discusses the technical background of the Android app, focusing on the use of JNI; Section 3 summarizes previous works concerning strategies to analyze Java and native code; Section 4 details the motivation and introduces our methodology for reverse engineering an Android app with JNI. Section 5 summarize the design and the implementation aspects of *ANDani*, while Section 6 and Section 7 introduce respectively the dataset that we used for the longitudinal analysis and their results, which highlight how and why malware uses the native code. Section 8 shows a concrete use case of the suspicious tags, investigating how our results can improve the reliability of a binary classification task. Finally, in Section 9 we discuss the limitations of our work and conclude in Section 10.

2 Android JNI Internals

The Android system supports apps written with different programming languages and frameworks. While apps were initially developed in Java, today, it is possible to write Android apps in Javascript (with the Cordova framework [17], which wraps the HTML and JavaScript code into a native app container), .NET (with the Mono [45] and Xamarin [38] projects), and Kotlin, the new official programming language for the Android platform. In cases where the app has to comply with very stringent performance constraints or interact with low-level components of the device, Android allows developers to introduce native components written in C and C++ into the app.

Although Android supports Java Virtual Machine-based (JVM) languages such as Java and Kotlin, the compilation process of Android apps differs from that of regular Java apps. On Android, the Java code is first compiled into the corresponding Java-bytecode, which is then compiled into Dalvik-bytecode DEX (`.dex` extension). For the native component, instead, the Android system provides an Android Native Development Kit (NDK), a set of tools containing compilers, debuggers, and build systems that allow the developer to compile native code for their Android app: at the end of the compilation process, the NDK generates native libraries as Executable and Linkable Format (ELF) files, in the form of Shared Object (`.so` extension). The interaction between bytecode and the native libraries, and vice versa, is made possible thanks to the Java Native Interface (JNI).

When all the code has been compiled, it is embedded in an Android app PacKage (APK), an archive containing different files among which all the program's code (such as `.dex` and `.so` files). When the APK is installed on an Android device, another compilation step – that only affects DEX files – takes place on the device. Dalvik-bytecode files are compiled Ahead-Of-Time (AOT) to generate an executable app for the target device architecture. This approach brings numerous improvements in terms of performance and battery life: since the bytecode has been compiled, the app will not require extensive CPU usage for Just-In-Time (JIT) optimizations. Native libraries, on the other hand, are not affected by this additional optimization step: in fact, they are already compiled for the architecture(s) in which the app will run. This means that if an app wants to be installed on several devices that differ in Application Binary Interface (ABI) and Instruction Set Architecture (ISA), it must contain native components compiled for each target architecture it wants to support. To date, Android supports the following ABIs: `armeabi-v7a, arm64-v8a, x86, x86_64`: in the past the system supported `ARMv5 (armeabi)`, and `32-bit` and `64-bit MIPS`, but they are no longer supported [23].

The Android system allows an app to invoke and use native code, whether in the form of shared objects or executable files, through four main techniques.

2.1 Native Library Loading

To allow the interaction between Java code and native components through JNI, libraries must first be loaded into the app's address space. An app can load these libraries by using the `load` or `loadLibrary` methods, which are present in both `java.lang.System` and `java.lang.Runtime` classes. The difference between the `load` and `loadLibrary` methods, for both the implementations, is that the first method requires the library name to be specified as an absolute path. In contrast, the second requires that the name passed as an argument must not contain a file extension or path, as the library will be automatically searched in the default path where the app is installed.

When the library is loaded, the linker calls the initialization functions. The ELF file format defines three sections that contain code (or pointers to code) that are in charge of initialization procedures: `.pre_initarray`, `.init`, and `.initarray`. The linker searches them in this order and runs the code of the present ones. In Android, the `.pre_initarray` section is ignored for shared libraries [26]. Finally, the linker invokes a JNI-specific initializer, the `JNI_OnLoad` function.

2.2 Bridging Functions

The JNI allows the interaction between Java and native components. Thus, it is possible to invoke a native function defined within a shared object from a Java method. Vice versa, the native component, always via JNI, can interact freely with the Java counterpart. For example, the native component can create objects, invoke methods of the Android framework or defined within the app itself, or even modify field values: all these operations are possible thanks to the use of *JNI Callbacks*.

In the Java code, the methods that are declared with the keyword `native` represent the functions defined and exported within the shared library, accessible from the app. The redirection of the execution flow and the mapping between the native method definition and its implementation is all handled via JNI. In particular, when a native library is loaded, the JNI tries to resolve the native methods dynamically and map them into the corresponding defined Java method [44].

These steps are possible thanks to the fixed structure in the *naming convention* of the native methods. For instance, in the following example

```
1  package xx.yyy;
2  class Clazz { public native String test(int x); }
```

the class `Clazz` declares a native method `test`. When the shared library containing the function is loaded, the system will search for the symbol corresponding to the function name: `Java_xx_yyy_Clazz_test(JNIEnv*, jobject, jint)`

The name of the function, translated from Java to native, is made of three parts: the `Java_` string, concatenated with a mangled fully-qualified class name of the related Java class, concatenated with the name of the method.

Furthermore, the definition of native methods requires the first additional argument always to be a pointer to `JNIEnv`. Then, in the case of a static method, JNI requires the second argument to be a pointer to the corresponding Java class (`jclass`); on the other hand, a pointer to the corresponding Java object (`jobject`). Both the first and the second arguments are implicit, and the developer does not directly handle them.

The third and last argument of the example, the type `int` defined in the Java method signature, matches the native type `jint`. For a complete list of Java primitive types and their machine-dependent native equivalents, please refer to [43].

The `JNIEnv` type – a `struct` when the shared object is written in C, or a `class` in C++ – contains pointers to functions that allow the interaction between the native component and the Android framework or the app itself. These functions are called *JNI callbacks*, and the most relevant are `NewObject`, `FindClass`, `GetMethodID`, `GetStaticMethodID`, and the `Call*` family (e.g., `CallVoidMethod`). Through these functions, the native code can, respectively: instantiate objects, find a reference to a class or a method, and then call a method.

Moreover, the `JNIEnv` provides the `RegisterNatives` function to dynamically map a Java method defined as `native` to its implementation in the shared library at runtime. However, in this case, there is no requirement to follow a fixed naming convention. For a complete listing of all the JNI functions, please refer to [42].

2.3 Native Activity

Developers who require their app to have high-performance in terms of execution speed, or that need to interact with low-level system components, may decide to develop *the entire app natively*. For this, the NDK introduces and supports the concept of "Android Native Activity." The native code implements the Android activity component, and its methods are invoked according to the activity lifecycle functions (e.g., `onCreate`, `onDestroy` [22]).

If the app does not contain any Java code, a (Java) "stub" is created at compile-time with the sole task of loading and running the native code, since it is released in the form of Shared Objects and therefore has to follow the entire loading process described above.

There are several requirements for the developer to create a native app: it must target an API level greater than 8, and it must specify whether it contains Java code via the `android:hasCode` attribute of the manifest. Then, each Activity defined as native must indicate in which library it is located: the name of the shared library is specified in the `android:name` attribute.

2.4 Process Execution Methods

Shared Objects are not the only types of ELF that can be executed within Android apps. The Android framework allows apps to execute shell commands, scripts, or ELF executable in a separate process through the `Runtime` class, with its `exec` methods, or via the `ProcessBuilder` class and its `start` method. These methods allow the app to execute binaries that do not contain, potentially, any JNI components. The execution of these new processes takes place in a different process, and therefore the interaction between the native component and the app is not handled by JNI. Therefore, as JNI interaction is not present in this scenario, this category is beyond the scope of our research. However, we analyze the scenario in which a binary or shell script executes within a function defined in a shared library using JNI.

Lastly, we would like to point out that, for the sake of brevity, we will refer to the Dalvik-bytecode as *Java*. We use this simplification to remain consistent with the *Java* Native Interface and avoid specifying on every occasion the distinction with high-level code that is itself compiled into bytecode.

3 Related work

There are mainly two areas of work relevant to this paper: the analyses focusing on the Java layer and those considering JNI.

Java. In 2013, Octeau et al. [40] implemented Epicc, an analysis framework based on Soot – a Java optimization framework proposed by Vallee et al. [56] – to resolve Inter-Component Communication (ICC) in Android apps. In 2014, Arzt et al. [9] proposed FlowDroid, a dataflow analysis framework for taint detection of the Java code of an Android app. It is a full context, object, and flow-sensitive taint analysis which considers the Android app lifecycle. FlowDroid extends the Soot framework and creates an app-level dummy Main class to collect all Android system events. In the same year, Wei et al. [60] proposed Amandroid to conduct static analysis for security vetting of Android apps. It builds a context and flow-sensitive inter-procedural control flow graph (ICFG) of the whole app and computes the point-to information to detect several security-related problems. In 2015, Li et al. [36] proposed a new static taint analyzer to detect privacy leaks among components in Android apps, named IccTA. It propagates the context information among different components to resolve call parameters and return values. In the same year, Gordon et al. [30] proposed DroidSafe, a static analysis framework able to resolve ICC and Remote Procedure Call calls to detect potential data leaks by tracking information flows. Then, Yang et al. [64] proposed AppContext, a static analysis approach to extract context security-sensitive behavior to assist the app analysis focusing only on the Java layer. In 2021, Wu et al. [61] proposed BackDroid, an inter-procedural analysis of Android app, with the primary goal of improving the performance of the static analyzer described earlier by implementing a novel technique named *on-the-fly bytecode search* which searches the disassembled app bytecode text just in time when a caller needs to be located.

JNI. In 2012, Yan et al. [63] proposed DroidScope, an emulation-based Android malware taint-analysis engine used to analyze the Java and native components (x86 and ARM architectures) of an Android app to track information leakage. In 2014, Qian et al. [46] performed the first large-scale study on information flows using JNI. This study leverages NDroid, a novel dynamic taint propagation tool based on QEMU, which tracks JNI and system library functions in Java and native code. Alfonso et al. in 2016 [3] performed an extensive analysis on the adoption of the native code on Android apps, highlighting potential usage of JNI, and proposed a new method to generate a native code sandboxing policy automatically. The same year, Sun et al. [51] proposed TaintART, a customized ART compiler that inserts the taint logic and retains the original ahead-of-time optimizations that perform taint analysis to track data flow. Rasthofer et al. [47] proposed Harvester, an hybrid analysis tool that combines static backward slicing to identify interesting code with the execution of the code for extracting runtime values. In 2017, Alam et al. [6] proposed DroidNative, a static Android malware detector based on the analysis of the native code. It introduces the concept of Malware Analysis Intermediate Language (MAIL) to create a high-level representation of the native code, which is then used to build a behavioral signatures template.

Xue et al. [62] presented Malton, a dynamic analysis platform built on Valgrind for malware detection based on information flow tracking on Java and JNI code. In 2018, Wei et al. [59] presented JN-SAF to conduct static cross-language dataflow analysis of Android apps to track information leaks through the Java and the native parts. JN-SAF builts the analysis of the Java part of the app on top of Amandroid [60]: the analysis of the native components instead – for both 32 and 64-bit versions of ARM, MIPS, PPC, and Intel architectures – relies on the Angr's symbolic execution engine [57], In 2019, Lee Sungho [35] proposed a novel JNI program analysis technique that combines the analysis of Java and C code separately to extract semantic summaries of C code from JNI programs. In 2020, Andarzian et al. [8] proposed the CTAN framework, which extends JN-SAF to improve its performance. The same year, Fourtounis et al. [18] proposed an approach to recover JNI callbacks in the native code: disassemble native binaries, recover static symbol information, and produce a model for statically linking the native callbacks. In 2021, Samhi et al. [49] proposed JuCify, a framework that combines Android bytecode and native code into a unified model to detect data leaks. The native code analysis is built on top of Angr, while the Java code analysis and the unified model rely on the Soot framework.

4 Motivation & Methodology

We open this section by showing a practical example of the limitations of antivirus engines in analyzing native code. Then, after defining what we mean by "suspicious," we illustrate our methodology guided by an example.

4.1 Native Components and Antivirus Software

To begin with, we show how the static module of many Android anti-malware engines often ignores the presence of native components. In this respect, we collected famous exploits by querying GitHub and The Exploit Database [41]. We focused on generic, weaponized, ready-to-use exploits that compile for x86 and ARM32. We found four implementations: CVE-2011-1823 [11, 52], CVE-2014-3153 [12, 52], CVE-2016-5195 [13, 32], and CVE-2019-2215 [14, 33]. They date to 2011, 2014, 2016, and 2019, respectively. It is worth emphasizing that anti-malware engines are aware of these exploits; for example, the malicious app `cdde` [1] is labeled with the exploit name or the associated CVE (i.e., CVE-2016-5195 is also known as *DirtyCow*).

[1]Due to space limitation, throughout the paper, we just use the first four bytes of the sha256. The complete list is in Table 8.

Table 1. Evolution of detections (for those with more than zero) of our synthetic malware samples over time

			VT Scan Date (Year-Month)			
Hash	**CVE**	**Arch**	**2022-08**	**2023-02**	**2023-08**	**2024-11**
7383	2019-2215	x86	1/62	3/61	12/64	33/68
1f26	2019-2215	ARM32	1/61	2/61	13/64	27/69
e088	2011-1823	ARM32	1/61	2/62	7/64	6/68

Then, we created two native Android apps for each exploit, the first for the x86 architecture and the second for ARM32, for a total of eight synthetic malware samples. Each app loads and runs the exploit immediately at startup: it loads the library in the static constructor of the `Application` class and calls the function in charge of running the exploit directly from `JNI_OnLoad`, making the program flow to reach the exploit straightforward and trivial to analyze. The apps were signed with the default debug key and did not use any form of obfuscation or shrinking. The resulting eight apps were uploaded to VirusTotal and scanned with at least 61 different engines [2]. We repeated the scan every year for three consecutive years, from 2022 to 2024.

In the first analysis (August 2022), three apps were detected by only one engine (i.e., samples `7383`, `e088`, and `1f26`), and we obtained no detections for the remaining five (i.e., samples `b3c7`, `858e`, `f7b9`, `98f6`, and `5bd3`). We repeated the measurements twice with an interval of six months (February and August 2023) and finally one year later (November 2024). Those initially exhibiting zero detections remained stationary at zero, while those with one detection were detected by an increasing number of engines, as documented in Table 1. A variation in the number of detections over time is a well-known phenomenon. It occurs because anti-malware engines regularly update their models and signatures to improve their detection capabilities. For example, Zhu et al. [65] demonstrated how engines' decisions are influenced by the labelling of other engines, leading to collective improvements in detection.

As the current experiment employed only publicly accessible and widely recognized exploits, without any alteration or obfuscation, the observation that 5/8 samples had not yet been identified at the time of writing highlights a significant shortcoming in the existing commercial solutions, which this work hopes to help. It is also noteworthy that the remaining 3/8 samples were successfully detected within a year by a minimum of seven engines. We will therefore use this one-year range, from the first submission to the last analysis, to test whether this condition also applies to our malware dataset – as we will explain in Section 6.

4.2 Running Example

Listing 1. JNI example – Java side

```
1  package com.xmp;
2  public class MainActivity extends AppCompatActivity {
3  static { System.loadLibrary("xmplib"); }
4  public native boolean scheck();
5  public static native String whoknows(Object obj, boolean b);
6  public String getSubscriberId(){ return "no_id"; }
7  @Override protected void onCreate(Bundle b) {
8    super.onCreate(b);
9    ActivityMainBinding bind = ActivityMainBinding.inflate(getLayoutInflater());
10   setContentView(bind.getRoot());
11   String s = whoknows(getSystemService(Context.TELEPHONY_SERVICE), scheck());
12   bind.sampleText.setText(s); } }
```

[2]On VirusTotal, the number of engines available at any given time for an analysis is variable.

Listing 2. JNI example – Native side (C++ language)

```
1  jboolean JNICALL Java_com_xmp_MainActivity_scheck (JNIEnv* env, jobject thiz) {
2    return fopen("/system/xbin/sudo", "r") == NULL; }
3
4  static jstring sensitive
5  (JNIEnv* env, jclass jclazz, jobject obj, jboolean b) {
6    char* cName;
7    if (b) cName = "android/telephony/TelephonyManager";
8    else cName = "com/xmp/MainActivity";
9    jclass fclazz = env->FindClass(cName);
10   jmethodID method = env->GetMethodID(fclazz, "getSubscriberId", "()Ljava/lang/String;");
11   return (jstring) env->CallObjectMethod(obj, method);}
12
13 static JNINativeMethod nat_methods[] = {
14 {"whoknows", "(Ljava/lang/Object;Z)Ljava/lang/String;", (void*)sensitive },};
15
16 jint JNI_OnLoad(JavaVM* vm, void* reserved) {
17   JNIEnv* env = nullptr;
18   vm->GetEnv(reinterpret_cast<void **>(&env), JNI_VERSION_1_4);
19   jclass clazz = env->FindClass("com/xmp/MainActivity");
20   env->RegisterNatives(clazz, nat_methods, 1);
21   return JNI_VERSION_1_4; }
```

To better present our methodology, we continue the discussion guided by the code presented in Listing 1-2, which shows an example of an Android app that displays a string to the user in a blank Activity. The string is generated [line 1.11] from two native functions, `scheck` [line 1.4] and `whoknows` [line 1.5] implemented in the `xmplib` library (loaded at [line 1.3]). The C++ code declares three functions: `JNI_OnLoad` [lines 2.16-21], `Java_com_xmp_MainActivity_scheck` [lines 2.1-2] and `sensitive` [lines 2.4-11]. The native functions reachable from the Java code are those that respect the JNI *naming convention* [44] or the ones that are registered through the `RegisterNatives` callback. In particular, the `JNI_OnLoad` binds at runtime, through the `RegisterNatives`, the Java method `whoknows` to the native function `sensitive` [line 2.20]. Instead, the function `scheck` leverages the JNI name convention to register it and returns a boolean depending on whether or not the file `/system/xbin/sudo` exists – this is a trivial but common trick to check if the device has been rooted. Lastly, the `sensitive` function, depending on the value of its boolean parameter, uses `FindClass` to get the reference to the user-defined `MainActivity` class or the `TelephonyManager` of the Android framework, and, finally, invokes the `getSubscriberId` method of such class. Therefore, if `scheck` finds the `sudo` file, the code logic calls `whoknows` with a `True` argument and obtains the IMSI (considered very sensitive information) via `getSubscriberId`.

Although this is only an example, it shows two crucial aspects. First, these "jumps" between the Java and the native worlds and the `RegisterNatives` feature significantly complicate the static analysis, both from the perspectives of humans and machines. Second, calling a privacy-threatening API depending on the presence of a file showing whether the device has been rooted is not a clear malicious behavior because it depends on whether such personal data is being transmitted without adequate notice or consent. Still, it is a suspicious case that an analyst needs to investigate more thoroughly.

4.3 Suspicious Patterns

Through this work, we use the term *suspicious pattern* to describe any snippet of surreptitious code [39] (e.g., obfuscation and anti-tampering) that abuses the JNI execution cycle. More precisely, we define a *pattern* as a sequence (not necessarily consecutive because it could involve both Java and native code) of instructions that aim at preventing others from understanding the code, regardless of whether the effort is to uncover malicious purposes. Therefore, we emphasize that suspiciousness does not necessarily imply maliciousness. However, if an analyst were to observe one of these

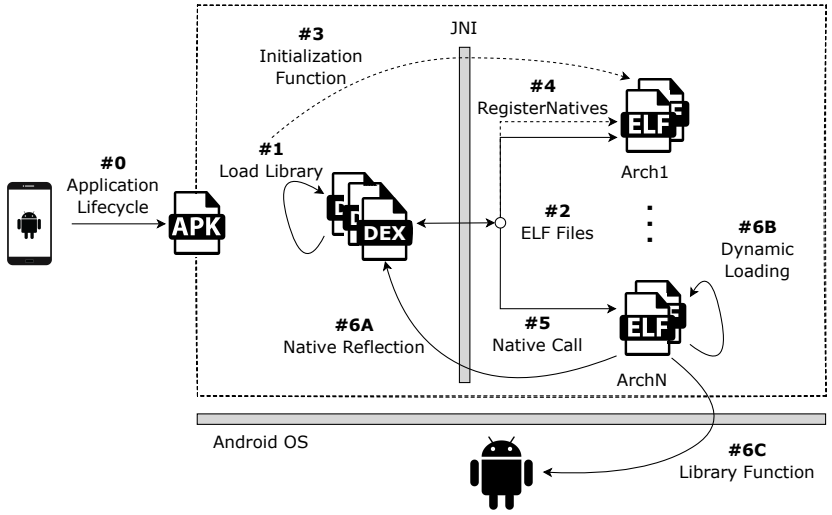


Fig. 1. JNI execution cycle

suspicious patterns, they should analyze them further as they *might* be used to protect harmful code (as shown in the running example in Section 4.2).

In the next section (Section 4.4), we illustrate the methodology we developed to guide both human and automated analysis by following the execution flow of JNI and describing the spots where suspicious patterns might lurk. Finally, in Section 4.5, we provide the complete list (Table 3) of all the cases we identified during our study. We also organize them into a clear taxonomy and assign a *tag* to each of them. Although our list is not exhaustive, as the malware ecosystem constantly evolves, our methodology details the methodology needed to identify new cases.

4.4 Methodology

As the first contribution of this paper, we propose a methodology to reverse engineer an Android app that uses the JNI. It has been developed through a collaborative effort between academics and industry experts, who jointly analyzed samples, consulted malware reports, and conducted scientific experiments. It comprises seven different *Steps* that characterize all the various aspects in which the JNI native code is involved in the apps' execution.

The execution order of runtime code is crucial to preserve when reverse engineering an Android app that uses the JNI. To closely follow the execution flow, we have broken down our analysis pipeline into the seven main steps, as represented in Figure 1.

The very first step, Step **#0**, extracts the possible entry points of the app from its Manifest file to understand the Android components (i.e., Activity, Service, Broadcast Receiver, Content Providers) and their lifecycles. Such information is fundamental to identifying all the application components that can be triggered and could potentially reach the JNI. For example, malware can gain persistence by registering a Broadcast Receiver for the `BOOT_COMPLETED` intent filter to start each time the device boots and loads the malicious native component. In our example, the `onCreate` method of the `MainActivity` [line 1.7] class is an entry point.

In Step **#1**, the Java code loads a native library. Calls to load methods mostly occur in the static constructor of the class that contains the native methods (e.g., line 1.3). An app can load these libraries by using the `load` or `loadLibrary` methods of the `System` and `Runtime` classes. The `load`

method requires the library name to be specified as an absolute path. In contrast, the `loadLibrary` requires that the name passed as an argument must not contain a file extension or path, as the library will be automatically searched in the default path. From a malicious perspective, the `load` method can be used to hide which ELF file is loaded. This type of analysis can reveal (e.g., sample `04CE`) whether a native library is not present in the APK but is downloaded at runtime once specific conditions are met, or whether the app loads a file that is not supposed to contain executable code (e.g., loads a PNG file as ELF file). Moreover, this approach can uncover the author's will to hide the actual library loaded by the app if the string passed as an argument to the load method is not defined in the code but computed at runtime.

In Step **#2**, JNI automatically loads the correct library according to the device architecture. However, malware can ship libraries with the same name, exposing and implementing functions with different names or semantics. For instance, if malware authors knew that specific antivirus solutions run the app in x86 emulators, they could avoid detection by restricting the malicious logic to the ARM architecture and placing a harmless code into the x86 library. The malicious component would evade the analysis since the sandbox will only load the x86 library. However, the malware would show its real behavior when executed on a real device that supports ARM.

In Step **#3**, the dynamic linker first invokes the initialization functions of the ELF file (e.g., `.init_array`), then the JNI automatically calls the `JNI_OnLoad` function (e.g., line 2.18). A malicious actor can hide the logic to perform evasive checks in the initialization functions. For example, we found samples (e.g., `019E`) using the `ptrace` function as an anti-debugging technique when it is loaded.

Furthermore, in Step **#4**, the `JNI_OnLoad` is used to dynamically link the JNI methods through the `RegisterNatives` API (e.g., line 2.22). In this way, the mapping between Java methods and native functions is no longer statically explicit but is performed at runtime. Thus, an attacker can perform environment checks and use the `RegisterNatives` to map different functions depending on their results (e.g., `7900`, discussed in Section 7).

At this point, Step **#5**, the Java methods can call the native functions of the loaded library (e.g., line 1.11). This transition is crucial as it might be impossible to determine the mapping between methods and functions statically and as different architectures might result in different semantics. For this reason, we designed three specific Steps (**#6A**, **#6B**, **#6C** in no particular order) to be followed once the execution moves from Java to the native library. Step **#6A** tracks the *native reflection*, which allows native code to manage Java objects through the JNI callbacks (e.g., create Java objects, invoke Java methods, or modify fields). For example, the `sensitive` method of the Listing 2 [lines 2.5-13] uses the JNI callbacks `FindClass`, `GetMethodID`, and `CallObjectMethod` to get a reference of the object `TelephonyManager` and call its method `getSubscriberId`. This possibility significantly complicates the analysis because it makes it impossible to statically determine which code will be executed without resolving the arguments of such methods. In addition, it is also worth noting that the native reflection can also use Java reflection features to hide methods or accessed fields. For instance, sample `4E4B` uses the `getDeclaringClass` method of the `java.lang.reflect.Method` class to dynamically retrieve the class representing an object, and then leverages the native reflection ()`FromReflectedMethod` callback function) to retrieve the corresponding method from the object.

In Step **#6B**, the native code can dynamically load and invoke exported functions of other libraries by relying on the `dlopen` and `dlsym` functions. This technique aims to conceal the usage of a particular shared library from static analysis, given that it is no longer present in the dependencies. Even if the system should prevent loading or linking those kinds of libraries since Android 7, we found multiple samples (e.g., `1306`) that use dynamic loading to load and call functions from a library that is not present in the APK.

The last step to consider (**#6C**) is the use of library functions that can affect security, for instance, by running exploits against specific subsystems or by performing environment checks (e.g., debugger

Table 2. Security-relevant library calls

Category	Library Calls
Dynamic Loading	dl(v)sym, dl(m)open
Execution	exec*, system, popen
File Permission	*chmod*, *chown*, access
Kernel Interaction	ioctl, syscall
Identity	get(e)uid, get(e)gid
Memory Protection	mmap, mprotect
Network	socket, listen, connect, gethostbyname
Open Special File	*open* <special_file_path>
Process Management	kill, ptrace, fork
Monitoring	inotify_*

detection). We report the complete list in Table 2, divided into nine categories: Dynamic Loading (e.g., `dlopen`), Execution (e.g., `system`), File Permission (e.g., `chmod`), Kernel Interaction (e.g., `ioctl`), Identity (e.g., `geteuid`), Memory Protection (e.g., `mprotect`), Network (e.g., `gethostbyname`), Open Special File (e.g., `open("/proc/version")`), Process Management (e.g., `ptrace`), and Monitoring (e.g., `inotify_add_watch`, because a recent work showed how an attacker can abuse inotify to perform state inference attacks on Android [48]).

4.5 Suspicious Tags

A *tag* is made of the concatenations of two strings interposed by the symbol "-"; the first string is the category, and the second is a title. Referring to our methodology, the category is one of the steps presented in Figure 1, while the title identifies the suspicious pattern within the category.

We report all tags in Table 3. Since in Section 8 we will use them in a machine learning algorithm for a binary classification task, we also specified their type (float or boolean) when we convert them to numbers for the feature vector. '<SYMBOL>' denotes that there is a specific tag for each suspicious library call family (see Table 2).

To give some examples, a node tagged with `NR_FINDCLASS-JAVA_REFLECTION` (i.e., category: "`NR_FINDCLASS`", and title "`JAVA_REFLECTION`") denotes a call to the `FindClass` callback and its argument refers to a Java class related to reflection; thus, the developer is using native reflection to use Java reflection. Also, the `REGISTERNATIVES-CLASS_NOT_IN_APK` tag refers to a call to the `RegisterNatives` callback, but its class argument refers to a class not defined in any DEX file; thus, that code is not available statically. Moreover, in our running example (Section 4.2), by using the `GetMethodID` [line 2.11], the developer could obtain a reference to the `getSubscriberId` method of the `android/telephony/TelephonyManager` class, which requires a dangerous permission (also known as runtime permissions [28], i.e., permissions that allow actions which significantly affect the system and other applications). Thus, our methodology tags this code with the `NR_METHOD-WITH_DANGEROUS_PERM` tag.

It is worth noting that suspicious tags serve three main purposes. First, they can assist a human analyst by showing precisely which parts of the code need to be inspected because they might conceal surreptitious code facets, such as evasive techniques, malicious behavior, or protection mechanisms. Second, it provides automated analysis systems with target locations that could be investigated using more costly but more precise analysis routines (e.g., symbolic execution or dynamic analysis). Finally, the suspicious tags can improve classification tasks, as shown in Section 8.

Table 3. List of suspicious tags.
†: float computed as $\frac{\#\ of\ features}{total}$
§: boolean

Step	Category	Category Description	Title	Example
#0	J_NATIVE_METHODS	Presence/absence of native methods and the entry point from which they can be reached	NO_NATIVE_METHOD†	
			NO_REACHABLE†	
			APP_LIFECYCLE_EP†	
			ACTIVITY_LIFECYCLE_EP†	
			EXTERNAL_DEX†	
			SUSPICIOUS_INTENT§	
#1	J_LOAD_METHODS	Presence/absence of load methods and the entry point from which they can be reached	NO_LOAD_METHODS§	
			PATH_LOAD_METHOD†	
			APP_LIFECYCLE_EP†	
			ACTIVITY_LIFECYCLE_EP†	
			SUSPICIOUS_INTENT§	
			EXTERNAL_DEX†	
			NO_ELF_NAME†	
			ELF_IN_LIB_AND_NOT§	
#2	CODE_LOCATION	Code file in suspicious location or with extension name mismatch	ELF_IN_ARCHIVE§	
			DEX_EXT_MISMATCH§	
			ELF_EXT_MISMATCH§	
#4	REGISTERNATIVES	`RegisterNatives` callback	UNRESOLVED_METHODS§	
			MULTIPLE_PATH§	
			CLASS_NON_IN_APK§	
#6A	NR_FINDCLASS	Native Reflection: `FindClass` callback	ANDROID_MANAGER§	
			CONTEXT§	
			CLASSLOADER§	
			JAVA_REFLECTION§	`java.lang.reflect.Method`
			THREAD§	
			SYSTEM§	
			CRYPTO§	`javax.crypto.Cipher`
			APP_INFO§	`android.content.SharedPreferences`
			ZIP§	
			ANDROID_INTERNALS§	`android.app.LoadedApk`
			STACK_TRACE§	`java.lang.StackTraceElement`
			EXCEPTION§	
			PARTIAL_RESOLUTION†	
			NO_RESOLUTION†	
	NR_METHOD	Native Reflection: `GetMethodID` callback	WITH_DANGEROUS_PERM§	
			ANDROID_MANAGER§	
			CONTEXT§	`getSystemService`
			SENSIBLE_INFORMATION§	`getImei`
			CLASSLOADER§	`loadClass`
			JAVA_REFLECTION§	`getClass`
			THREAD§	
			PERMISSION§	
			STACK_TRACE§	`getStackTrace`
			PARTIAL_RESOLUTION†	
			NO_RESOLUTION†	
#6B	DYNAMIC_LOADING	The tags report the usage of library call to dynamically load and invoke exported functions of other libraries	<SYMBOL>§ (see Table 2)	`dlsym(fd, "chmod")`
			NO_RESOLUTION§	
			ANDROID_DVM_ART§	`libdvm.so`
#6C	SUSP_LIB_CALL	Suspicious library calls	<SYMBOL>§ (see Table 2)	`execve`
			CREATEJAVAVM§	`JNI_CreateJavaVM`
	LIB_CALL_SUSP_PARAM	Suspicious argument to the library calls	<SYMBOL>§ (see Table 2)	`open("/proc/version")` `open("/sys/devices")`
			CREATEJAVAVM§	`JNI_CreateJavaVM`
#6A #6C	STRING	Presence of meaningful strings	CLASSLOADER§	
			ANDROID_INTERNALS§	
			PROPERTIES§	`ro.product.cpu.abi`

5 Suspicious Analysis Framework

This section describes our second contribution: *ANDani*, a cross-architecture analysis framework that implements our analysis methodology.

5.1 Overview

Our system receives an APK file as input and returns an Inter-Procedural Control Flow Graph (IPCFG) with each node eventually tagged (using the suspicious tags described in Section 4.5) in case it detects a suspicious pattern.

It first unpacks and extracts its DEX, JAR, and ELF files. This operation is performed recursively on all the archives (e.g., ZIP, TAR) inside the APK. It also parses the `AndroidManifest.xml` file to extract relevant information, such as the Android components, the required permissions, and the various intent filters. Next, *ANDani* starts the analysis by computing an IPCFG for both the Java and the native code. First, it computes the IPCFG of every code file, then merges them into a single IPCFG, keeping track of whether the code file was found in a standard or non-standard location within the APK. This information is crucial to identify potentially malicious code in non-standard locations (e.g., in an archive file).

The IPCFG is based on two types of nodes: *code blocks* that are the traditional basic blocks that are interrupted by function calls, and *call blocks* that represent the function calls or method invocations. Moreover, our analysis also considers that from a call block of a Java native method, we can have multiple edges to different native functions of different architectures. In the same way, when we deal with the *native reflection* (see Step #6A of Section 4) where the function can take multiple arguments (e.g., lines 2.7-11), the graph can also have different edges to Java methods.

The *Bytecode module* handles the IPCFG for all the bytecode components, and it is built on top of the Soot framework [31, 56]. For the native components, the analysis is performed by the *Native module*, which leverages the Ghidra [4] API to process the ELF files.

Performance. On our machine, an Ubuntu 20.04 with 64 CPUs (Intel Xeon 8160 @ 2.10GHz) and 128 GiB of RAM, we measured an average execution time of 614 seconds (std. dev. 182) for a single APK. Such variance is due to the fact that the analysis duration of *ANDani* grows up proportionally with the code (i.e., ELF or DEX files) in the APK.

5.2 Bytecode Module

The Bytecode module performs the analysis of DEX and JAR files. It is written in 1814 lines of Java code, and it is based on the Soot framework. The module starts by translating the bytecode into Jimple, a three-address code [5] intermediate representation that Soot needs to build the preliminary IPCFG.

Soot suffers from many known limitations in the case of parallel and asynchronous Java classes, i.e., those that extend or implement `Thread`, `Runnable`, `AsyncTask`, or `Timer` (we show an example in Appendix 10.3). Such drawbacks make the IPCFG incomplete, and therefore, we had to add the necessary code –complete with test cases– to handle them.

Once the IPCFG is computed, our system continues the analysis by identifying the calls to the *load* methods that allow loading native libraries. Each time a call is found, it tries to resolve the argument (a string) to identify which native library is loaded. For this, we perform a backward intra-procedural taint data analysis. This is sufficient in most cases because, as we previously mentioned, calls to load methods almost always occur in the static constructor of the class that contains the native methods, which also references the plaintext string with the library's name.

The system then repeats this procedure for each DEX and JAR file, and at the end of this phase, it produces the IPCFG of the whole bytecode found in the APK. Then, the analysis identifies the

entry points of the app by combining the information from the manifest file (e.g., the `onCreate` method of the main activity) and the list of entry point methods of previous studies [9, 60]. As the last step, the system identifies all native methods and extracts their signatures, which serve as input to the next module.

5.3 Native Module

The Native module is written with 5198 lines of Java code, and it uses the Ghidra reverse engineering framework API to perform headless analysis of all ELF files. The analysis leverages the Ghidra *P-code* intermediate representation to model the behavior of many architectures. Since the signatures of the Java native methods are fundamental to propagate types of information in the native code properly, this module is executed after the Bytecode module. The output of the Native module is the final, complete, and merged IPCFG.

The analysis performed at this stage can be divided into four different phases:

I) ELF information and JNI entry point identification. Given an ELF file, the module extracts generic information from the ELF header (e.g., architecture, sections, segments, symbols, strings), and it initializes the set of the JNI entry points, i.e., the native functions that can be reached by Java code. To start, we consider as JNI entry points the functions whose symbol name respects the JNI naming convention – i.e., symbol name starts with `Java_` or `JNI_` – and all ELF initializers. If we consider the example shown in the previous section, the entry points are: `JNI_OnLoad` [line 2.16] and `Java_com_xmp_MainActivity_scheck` [line 2.1].

II) Entry point arguments type definition. The module iterates the entry points: for `JNI_OnLoad` and ELF initializers, if present, it creates an edge from the respective Java load method to the first of these being called, and the others are propagated. For the `Java_` functions, given the signatures of Java native methods collected from the Bytecode module, it creates an edge from the corresponding Java method and applies the proper JNI data type to all the input parameters. In case a `Java_` function is not found, it just applies the `JNIEnv*` type to the first one. Once the argument types are updated in the JNI entry point, the module propagates them to the called functions. If we consider our initial example, in that case, we have an edge from the `loadLibrary` [line 1.3] to the `JNI_OnLoad` [line 2.16].

However, in case the native functions are dynamically registered through the `RegisterNatives` (e.g., lines 2.20-22), the function names may not start with `Java_`, as it is the case in our example for the `sensitive` function. To handle these cases, the module searches for all `RegisterNatives` calls in the code of the entry points, and, for each of them, it performs a backward taint data analysis again on its second and third arguments. The second argument of the `RegisterNatives` is a `jclass` object obtained from a call to the method `FindClass` of the `JNIEnv`. In turn, the `FindClass` takes a string with the corresponding Java class name; it is marked as tainted and searched backward. The third argument is a `JNINativeMethod` that contains the mapping between the Java methods and native functions. Lastly, after this procedure, the module re-iterates the JNI entry points to apply the correct data types. This phase is crucial to get the correct types, especially `JavaVM` and `JNIEnv` since they expose all the JNI callbacks.

Following the example, the module, after correctly resolving the arguments, can now create an edge from the Java method `fun2` to the `sensitive` native function and apply the correct type to its arguments.

III) Graph construction. At this last phase, since all the entry points and the types of their arguments have been retrieved, the module can create and merge the final IPCFG. First, the module follows call instructions with a depth-first strategy, propagating the types of arguments to the called

functions each time and searching for interesting functions. It is worth remembering that we are not just interested in JNI callbacks but also specific Android system library calls, as their arguments can reveal helpful information about the app's capabilities. The analysis of the information collected by analyzing the arguments of these calls is discussed in more detail in Section 7.

Once the module finds an argument of interest, it marks it as tainted, and it performs a recursive backward taint analysis on the tainted arguments to resolve them until a loop is detected or *ANDani* resolves it. For example, *ANDani* would try to resolve the path of each opened file to detect suspicious accesses to Linux special files, such as `/proc/version` to retrieve the version of the Linux kernel [34] (sample `0259`). This phase of the analysis also considers all the values a variable can take among those that were able to recover.

In case arguments of JNI callbacks that interact with Java objects (e.g., `CallObjectMethod`) are correctly resolved, *ANDani* creates an edge between this call block and the corresponding block of the Java CFG. E.g., in our running example, the graph construction would add two edges (because the argument of the `FindClass` is not unique) from the `CallObjectMethod` [line 2.11] to the `getSubscriberId` method: the first to the Android framework `TelephonyManager` class, and the second to the `MainActivity` class [line 1.6].

IV) Suspicious tagging. In the final phase, *ANDani* visits the IPCFG, and if it finds a match with one of the suspicious patterns, it assigns the corresponding tag (Table 3) to the node (or nodes) where the violation occurs. From a practical standpoint, we developed a Python script for this purpose, with a modular architecture that is easy to extend with new patterns.

5.4 Comparision with state-of-the-art tools

During the design phase of *ANDani*, we investigated the state-of-the-art tools for static analysis to understand which technologies are best to rely on.

As discussed in Section 3, most of the works focus on the Java layer and do not handle native code. However, we noticed that there is a tool in common among the most cited works (e.g., Epicc [40], and Flowdroid [9]): the Soot framework [56], which is still under active development [31], and therefore it was our choice since the alternatives are no longer supported (e.g., Amandroid [60]). However, as already discussed, we had to improve the analysis of some typical Android mechanics, namely, the detection of entry points and concurrent execution management.

Then, for inter-language analysis, we identified JN-SAF [59] and JuCify [49]; these tools perform taint analysis between different code layers to detect data leaks. As for the analysis of the Java layer, the former is based on Amandroid, while the latter is based on Soot. On the other hand, they both use Angr [57] to analyze the native layer.

We tested Angr with the same configuration of JuCify. However, from several preliminary tests on native Android libraries taken from real-world goodware apps (the same we used in Section 6) we discarded it because the one-hour timeout was often reached (because of the path explosion typical of symbolic execution), thus making it unsuitable for our large-scale measurement. Moreover, JuCify's approach is not suitable for our needs. Using Soot, they lift Java code to Jimple, a 3-address intermediate representation (provided by Soot); then, they extended Angr to lift native code to Jimple so they can reason on a unified representation. The issue is that we are interested in every single instruction in the native code, as we need as much precision as possible; instead, JuCify just considers call instructions because its purpose is to create a call graph while we need the interprocedural control flow graph. Regarding JN-SAF, it is no longer maintained and does not investigate specific aspects of native code we are interested in (e.g., library calls and their arguments). Given that it is also based on Angr, we did not investigate further and then decided to write the analysis of native code from scratch.

Table 4. Distribution of the AndroZoo suspicious samples

	Detection Range				
Year	**Fam. 1**	**Fam. 2**	**Fam. 3**	**Singleton**	**Total**
2010/11	5.0%	5.0%	5.0%	8.4%	5,065
2012/13	5.0%	5.0%	5.0%	7.0%	15,458
2014/15	5.0%	5.0%	5.0%	16.5%	19,127
2016/17	5.0%	4.8%	4.4%	37.3%	20,268
2018/19	5.0%	5.0%	4.3%	48.5%	19,610
2020/21	3.7%	0.5%	0.5%	84.3%	18,301
Total:					97,829

6 Dataset

To perform our analysis, we built a comprehensive dataset of Android apps, which is divided into *malware* samples collected over the past ten years and a *goodware* dataset of benign apps. All malicious samples are downloaded from AndroZoo [7]. In addition to the file, AndroZoo also provides the date associated with the APK file, the number of antivirus (AV) engines that detected the app as malicious on VirusTotal (VT), and the first submission date to VT.

However, according to their documentation, most apps from Google Play have 1980 as the APK date. Therefore, we assigned each app to a year by applying the following procedure: if a year was present and had a plausible value, i.e., other than 1980 and between 2010 and 2021, we consider that to be the year. Otherwise, we assign the year of the app as the year in which the first submission in VT was performed. On the other hand, benign apps were collected among the most downloaded apps from the Google PlayStore for each of the 50 categories [25]: 15 categories are related to *Games* while the remaining 35 vary from *Communications* to *Social*.

Since this research focuses on the usage of the native component via JNI by Android apps, our dataset consists only of apps that make use of this technology. Therefore, each app in our dataset respects at least one of the following two constraints: it must contain a DEX file with a declaration of at least one `native` method that is not defined in the standard Android libraries, or it must contain an ELF Shared Object file with a JNI entry point method.

Goodware. We collected the package names of the 500 most downloaded free apps for each of the 50 official Google Play Store categories. We extracted this information using *Google Play Scraper* [15], and we downloaded the samples with *Playstore Downloader* [19]. The tool was able to download 27,665 apps successfully. After our pre-filtering, which only retained apps that use native components, we were left with 15,647 samples. The fact that more than half (57%) use a JNI component is a clear sign that, nowadays, native components constitute a fundamental part of the Android userspace ecosystem.

Malware. We considered "malware" any sample with at least five AV detections to minimize false positives, as recommended by Zhu et al. [65]. Moreover, guided by the results of Section 4.1, we downloaded the corresponding VT report, and we verified that the date of the last analysis was at least one year after the first submission; if not, we re-submitted the sample to VT for a fresh scan. In this manner, we incorporated a number of malicious samples that might have been overlooked at the time of their initial submission.

Then, Androzoo does not indicate the family the samples belong to, so we had to download the respective report from VirusTotal and determine the family via AVClass2 [50]. From a preliminary analysis, we found that the samples, grouped by year, are overrepresented by a few families (e.g., among 10k random APKs 41% of the malicious samples in the year 2014 belong to just three families).

We, therefore, opted to group samples by pair of years, with a maximum of 5% for each family, and the sha256 hash of each sample belongs to just a pair of years (namely, the intersection between the samples for each pair of years is empty).

We ended up with 97,829 native malicious apps, whose distribution is summarized in Table 4, where we report the percentage of the three most frequent families. It is worth noting that the number of malware that AVClass2 cannot assign to a family (*Singleton*) has increased over the years. This is due to the fact that the number of AV engines has increased, and there are more inconsistencies in the naming convention of family labels; furthermore, we observed that in the Androzoo dataset, the average number of detections (also by referring to updated VirusTotal reports) in recent malware is lower than the old one.

7 Results

This section presents and discusses the results of the longitudinal measurement we conducted over malware and goodware datasets. To better understand why malware uses native code and how it is tied to the app's lifecycle, we will report the results according to the seven main steps of our methodology (discussed in Section 4.4). Since we have grouped the malware into pairs of years, we will refer only to the highest year in the pair (e.g., 2011 refers to the pair 2010/2011) to improve readability. For the same reason, we omit the decimal place from percentages when not strictly necessary.

7.1 Application Lifecycle

The first question we want to answer with our measurement is *when* apps invoke JNI methods, i.e., whether the native component comes into play immediately after the application starts or whether it is only invoked when specific conditions are met. The formulation of this question stems from the fact that understanding whether there is a difference between goodware and malware could help improve analysis by prioritizing the prevailing case.

For each of the native functions, *ANDani* first visits the Java IPCFG to verify if a native method is *reachable* (we consider a method to be reachable if there is at least one block of the IPCFG that calls such method) and, if it is, it extracts all the possible entry points of the different paths that lead to it. This first analysis shows that among goodware, 91% of the native functions are reachable from nearly all (99.8%) of the DEX files in the standard location. Thus, the code invoking the native component is easy to identify and is not obfuscated nor dynamically loaded. Moreover, it is essential to highlight how 94% of these functions are reachable only under specific *user interaction* with the app, such as a click on a GUI item. Among the remaining, 4.4% of the native functions are reachable from the lifecycle methods of Activity components, and the remaining (1.2%) is triggered at the app's startup, from static constructors, or other Android components (i.e., Service, Broadcast Receiver, and Content Provider).

The picture is utterly different for malware. In fact, among malicious apps, the average number of reachable native functions has decreased from 81% in 2013 to 21% in 2021. Moreover, the number of reachable native code only from DEX files not located in the standard position is always higher than 2%. These results suggest that malware uses resources loaded at runtime to invoke native methods. Furthermore, since 2017, more than 34% of the native functions have been reachable at the startup (i.e., directly from the Application class), with a peak of 53% in 2021. This observation is significant because it shows how malware, unlike goodware, tries to start the native component as quickly as possible. Comparing the 2021 percentages for both malware and goodware, we observe that for the malicious applications, 55% of the entry points are Application (53%) or Activity (2.4%) lifecycle methods, and only 44% are related to user interaction. This highlights once more that

current malware mainly invokes native code at the beginning of the process and does not wait for the user to interact with the application.

Another interesting aspect that differentiates the use of alternative entry points used by goodware and malware concerns Broadcast Receivers. Although their use is very limited in percentage, our data shows that malicious apps are more prone than goodware to use broadcast receivers to invoke native functions when they are notified that the user is present, an existing app has been added or removed from the device, an external power has been (dis)connected to the device, or the device boots.

The final analysis measured when an app declares Java native functions that are not exposed by shared libraries and vice versa (i.e., exported JNI entry points not declared in Java code). Our results show that these discrepancies are much more prevalent in malware than in goodware. For instance, in 2017, 49% of malicious apps exported JNI entry points were never declared in the Java code, and it grew over the years until 84% in 2021. In goodware, this behavior only appears in 21% of the apps. One possible explanation, confirmed later in this section, is that malicious apps dynamically load Java code from native and then use this new Java code to invoke other exported native functions. This cycle of redirection between Java and native layers is a form of obfuscation that makes static analysis much more complicated to perform.

Observations: The native methods declared by benign apps are reachable in most cases from the main application code, and the trigger for their execution is very often dependent on user interaction. Concerning malware, we observed a significantly different trend: the average number of reachable native functions from the main application codes has significantly decreased, consequently implying a strong presence of Dynamic Code Loading. Moreover, we noticed how the invocation of native methods is almost immediate and occurs mainly without user interaction. Finally, malware often uses Broadcast Receivers to wait for a particular event and trigger the native code. Therefore, during manual investigation, we need to prioritize the analysis of code that is invoked directly and without interaction.

7.2 Load Methods

Android apps can load native libraries through the `load` and `loadLibrary` methods. We recall that the former accepts the full path of the library, while the latter accepts only the name of the library – which is loaded from the default folder.

The 86% of loading operations in the goodware dataset load shared libraries directly from the standard location using the `loadLibrary` method. Until 2013 this was also the preferred method among the malicious app, with more than 89% of such operations relying on `loadLibrary` and only 11% on the `load` method. From 2013 to 2021 instead, this percentage steadily decreased, and today the `load` method accounts for 42% of loading operations against 14% in goodware.

A second crucial aspect is when these libraries are loaded to make the native methods accessible. The JNI common practice suggests loading the libraries within the static constructor so that native methods will be immediately available and exposed to the rest of the application. We measured this from 2011 to 2015: more than 69% of malware was loading libraries from a static constructor. However, from 2017, we observed a change that saw samples loading libraries in other points of the app's execution until 2021, when 80% of malware loads native libraries from other code locations. Goodware reinforces this phenomenon by loading libraries from different entry points, but still, 42% of loading operations are performed by static constructors.

Other interesting aspects of the Application Lifecycle analysis concern user interaction and response to specific system events. In the first case, our experiments show that 52% of goodware loads native libraries only in response to user interaction, while malware performs this behavior

only in 16% of the samples. Moreover, malware tends to use broadcast receivers to load native libraries in response to particular events, such as an external power having (dis)connected to the device, an external media being (un)mounted, or the device boots.

Finally, we analyzed the names of the libraries that goodware and malware load. Our data shows that, throughout the years, malware loads significantly more libraries related to packing (e.g., `jiagu` [1]), obfuscation, encoding/encryption, or audio recording. On the other hand, a high percentage of goodware includes libraries from well-known frameworks, such as Unity [53] and Flutter [16].

Observations: Goodware is much more adherent to the good practices, like loading libraries via the `loadLibrary` method, thus making the analysis easier – given that they are loaded from a single location and must be present in the APK. This was the same trend observed in malware until 2015, while it is changing in favor of the use of `load` method. This fact brings numerous problems to static analyses as it may not be possible to know in advance what library will be loaded or where it is located. In addition, recent malware is more likely to include protection libraries, such as packers and obfuscators.

Table 5. Supported architectures over the years [%]

	ARM 32-bit	ARM 64-bit	x86	x86_64	MIPS 32-bit	MIPS 64-bit	Others
Good '21	80.49	81.26	59.32	43.27	11.18	6.71	0.15
Mal '11	99.98	1.68	7.65	0.68	4.05	0.10	0.00
Mal '13	99.99	0.33	8.92	0.13	2.59	0.04	0.01
Mal '15	99.99	3.52	38.42	1.77	12.29	1.53	0.00
Mal '17	99.99	30.93	65.49	22.86	17.86	9.97	0.18
Mal '19	99.96	46.42	69.98	29.31	8.90	5.36	0.50
Mal '21	99.54	78.51	87.97	45.69	2.19	1.20	1.43

7.3 ELF files

To support different architectures, APKs include ELF libraries in `lib` folder, which are in turn divided into different subfolders, one for each architecture supported by the app. Table 5 details the evolution of the architectures supported by malware and goodware.

Google Play introduced the *App Bundle* [24], a new publishing format to generate and serve optimized APKs for each device configuration. This ensures that new applications that use native libraries do not have to ship their APK with multiple versions of the same native library since the correct version will be directly shipped at installation time. Despite this new feature, we observed that 75% of the goodware still ships the same library object for multiple architectures, and more than 11% still includes MIPS – even though it is no longer supported [23]. In comparison, almost all malware samples include ELF files for ARM32, and the number of samples with multiple architectures grew over the year, from 9% in 2011 to 74% in 2021. Since 2017, malware samples also contain few ELF files that target different architectures (e.g., `SPARC` and `PowerPC`), which grows up to 1.4% in 2021. We also observed a small percentage (about 0.1% every year) of ELF files with a broken header section, while none of the goodware had this peculiarity. By inspecting some of these cases manually, we identified a common technique that is used to prevent static analysis: the authors have removed a part of the ELF header from the libraries, and the missing part is only restored at runtime.

Afterward, we analyzed where the ELF files are located in the APK. Most of the goodware (78%) contain ELF files only in the standard `lib` folder, while 10% ships ELF files only in non-standard (e.g., `assets` folder or archives) locations. On the other hand, about 85% of the malware contains

ELF files in a non-standard location, which reflects the high usage of the `load` method in malware. Moreover, malicious APKs also embed ELF libraries in archives or try to disguise the analyst by hiding the executable with a wrong and harmless extension name (i.e., different from `.so`). More than 5% of ELF files in the malware samples were extracted from an archive or have an extension that did not match the file type. The most common wrong extension names we found are: `png`, `jar`, `sdk`, and `lib`.

We also found another interesting phenomenon: since 2017, more than 1% of malware contains an ELF file with the same name that targets the same architecture both in the standard folder and in a non-standard location. Therefore, we decided to manually examine those ELF files with the same name and architecture. We computed the Jaccard similarity coefficient between the set of JNI entry points and obtained an average index of 0.7 when there should be no difference. It indicates that the two files have very different entry points. For instance, sample `213c` contains an ARM library in both the `lib` and the `assets` folders. The ELF library under the `assets` folder exports one entry point more than the one in the `lib` folder, and such an entry point is in charge of executing a well-known exploit [52] to escalate privileges and obtain root capabilities. In general, from 2015, more than 6% of such libraries (with a peak of more than 30% in 2019) implement the `JNI_OnLoad` entry point only in the non-standard location.

Finally, we searched how many ELF files are shared among different malware samples and found that the percentage of ELF files shared between at least two malicious APKs is 46% (60,895/131,325). Among the top 1,000 shared ELF files, 59 (6%) have more than four detections, and some have up to 41 detections in VirusTotal. Interestingly, among the top shared ELFs in a non-standard location (e.g., `asset` folder), more than 10% has a VT detection higher than five. Moreover, 7% of the shared ELFs are contained in different places depending on the malware. It highlights how malware tends to include malicious executables not in the `lib/` folder, reinforcing the results of the previous sections that malware leverages the `load` method to get libraries from various locations. For instance, the `6c6e` ELF file is shared among 293 APKs, and it is loaded from the `res/` folder. We then measured the number of years in which these malicious ELFs were observed in our dataset, and we obtained a median of four years. The most extreme case is `d867` which is shared by 54 APKs from 2011 to 2021 and has 32 detections on VT. As it turns out, malware authors are not too concerned about antivirus software and do not even try to change the hash of the malicious components they kept reusing year after year.

Last, our data shows that 6% of the ELF files is shared between at least one malware and one goodware sample. This highlights how malware provides goodware-like features to fool the final user (e.g., repackaged apps), or goodware's developers try to protect their code with obfuscation techniques.

Observations: From these results, we can conclude how malware has adapted to support more and more architectures. This can indicate many factors, the first of which is to try to be effective on as many devices as possible. Changing the file's extension is a trivial and common trick, while removing the ELF header and restoring it at runtime is certainly more sophisticated and, thus, less prevalent. Using the same name for different libraries is a clear sign that the dispatch might change depending on some checks. We have noticed that the reuse of malicious native libraries is a frequent practice that lasts over the years, showing carelessness in concealing it. Finally, we highlighted how malware and goodware have native components in common to protect their code from reverse engineering.

7.4 Initialization Functions & JNI_OnLoad

When the dynamic linker loads a library, it first calls the initialization functions defined in well-known ELF sections. Samples in both datasets have about the same percentages of initialization functions distributed across sections: while the `.init` section is used by less than 1% on average, `.init_array` is more prevalent, being present in roughly 30% of the apps.

After the ELF initialization functions are executed, JNI calls `JNI_OnLoad`. Here we noticed an interesting trend: while the usage of `JNI_OnLoad` in the goodware dataset is around 60%, the percentage of malware that uses it had been steadily increasing over the years until 2017, after which it remained stable at 92%. This brings another important observation: `JNI_OnLoad` is the function where usually the `RegisterNatives` is used to register the methods dynamically, thus hiding the mapping between Java and native functions. Therefore, it is possible that malware use this construct more frequently, specifically to prevent the detection of which native methods are executed by the sample. During the years, the percentage of such malware that exports only the `JNI_OnLoad` as JNI entry point decreased from 48.1% to 18.1%, while, for goodware, it is stable at around 42%. This shows that malware authors jointly use both techniques to register the JNI entry points to hide the mapping for specific functions. Moreover, we measured the average number of branches of the `JNI_OnLoad`: for malware, it grew up over the years until 91 (std. dev. 131) in 2021, while, for goodware, it is only 44 (std. dev. 110); a clear sign that this function is often abused by malware authors and must be analyzed with particular care.

ANDani tries to resolve which methods are dynamically registered by computing the arguments of the `RegisterNatives` calls. The usage of this callback in malicious apps has increased over the years up to 71% of samples, while it accounts only for 30% of the goodware samples. We recall that the `RegisterNatives` dynamically maps a Java method defined as `native` to its implementation in the shared library at runtime, accepting two distinct input parameters: the Java class and the function-method mapping. *ANDani* resolved the mapping in 88% of the cases for benign samples, thanks to the fact that goodware often uses hardcoded values. For malware, the percentage decreases to 79% averaged over the years as a consequence of the fact that these samples tend to obfuscate the parameters' value. Conversely, resolved Java classes are approximately 60% for goodware and 55% for malware since 2019. For the arguments we were able to resolve, we inspected whether the related classes were defined within the code of the APK, the framework, or if they were not present in either. We identified how the number of Java classes used in the `RegisterNatives` that are not present in the APK is higher in malware. In goodware, 78% of the resolved classes point to defined references (and all the other points to Android framework classes, such as `androidx.renderscript.RenderScript`), while for malware since 2019, this only accounts for 36% of the classes.

We also noticed that malware samples perform various checks (e.g., environment controls, anti-debugging checks, etc.) and map different functions depending on their results. For example, we identified one APK (`7900`), which loads a native library (`32cd`), and it leverages this technique to invoke the `RegisterNatives` callback and to map different Java methods depending on the context in which it is analyzed. A more detailed manual analysis revealed that the sample decrypts a string, and if a class with the same name exists, it maps the native functions to such class; otherwise, if the check fails, it decrypts another string and repeats the procedure.

Observations: The investigation of the `JNI_OnLoad` function is a crucial aspect in the analysis of Android malware. In particular, using `RegisterNatives` with different hidden arguments, based on environment checks, with the goal of mapping different functions at runtime, can be considered an anti-analysis technique for both static and dynamic analysis – that, to the best of our knowledge, has not yet been documented.

7.5 Native Behavior

Finally, the native libraries (steps **#6A**, **#6B**, **#6C** of our methodology presented in Section 4).

Native reflection. A shared library might leverage the JNI callbacks to communicate back with the Java world. Our measurement revealed that malware samples, especially the most recent ones, adopt this behavior more often than goodware.

Over the years, the usage of native reflection by malicious apps significantly increased, reaching over the 90% in 2021. On the other hand, the adoption of JNI callbacks in benign software is present in 57.1% of the applications that use native components. In particular, among the several JNI callbacks available, we noticed that the usage of the `FindClass` and `GetMethodID` callbacks are about 35% higher in malicious apps than in goodware. In addition, *ANDani* succeeded in recovering over 81% of the arguments for goodware for both the callbacks, while it resolved less than 68% of `FindClass` and 70% of `GetMethodID` for malware from 2017. In half of the cases, both goodware and malware apps access classes and methods of the Android framework, while the remaining involve app-specific classes. About goodware, over 72% of these custom classes are present in the APK, whereas, in malware, this happens only in 21%. This second result could indicate how malware tries to communicate natively with Java components not present in plain within the APK. We suppose that these classes are present in obfuscated files or retrieved from the internet and loaded at runtime.

We investigated further the classes and methods of the Android framework accessed with the native reflection, and we have noticed a significant difference between the apps of the two datasets. In order of frequency, the native reflection is used by malicious apps to: load a DEX or a JAR file through the `DexClassLoader` (or its superclass `ClassLoader`) class, get a handle to a system-level service such as with the `getSystemService` method of the `Context` class, interact with Android managers, inspecting incoming exceptions, and perform crypto and encoding operations. The adoption of such techniques is approximately six times more frequent in malware than goodware, which is less than 4%.

Concerning the analysis of the Android managers, our result shows that malware mainly interacts with `PackageManager` to retrieve app information or verify the permission through the `checkPermission` method, `WifiManager` to check the connection, and `TelephonyManager` to retrieve sensitive information, such as getting the IMEI and IMSI with `getDeviceId` and `getSubscriberId` methods. Collecting unique identifiers for the device (IMEI) and SIM card (IMSI) is a well-known procedure malware uses to profile the victim. However, they moved this logic into the native code over the years.

Moving to the exploitation of the `ClassLoader` from native code, we noticed how recent malware loads the target Java classes by directly invoking Java methods, such as `loadClass`. This technique can be used to replace the `FindClass` JNI callback, making the analysis much more complicated. Besides, we notice that malware in 2021 tends to leverage the Java reflection technique in the native code; for instance, the `ToReflectedMethod` and `FromReflectedMethod` callbacks are used four times more in malware than goodware.

Moreover, it is interesting to highlight how the malicious apps natively retrieve and handle the stack trace, using the `getStackTrace` method of the `Throwable` class to inspect its content. This

technique is applied as a form of anti-hooking: by looking at the content of the stack trace, an app can detect the presence of either the Cydia Substrate or the Xposed framework (e.g., sample `4a7e`), as both manipulate the call stack [10].

Finally, every time we found a native reflection pattern, we applied the technique described by Aafer et al. [2] to check if the method needed: no permission, normal permission, or dangerous permission. If the argument of `FindClass` could not be statically retrieved, we used the argument of `GetMethodID`. In fact, the Android framework method names are often unique, and in case we found multiple matches (like `read` or `open`), we excluded these cases from our analysis. First, more than 10% of malware, regardless of the year but with a gradual increase in such percentage over the years (up to 16% in 2021), invokes methods that require normal/dangerous permission; on the other hand, this percentage is negligible for goodware. Then, on average, comparing recent malware and goodware, we obtained respectively: 84% (vs. 94% for goodware) of the methods required no permissions, 8% (vs. 3%) required normal permissions, and 6% (vs. 3%) required dangerous permissions. This shows how malware abuses native reflection to perform privileged operations and, in particular, malware invokes methods for reading SMS, accessing the location, and reading contacts.

Observations: The inspection of *Native reflection* confirms and brings to light new techniques with which malware executes dynamic code loading exploiting the native layer. While malware increasingly tends to interact with classes that are loaded at runtime, this behavior is rarely exposed by goodware. In addition, to make the analysis and the identification of harmful patterns more challenging, malware tends to move malicious techniques from Java to the native layer (such as anti-hooking or accessing sensitive user information), and they start to replace JNI callbacks with a direct invocation of Java methods.

Library function. An ELF file might rely on external functions exposed by other shared libraries, in which symbols are dynamically resolved or included in the ELF file during the compilation (i.e., statically linked ELF files). Our analysis reveals that almost all the apps in the goodware and malware datasets import `libc.so` (which in Android include also `libpthread.so` and `librt.so`), `libm.so`, and `liblog.so` libraries. The only noteworthy relevant differences are the `libz.so` (a compression/decompression library), used by 90% of malware and 6% of goodware. This discrepancy is due to the fact that malicious apps often decompress components that will be used at runtime (e.g., sample `05b4`).

We identified several discrepancies between goodware and malware in the prevalence of usage of security-relevant functions (Table 2). For instance, in 2011, only 3% of the malicious apps used `chmod`, and 7% used `mprotect`, which are respectively used to change the owner of a file and the permission of a mapped area in the memory. The use of these functions has grown steadily over the years to the point where, nowadays, more than 59% of the samples in our dataset use the first function, and 87% use the second.

Then, we investigated the files that are opened through the `*open` family. In the first case, malware use native code to open or access the files under the `/dev` and `/proc` folders more often than goodware. For instance, in 2021, 54% of malware and 2% of goodware opened the `/proc/version`. Another common target in the `proc` folder is `/proc/self/maps`, which describes the virtual memory in a process, and it is used by 84% of malware and 10% of goodware. Checking the contents of the `maps` file by an application can provide information about injected libraries, and it is a known technique used especially by malware to identify frameworks such as Frida. On the other hand, identifying access to device drivers located under `/dev` is another crucial aspect over the years; numerous vulnerabilities have affected that subsystem (e.g., the recent Use-After-Free

vulnerability in the Android Binder [21]). In fact, the results highlight how recent malware is more prone to open drivers, such as `/dev/tty` to read the output of processes or `/dev/ashmem` to share large quantities of memory among processes, in which vulnerabilities have been found over the years [20].

Furthermore, some malware checks device-related information or opens system shared libraries, while the number of goodware is negligible (< 0.1%). For instance, in 2021, 10% of malicious apps verify if an SELinux policy is enabled accessing the `/sys/fs/selinux/enforce` file, and 9% of malware explicitly interacts with the `/system/bin/linker` to load and run a dynamic executable, even if they are contained in a ZIP file. These are new techniques that have grown in the last years, performed by malware to detect the environment where they are executed or evade anti-malware controls.

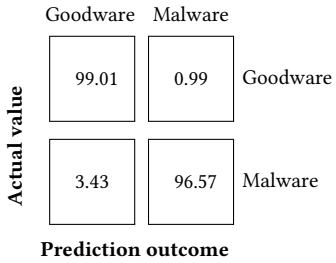
Observations: The analysis reveals that malware is more prone to call security-relevant library functions, indicating operations that require further investigation. Moreover, we highlight the high discrepancy in the usage of `libz` for (de)compression and network-related functions, which are probably used to retrieve resources at runtime. Finally, malware often reads the content of some particular files to perform environment controls (e.g., emulator/sandbox), interacts with low-level components (e.g., linker) to bypass common checks, and runs command line programs more frequently than goodware.

Dynamic loading of DLLs. Dynamic loading refers to the ability to load and invoke functions of other shared objects at runtime without the need to link the library to the executable. In particular, this technique is based on two specific library functions: `dlopen` to load the library and `dlsym` to retrieve a pointer to the target function. The prevalence of this technique had increased over the years, from 2011, when 24% of malware employed it, until today when the percentage is over 90%. In comparison, around 55% of benign apps in our dataset perform dynamic loading.

Looking at libraries and functions invoked with such technique, we found that for goodware, half of the loaded shared objects are well-known Android libraries, while more than three-quarters are for malware. Among the remaining quarter, most of the libraries are not included in the APK – we suppose these libraries are present in obfuscated files or retrieved from the internet and loaded at runtime. The most common libraries that are dynamically loaded are related to (de)compression and decryption/encryption operations; for example, the `uncompress` function of the `libz.so` library is dynamically loaded from more than 90% of malware, but only from 0.4% of goodware apps. This finding again reinforces our idea that malware uses native components to prepare resources that will be used at runtime to evade static analysis.

Even if some Android libraries are widely loaded from both goodware and malware, on average, their usage is very different, and this can be used to pinpoint suspicious operations. For instance, the `libc` library is loaded though `dlopen` by more than 30% of goodware and malware. However, malicious apps rely more on dynamic loading to invoke `libc` functions such as `__system_property_get` to retrieve the value of device-related properties and `chown` to modify the owner of some resource. In addition, we found an unconventional and rather peculiar use of these functions by malicious applications in which few apps called `dlopen` and `dlsym` to obtain a function pointer to `dlopen` and `dlsym` themselves, and use that later on in the execution. Lastly, we observed how at least the 9% recent malicious apps load Dalvik and ART runtime libraries, namely `libdvm.so` and `libart.so`, but this phenomenon occurs in less than 2% of goodware – such libraries can be (ab)used to bypass Android Runtime restrictions [54].

Table 6. Left: confusion matrix; right: classification report



Metric	Goodware	Malware
Precision	96.57	99.02
Recall	99.01	96.57
F1-score	97.77	97.78
Accuracy	99.01	96.57
Accuracy	97.79	

Observations: Most malware abuses the dynamic loading of DLLs, and, very often, the loaded library is "generated" at runtime, while goodware does the opposite. This reinforces the consideration that malware is more prone to prepare resources (e.g., decrypt or download code) from native code in an evasive way. Moreover, most loaded functions could hide suspicious operations, such as (de)compression or permission management.

8 Use case: binary classification

In the previous section, we discussed the many facets of native code execution in Android apps and highlighted core differences between how benign and malicious applications use native code. In this section, we studied to what extent the suspicious tags assigned by our system can be used to detect malware. This analysis allowed us to show a practical use of our methodology to extract suspicious tags and investigate which suspicious patterns are more prevalent in malware w.r.t. goodware as they have a more significant impact on the classification task.

For this task, we built a dataset using all the 15, 647 goodware at our disposal and sub-sampling the same amount of malicious apps collected in 2021. We selected all samples classified as "Singleton" (15, 427) (those for which AVClass2 was unable to determine the family), and we sampled the remaining 220 malicious apps one per family to avoid bias towards particular families. We extracted the suspicious tags defined in our methodology for each sample and created a vector of 74 features (62 booleans and 12 floats). In Table 3, we annotated each tag with the data type used in the vector. We then used a Random Forest classifier due to its ability to handle numeric and categorical features without needing encoding. We set the split criterion of the algorithm to be the Gini impurity and tune the remaining hyperparameters (such as the number of trees, their depth, and the number of features to consider when splitting a node) by using the Out Of Bag (OOB) error computed during the training phase. As reported in Figures 2 and 3 in the Appendix, we obtained the optimal OOB error when considering 141 trees with depth 32 and the *sqrt* as a metric to define the number of features to test when extending a node. We used a 10-fold cross-validation approach to train and test our classifier, each round training the model on k-1 folds and testing its performance on the remaining fold —different for each round— to measure how well the classifier generalizes on unseen samples. Table 6 summarizes the average performance obtained on the test sets, including accuracy, precision, recall, and F1-score. Even though we opted for a simple classification scheme, our results were surprising. In fact, by simply leveraging the suspicious tags to distinguish between goodware and malware, the average error rate was 2.21%, with an accuracy, respectively, of 0.99 and 0.96 and a mean F1-score of 0.97.

Feature importance. We ranked the features used in the classifier based on their Mean Decrease Impurity (MDI) and reported in Table 7 the top-10 of them together with their relative importance normalized as a percentage.

The tags belonging to the categories `J_NATIVE_METHODS` and `J_LOAD_METHODS` (the first two steps of our methodology, which capture native/load methods in the Java code and the entry point from which they can be reached) accounts for almost 50% of the total feature relevance. This shows that the insights gleaned in Section 7.1 and 7.2 represent the most discriminating traits for the classification of goodware and malware; namely, our results suggest that a reliable indicator of malicious behavior is when an app reaches the native code without user interaction and such native code is not statically available.

Furthermore, four of the top-10 entries belong to the category `SUS_LIB_CALL`, which denotes the use of security-relevant functions within the native code. Interestingly, the most impactful are the 'Memory Protection' calls `mmap` and `mprotect` that are a prerequisite to execute dynamically loaded code. Finally, the fifth entry indicates the presence of `build.prop` key strings. The `build.prop` is a file that contains build properties and settings in the format `key=value`. Some contents are specific to the device or manufacturer, while others vary according to the operating system version. Retrieving these values significantly impacts security because attackers can fingerprint the device and engage in evasive behavior or select a valid exploit for the system.

Classification errors. In the last part of our analysis, we investigated the root causes of classification errors and whether those were attributable to any particular characteristics of the samples. In our setting, we define a false positive (FP) as a classification error in which a benign sample is labeled as malware; vice versa, the classifier produces a false negative (FN) when predicting malware as goodware. We repeated our classification task 100 times by using independent folds for each experiment, thus resulting in training and testing 1K different classifiers. The rationale behind this choice is to isolate those samples that are always mispredicted as FPs (0.5% - 81/15, 647) or FNs (3.1% - 483/15, 647). These samples were further investigated by analyzing the tags extracted with our methodology and by resorting to manual reverse engineering for a subset of them.

When narrowing down to FNs, we detected that the model errs when the malicious logic (e.g., sample `c227`) is fully contained in `classes.dex` files – which is out of our scope, and it includes well-known legitimate native libraries. For example, sample `58b3` only ships two open-source libraries in the standard location, namely LAME to manipulate MP3 files, and a second one that provides WebRTC capabilities. However, many samples contain the definition of some native methods in the Java code, but do not contain the relative ELF file, neither in standard nor in non-standard locations. The respective sample is then characterized by the sole presence of the `NO_ELF_NAME` tag (`J_LOAD_METHODS` category), whereas almost all the other tags are missing. In such a case, the model does not have enough information for an accurate classification.

On the other hand, we discovered that misclassifications of goodware (FPs) are mainly due to the heavy usage of Dynamic Code loading (`DYNAMIC_LOADING` category) or suspicious library calls (Table 2). In particular, almost all FPs are samples with native libraries that try to protect the intellectual properties of the developers with integrity checks, obfuscation, and packing techniques. By nature, such techniques generate exactly surreptitious code that represents the core of our analysis.

Table 7. Top 10 features sorted by MDI score

Tag Category	Tag Title	MDI score (%)
J_NATIVE_METHODS	NO_REACHABLE	15.88 %
J_LOAD_METHODS	APP_LIFECYCLE_EP	13.17 %
J_NATIVE_METHODS	APP_LIFECYCLE_EP	11.50 %
J_LOAD_METHODS	PATH_LOAD_METHOD	8.87 %
STRING	PROPERTIES	7.15 %
J_NATIVE_METHODS	ACTIVITY_LIFECYCLE_EP	5.90 %
SUSP_LIB_CALL	MEMORY_PROTECTION	4.02 %
SUSP_LIB_CALL	PROCESS_MANAGEMENT	3.65 %
SUSP_LIB_CALL	IDENTITY	3.45 %
SUSP_LIB_CALL	PERMISSION	2.83 %
	Average	1.35 %
	Standard deviation	0.24 %

Takeaways: Using only the suspicious tags extracted from native libraries for binary classification task yielded promising results: the classifier can distinguish between the two classes with an average error of 0.02 and achieves an F1 score of 0.97. This fact highlights how and which native code suspicious patterns correlate more to malware vs. goodware and whose presence may indicate a *potential* malicious pattern. In particular, the features extracted from Step #1 and Step #2 of our methodology account for almost 50% of the total relevance.

9 Limitations

We performed the first longitudinal analysis of native components in Android malware, which allowed us to identify several suspicious uses related to the JNI code. Moreover, we showed how our automatically assigned suspicious tags could pinpoint the code region to inspect and speed up the analysis process, examining the behavior for all supported architectures. Our work significantly differs and complements all state-of-the-art tools for both the type of analysis performed and the goal of the analysis. In particular, *ANDani* is not limited only to a Java to native dataflow analysis: it also analyzes all aspects of the native components, from the entry point analysis and the triggering condition of JNI methods to the suspicious library function invoked by the native code. With this design, we can improve the automatic detection of Android malware in a binary classification task by including the native-related features extracted by *ANDani*. However, our implementation has all the intrinsic limits of the static analysis, in particular, the backward taint data analysis [37] and the fact that we are not able to analyze the code whether it is dynamically loaded (i.e., not statically present in the APK). Given the current spread of droppers in the PlayStore [55], this is an issue that needs to be addressed; in particular, a possible future work concerns integrating *ANDani* with a dynamic analysis pipeline to analyze DEX and ELF files that are dynamically recovered.

Furthermore, in our experiment in which we submitted synthetic malware to VirusTotal, we demonstrated that some malicious apps are not detected – and are unlikely to be detected in the future – while for others, it is simply a matter of waiting for the AV vendors to update their databases. This represents another intrinsic limitation faced by researchers in this field. The ability to identify sophisticated malware is often contingent upon the willingness of companies to share the results of their meticulous manual analysis, which is conducted using a multitude of tools and techniques. It can be argued that this is a recognized limitation of all studies that rely on ground truth from publicly available tools. In order to mitigate this issue, it should be noted that our work does not claim absolute reliability of the dataset. Rather, it employs the dataset as a representative sample with which to evaluate broader trends. This enables the identification of significant deficiencies in existing detection mechanisms and the illustration of methods by which native payloads can evade detection. The acknowledgment of the limitations of ground truth sources does not invalidate the

broader insights provided by our methodology into the weaknesses of current detection strategies. In conclusion, the findings presented in this paper are not intended to be generalized to all malware but rather to provide evidence of a growing reliance on native code.

It should be noted that although our study focuses on native code, the presence of a malicious component in the DEX code rather than in native libraries did not affect our measurements. In the event that a malicious component is present in DEX, our objective was to measure any suspicious native patterns that might be employed to disguise further malicious logic or evasive code. However, a limitation of our measurements is that we were unable to ascertain whether and what (both good and bad) developers are concealing through the use of such suspicious patterns. Nevertheless, our work provides a foundation for further intriguing avenues of research.

10 Conclusions

We consider this study to be of considerable relevance in the present and future. In 2016, Afonso et al. found that 40% (446,562/1,208,476) of goodware used JNI, while in our study (seven years later), this percentage increased to 57%. Moreover, modern chipsets often include a Neural Processing Unit to accelerate artificial intelligence computations; thus, Android started offering Neural Networks APIs (NNAPI) for interacting with it [27]. However, due to performance reasons, developers can only interact with NNAPIs through native code, making JNI an increasingly crucial component of the Android operating system – as is also evidenced by the recent introduction of Rust [29] as a safer alternative to C/C++.

On the other hand, malware authors are always up to speed and saw in JNI an opportunity to hide their intentions better. In fact, the performance of our ML classifier revealed a statistically significant distinction in the utilization of suspicious native code between malware and goodware within our dataset.

In conclusion, with this paper, we have tried to contribute to this cat-and-mouse game, hoping that our methodology and suspicious tags will be the building block for future research on Android malware and its detection.

Acknowledgements

Our heartfelt appreciation to the illustrious Slasti Mormanti for is sage guidance, boundless wisdom, and unwavering commitment during the development of this paper.

This work benefited from two government grants managed by the French National Research Agency with references: “ANR-22-PECY-0007” and “ANR-23-IAS4-0001”.

References

[1] 2022. Jiagu. http://jiagu.360.cn/. Accessed January 6, 2025.

[2] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. 2018. Precise Android API protection mapping derivation and reasoning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1151–1164.

[3] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. 2016. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *The Network and Distributed System Security Symposium*. 1–15.

[4] NSA National Security Agency. 2022. Ghidra: A software reverse engineering (SRE). https://ghidra-sre.org/. Accessed January 6, 2025.

[5] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.

[6] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. 2017. DroidNative: Automating and optimizing detection of Android native code malware variants. *computers & security* 65 (2017), 230–246.

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

[8] Seyed Behnam Andarzian and Behrouz Tork Ladani. 2020. Compositional Taint Analysis of Native Codes for Security Vetting of Android Applications. In *2020 10th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, 567–572.

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

[10] Neil Bergman. 2015. Android Anti-Hooking Techniques in Java. https://d3adend.org/blog/posts/android-anti-hooking-techniques-in-java/. Accessed January 6, 2025.

[11] The MITRE Corporation. 2011. CVE-2011-1823. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1823. Accessed January 6, 2025.

[12] The MITRE Corporation. 2014. CVE-2014-3153. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3153. Accessed January 6, 2025.

[13] The MITRE Corporation. 2016. CVE-2016-5195. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-5195. Accessed January 6, 2025.

[14] The MITRE Corporation. 2019. CVE-2019-2215. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-2215. Accessed January 6, 2025.

[15] facundoolano. 2022. Google Play Scraper. https://github.com/facundoolano/google-play-scraper. Accessed January 6, 2025.

[16] Flutter. 2022. Flutter. https://flutter.dev/. Accessed January 6, 2025.

[17] The Apache Software Foundation. 2022. Apache Cordova Framework. https://cordova.apache.org/. Accessed January 6, 2025.

[18] George Fourtounis, Leonidas Triantafyllou, and Yannis Smaragdakis. 2020. Identifying java calls in native code via binary scanning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 388–400.

[19] Gabriel Claudiu Georgiu. 2022. Playstore Dowloader. https://github.com/ClaudiuGeorgiu/PlaystoreDownloader. Accessed January 6, 2025.

[20] Google. 2017. BitUnmap: Attacking Android Ashmem. https://googleprojectzero.blogspot.com/2016/12/bitunmap-attacking-android-ashmem.html. Accessed January 6, 2025.

[21] Google. 2020. Android Use-After-Free in Binder. https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2019/CVE-2019-2215.html. Accessed January 6, 2025.

[22] Google. 2022. The Activity Lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle. Accessed January 6, 2025.

[23] Google. 2022. Android ABIs. https://developer.android.com/ndk/guides/abis. Accessed January 6, 2025.

[24] Google. 2022. Android App Bundle. https://developer.android.com/guide/app-bundle. Accessed January 6, 2025.

[25] Google. 2022. Android App Categories. https://support.google.com/googleplay/android-developer/answer/9859673. Accessed January 6, 2025.

[26] Google. 2022. Android linker source code, call_constructors method. https://android.googlesource.com/platform/bionic/+/master/linker/linker_soinfo.cpp#516. Accessed January 6, 2025.

[27] Google. 2023. Neural Networks API. https://developer.android.com/ndk/guides/neuralnetworks. Accessed January 6, 2025.

[28] Google. 2023. Permissions on Android. https://developer.android.com/guide/topics/permissions/overview. Accessed January 6, 2025.

[29] Google. 2024. Android Rust introduction. https://source.android.com/docs/setup/build/rust/building-rust-modules/overview?hl=en. Accessed January 6, 2025.

[30] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information flow analysis of android applications in droidsafe.. In *NDSS*, Vol. 15. 110.

[31] Sable Research Group. 2022. Soot - A Java optimization framework. https://github.com/soot-oss/soot. Accessed January 6, 2025.

[32] j0nk0. 2019. Android DirtyCow. https://github.com/j0nk0/GetRoot-Android-DirtyCow. Accessed January 6, 2025.

[33] kangtastic. 2019. CVE-2019-2215 Exploit. https://github.com/kangtastic/cve-2019-2215. Accessed January 6, 2025.

[34] Michael Kerrisk. 2021. proc.5. https://man7.org/linux/man-pages/man5/proc.5.html. Accessed January 6, 2025.

[35] Sungho Lee. 2019. JNI program analysis with automatically extracted C semantic summary. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 448–451.

[36] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.

[37] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67–95.

[38] Microsoft. 2022. Xamarin. https://dotnet.microsoft.com/apps/xamarin. Accessed January 6, 2025.

[39] Jasvir Nagra and Christian Collberg. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection.*

[40] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*. 543–558.

[41] OffSec. 2009. The Exploit Database. https://www.exploit-db.com/. Accessed January 6, 2025.

[42] Oracle. 2022. JNI Functions. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html. Accessed online: January 6, 2025.

[43] Oracle. 2022. JNI Types and Data Structures. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html. Accessed January 6, 2025.

[44] Oracle. 2022. Oracle JNI. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html. Accessed January 6, 2025.

[45] Mono Project. 2022. Mono Project. https://www.mono-project.com/. Accessed January 6, 2025.

[46] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin TS Chan. 2014. On tracking information flows through jni in android applications. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 180–191.

[47] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques.. In *NDSS*.

[48] Antonio Ruggia, Andrea Possemato, Alessio Merlo, Dario Nisi, and Simone Aonzo. 2023. Android, Notify Me When It Is Time To Go Phishing. In *EUROS&P 2023, 8th IEEE European Symposium on Security and Privacy*.

[49] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. 2022. JuCify: A Step Towards Android Code Unification for Enhanced Static Analysis. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 1232–1244.

[50] Silvia Sebastián and Juan Caballero. 2020. Avclass2: Massive malware tag extraction from av labels. In *Annual Computer Security Applications Conference*. 42–53.

[51] Mingshen Sun, Tao Wei, and John CS Lui. 2016. 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*. 331–342.

[52] Hacking Team. 2015. HackingTeam Exploits. https://github.com/f47h3r/hackingteam_exploits/tree/master/android. Accessed January 6, 2025.

[53] Unity Technologies. 2022. Unity. https://unity.com/solutions/mobile/android-game-development. Accessed January 6, 2025.

[54] Romain Thomas. 2019. Android Runtime Restriction Bypass. https://blog.quarkslab.com/android-runtime-restrictions-bypass.html. Accessed January 6, 2025.

[55] ThreatFabric. 2021. 300.000+ infections via Droppers on Google Play Store. https://threatfabric.com/blogs/deceive-the-heavens-to-cross-the-sea.html. Accessed January 6, 2025.

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

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

[58] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep ground truth analysis of current android malware. In *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, 252–276.

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

[60] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2014. 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*. 1329–1341.

[61] Daoyuan Wu, Debin Gao, Robert H Deng, and Chang Rocky KC. 2021. When Program Analysis Meets Bytecode Search: Targeted and Efficient Inter-procedural Analysis of Modern Android Apps in BackDroid. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 543–554.

[62] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 2017. Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for {ART}. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 289–306.

[63] Lok Kwong Yan and Heng Yin. 2012. Droidscope: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis. In *21st {USENIX} Security Symposium ({USENIX} Security 12)*. 569–584.

[64] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. 2015. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 303–313.

[65] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. 2020. Measuring and modeling the label dynamics of online {Anti-Malware} engines. In *29th USENIX Security Symposium (USENIX Security 20)*. 2361–2378.

Appendix

10.1 Hash of the samples

Table 8. Sha256 of the samples mentioned in the paper

019e12c7233d7324667d9e49aba4787c67204c5c8f2c38754f469a5b600bddde
0259d084a78ddc98e663ae5799898a0afb4d021c6486cb61bdf7285731476d61
04cef547b64e459936dd243bfb19575bc905f6271c94723788000088f9e7e278
05b4c4dd8bf9f376c767330e649d725ad35c0c9c3b1b2dbbfab7f39e90c5bac4
13068932cd52ffa257fa35bba7860e618416f0d53eecd7650a7700607220d4c0
1f267514222943779bdd642b9c7322a31a87d8f17790be4f31d59c2f4fade4d3
213c997dc02dfc4e83e872243c9217c7481a18a386b4fd79c049a5e27dad97f0
32cd907d3343c44180294a7c279c2a5f139a6ee443cbf443eb2bd663bca37c6e
4a7e913d491f715bb00b37ad5b8802a00c919070486212e8d1d1a802f4bdf6bf
4e4be579cffdd690cef4bb0d779d66ede95cfd955eb27eb797e0704f59d61e6d
58b34234bd375ac81753cb8cc793a60cf9f0a220383bf332d15ce51917488623
5bd3e6f49aaab9e7fe566d92cceb9a5701a072426434de5bb2cdbc34a7d265f2
6c6eeed1b91913db0d6232edb1979c67d6fb48ca3da4f83dc49fb565a4e5f4fe
73837b030f031d532741b7e84068aabed24e7a6ac118c4272005e6ecd18a17d7
79009a3bbdb9f73faa3d8b3a35306957fbd2bfb362d0c2d658079ff6a49b69e0
858ebffaa54e40cc4787280da60e5854e8776359340bdf5287e32a580878a2c0
98f6d51dff0cfde3ca2aab65ec4edc0f9054900dacc0817f39ff018625c83fee
b3c7aa8b70edd76d9463c458a74a8c61266b90d946e4d8b329a14b3e597142ce
c227edef2d823059f261b2101a21c4deeda2ee016671ce06b28dde0297018550
cdde49edda06e3856755e5b847892ee91fb3ac334595328b1a742d9b898992a7
d86731c8fa5ed48f13fadbf761a0869697dd56bbf963028e57d35395cf217f74
e0881b869add4b86628abb53255990aabb5db2548b259ecb04d03834dcf54d38
f7b906ec2ce1c39979092dbd220d0b9bf7fb770122c4de31e239935aa4763fea

Due to space limitations and readability, we never used the whole sha256, but only the first 4 bytes. In Table 8, we report each complete hash alphabetically, with the first 4 bytes in bold. Each entry is a hyperlink to the corresponding VirusTotal webpage.

10.2 Random Forest hyperparameters tuning

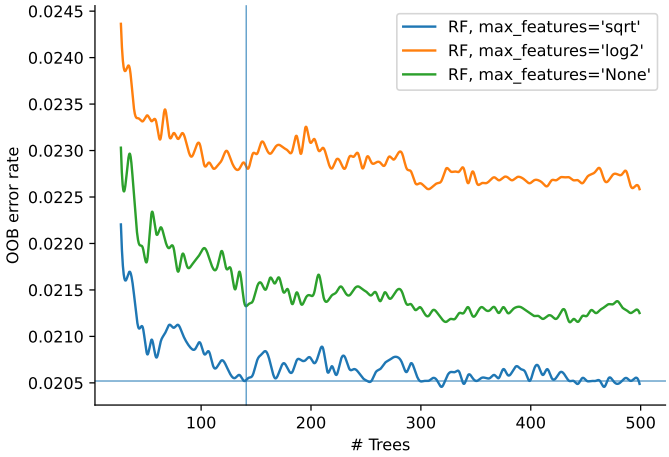


Fig. 2. OOB error to determine the optimal number of trees and the number of features to test when splitting a node. The optimal value for the number of trees has been chosen using the Elbow method, after calculating that there is no performance improvement above 259 trees.

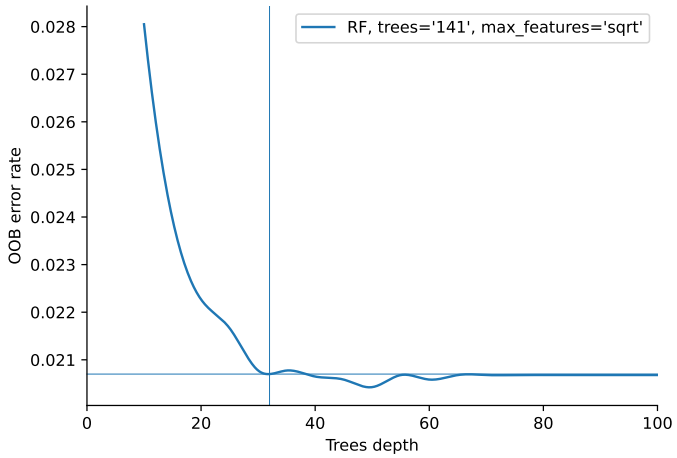


Fig. 3. OOB error to determine the optimal depth of trees. The optimal value for the depth of the trees has been chosen using the Elbow method and of trees is chosen using the Elbow method after calculating that there is no performance improvement above a depth of 34.

10.3 Example of Thread handling

Listing 3. Example of how start a new thread in Java.

```
1  class MyThreadA extends Thread {
2    void run() {
3      System.out.println("My Thread A - called");
4    }
5  }
6
7  class MyThreadB extends Thread {
8    void run() {
9      System.out.println("My Thread B - never called");
10   }
11 }
12
13 class Main {
14   static void start_thread(Thread t) {
15     t.start();
16   }
17
18   static void main(String[] argv) {
19     start_thread(new MyThreadA());
20   }
21 }
```

Listing 3 shows a simple Java snippet where the `run()` method of a Thread subclass is executed through the `Thread.start()` call. The code contains two Thread classes, respectively `MyThreadA` and `MyThreadB`, but only the first one is used. The *Bytecode module* has been designed to consider the context of the call and to propagate the arguments. In this example, the module is able to create an edge from `Main.start_thread` to `MyThreadA.run()`.