Doctoral Dissertation

CROSS-ENVIRONMENT DYNAMIC SYMBOLIC EXECUTION ON ANDROID/APK FILES AND ITS APPLICATION FOR MALWARE ANALYSIS

NGUYEN THI VAN ANH

Supervisor : MIZUHITO OGAWA

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
Information Science
September 2024

Abstract

Modern applications often run across multiple environments. A high-level language can invoke native extensions, typically written in C/C++ code, resulting in more efficient applications and increased productivity since legacy code can be reused. However, the use of native code introduces safety concerns that can lead to security breaches, potentially violating security protocols. In this work, we introduce a novel tool, HybridSE, to analyze Android applications with native code.

HybridSE distinguishes itself by integrating the strengths of established Dynamic Symbolic Execution (DSE) tools—SPF (Symbolic Pathfinder) and CORANA/API, which were originally designed for Java and ARM architectures, respectively. Enhanced with a specialized taint analysis module, HybridSE effectively addresses data leaks in real-world applications and malware, demonstrating a notably low false positive rate in our evaluations.

We assess the performance of HybridSE in two key aspects: control flow and data flow analysis. Regarding control flow, we utilize the generated graphs and apply graph similarity to two tasks: malware family classification and Android packer classification. The graphs generated by HybridSE, when used as features in classification tasks, yield results comparable to those of state-of-the-art classifiers.

In terms of data tracking and detecting data leakage, HybridSE demonstrates higher precision compared to other tools, effectively reducing false positives caused by over-approximation. Unlike static taint analysis tools, HybridSE avoids issues related to array handling and Java reflection. By generating accurate cross-environment control flow graphs for both Java bytecode (.dex) and native code (.so), our taint analysis method has successfully detected 139 data leaks in real-world Android malware. Through our analysis with HybridSE, we have made several observations on how data leaks occur, including a detailed examination of the Lotoor family, which remained active until 2022.

Keywords: Android mobile security, Symbolic execution, Taint analysis, Packer identification, Malware classification

Acknowledgments

First and foremost, I wish to express my deepest gratitude to my main supervisor, Professor Mizuhito Ogawa. Without his vision, this dissertation would not have been possible. I have learned the majority of my background and research methods from him. His deep expertise and constructive criticism have been invaluable in helping me enhance my work and guiding me toward the right approach to research. I also deeply appreciate his genuine care for the personal well-being of both myself and other students.

I would like to extend my sincere gratitude to my advisor, Professor Jean-Yves Marion at LORIA, University of Lorraine, for his valuable direction and kind assistance. He supported me during my internship and introduced new approaches and ideas to guide my research. I learned a great deal from him about writing research papers and constructing research studies.

I also want to thank my second supervisor at JAIST, Associate Professor Nao Hirokawa, for always giving detailed comments and valuable feedback, particularly on how to write concise logical formulas and systematically explain my research. My thanks also go to the committee members - Professor Nguyen Le Minh (JAIST), Associate Professor Razvan Beuran (JAIST), and Professor Katsuna Yoshioka (Yokohama National University YNU) - for their insightful comments that improved my dissertation. I am also thankful to the professors at JAIST for accepting my application and granting me the opportunity to learn and pursue my PhD.

My thanks extend to the Ogawa Laboratory team - Ms. Kuniki Hase, Professor Keita Yokoyama, my labmate Nguyen Thi Anh Thu, and other students - for their warm welcome, kindness, and assistance in both research and personal matters. I have thoroughly enjoyed our discussions at the lab. I am also grateful to my university professor at UET, To Van Khanh, for his recommendation to JAIST and his ongoing support. Additionally, I appreciate the CARBONE team at LORIA, including Jean-Yves Marion and Guillaume Bonfante, for their discussions that provided new ideas and perspectives for my research. I also want to acknowledge Vu Viet Anh, the original implementer of CORANA, whose work helped build the foundation for my thesis.

I am grateful to my Japanese teachers at NIEA for their warm welcome and their Japanese instruction, which greatly enriched my life in Japan. I also want to thank my friends, who have made my life at JAIST fruitful.

The MEXT scholarship has been crucial in supporting my life financially and allowing me to focus on my research. Additionally, JAIST's support made it possible for me to complete my internship at LORIA in France.

Finally, I would like to thank my parents, sisters, and other family members for their kind encouragement and continuous support. I am profoundly grateful to my husband, Dang Tran Binh, for being there for me every step of the way.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem statement

Android OS is now extensively used not only on smartphones but also on tablets and various other smart devices. These devices function as portable computing platforms, granting access to sensitive system resources such as device numbers and personal data, including emails and contact lists. As a result, data compromise is a significant concern in Android security [2]. A substantial amount of research has focused on detecting Android data leaks and malicious behavior using both dynamic [3] and static [4, 5] analysis techniques.

While Java, a high-level language, can leverage native code, typically written in C/C++, to improve application efficiency, this practice has also raised considerable security concerns in the context of Android security [6]. Native code can introduce safety and security issues that may be missed by higher-level language analyses [7]. We examine the unique characteristics of the Android environment and explain why native code poses a significant security challenge.

1.1.1 Cross-environment nature of Android/APK files

Google supplies *Android Native Development Kit* (Android NDK) from 2009, which allows developers to write C/C++ code for Android and cross-compile it to multiple architectures, e.g., ARM, x86, and MIPS. Malware developers began to obfuscate bytecodes compiling into native code to bypass bytecode level analyses.

Although Android APKs are developed in Java, there are several differences. An Android application (Android APK) starts with a Java variant called Dalvik bytecode (`classes.dex`) and may include native code, such as Android library functions and user-defined functions in `.so` shared libraries. This feature is convenient for reusing legacy code, boosting performance, and accessing devices directly.

Note that an Android application allows multiple entry points activated by facilities such as `Activities` and `Services` due to user actions, messages, or system events.

While Dalvik bytecode can be easily decompiled into Java bytecode, native library code requires reverse engineering of binary code (for each architecture: ARM, x86, etc.). Another difficulty lies in the calling conventions between bytecode and native code. It is important to account for the differences between Java conventions and ARM/C/C++ conventions.

1.1.2 Security risks posed by native code

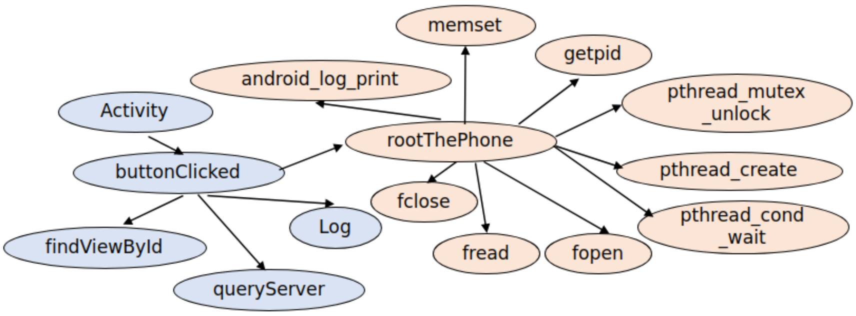


Figure 1.1: Simplified Towelroot callgraph with blue node in bytecode and red node in native code.

Towelroot is a textbook illustration of how to "root" an Android device. This example will enable us to explain how a cross-environment application works and to see how information can escape. The majority of static analysis tools for Android tend to neglect native code. Many spyware have taken advantage of the lack of native code analysis to bypass data protection. Callgraph (Fig. 1.1) of *Towelroot* shows no suspicious calls in the Java component, but numerous external calls to the Linux kernel from the native code (red node).

Towelroot gains root access by exploiting a vulnerability in an old Linux kernel. The exploit leverages a vulnerability in the *Fast Userspace Mutex (Futex)*, accessed through the *pthread* library.

That said, users trying to take control of their smartphones with *Towelroot* run the risk of confidential data being leaked. Let's take a closer look at *Towelroot*'s code. Listing 1.1 shows a key snippet.

```
 1 %%% JAVA CODE in towelroot.apk
 2 public class MainActivity extends Activity {
 3 static{ System.loadLibrary("libexploit.so");}
 4  public native String rootThePhone();
 5  public void buttonClicked(View view) {
 6    TextView tv=(TextView) findViewById(R.id.textoverwrite);
 7    if (queryServer(false)) {
 8      tv.setText(rootThePhone()); // CALL to native
 9      queryServer(true);
10      this.didrun = true;}
11     }}
12
13 %%% NATIVE ARM CODE of rootThePhone()
14 jstring Java_libexploit_rootThePhone() {
15     0x10d44 add r5,r0,r7
16     0x10d48 bl getpid   //SOURCE: getpId() Taint r0
17     0x10d4c cpy r3,r0                   Taint r3
18     0x10d50 mov r0,#0x4    = 4          Clear r0
19     0x10d54 cpy r1, r4     = "towelroot"
20     0x10d58 cpy r2, r5     = "native running with"
21     0x10d5c bl __android_log_print params: r0, r1, r2, r3
22                  //SINK: __android_log_print
23     0x119b8 ldr r0,[sp,#114c]
24     0x119c0 bl pthread_create
25     0x119c4 ldr r1,[sp,#114c]
26     0x119ec bl pthread_mutex_lock
27 }
```

Listing 1.1: The cross-environment application towelroot.apk with an ARM native code

Towelroot (Line 8) calls `rootThePhone()` with `id` (Line 6) as an argument. The native function `rootThePhone()` is declared using the `native` keyword (Listing 1.1, Line 4, and is registered statically by `System.loadLibrary()`). The native method in the bytecode and the native code are tied by the naming convention of JNI (Java Native Interface). The JNI establishes a one-to-one mapping between the name of a native method declared in Java and the name of its counterpart residing in a native library. In this case, `rootThePhone()` is mapped to `Java_libexploit_rootThePhone`.

`rootThePhone()` accesses the physical components and gains root access. But, we can also observe a potential data leak of the process `id`, which is possibly not harmful. The native function `__android_log_print` is used to send the information obtained from the `getId()` call in the native part.

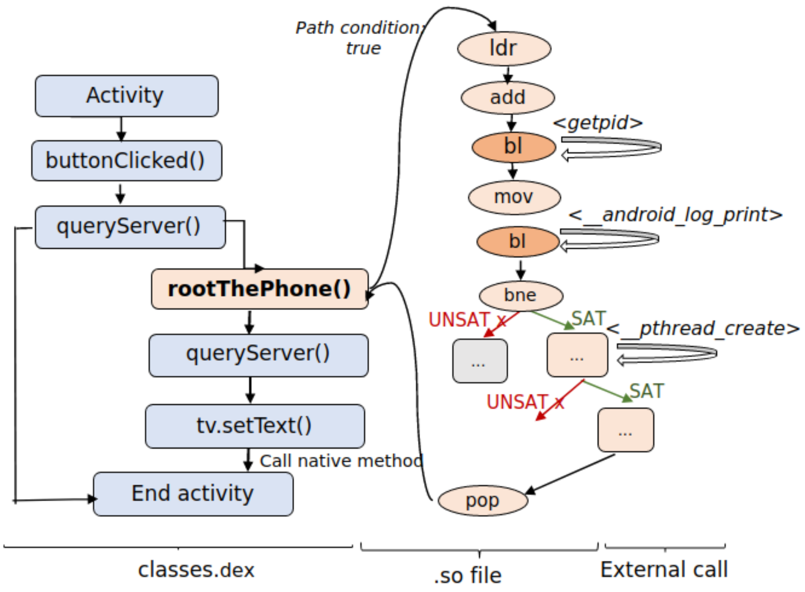


Figure 1.2: Control flow graph for *Towelroot* in Listing1.1

1.1.3 Why DSE is needed?

We emphasized the necessity of cross-environment Dynamic Symbolic Execution for Android applications.

The need to analyze native code has led to a surge in the development of taint analysis tools for Android Native code, encompassing both dynamic and approaches, such as ARGUS-SAF [8], JuCify [9], TaintArt [10] and OATs'inside [11]. Dynamic taint analysis tools such as TaintArt [10] and ViaLin [12] can detect some malicious behaviors. However, they are unable to analyze behavior that is not activated at runtime, such as trigger-based behaviors and VM-aware actions, potentially causing them to miss hidden triggers within the code.

On the other hand, static analysis tools such as Argus-SAF [13] and JuCify [9] adopt an over-approximation to cover all execution paths. A significant advantage of static analysis lies in its ability to be automated and scaled quite well. However, it is prone to produce false positives, struggles against code obfuscation, and runtime-related components like Java reflection and dynamic class loading [14, 15].

To balance both techniques, we proposed a cross-environment Dynamic Symbolic Execution framework (DSE) for Android Native code. On Windows, symbolic execution on binary code has attracted attention for deobfuscation to obtain precise control/data flow. Additionally, it has proven effective in identifying vulnerabilities, such as integer overflow. However, within the Android domain, existing DSE tools [16, 17, 13] focus solely on Dalvik bytecode. They either ignore or treat native code as a black box, leading to a shortfall in assessing the behaviors embedded within native code.

Constructing a DSE tool for Android presents a challenging task due to the Android framework's heterogeneous nature. An Android APK file includes various components, such as Dalvik bytecode (.dex), native code (.so), and the AndroidManifest.xml, which specifies data access permissions. Hence, the execution of an Android application frequently traverses diverse environments, moving back and forth.

Symbolic execution tools for Android APK files are mostly Java-based and handle native code as black boxes, i.e., they simply get results of native code by concrete execution, instead of symbolic execution.

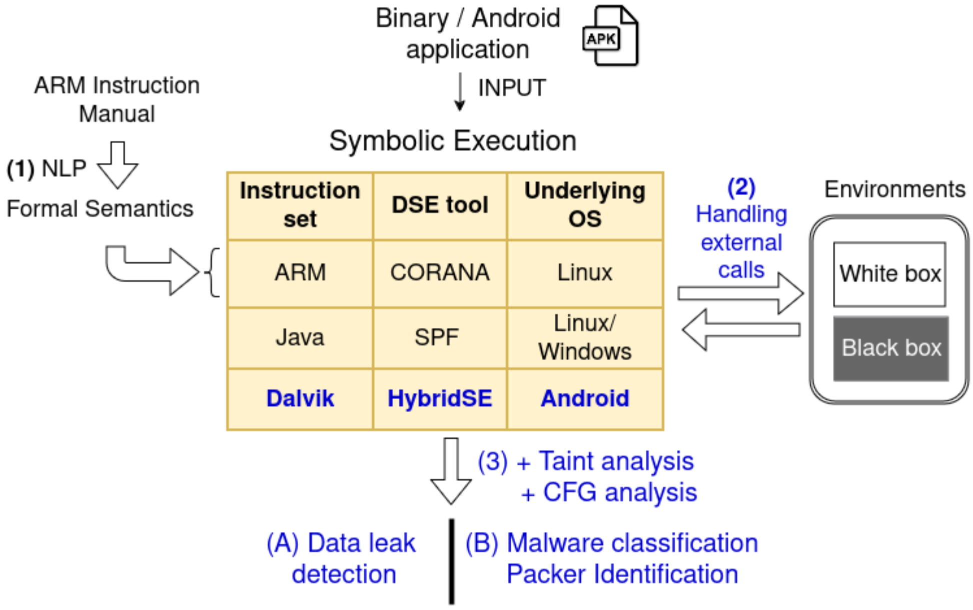
Some tools, such as ANGR [18], perform symbolic execution across both .dex (bytecode) and .so (native code) files. However for this, ANGR translates both file types into a single entity using the Jimple intermediate language. Consequently, its effectiveness is constrained by the translation from native code (such as ARM) to Jimple. This transition is challenging to extend, as Jimple maintains a class hierarchy similar to Java, which can complicate direct device access.

Our primary goal is to develop a DSE tool capable of analyzing cross-language Android applications while being resilient against obfuscation techniques. To this end, we introduce HybridSE, which integrates two established DSE tools, SPF [17] and CORANA/API [19]. Furthermore, we have enhanced its functionality by adding a taint analysis module.

1.1.4 Proposing a DSE tool - HybridSE

When targeting malware, there are *PC malware*, *IoT malware*, and *mobile malware*. *PC malware* mostly focuses on x86 with typical OSs, e.g., Windows, and Linux. It often uses heavy obfuscations to bypass anti-virus software, which is typically introduced by a *packer*. *IoT malware* is often naive because of the absence of anti-virus protection in IoT devices. However, the target platforms of IoT malware vary a lot whereas the target OS is often Linux-based. For *mobile malware*, Android, with its more open ecosystem, tends to have a higher prevalence of malware compared to iOS. Mobile malware on Android often leverages multiple components of the operating system to exploit permissions, access the file system, intercept SMS messages, use root exploits for elevated privileges, and manipulate app components like activities, services, and broadcast receivers.

In this work, we propose a DSE tool for Android applications with native code, called HybridSE, as shown in Figure 1.3.


Binary / Android application
APK
INPUT
ARM Instruction Manual
(1) NLP
Formal Semantics
Symbolic Execution
(2) Handling external calls
Environments
White box
Black box
(3) + Taint analysis + CFG analysis
(A) Data leak detection
(B) Malware classification Packer Identification

Instruction set	DSE tool	Underlying OS
ARM	CORANA	Linux
Java	SPF	Linux/ Windows
Dalvik	HybridSE	Android

Figure 1.3: Overview of the DSE Systems described in this dissertation, with the contributions of this work indicated in Blue

The process of implementing and utilizing a DSE tool involves three stages:

1. *Instructional semantics definition*: This defines the execution and interpretation of instructions within the symbolic execution engine.

   Symbolic PathFinder (SPF) [17], which was developed by NASA, integrates formal semantics of Java bytecode within the JPF VM framework, enabling SPF to conduct DSE on Java programs. CORANA [20] has successfully automated the extraction of formal semantics from specifications for ARM.

   Execution on Android involves navigating Dalvik bytecode instructions and binary code within native libraries. We utilize the well-defined semantics from existing tools, SPF for Java and CORANA for ARM binary code, although some modification is needed to adapt SPF for Android (Figure 1.3 - 1).

2. *Handling external calls*: This manages interactions with external libraries, system calls, or other software components.s Based on the

visibility of an application's components to the user process, we can categorize them as either black-box or white-box. HybridSE adds the handling of black-box components, including OS syscalls in native code, and white-box components, including calls to native libraries.

3. *Analysis implementation*: This involves developing analysis modules within the DSE tool and applying them to malware analysis tasks. We implemented two analysis modules in HybridSE: taint analysis and CFG generation. Accordingly, we performed two tasks on a set of current Android malware: detecting data leaks and classifying malware families and used packers using CFG similarity generated by HybridSE.

1.2 Contribution

This thesis's contributions are:

- This thesis presents a novel symbolic execution framework tailored for cross-environment platforms, encapsulated within a DSE tool named HybridSE.

- HybridSE[1] is a pioneering tool for DSE, designed for analyzing cross-language Android applications. Notably, HybridSE leverages its ability to perform DSE across both Java code and native code.

  HybridSE combines 2 DSEs, SPF, and corana, by integrating both bytecode-level semantics and low-level semantics, taking into consideration interactions with system calls. The binding layer between bytecodes and native extensions poses another significant challenge. To address this, we've defined a conversion process for the calling convention between Java and native ARM. This process automatically extracts the interface during native calls. This interface is essential for cross-platform calls, detailing arguments passing, data type conversion, and memory allocation. Extracting type information is crucial for constructing these interfaces. HybridSE automates this process by extracting data type information from the Linux Manual Page [19].

- We assess HybridSE by constructing cross-environment control flow graphs of more than 10,000 applications that contain native code invocations. We also extensively discuss the limitations of HybridSE, stemming from the analysis of the datasets considered.

[1] figshare.com/s/45b91d138c44e2e55ddd

- We assess the performance of HybridSE in two key aspects: control flow and data flow analysis.
  - Regarding control flow, we evaluate the efficacy of the generated CFG by comparing it with the call graph produced by the static analysis tool Flowdroid.
  - Subsequently, we utilize the generated graphs and apply graph similarity to two tasks: malware family classification and Android packer classification. In both tasks, the graphs generated by HybridSE, when used as features in classification tasks, yield results comparable to those of state-of-the-art classifiers.
  - In terms of data tracking and detecting data leakage, we showcase that HybridSE exhibits greater precision compared to other tools, thereby minimizing false positive alarms resulting from overapproximation. Unlike static taint analysis tools, HybridSE does not suffer from weaknesses related to array handling and Java reflection. Consequently, HybridSE yields accurate results on tasks involving these aspects.
  - By generating precise cross-environment control flow graphs for both Java bytecode (.dex) and native code (.so), our taint analysis method identifies data leaks through experiments conducted on real-world Android malware.
  - HybridSE successfully detected 139 malware data leaks. From our analysis with HybridSE, we have drawn several observations about how data leakages occur. In particular, we carefully examined the *Lotoor* family, which was active until 2022.

1.3 Outline

This thesis is structured as follows.

Chapter 1 explains the motivation behind our work on Dynamic Symbolic Execution (DSE) for cross-environment Android applications. After presenting the features of the Android framework, we show the challenges that Android analyses are facing and our motivation for building a precise and complete analysis of native Android applications.

Chapter 2 outlines the structure of an Android/APK file and obfuscation techniques that frequently occur in Android. While native code provides developers an incredibly effective tools, it also introduces serious security issues to the Android framework.

Chapter 3 explains dynamic symbolic execution (DSE) across heterogeneous environments, targeting a combination of SPF and CORANA/API.

In this chapter, we first discuss symbolic execution and the choices when handling heterogeneous platforms. Then, we present the components of a multi-language environment DSE, named HybridSE, for an Android application running on an ARM-based device.

We divide platforms into two types, the black box, and the white box, depending on their visibility. A black-box platform prohibits tracking the data and control flows. In contrast, distinct components written in multiple programming languages are white-box platforms. Different from existing DSE tools, we combine platform-specific DSE tools for each white-box component (native code) to keep execution across different platforms. For the black box component (system call), we concretize symbolic values in the required arguments and execute the call in the operating system kernel.

The specific handling of each type of call in an Android application is described in this chapter.

Chapter 4 and 5 presents the design and implementation of our DSE tool, HYBRIDSE, specifically tailored for APK files.

HYBRIDSE combines SPF and CORANA/API to perform DSE for Android applications that contain native code and further external calls in the operating system. The underlying mechanism is implementing connection interfaces that obey calling conventions between different platforms for maintaining the environment and path constraint update.

The chapter also elaborates on taint analyses, including the implementation of our taint analysis module within HYBRIDSE.

Chapter 6 discusses the performance of HYBRIDSE in tracing Android applications through control flow graph construction. Subsequently, we evaluate the control flow graph generated by HYBRIDSE in two classification tasks utilizing graph similarity.

Chapter 7 presents the results and effectiveness of taint detection on Android apps and malware. The result shows that HYBRIDSE can identify the correct data leaks in a well-defined benchmark and real-world spyware.

Chapter 8 discusses related works, while **Chapter 8** provides concluding remarks for the thesis.

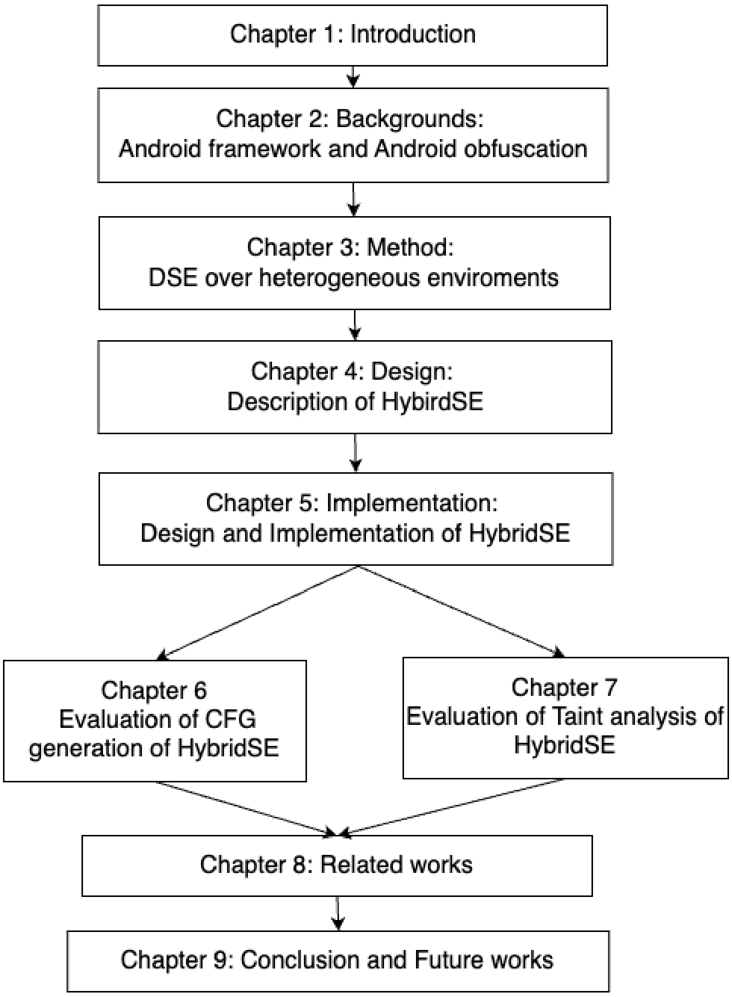

Chapter 1: Introduction
Chapter 2: Backgrounds: Android framework and Android obfuscation
Chapter 3: Method: DSE over heterogeneous enviroments
Chapter 4: Design: Description of HybirdSE
Chapter 5: Implementation: Design and Implementation of HybridSE
Chapter 6 Evaluation of CFG generation of HybridSE
Chapter 7 Evaluation of Taint analysis of HybridSE
Chapter 8: Related works
Chapter 9: Conclusion and Future works

Figure 1.4: Thesis construction

Chapter 2

Android framework and Android obfuscation

This section provides an overview of the cross-environment framework in Android, highlighting the need for a cross-environment Dynamic Symbolic Execution (DSE) tool. Android applications often operate across diverse environments, this requires an analysis tool that can handle multiple interactions between different components and a high resilience level against obfuscation techniques. These challenges underscore the rationale behind the development of a cross-environment DSE tool tailored for Android.

2.1 Android framework

The Android architecture consists of multiple layers, such as the Linux kernel, native libraries, runtime, application framework, and applications, which together enable the functioning and interaction of Android devices and applications.

An APK (Android Package) file encapsulates various components. The AndroidManifest.xml file contains essential information such as the package name, version, required permissions, and possible entry points through components like activities and services. The *classes.dex* file holds the compiled Java bytecode, which contains the payload of the application. The Resources and *assets* directories store app resources, while the META-INF directory contains metadata and package signature files. Optional native libraries (.so files) are stored in the */lib* directory.

Among these components, the application code resides in the *classes.dex* file and the .so files in the /lib directory (and occasionally in the /assets directory). The link between bytecode in *classes.dex* and native code in *.so*

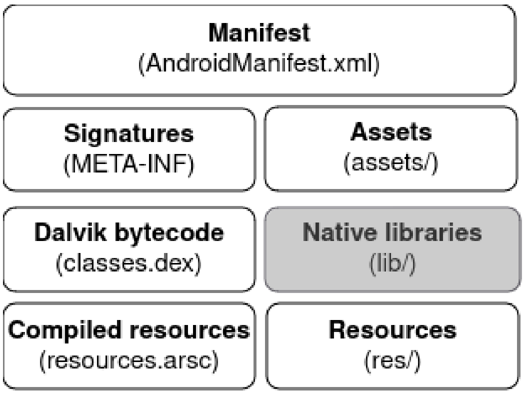


Figure 2.1: Structure of an APK file

files are facilitated by the Java Native Interface (JNI).

Native methods can be registered by JNI either statically or dynamically. Static registration explicitly declares native methods within Java classes using the 'native' keyword. At runtime, the native method in the .so file and Java classes is mapped by the JNI naming convention. In dynamic registration, developers utilize JNI's *RegisterNatives()* function to link Java methods with their corresponding native implementations. In both static and dynamic registration, the *JNI_OnLoad()* function is invoked at the start of native code execution.

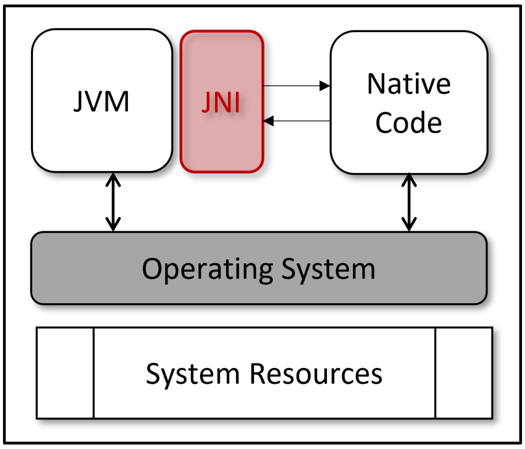


Figure 2.2: Java Native Interface

2.2 Obfuscation techniques

Obfuscation is the act of complicating code or data to deliberately make code or data more difficult to understand or reverse-engineer. Android packers employ various obfuscation techniques to hinder the analysis and monitoring of Android application behavior. Measures like anti-debugging, Virtual machine(VM)-awareness checks, or behavior-triggering mechanisms are often utilized to detect tracking on emulators and respond by either altering behavior or crashing the application. While these methods primarily target dynamic analysis, static analysis is also deterred through the application of multiple obfuscation techniques.

Common obfuscation techniques used by Android apps include identifier renaming, string encryption, multi-dex, and reflection. Some of these techniques, such as control flow obfuscation using opaque predicates, identifier renaming, and string encryption, are employed across various platforms (for both bytecode and binary code). However, certain methods are specific to the Android and Java frameworks, such as multi-dex and Java reflection.

This section explores common obfuscation techniques detected on Android, aiming to justify the adoption of DSE as the most resilient approach to combatting obfuscation.

Identifier Renaming. For readability, developers typically use meaningful names for code identifiers, following different naming conventions. However, these meaningful names help reverse engineers understand the code logic and quickly locate target functions. To minimize information leakage, identifiers can be replaced with meaningless strings.

```
public String m3a163f7d(TelephonyManager telephonyManager) {
    return telephonyManager.getDeviceId();
}
public C0010p4a8a08f0[] m363b122c() {
    if ((11 + 19) % 19 > 0) {
    }
    return (C0010p4a8a08f0[]) mc09695e2(f110M, new
        C0010p4a8a08f0[0]);
}
```

Listing 2.1: Identifier renaming in an Android malware

String Encryption. String encryption is used widely to hide original plaintexts by cryptographic functions. To effectively retrieve the original strings, a correct decryption function must be applied during reverse engineering. This process often relies on the experience of the security specialist. Alternatively, the strings can be retrieved at runtime after the decryption function has been executed.

Java Reflection. Reflection is a Java feature that allows the creation of new object instances and invoking methods at runtime. As an obfuscation technique, reflection can call specific functions implicitly, which allows programs to hide their behaviors from state-of-the-art static analysis tools. Consequently, malware developers often use reflection extensively to conceal malicious actions. Listing 2.2 shows an example of such cases. In the example, sensitive APIs such as `getLastKnowLocation` and `getBestProvider` are not directly declared but are obfuscated by reflection calls and put into a list. Only at runtime can the corresponding method be invoked via `obfuscatedMethods.get(i).invoke()` at Line 14. Reflection can also be combined with string encryption to hide sensitive function names, which are only revealed at runtime. SPF Virtual Machine can resolve reflection at runtime dynamically, allowing the correct invocation of API calls in Android applications.

```
1  public class AdvancedApiReflection {
2    private static final List<Method> obfuscatedMethods = new ArrayList();
3      static {
4      obfuscatedMethods.add(LocationManager.class.getDeclaredMethod("getBestProvider", Criteria.class, Boolean.TYPE));
5      obfuscatedMethods.add(LocationManager.class.getDeclaredMethod("getLastKnownLocation", String.class));
6      obfuscatedMethods.add(ConnectivityManager.class.getDeclaredMethod("getActiveNetworkInfo", null));
7      obfuscatedMethods.add(LocationManager.class.getDeclaredMethod("getLastKnownLocation", String.class));
8      obfuscatedMethods.add(TelephonyManager.class.getDeclaredMethod("getSimSerialNumber", null));
9      obfuscatedMethods.add(TelephonyManager.class.getDeclaredMethod("getSubscriberId", null));
10     obfuscatedMethods.add(TelephonyManager.class.getDeclaredMethod("getVoiceMailNumber", null));
11     }
12
13 public static Object obfuscate(int i, Object obj, Object[] objArr) {
14     return obfuscatedMethods.get(i).invoke(obj, objArr);
15 }}
```

Listing 2.2: Reflection is used to hide sentitive API in an Android malware APK

Multi-dex. Before the Android platform version 5.0 (API level 21), apps were restricted to a single *classes.dex* bytecode file per APK. In later versions, Multi-dex allows Android application authors to split an application's bytecode across multiple DEX (Dalvik Executable) files. This is typically necessary for applications that exceed the 65,536 method limit and also adds an additional layer of protection. Static Android analysis tools such as *apktool* and *dex2jar* can face difficulties when dealing with Multi-dex.

Packing. To thwart static analysis, Android packers employ various measures to shield both DEX files and so files (Figure 2.3).

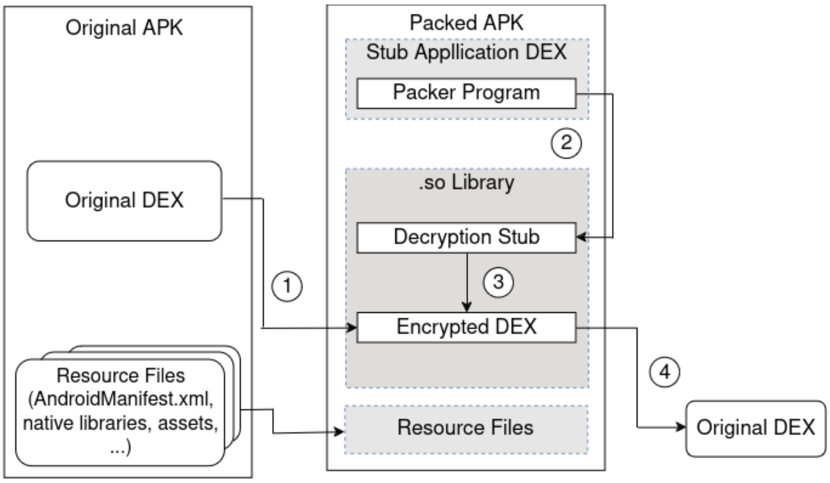


Figure 2.3: DEX encryption mechanism on APKProtect [1]
1 - Packing, 2 - Executing at runtime, 3 - Decrypting, 4 - Unpacking

DEX files are typically safeguarded through encryption, dynamic loading (i.e., dynamically releasing protected data into memory for execution at runtime), dynamic modification (i.e., altering DEX files in memory while the app is operational), obfuscation, and re-implementing with native code. Additionally, some packers utilize virtual machine-based protection methods, translating Dalvik bytecode into a customized bytecode format and integrating a tailored virtual machine to interpret them during app execution on a device. For .so files, Android packers utilize techniques such as ELF file packing or obfuscation tools like Obfuscator-LLVM.

2.3 Disscussion

Regarding obfuscation techniques, there is no one-size-fits-all solution. Anti-debugging methods like VM awareness and behavior-triggering frequently hinder dynamic analysis. Static cross-environment tools face challenges with obfuscation techniques such as control flow obfuscation, opaque predicates, dead-code insertion, and self-modification. Techniques specific to Android and Java environments, such as Java reflection, multi-dex, and dynamic loading, further complicate static analysis.

The rationale behind combining 2 DSE tools - SPF and CORANA, is their effectiveness in handling various obfuscation techniques, though they may still struggle with platform-specific methods such as multi-dex and dynamic loading. To our knowledge, existing Android taint analysis tools like ARGUS-SAF and JUCIFY have not yet addressed challenges posed by multi-dex and dynamic loading in packed applications.

Chapter 3

DSE over heterogeneous environments

A conventional DSE framework targets the sequential execution of a program on a single platform. However, real-world programs are often neither self-contained nor in uniform environments. They mostly operate in heterogeneous platforms that differ in the environment structure, the language descriptions, and the privilege hierarchy.

We focus on Android APK files. An Android apk file consists of Dalvik bytecode (.dex), native code (.so), and Manifest.xml, which includes permission for data access. Hence, its concrete execution goes across the environments of Dalvik bytecode, native code, and Android library functions.

This chapter discusses the dynamic execution of a cross-environment and the calling conventions necessary for transferring between environments. We categorize system calls as either black box or white box, discussing the different approaches for handling each type. Finally, we validate our chosen methods.

3.1 Symbolic Execution for instruction sets

Symbolic execution (SE) [21] associates formulas to each execution step, obeying the Hoare triple inference rules

$$\{\mathit{Pre\text{-}condition}\}\mathit{Command}\{\mathit{Post\text{-}condition}\}.$$

In the original form of Hoare logic, at each step of the execution, a fresh variable name is introduced. In an actual implementation of dynamic symbolic execution (DSE) tools, instead of the variable name conversion, the environment model and the path condition are separated such that the path condition contains only symbolic values as variables.

To build a DSE tool for binary code, the formal semantics of each instruction is required. Our motivation is malware analysis, which is mostly a user-level process and contains only *serializable* [22] multi-threads, e.g., fork the independent scanning processes. We limit the target of DSE for instruction sets on the sequential execution only (forgetting the multi-stage cache and the out-of-order execution), and the operational semantics is simplified as a transition system over *symbolic states.*

Definition 3.1.1. A *symbolic state* at a location i with an instruction `inst` is the tuple $\langle \alpha_i, (CFlow, Env) \rangle$ where

- Sym is the set of symbolic values s,
- α_i is a path condition (the *pre-condition* of `inst`) at i with $Var(\alpha_i) \subseteq Sym$, $Var(\alpha_i)$ returns the set of variables within the path condition α_i.
- $Env = \{VarEnv\}$ is a set of environment variables, where an environment variable is $VarEnv : Name \to Val$ with $Val = \{0,1\}^k \cup Expr$.
  $Expr$ denotes the set of expressions that operate on constant values and symbolic values.
- $CFlow \in (Inst \times Loc)^*$ is a path to the predecessor of `inst`.

k is typically either 32 or 64.

The Hoare logic inference rule for an instruction `inst` (from the *pre-condition* to the *post-condition*) is directly deduced from its operational semantics. Let i be the program counter.

$$\underset{\textit{Operational semantics}}{\frac{Env}{Env'}\ \texttt{[inst]}\ \text{if}\ \psi} \quad \Rightarrow \quad \underset{\textit{Hoare logic inference}}{\frac{\langle \alpha_i,\ (CFlow, Env) \rangle}{\langle \alpha_i \wedge \psi,\ (CFlow.(\texttt{inst}, i),\ Env' \rangle}}\ \texttt{[inst]}\ \text{if}\ \psi$$

Note that the choice of logic for the base of Hoare logic decides the reasoning ability. For instance, a bit sequence stored at a memory address can be interpreted as a value or an address point to another location. For precise description, Hoare logic must be able to describe the *points-to* relation, which is not easy. In practice, a common backend reasoning engine of SE tools is an SMT solver, in which no suitable backend theory seems to be prepared. We consider an environment model as described in Fig. 3.1. Although model components may differ, platforms mostly share similar environment models, which often include

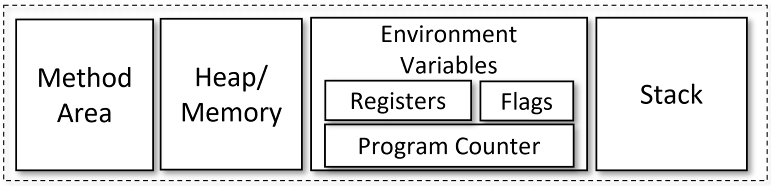


Figure 3.1: Environment model

- *the stack* (e.g., JVM Stack in Java and Stack in x86, ARM) to store local variables and temporary data.
- *a memory* (e.g., Heap in Java, Data Area in x86, and Memory in ARM) contains the program data and uses it for dynamic allocation.
- *a method area* that stores the code segment and in some cases, the instruction code.
- *environment variables* such as registers, flags, and the program counter (PC) (though there are no flags in Java).

For instance, a DSE of ARM instruction can be defined with the environment model *Env* includes:

- $R : Reg \to Val$ is a set of registers

$$Reg = \{r_0, r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}, r_{11}, r_{12}, sp, lr, pc, apsr\}$$

- $F : Flag \to Bool$ is the set of flags $Flag = \{N, Z, C, V, Q, GE\}$.
- $M : Mem \to Val$ is a set of memory locations $Mem = \{m_0, m_1, m_2, ...\}$.
- $S : Stack \to Val$ is the stack $Stack \subseteq Mem$ (and $S \subseteq M$).

At the beginning of execution, all environment variables above are initialized as symbolic values. The operations on Bit-vector theory in the DSE of ARM instructions follow the SMT-LIB (Satisfiability Modulo Theories Library) standard.

Example 3.1.1. Figure 3.2 illustrates an example of an ARM assembly code snippet and the corresponding path condition generated during dynamic symbolic execution (DSE). Initially, at *_start*, the environment is initialized with symbolic values, such as $r_0 = r_0$_SYM, $r_1 = r_1$_SYM, and so on. After executing the instruction `mov r0, #2`, the environment updates to $r_0 =$ #2. At state `n1`, the instruction `cmp r0, #4` followed by `bleq` causes the execution to split into two branches: one where r_0 equals 4 and another where r_0 does not equal 4. These branches produce two path conditions, which in SMT-LIB are represented as $(= r_0\ 4)$ and (not$(= r_0\ 4)$). Solving with an SMT solver, $(= r_0\ 4)$ is satisfiable.

Continuing the DSE, at states n4 and n5, the satisfiable path conditions are reported as bvand ((= r_0 4) (bvslt (bvsub r0 #5) #0)).

```
    _start:
n0:     mov  r0, #2
        mov  r1, #2
        add  r0, r0, r1
        cmp  r0, #4
n1:     bleq  func_1

n2:     subs r1, #5
        blmi  func_2
n5:     ...

    func_1:
n3:     mov  r1, r0
        bx   lr

    func_2:
n4:     mov  r0, r1
        bx   lr
```

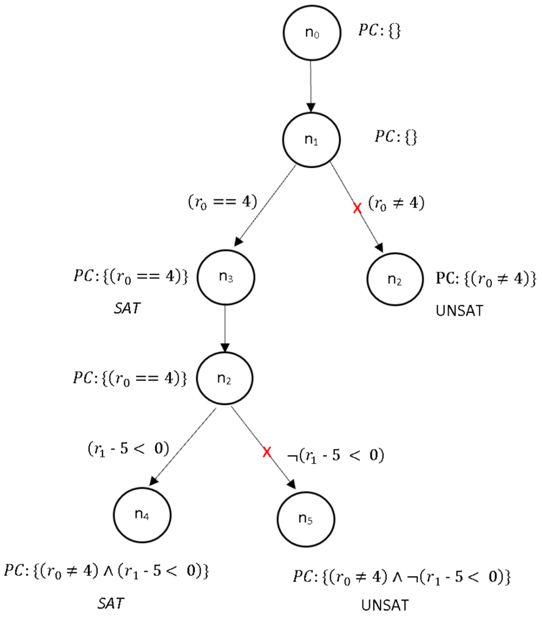


Figure 3.2: Example of path condition generated throughout ARM execution

3.2 DSE implementations in binary code

There are lots of tools for high-level programming languages, such as C/C++ and Java are developed (e.g., KLEE [23], CUTE [24], and SPF [17]). For binary code, McVeto [25] is an early static symbolic execution example, and from around 2015, several dynamic symbolic execution tools have become available, such as MAYHEM [26], KLEE-MC [27], CoDisasm [28], S2E [29], angr [18], BINSEC [30] and BE-PUM [31].

Different from high-level programming languages, binary code has no syntax, i.e., no grammar constraints on the order of instructions, no distinction on data and code. Further, the control flow graph is implicit, whereas a high-level programming language obtains it for free during the parsing.[1] The control flow graph construction, equivalently the disassembly, becomes a challenge when malware adopts the obfuscation techniques. The

[1] For object-oriented languages, an inter-procedural control flow like a call graph requires a *points-to* analysis [32, 33].

syntactic disassembler, e.g., CAPSTONE[2] and IDApro[3], are easily cheated by the obfuscation techniques, especially combined with indirect jumps and self-modification to confuse the next control point. Dynamic analyses are also cheated by VM awareness, anti-debugging, and/or trigger-based behavior [34]. Dynamic symbolic execution (DSE) on binary code is considered the most powerful (though heavy) [35, 36].

When targeting malware, there are *PC malware* and *IoT malware*. *PC malware* mostly focuses on x86 with typical OSs, e.g., Windows, Linux, and macOS. It often uses heavy obfuscations to bypass anti-virus software, which is typically introduced by a *packer*. On the other hand, *IoT malware* is often naive because of the absence of anti-virus protection in IoT devices. However, the target platforms of IoT malware vary a lot whereas the target OS is often Linux-based. For developing DSE tools for binary code, the instruction level covers a single context, and the definition of the formal semantics is the target task. A popular approach is to translate into an intermediate language (IL), e.g., VEX, LLVM, and BAP (used in angr, KLEE-MC, and MAYHEM, respectively), by using a common disassembler like CAPSTONE. This makes different platforms share the same DSE implementation, but the drawback is the difficulty to handle obfuscations, which will cheat disassemblers.

An alternative approach is a platform-wise DSE implementation. The drawback is the heavy implementation effort for various platforms, which will be assisted by automatic extraction of the formal semantics from (possibly not formal) specifications. We have successfully tried this approach in the past, e.g., BE-PUM [37] for x86, CORANA [20] for ARM, and SyMIPS [38] for MIPS.

As in Chapter 2, an Android APK file consists of Dalvik bytecode, native code, and Android library (OS) function calls. There are several formal method tools for Android APK files, such as *JPF-Android* [39], jpf-mobile [40], *SynthesisSE* [13], and ANGR [18]. The former three are based on JPF and mostly work as model checkers. They treat native code as a black box component, i.e., either out of support or handling by concrete execution (testing) in the Android environment.

The last ANGR is the only working symbolic execution tool that supports Android with user-defined native code as a white box callee. It converts ARM native code into Python description, and further into the intermediate representation (IR) SootIR. Dalvik bytecode is also converted into SootIR via Java. Then both of them are uniformly analyzed on SE on SootIR.

All existing tools depend on Java-based tools, and often *dex2jar* is used

[2] http://capstone-engine.org

[3] https://hex-rays.com/products/ida

to adapt Dalvik bytecode. *dex2jar* statically translates .dex to .class files. The translation is lightweight and practical since both Dalvik and Java bytecodes are originally compiled from Java. Our aim is to connect SPF and CORANA/API seamlessly as white boxes to each other for Android apk files.

3.3 DSE for cross-environments

3.3.1 Calling convention between different environments

The memory allocation convention and the datatype convention specify how a platform stores its data types in the memory of each platform. They may share a set of equivalent data types with different terminology. For example, the boolean types in Java and C are `Boolean` and `bool`, respectively. `String` is a specific type in Java, while C defines a string by an array of chars, terminated by "`\0`". When a call between different platforms occurs, the interface is required for passing the arguments and the return values across environment models. They are specified as the ***calling convention***, e.g., how to pass the arguments, and how to convert *the datatypes* and *the memory allocations* of values. For instance, while the x86 calling convention uses the stack to pass arguments, the ARM calling convention uses the registers for the first three arguments and pushes the remaining onto the stack.

The memory allocation convention and the datatype convention specify how a platform stores its data types in the memory of each platform. They may share a set of equivalent data types with different terminology. For example, the boolean type in Java and C are `Boolean` and `bool`, respectively. `String` is a specific type in Java, while C defines a string by an array of chars, terminated by "`\0`".

When the caller passes the environment to the callee, there are two choices, *copy* or *share* the environment. A typical choice is the former, especially when platforms have different memory allocation and datatype conventions. When copying primitive type arguments, they are directly converted to the corresponding types in the other. For pointer types (e.g., string, list, and array), the whole data structure that is pointed to needs to be copied, which is traced from the pointer value. Since the datatype specifies how to trace the data structure in the memory, either the caller or the callee needs to know the arguments' data types and the return values. For instance, when Java calls ARM native code in an APK file via JNI (Java Native Interface), the caller side knows. When ARM native calls a system function, the user mode process is interrupted and OS handles the interface. Thus, the callee side also knows.

3.3.2 Handling black box callees

We cannot observe the data and control flow of a *blackbox callee*, e.g., tasks running on the operating system, closed-source components, or no SE tools are available. Thus, a black box callee during DSE is approximated in some ways. One possibility is their manual modelling, which may be too expensive or even impossible. Instead, we have two reasonable choices: (1) return new symbolic values (*over-approximation*), or (2) execute with a satisfiable concrete instance (*under-approximation*). We call the latter *"concretization"*.

The former is useful to detect *VM awareness* and *trigger-based behavior* [41], such as *April fool attack* (which occurs only at specific time) and *STUXNET* (which works only at specific IP addresses). However, its unbounded usage will quickly make DSE intractable. The latter reduces the symbolic execution to the concrete execution with a satisfiable instance of the path condition. This is reasonable when the result of the external call will not affect later conditional branches, e.g., scan the ports and try to connect with them. Only the possibility is an *error*, e.g., *not found*, which is detected as an inconsistent datatype of the return value. *Minesweeper* [41] is an early example of manually switching such options depending on callees.

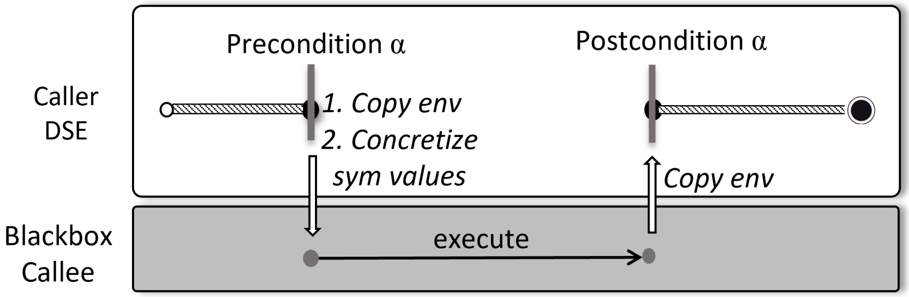


Figure 3.3: Call to a black-box platform

There are several methods for concretization in the existing implementation.

1. *Copy the whole execution*
   The symbolic execution engine parallelly maintains both the symbolical represented environment model and the corresponding concrete environment. For instance, SPF runs on JPF and uses jpf-nhandler plugin [16] to transfer the whole execution from the JPF to the host JVM when a native call occurs. At the return from the native call, SPF continues with the identical path condition, but with instantiated variables that have a dependency on the native call.

2. *Copy the environment and concretize symbolic values*

   (a) Eager concretization on all symbolic values
   The call to the black box callee is executed in the actual system and the return result updates the environment of SE. This reduces the current branch in SE to a single concrete execution. Thus, at the return, the path condition becomes *true*.

   (b) Lazy concretization on required symbolic values
   The arguments of the call are instantiated to execute in the actual system. Same to (a), the return results update the environment. However, only the arguments required at the call are concretized, and other symbolic values and expressions out of the context are left unchanged. The path condition is set identical, i.e., the same *pre-* and *post-* conditions, except for instantiating with the lazy concretization.

Our approach for black-box callee [19] follows (2).(b) to keep values symbolic as much as possible. To concretize symbolic values, we use an SMT solver to randomly find a satisfying value that meets the current path condition.

Either case obeys the same calling convention 3.3.1, in which the extraction of type information is crucial for tracing the points-to relation of values. Data type information of each platform is needed and often it can be automatically retrieved from the developers' documentation. We have some examples.

- For x86-32 on windows, BE-PUM handles Windows-API calls [42], in which the data type information is extracted from Microsoft Developer Network (MSDN).
- For ARM-32 on Android, CORANA/API [19] (which is an extension of CORANA) handles Android system function calls, in which the data type information is extracted from Linux Manual Page.

First, they are based on the argument name convention of the pseudo code descriptions in manuals. Second, sentence similarity analysis can often further classify the data types [42].

After the argument types are detected, the type conversion relation and the theory correspondence needs to be prepared. Table 3.1 describes the difference between the Java, ARM, and C/C++ platforms. From ARM to C, system calls are wrapped by C standard library functions in the GNU C Library (GlibC)[4]. Hence, the argument types of Linux API can be automatically retrieved from GlibC documentation by applying name conventions [19].

[4] `www.gnu.org/software/libc/`

From Java to ARM, argument types are directly shown in the JNI declaration in the Java class.

	Java	ARM	C
Calling convention	Stack-based[5]	AAPCS[6]	C calling convention[3]
Data types	Java types	32 Bit-Vector	C types

Table 3.1: Java and ARM data type comparison

3.3.3 Handling whitebox callees

Modern applications (e.g., Java, Android, and .NET programs) combine the main block with the native code, in which its data and control flow are visible in the user-level process. They are *whitebox callees*. We have two choices.

- *Convert the program into a single context*
  The code in the different platforms is translated into a single platform, e.g., native code into bytecode and C/C++ code into Java [43]. However, this semantics conversion proves to be difficult.

  ANGR [18] translates both Java/DEX bytecode and native code, e.g., C/C++, ARM, x86, MIPS, into an intermediate representation of SootIR. This approach limits the deobfuscation ability, i.e., it may be cheated by the combination of self-modification and indirect jumps.

- *Combine DSE tools of individual platforms*
  Interfaces between different DSE tools follow the calling conventions to keep track of the execution.

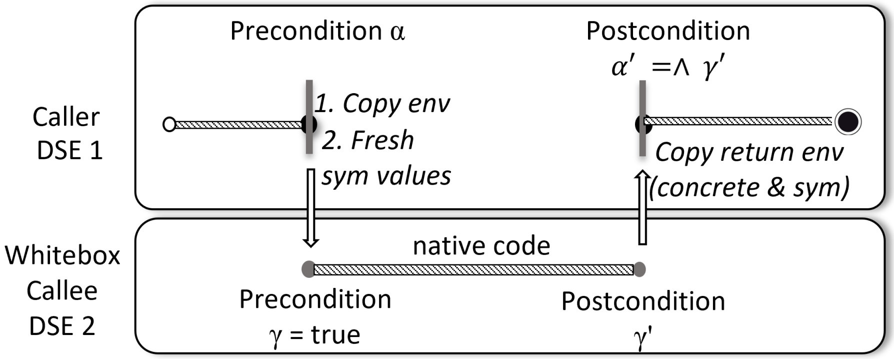


Figure 3.4: Call to a white-box platform

Our choice is the latter, e.g., in an APK file, the Java bytecode is analyzed by SPF, and ARM native code is by CORANA/API [19].

The symbolic execution in the white box callee starts with the clean environment, i.e.,

- If the arguments contain symbolic values, their values are set to fresh symbolic values (with the constraints between existing symbolic values).

- The initial path condition is set to *true*.

After the SE in the white-box callee is over, the conjunction of path conditions of the caller and the callee is taken.

Note that, as Table 3.1 shows, the data type conversion occurs when crossing the environments. Such conversion also leads to the backend theory conversion. For instance, the symbolic execution on high-level programming languages often uses LIA (Linear Integer Arithmetic), whereas on binary codes use BitVector.

3.4 Discussion

In this section, we introduce a framework for implementing a DSE tool for cross-environment platforms. Most existing DSE tools for Android, such as jpf-mobile, JPF-Android, and SynthesisSE, treat native library code as a black box and execute it concretely. A black-box approach restricts tracking data and control flows. However, Android APK files are typically deployed with native code libraries, which are white boxes. To achieve the most comprehensive program flow, we propose a DSE framework that accommodates environments with components of varying visibility levels. The transition between DSE between black-box and white-box components needs to be handled carefully by interfaces that abide by the calling conventions.

Chapter 4

Description of HybridSE

We present the components for cross-environment DSE, named HybridSE, for an Android application running on an ARM-based device. Different from existing DSE tools, we use platform-specific DSE tools for each white-box component and keep track of the environment and the path condition update throughout the execution across different platforms. For Android APK, we combine two DSE tools Symbolic Pathfinder SPF and corana/api, which are for Java and ARM code, respectively, to analyze white-box native code.

After discussing the components, we provide an overview of the system of HybridSE. This includes the strategy for generating the control flow graph and, atop the DSE engine, an added taint analysis module.

4.1 DSE components

4.1.1 DSE for Java Bytecode: Symbolic PathFinder

Java bytecode is the instruction set of Java Virtual Machine (JVM) and can run regardless of the underlying processor architecture. JVM uses Stack to hold its local variables and temporary data, and also to manage method invocations and their returns. Besides JVM Stack, Native Method Stack is prepared for native methods.

JPF [44] is an extensible Java analysis tool and its core is a customized JVM that supports multiple analysis strategies. Symbolic PathFinder (SPF) [17] is a symbolic execution extension built on top of Java PathFinder (JPF). Instead of the standard JVM, SPF defines the operational semantic descriptions of Java bytecode instructions by adding new symbolic classes to deal with symbolic operands (Fig 4.1). SPF keeps both symbolic and concrete executions in parallel. At library function calls, SPF passes only the concrete execution from the JPF custom VM to the host JVM. The result of the symbolic expression is suspended and later used to generate path conditions.

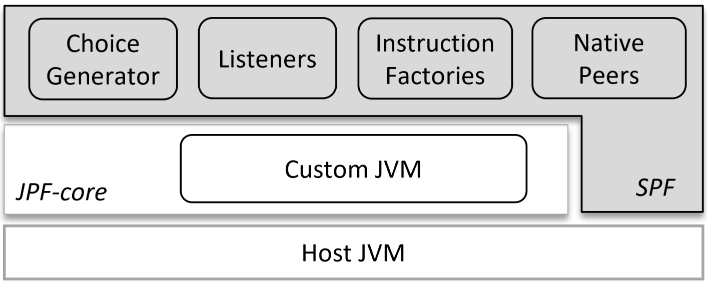


Figure 4.1: Symbolic PathFinder

4.1.2 DSE for ARM instruction: CORANA

Nowadays, the use of native code for mobile applications steadily increases and 95% of the mobile devices run on ARM CPUs. ARM is a RISC instruction set with 4 Cortex series A (Android), M (Micro Controller), R (Real-time), and recently X (high-level CPU). Although each variation of a Cortex has around 200 instructions only, each cortex has 10-20 variations, which are either 32-bit or 64-bit instructions. An Android APK file specifies the native code in either ARM 32 bits, ARM 64 bits, or x86, ignoring the differences among Cortexs of ARM.

CORANA (Cortex Analyser) [20] is a preliminarily DSE tool focusing on 32 bits instruction set of ARM Cortex-M, which is implemented based on the semi-automatically extracted formal semantics from ARM Cortex-M manual[1]. The semantics of each ARM instruction is represented as a Java method built on top of a customized **BitVec** class, which is a pair $\langle bs, s\rangle$ of a **BitSet** 32-bit vector bs and a string s. Corresponding to the BitVector theory of SMT solvers, 35 basic methods are prepared for the binary symbolic execution engine.

Note that CORANA adopts the Bit-Vector theory of SMT solvers as the base of Hoare logic. Thus, the *points-to* relation cannot be described by formulae. Therefore, the *points-to* relation on concrete addresses can be traced, but the *points-to* relation on a symbolic value simply requires a fresh symbolic value.

[1] `https://developer.arm.com/documentation`

4.1.3 ARM-Linux Kernel call: CORANA/API

For black-box calls from ARM native code to the operating system kernel, we follow the API stub of CORANA/API, i.e., concretize symbolic values in the required arguments and execute in the kernel.

An external call to a different environment requires (1) passing the arguments and the environment when the call occurs, and (2) receiving the output and the environment update when the call is over, which follows the calling convention.

Note that the arguments, the output, and the environments may contain pointer values, for which tracing pointers are required. Thus, the detection of types of each value is needed [19]. The environment transfer is partial in the sense that the transfer is only in their reachable and visible areas.

A Linux system call (or Linux external library call) in CORANA is a black-box component since the system process is invisible from the user process. There are 3 choices (a) model the black-box component, (b) introduce a new symbolic value as the output, or (c) concretize symbolic values for concrete execution in the OS. (a) is often expensive, and (b) fits the *trigger-based behavior*. Our current choice is (c) to cover typical scanning loops, e.g., the port scan. Note that we keep the concretization as minimal as possible, i.e., only for needed values. After the execution in the OS, it updates the environment of CORANA.

The path condition is kept unchanged since the constraints of conditional branches in the black box are inaccessible and cannot be observed from the user process.

4.1.4 Java-ARM communication

For combining DSEs, the arguments, the output, and the environments may include symbolic values, and it also requires (3) the path condition update. Fig. 4.2-right describes the white-box call from SPF to CORANA. The symbolic execution in SPF is presented by Java environment variables α, β and the path constraints Φ_{java} on these symbolic values. At the point of the native method F invocation, the arguments α, β for the native method are passed to CORANA. It starts with the initial environment α, β and the initial path condition $\Phi_{native} = true$. At the end of the native code, the return value *ret* of CORANA, which can either be a symbolic or concrete value, and the path condition Φ'_{native} are passed to the environment of SPF. Then, the postcondition of the white box call is updated as $\Phi'_{java} = \Phi_{java} \wedge \Phi'_{native}$.

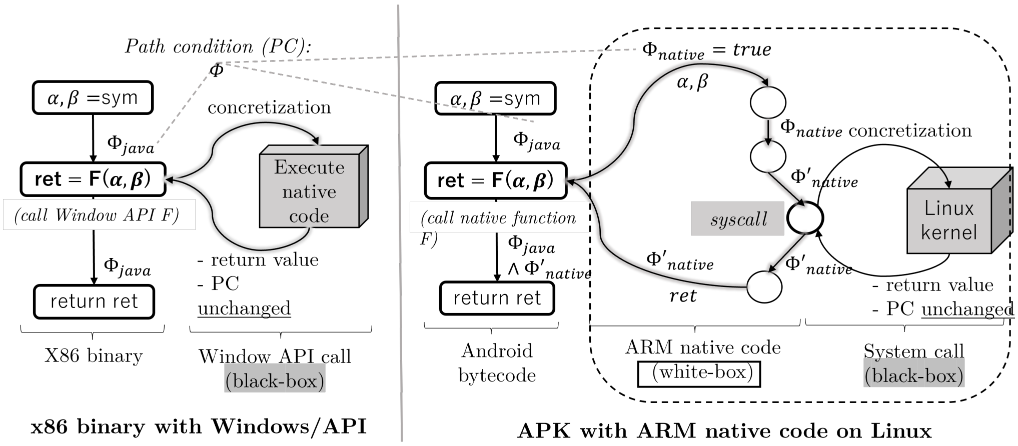


Figure 4.2: Call handling for Android on ARM compared to Windows x86

4.2 HybridSE Overview

4.2.1 Preprocessing

SPF requires Java bytecode and a configuration file (.jpf) as prerequisites, instead of Dalvik bytecode, and CORANA/API requires an ARM binary (.so) file. As preprocessing, we use *apktool* to decompile the APK file, extracting resources including the AndroidManifest.xml file, Dalvik bytecode, and other assets. Then, *dex2jar* converts Dalvik bytecode (.dex) to Java bytecode (.jar) (Fig. 4.3).

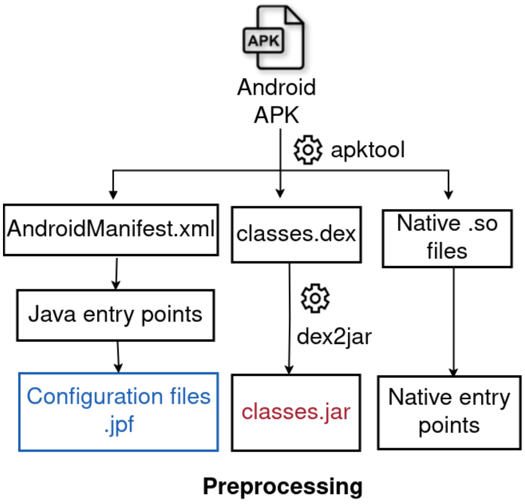


Figure 4.3: Preprocessing in HybridSE

- *From AndroidManifest.xml:* Identify potential entry points such as Activities, Services, AsyncTask, and Application, we generate a dummy Main Java class to initiate the entry points. Subsequently, SPF configuration files are created based on them, specifying analysis parameters.
- *From Dalvik bytecode:* The Dalvik bytecode (classes.dex) is converted into Java bytecode (classes.jar) using *dex2jar*.
- *From Native code:* After extracting the APK file, we search for .so files in the */lib* and */asset* directories. Ghidra[2] is employed to extract the symbol table, which helps us to locate registered native functions and their respective positions in the binary. Each function in the native code act as an entry point when called from Java.
- *Mapping a native function registered in Bytecode to its corresponding region in Native code.* Native functions can be resolved either statically through JNI naming conventions or dynamically via the *JNI_OnLoad()* function.[3]

4.2.2 Construction of Cross-environment Control Flow Graph

Definition 4.2.1. An **instruction** is a fundamental unit of executable code in a binary representation. An instruction I can be defined as:

$$I = (\text{Address}, \text{Opcode}, \text{Operands})$$

where:

- Address is the location in memory where the instruction is stored.
- Opcode is a code that identifies the operation to be executed.
- Operands are data or addresses that the operation acts upon.

Instructions are sequentially executed by a processor or interpreter, forming the basis for program execution and control flow.

Definition 4.2.2. A Control Flow Graph (CFG) of a binary code P is a directed graph CFG $G = (E, V)$ that depicts the execution process of P. In this graph, each node represents a pair ⟨Address, Instruction⟩, corresponding to an address and its associated instruction.

[2] `github.com`NationalSecurityAgency/ghidra/releases/tag/Ghidra_10.3.1

[3] docs.oracle.com/en/java/javase/17/docs/specs/jni/design.html

Unlike most static analysis tools [9, 13, 6] that construct CFGs by pre-constructing native CFGs and then mapping the calls of bytecode CFGs with native CFGs, HybridSE adopts a different strategy. It constructs a cross-environment CFG in an on-the-fly depth-first-search manner. It converts between two DSE engines, SPF and corana/api, depending on the current instruction being executed.

Algorithm 1 Constructing cross-environment CFG

```
Input: entry point $jentry$ of Java classes, and native code $C$
Output: A cross-environment CFG $G$ for $jentry$
  $G \leftarrow \emptyset$
  for $insn$ in SPF.runDFS($jentry$) do
      $G \leftarrow G \cup \{insn\}$
      if $isJNICall(insn)$ then
          $N \leftarrow \emptyset$                                         ▷ Native CFG N
          $jni\_args \leftarrow$ Java2Native.call(SPF.getStack())
          $N \leftarrow$ CORANA.runDFS($C, jni\_args$)
          $n\_result \leftarrow$CORANA.getReturn()
          Native2Java.return($n\_result$)
          $G \leftarrow G \cup N$
      else
          SPF.execute($insn$)
      end if
  end for
  return $G$
```

For each Java entry point, a CFG is individually constructed with Algorithm 1. Each Java bytecode instruction is executed and added to the CFG. When encountering a JNI call to native code, the execution is transferred to HybridSE through an interface communication between Java and native code.

Similarly, the native code in *.so* files is incrementally traced by HybridSE's engine. External function calls are handled by Syscall stubs, which directly execute in the OS. In cases of indirect jumps, an SMT solver resolves the path condition and testing determines the next location. After native code execution, the control returns to Java, integrating the new CFG of the native code into the CFG of the bytecode.

4.3 Taint analysis module

Taint analysis is a program analysis method that examines the flow of information between specific source and destination points within a program. In

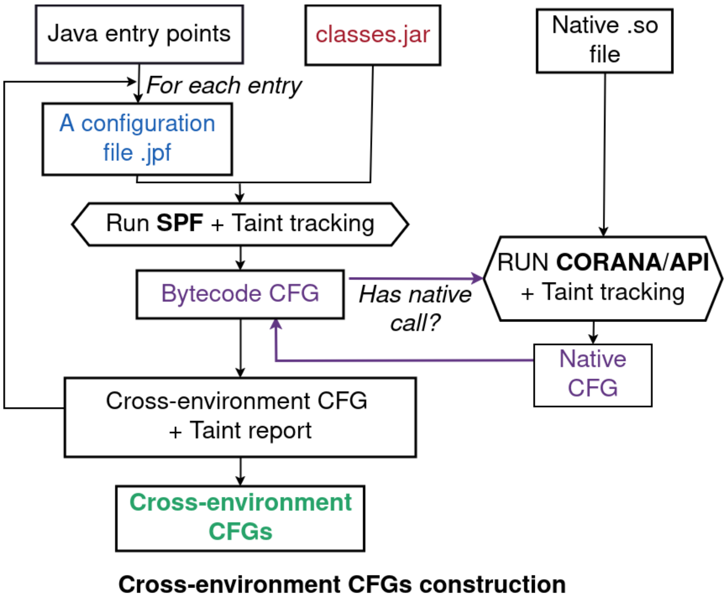


Figure 4.4: An overview of HybridSE system

security, especially in the mobile software domain, taint analysis is an effective tool to uncover potential malicious behaviors within mobile applications. It helps determine whether these apps inadvertently expose user-sensitive information to unauthorized parties.

We've integrated a taint module atop the DSE engine to capitalize on the cross-environment analysis capabilities offered by HybridSE.

4.3.1 How HybridSE detects data leakage?

Example 4.3.1. The *native_leak_array.apk* sample below is modified from the NativeDroidBench benchmark to illustrate a data leak originating from Android code, and its destination located within ARM native code.

```
public static native void send(String imei);
private void leakImei() {
    String[] strArr = new String[10];
    TelephonyManager tel = getSystemService("phone");
    strArr[1] = tel.getDeviceId();      //strArr[1] is tainted
                //SOURCE: TelephonyManager.getDeviceId()
    send(strArr);
}
```

Listing 4.1: Source in Android code

```
1  void Java_native_leak_MainActivity_send
2      (JNIEnv *jniEnv, jobject this, jobjectArray strArr) {
3
4    jobject data_clean, data_imei;
5    data_clean = jniEnv.GetObjectArrayElement(jniEnv,strArr,0);
6          //data_clean = strArr[0]
7    data_imei = jniEnv.GetObjectArrayElement(jniEnv,strArr,1);
8          //data_imei = strArr[1]
9    __android_log_print(4,&10648,&10650, data_imei);
10    //SINK: __android_log_print()
11   return; }
```

Listing 4.2: Sink in native code

After the Activity starts, it eventually invokes the `leak_imei()` method, which retrieves the IMEI device number through the API call `getDeviceId()` and stores it in an array of Strings. The taint source is recognized as `getDeviceId:()Ljava/lang/String;`. The array that contains the IMEI string is then passed through the native function `send()`, which in this case, is mapped to `Java_native_leak_MainActivity_send()` in *native_leak.so*.

Therefore, it is desirable for a taint analysis tool capable of traversing both bytecode and native code. Presently, existing taint analysis tools for Android that address both bytecode and native code such as JN-SAF[8] and JuCify[9] employ static methods, utilizing Class Hierarchy Analysis (CHA) for Java bytecode and Symbolic execution for native code. Despite the speed and efficiency of static approaches, they are susceptible to over-tainting and lack resilience against obfuscation techniques that may be present in either bytecode or native code. In example 4.3.1, if instead of `data_imei`, `data_clean` is published through the sink function at Line 9, static taint tools will suffer from over-tainting and report false positive data leak.

To produce precise and complete control and data flow of Android native code, we propose a DSE framework called HybridSE that combines existing DSE tools of bytecode and native code. In a concrete execution, Java Native Interface (JNI) bridges the gap between Java byte code and the native code (often compiled from C/C++). Following the JNI mechanism, HybridSE implements the interface to establish connections between SPF for Dalvik/-Java bytecode and corana/api for ARM 32-bit binary code with external call handling.

We demonstrate how HybridSE will apply taint analysis on Example 4.3.1 where the `strArr` is passed from bytecode to native code.

At the point of JNI call invocation, the initial default parameter is the `JNIEnv` structure containing all the JNI function pointers, with the second

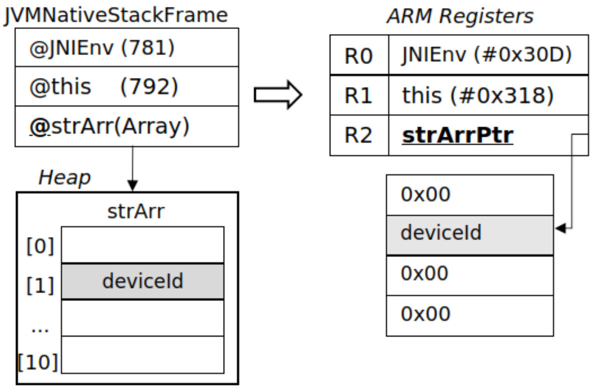


Figure 4.5: How environment is transfer in Example 4.3.1

parameter being the `this` pointer indicating the current method. The function arguments are sequentially placed into the later slots in `JVMNativeStackFrame`. For `send()` method invocation (Listing 4.2), the parameters are put into `JVMNativeStackFrame` as Figure 4.5, then the execution code is transited from Java bytecode to 32-bit ARM code.

Figure 4.3.1-right shows the data leak detected by HybridSE. For each instruction, the ARM taint rule is applied step-by-step. We discuss the details of the propagation and sanitizing rule in Section 4.3.3. We identify the API call `__android_log_print` as a sink in the native code. At 0x10652, the 4th parameter is tainted, thus, concluding there is a data leak from source to sink. In this case, the source is located in Android code and the sink is in native library code.

4.3.2 Taint analysis scenarios

A scenario of a taint analysis is a pair of a source method and a sink method, where the former retrieves data considered private (e.g., `getDeviceId()`) and the latter transmits data out of the application. A taint analysis detects possible dataflow paths of data leakage, i.e., from a source to a sink.

Table 4.1 shows an example list of scenarios, which is inspired by the list in Argus-SAF and expanded for native code. For instance, the API call `fopen("/proc/version", "rb")` is often used to read Linux kernel version) as a source method, which was missing in Argus-SAF. On the contrary, `Handler.obtainMessage` isn't included in the list of sources because it generates a new empty message instance, rather than retrieving a message from the Android handler's message queue [45]. Sources and sinks in Java are API invocation statements, e.g., `invokevirtual getDeviceID()`, `invokevirtual httpPost.getEntity()`. Sources and sinks in native code have two possibilities:

- Library function of OS system (e.g., `open 'proc'`, `getpid()`, `android_log_print()`)

- JNI callback, which enables invoking Java methods from native code. Listing 4.3) and Fig. 4.6 illustrates a potential data leak scenario involving JNI callback.

```
0x106cc adr r2, [s_getDeviceId_00010754]
0x106ce adr r3, [s_()Ljava/lang/String_00010760]
0x6d0  ldr  r6, [env,#0x84]
0x6d4  mov  r0, r4
0x6d6  blx  r6 <JNIEnv_getMethodID()>
0x6d8  mov  r2, r0
0x6da  mov  r0, r4
0x6e6  b.w 0x115b0 JNIEnv_callObjectMethod()
  ;SOURCE: Call getDeviceId()java/lang/String
```

Listing 4.3: Java method is invoked from native code in native_source.apk of Dataset 1 - Chapter 7

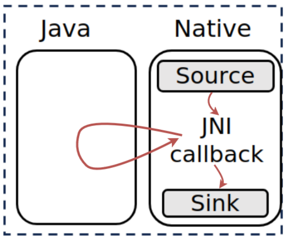


Figure 4.6: Data leakage scenario of Listing 4.3

In either case, we observe an Android malware dataset (Dataset 2 in Chapter 7), that the most frequently utilized source APIs detect device information, e.g., `BUILD.model`, `getDeviceId()`, `getLine1Number()`. The predominant sinks are httpPost-related APIs or print statements.

4.3.3 Cross-environment taint propagation

A taint analysis module in HybridSE has three steps.

1. Identify a source and give a taint tag on it. (`inject()`)

2. Propagate taint. (`propagate()`)

Sources	Details
BUILD.	Model and version
getDeviceId()	IMEI
getLine1Number()	Phone number
getLocation()	Location, country
getOutputStream()	HTTP connection
open '/proc'	Kernel version
Sinks	
Log	output to console
HttpPost.setEntity()	send to server
write()	write to file
SharedPreferences	save to object
Messenger	send text message
android_log_printf, sprintf	printing syscall

Table 4.1: Captured sources and sinks from detected data leaks

3. Recognize a sink and check it for data leaks. (`sink()`)

The propagation for Java assignments, method calls, and returns are DEF-USE chain manners of classical dataflow analyses. The propagation `propagate()` assigns a new taint tag for a variable when one of the parameters is tainted, and the taint tag is propagated until the value is redefined.

The concern is on the data type structures and the object structures. Each **primitive type** variable, **string** object, and **class** object are considered as a single taintable object. The **compound data**, e.g., **field** assignments and **arrays**, keep the taint tag of each element individually.

In 32-bit ARM native code, memory is structured in sets of 32-length words. For native code, instead of monitoring a taint tag for each bit, HybridSE assesses the taint tag for every 32-bit vector, referred to as a word.

When across environments, taint tags are seamlessly propagated during environment transitions. That is, if a value is copied between two environments, its associated taint tag is also copied according to the data type conversion.

Bytecode call to native code: a white box transfer

We explore Java calls to native code located in the .so library. HybridSE adheres to the calling convention from stack-based Java to register-based ARM. At the native code invocation, the `JVMNativeStackFrame` objects are transferred to ARM registers as 32-bit vectors. In the case of **arrays**, following the concrete execution, both the Java and native sides are aware of the array size. Therefore, the taint tag can be mapped and accessed using an index. On the other hand, a nested data structure, e.g., field, is difficult to determine the size, and the entire object is regarded as tainted.

Native code call to bytecode: a black box transfer

JNI provides a standard interface to Java functions (≈ 230 interfaces)[4]. JNI allows "callback" operations that enable Java method invocation from the native code by the method name and the signature (Listing 4.4). HybridSE treats a JNI call to Java as a black-box call such that if any of the arguments of the call is tainted, the return is treated as tainted. In the Listing 4.4, the `methodID` at Line 1 are gotten from the source API `getDeviceID`, and `methodID` is tainted. Hence, the return value of `CallObjectMethod` is tainted.

```
1 jstring Java_getImei(JNIEnv* env,jobject this,jobject* context) {
2 ...
3 methodID1=env.GetMethodID(env,p_Var1,"getSystemService","(Ljava/lang/String;)Ljava/lang/Object;");
4
5  cls=env.FindClass(env,"android/telephony/TelephonyManager");
6  methodID=env.GetMethodID(env,cls,"getDeviceId","()Ljava/lang/String;");
7
8  return _JNIEnv::CallObjectMethod(env, serviceObj, methodID);
```

Listing 4.4: JNI callback code in C

Not all JNI callbacks are over-approximated. For JNI callbacks that manipulate Java objects, strings, and arrays, we manually prepare stubs instead of using over-approximation. For instance, consider the following call:

```
jobject GetObjectArrayElement(JNIEnv *env, jobject array,int
    index); // Returns array[index]
```

This call returns the element of the array at `index`.

4.4 Discussion

Currently, we set several assumptions to combine SPF and corana/api. First, we capture the effect of the native function call via its return values without tracking the side effect. This is quite reasonable since different platforms are not easy to pass the side effects. Second, we handle a subset of

[4] docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/functions.html

types (e.g., primitive types and the string, arrays) and operations (e.g., `bvsub`, `bvadd`, `bvslt`, and `bvuge`) for Bit-Vector operations for the target of the LIA to Bit-Vector conversion).

Chapter 5

Design and Implementation of HybridSE

This chapter details the implementation of HybridSE, which consists of an extended CORANA with a taint module for ARM binaries, an adapted SPF with a taint module for Java bytecode, and a communication connector.

5.1 HybridSE architecture

We implement a cross-environment analysis tool HybridSE[1] for APK files (Fig. 5.1). Its preliminary goal is to generate control flow graphs (CFGs) and trace data across Java bytecode and native library calls.

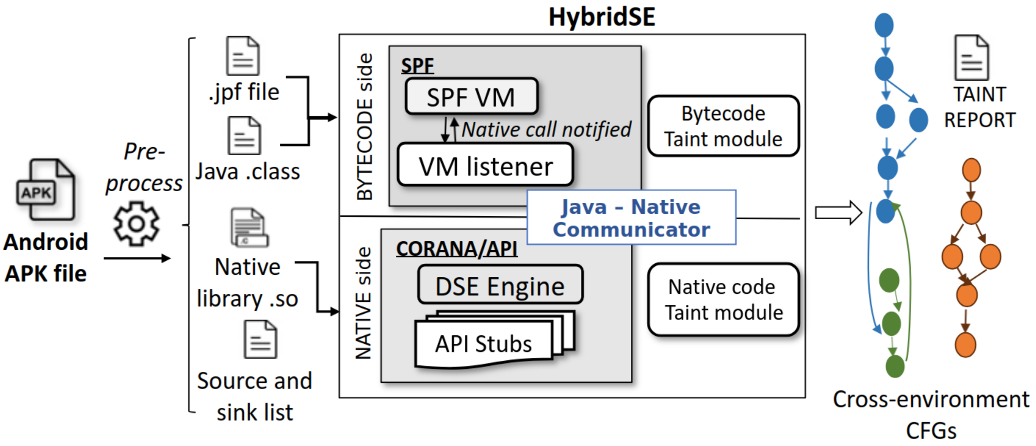


Figure 5.1: HybridSE architecture

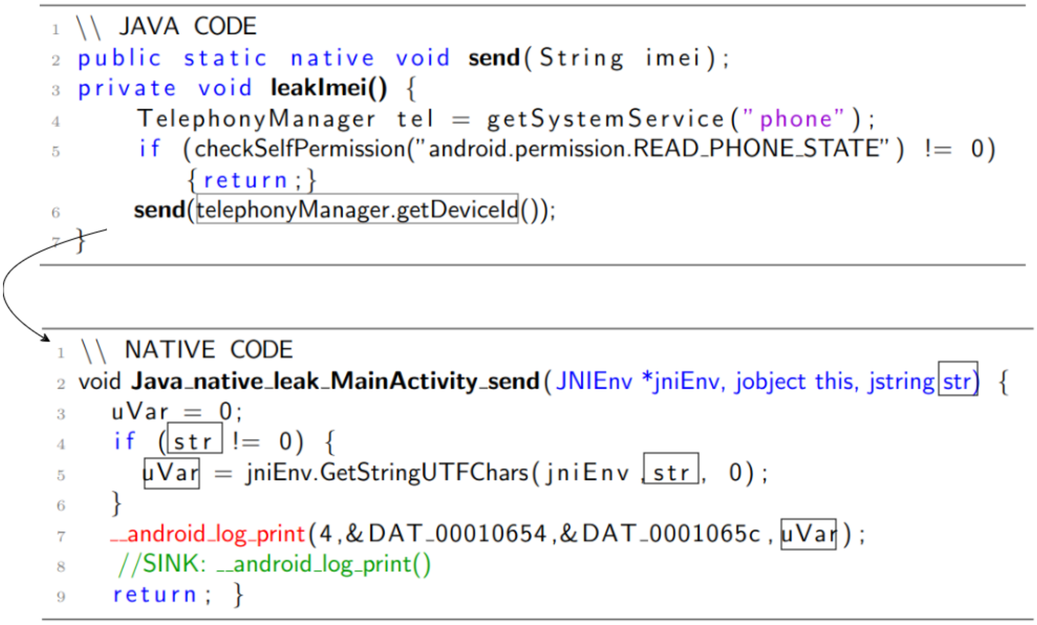
[1] https://github.com/vananhnt/HybridSE.git

HybridSE only requires the APK file as input. The outputs include detected data leaks, the generated control flow graph (CFG), and a detailed execution trace. The system is designed for comprehensive analysis of both Java bytecode and ARM native code. On the bytecode side, the Symbolic PathFinder (SPF) Virtual Machine executes Java bytecode.

The VM Listener in SPF listens for events. For HybridSE, we have implemented a specific SPF listener, consisting of 1200 lines of code, to detect native call invocations and transfer execution to the native side.

On the native side, the corana/api environment, featuring a Dynamic Symbolic Execution (DSE) Engine and API stubs, handles the symbolic execution of native code. Taint analysis modules on both sides track the flow of sensitive information. The Java-Native Communicator serves as the bridge between the bytecode side and the native side. When the VM Listener encounters a native call, it uses this communicator to relay the information to corana/api to start the native analysis.

Example 5.1.1. We illustrate an example featuring the *native_leak.apk* sample from the NativeDroidBench benchmark. The source code of this program, shown in Figure 5.2, is spread across two environments: Java code and native code.



```
\\ JAVA CODE
public static native void send(String imei);
private void leakImei() {
    TelephonyManager tel = getSystemService("phone");
    if (checkSelfPermission("android.permission.READ_PHONE_STATE") != 0)
        {return;}
    send(telephonyManager.getDeviceId());
}
```

```
\\ NATIVE CODE
void Java_native_leak_MainActivity_send(JNIEnv *jniEnv, jobject this, jstring str) {
  uVar = 0;
  if (str != 0) {
    uVar = jniEnv.GetStringUTFChars(jniEnv,str, 0);
  }
  __android_log_print(4,&DAT_00010654,&DAT_0001065c, uVar);
   //SINK: __android_log_print()
  return; }
```

Figure 5.2: Source code of the *native_leak.apk* example

IMEI (International Mobile Equipment Identity) is a unique identifier assigned to a specific mobile device. Since the IMEI can be used for tracking devices through networks, targeting by scammers and thieves, or adding devices to blacklists, its leakage poses significant security and privacy risks.

The application starts with the Java function *leakImei()*. After obtaining the IMEI number, it sends the IMEI at Line 6 in the Java code. The *send()* function, a native function, is mapped to *Java_native_leak_MainActivity_send* in the native code via a JNI call. HybridSE captures this execution flow with a cross-environment CFG. The results are shown in Figure 5.3 and the taint report in Listing 5.1.

```
ENTRY POINT: MainActivity_onCreate
Leaks: 0
ENTRY POINT: MainActivity_onRequestPermissionsResult
    LOC 42 SOURCE android.telephony.TelephonyManager
    LOC 42 SOURCE getDeviceId()Ljava/lang/String;
    LOC 64e SINK: __android_log_print
Leaks: 1
```

Listing 5.1: Taint report generated by HybridSE

5.2 CORANA extension

The core components of corana [20] include a step-wise *Binary Parser* that parses binaries using Capstone, and an *Emulator* module that implements the semantics of ARM instructions. The original input for corana is a standalone ARM binary file. To adapt corana for analyzing shared object libraries (.so files), we need to use *Ghidra* to retrieve the symbol table, which provides the list of native functions and their locations within the code. The corana/api extension builds on these core components and includes a specialized module for handling API calls, along with a taint engine that enables granular tracking of data flows (Figure 5.4).

corana/api receives a .so file, and *Ghidra* is used to find information on functions, identifying the start of the JNI method in the binary code. The binary code of the JNI function is then parsed by Capstone into disassembled code.

The dynamic symbolic execution begins with an initial environment and path configuration, specifying the starting state for the analysis. Each disassembled instruction is executed by the *Emulator*, which uses *API Stubs* to handle external calls and the *Taint Engine* to track data flow. Finally, the output reflects the environment and path after the analysis, including the results from the taint analysis.

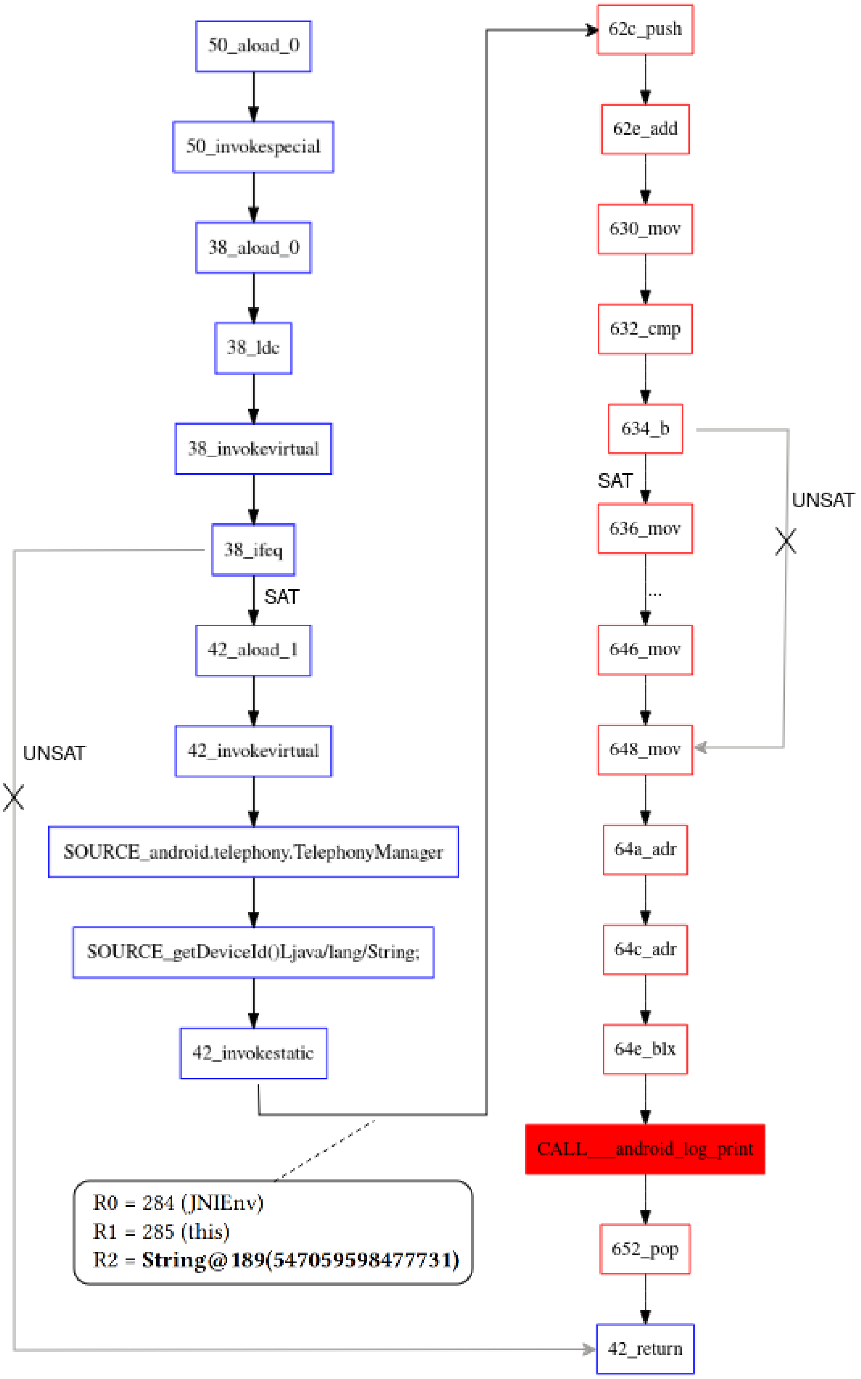


Figure 5.3: Cross-environment CFG snippet generated by HybridSE for native_leak.apk (blue = bytecode, red = ARM code)

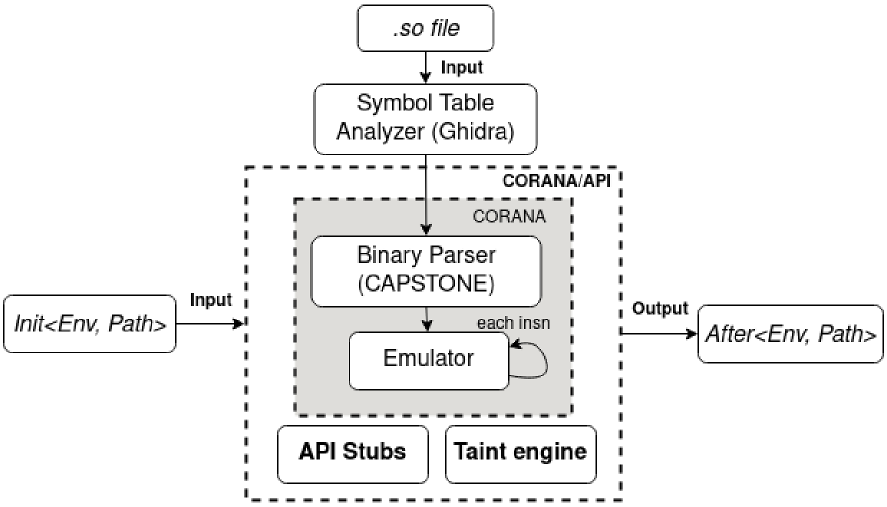


Figure 5.4: CORANA/API extension with a taint engine

In the next sections, we will explain how we added system call handling through the integration of API Stubs and introduced taint-related variables for data tracking.

5.2.1 API Stubs of system calls

The API Stub is an under-approximation of a system function call that concretizes call parameters and executes them in the operating system. Java Native Access (JNA) enables Java programs to call native functions if the function declarations are provided. Thus, creating an API Stub requires accurate library function information. In [19], API elements like function names, parameter fields, and return types were extracted from Linux function declarations. Based on the template in Listing 5.2, a total of 1,129 API Stubs have been generated to manage external calls in ARM binary code.

```
public static void $functionName(Environment env){
//1: Get original parameters from registers
        BitVec t0 = env.register.get('0');
        BitVec t1 = env.register.get('1');
//2: initialize input parameters
        $type0 param0 = new $type0();
        $type1 param1 = new $type1();
//3: read parameter values from memory
        param0 = env.memory.$getMemoryValue(t0);
        param1 = env.memory.$getMemoryValue(t1);
//4: spawn the API call
$return_type ret = CLibrary.$functionName(param0,param1,...);
//5: update registers and memory
        env.register.set('0', new BitVec(ret));
        env.memory.$setMemoryValue(t0, param0);
        env.memory.$setMemoryValue(t1, param1);
}
```

Listing 5.2: Template for API Stubs

Example 5.2.1. Fig. 5.5 illustrates the concretization of the *gettimeofday* call and the environment update according to the ARM calling convention. First, the information on the C library function

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

is extracted from Linux specifications as

Function name ($functionName)	**gettimeofday**	
Parameter type ($type0, $type1)	struct timeval	struct timezone)
Return type ($return_type)	**int**	

Then the call is executed, updating the environment as shown in Figure 5.5.

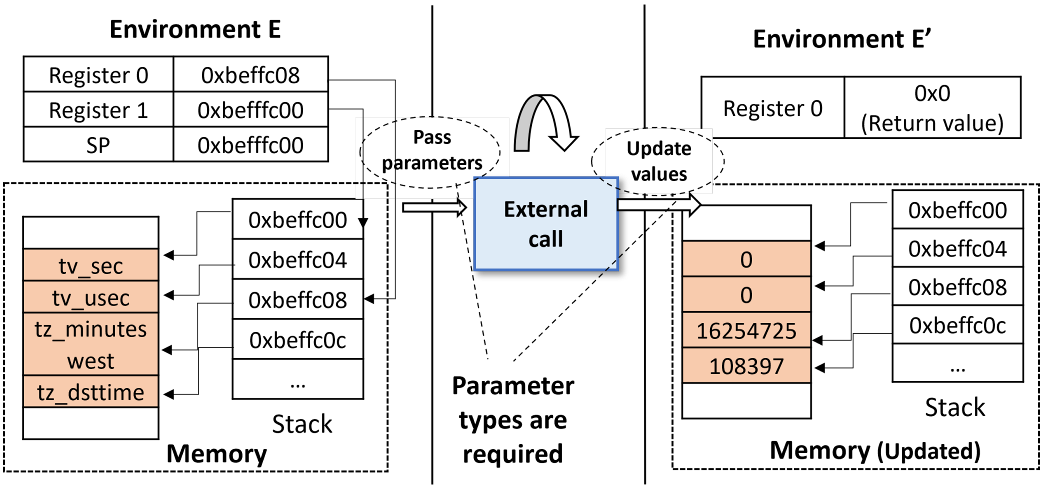


Figure 5.5: Example of a concretized external call in the ARM environment

5.2.2 Native taint engine

Figure 5.6 shows the core components that define the ARM architect in the original CORANA implement, which are the Environment, BitVec, and Emulator. In CORANA, an environment is defined to mirror the ARM instruction

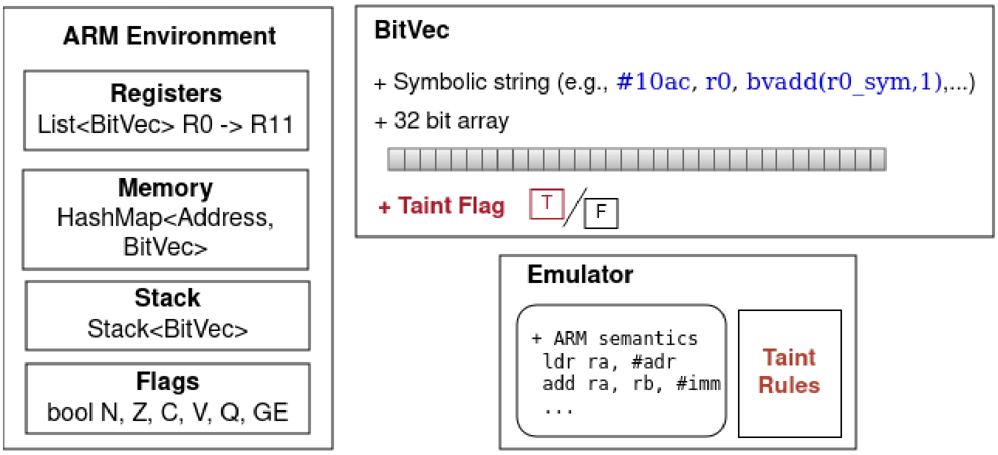


Figure 5.6: Core classes regarding environment definition in CORANA. Each BitVec object represents a 32-bit memory word during execution.

set architecture, encompassing four components: Registers, Memory, Stack, and Flags. Each 32-bit memory unit in concrete ARM execution is represented by a BitVec class in CORANA. This class features a symbolic string that represents data, which can be concrete hexadecimal, a symbolic variable, or an expression of symbolic variables. Additionally, a BitVec object includes a 32-length array composed of 0-1 bits when it represents concrete data. Built upon these foundational elements—Environment and BitVec—is the Emulator class, which implements the operational semantics of ARM instructions.

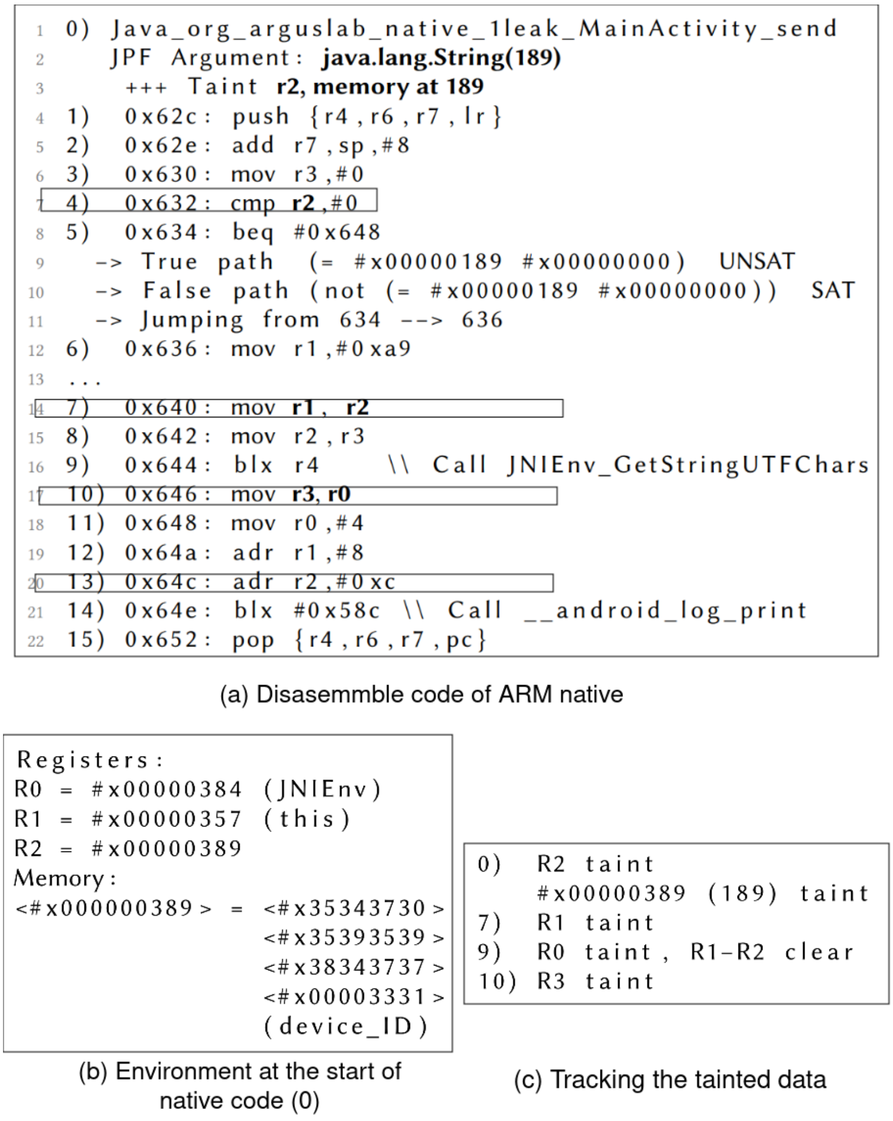
All data within the DSE engine of CORANA is represented by BitVec objects. We leverage the well-defined operational semantics of DSE for ARM, which operates on BitVec objects. Adding a taint tracking variable to each BitVec object, each representing a 32-bit memory unit in the execution, allows us to track data flow throughout the execution effortlessly.

In the Emulator, each instruction's semantics operate on BitVec objects. We have integrated taint rules into the Emulator to modify the taint flag of the operated BitVec objects. After each instruction execution, the environment, path condition, and taint tags are concurrently updated.

Example 5.2.2. Continuing Example 5.1.1. Figure 5.7 shows how data is tracked in ARM native. Figure 5.7 - a is the dissembler code of the native part in native_leak.apk in Figure 5.2. At the start of the native function, a *java.lang.String* representing *device_ID* is passed into the native function. Consequently, the register R2 is translated into the hexadecimal value of 189, which is #x00000389, and the string is stored in memory at #x00000389

(Figure 5.7 - b).

The memory slot and registers are tainted and sanitized at lines a-7, a-9, and a-10, depending on the operation. At line a-7, R1 is assigned the value in R2, carrying the taint tag from R2 into R1, and similarly at line a-10. At line a-13, R2 is assigned a fresh value #0xc, thus clearing the taint tag in R2.



```
0) Java_org_arguslab_native_1leak_MainActivity_send
   JPF Argument: java.lang.String(189)
    +++ Taint r2, memory at 189
1)  0x62c: push {r4,r6,r7,lr}
2)  0x62e: add r7,sp,#8
3)  0x630: mov r3,#0
4)  0x632: cmp r2,#0
5)  0x634: beq #0x648
  -> True path  (= #x00000189 #x00000000)  UNSAT
  -> False path (not (= #x00000189 #x00000000))  SAT
  -> Jumping from 634 --> 636
6)  0x636: mov r1,#0xa9
...
7)  0x640: mov r1, r2
8)  0x642: mov r2,r3
9)  0x644: blx r4     \\ Call JNIEnv_GetStringUTFChars
10) 0x646: mov r3, r0
11) 0x648: mov r0,#4
12) 0x64a: adr r1,#8
13) 0x64c: adr r2,#0xc
14) 0x64e: blx #0x58c \\ Call __android_log_print
15) 0x652: pop {r4,r6,r7,pc}
```

(a) Disasemmble code of ARM native

```
Registers:
R0 = #x00000384 (JNIEnv)
R1 = #x00000357 (this)
R2 = #x00000389
Memory:
<#x000000389> = <#x35343730>
                <#x35393539>
                <#x38343737>
                <#x00003331>
                (device_ID)
```

(b) Environment at the start of native code (0)

```
0)  R2 taint
    #x00000389 (189) taint
7)  R1 taint
9)  R0 taint, R1-R2 clear
10) R3 taint
```

(c) Tracking the tainted data

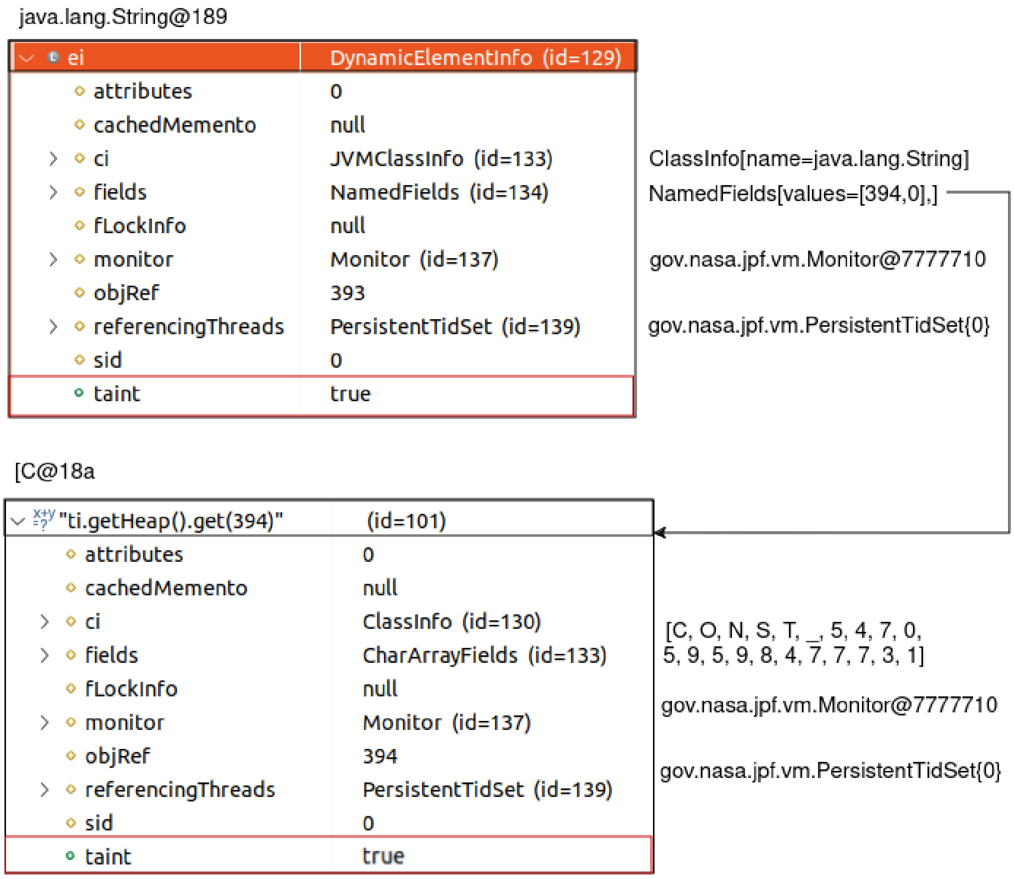
Figure 5.7: Dissamble code of native_leak.apk and tracked tainted data

5.3 SPF extension

5.3.1 Bytecode taint engine

Both the native taint engine and the bytecode side need to implement three operations: *inject()*, *propagate()*, and *sink()*. The *inject()* and *sink()* functions are implemented based on how to recognize source and sink APIs as discussed in 4.3.2.

In SPF, ElementInfo is the most crucial class for representing the state of an object or a class. It holds comprehensive information about the object throughout execution, including fields, references, and other attributes.



java.lang.String@189

ei	DynamicElementInfo (id=129)	
attributes	0	
cachedMemento	null	
ci	JVMClassInfo (id=133)	ClassInfo[name=java.lang.String]
fields	NamedFields (id=134)	NamedFields[values=[394,0],]
fLockInfo	null	
monitor	Monitor (id=137)	gov.nasa.jpf.vm.Monitor@7777710
objRef	393	
referencingThreads	PersistentTidSet (id=139)	gov.nasa.jpf.vm.PersistentTidSet{0}
sid	0	
taint	true	

[C@18a

"ti.getHeap().get(394)"	(id=101)	
attributes	0	
cachedMemento	null	
ci	ClassInfo (id=130)	
fields	CharArrayFields (id=133)	[C, O, N, S, T, _, 5, 4, 7, 0, 5, 9, 5, 9, 8, 4, 7, 7, 7, 3, 1]
fLockInfo	null	
monitor	Monitor (id=137)	gov.nasa.jpf.vm.Monitor@7777710
objRef	394	
referencingThreads	PersistentTidSet (id=139)	gov.nasa.jpf.vm.PersistentTidSet{0}
sid	0	
taint	true	

Figure 5.8: ElementInfo object of IMEI String(@189) in Example 5.1.1

By adding taint tracking as an attribute in the ElementInfo class, taint tags are updated with each operation performed on the ElementInfo object.

5.3.2 Cross-environment communicator

In SPF, listeners provide access to information about the Virtual Machine (VM) and current thread when specific events occur, such as *executeInstruction* (before instruction execution) and *instructionExecuted* (after instruction execution). We utilize this feature to create the cross-environment communicator as an SPF Listener. Our custom listener intercepts and retrieves current SPF information during native call invocations. The native call is processed by CORANA/API, and once the analysis is complete, the results are sent back to the listener, allowing SPF to continue its analysis. As discussed in Chapter 3, connecting different DSE tools requires establishing interfaces for communication between environments, specifically between ARM and Java.

Transfer environment

To facilitate environment transfers, the calling conventions of both environments must be adhered to.

- ARM calling convention: Register R0-R3 are used to pass the first 4 parameters, while the rest are put on the stack. Return value or pointer of return memory are placed in R0.

- Java calling convention: In Java, objects reside in the Heap, while method parameters use the Stack. Stack variables can hold either primitive values or references to Heap objects. A stack frame with method parameters is allocated on the Stack when a method is called. The NativeStackFrame is a special case for JNI method calls, where the first parameter is a *JNIEnv* pointer and the second is *this* pointer.

Based on this convention, Algorithm 2 outlines the process of transferring Java parameters to the ARM environment during a native method invocation. The translation is divided into three types of Java parameters: *Arrays*, *References*, and *Primitives*. For *Array* and *Reference* parameters, each item or field is translated, a pointer is placed into ARM registers, and corresponding memory in the ARM environment is created. For *Primitives*, Java primitive values from the Stack are directly translated and placed into ARM registers.

The reverse direction is simpler. After completing the native execution (i.e., executing CORANA/API) and retrieving the return value or memory from register R0 in ARM execution, we apply the `BitVec2LIA` operation to obtain the return value in Java.

Figure 5.9 provides an example of translating from the Native Method Stack in SPF to the Registers in the ARM environment in CORANA/API, demonstrating the cases of Integer and String.

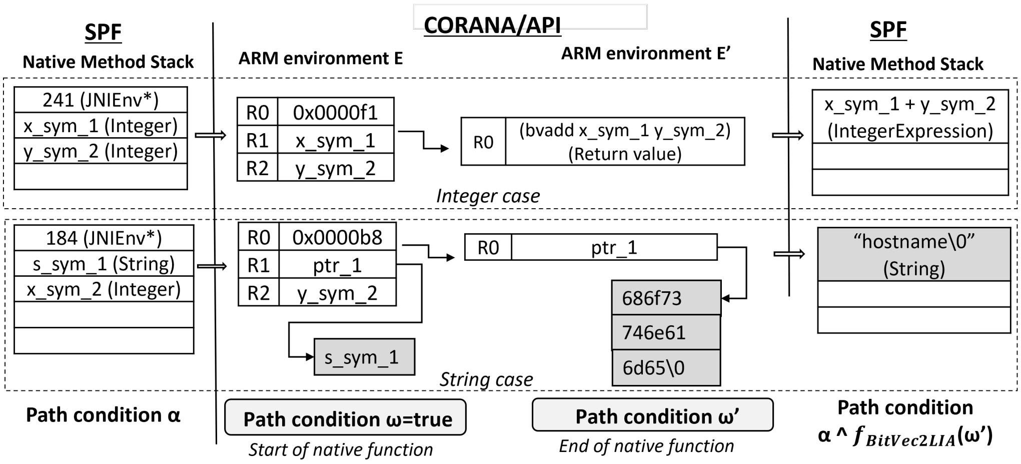


Figure 5.9: Example of a inter-environment translation between SPF and CORANA/API, back and forth

Combining path condition

The communicator initializes the path condition as $\omega_{BV} = \texttt{true}$ at the call. At the return to SPF, the difference ω'_{BV} computed by CORANA/API is converted to $\omega'_{LIA} = f_{BV2LIA}(\omega'_{BV})$ from Bit-Vector to LIA. Then, the postcondition α'_{LIA} is the conjunction of the precondition α_{LIA} and ω'_{LIA}.

The LIA from/to Bit-Vector conversion is based on `Java.util.BitSet` library. For the conversion of linear expressions, common operations (e.g., *bvsub*, *bvadd*, *bvslt*, and *bvuge*) are supported.

5.4 Discussion

One important consideration when implementing DSE for Android with the target of malware applications is ensuring the safety of the DSE implementation.

HybridSE operates in a Linux environment that is unable to perform concrete execution of Android applications. This setup helps to prevent an unauthorized automatic start of the Android application.

The DSE has three levels of operation:

- Symbolic Execution Level: At this level, symbolic operations are conducted on the emulator with semantics prepared for safe execution. This level is considered safe because it operates within a controlled environment.

- Java Virtual Machine (JVM) Level: Here, Java library calls are handled by SPF, which operates within a separate virtual machine compared to the system JVM. This separation ensures that Java system calls cannot directly access the system JVM, adding an extra layer of safety.

- System Call Level: At the lowest level, system calls from native code are concretized and executed as API stubs in the Linux OS system. While API stubs allow DSE to directly invoke system calls and retrieve return values, they can introduce potential security risks. Improper management of interactions with the underlying system might expose vulnerabilities. For example, malware could exploit this by injecting malicious payloads or commands through command string injection, or by repeatedly executing attack commands, potentially leading to a denial of service.

  To address this problem, HybridSE prohibits API stubs from allocating new memory regions in the OS system. The DSE process runs as a user-level process without administrative privileges. Additionally, HybridSE is executed through a network-isolated virtual machine to add an extra layer of protection.

Algorithm 2 Pass arguments from JNI call in Java to ARM native

```
Input: a JVMInstruction SPF.jniMethod, and the current environment of
    Java thread SPF.currentThread
Output: an ARM Environment CORANA.nativeEnv
Initialize reg_counter = 2                    ▷ Starting from register R2 in ARM
jargs = jniMethod.getArguments()
for i in range(jargs.length()) do
    if jargs[i] is T_ARRAY then               ▷ Convert Java array to ARM array
        Initialize nArr
        for each jitem in jargs[i].getFields() do
            n_item = LIA2BitVec(jitem)
            nArr.append(n_item)
        end for
        Compute offset
        Put offset in nativeEnv.register[reg_counter]
        Allocate nArr in nativeEnv.memory.at(offset)
    else if jargs[i] is T_REFERENCE then      ▷ Convert Java Reference to ARM memory
        Compute offset
        Put offset in nativeEnv.register[reg_counter]
        Allocate LIA2BitVec(jargs[i]) in nativeEnv.memory.at(offset)
    else                                      ▷ Convert Java primitive to ARM BitVec
        native_value = LIA2BitVec(jargs[i])
        if reg_counter < 4 then
            Set native_value in nativeEnv.register
        else
            Push native_value onto nativeEnv.stacks
        end if
    end if
end for
return nativeEnv
```

Chapter 6

Evaluation on CFG generation of HybridSE

In this section, we evaluate HybridSE's ability to generate unified CFGs for both Java and native code parts. This allows us to demonstrate trends in native code utilization and obfuscation within Android malware over the years. We compared HybridSE with FlowDroid but encountered difficulties running JuCify properly. Additionally, these tools generate call graphs rather than control flow graphs, limiting the scope of comparison.

To evaluate the quality of the graphs generated by HybridSE and compare them with those from FlowDroid, we conducted graph similarity analysis for two malware analysis tasks: classifying malware families and identifying Android packers. Our findings indicate that the HybridSE's CFGs provide more structure, enabling a more accurate representation of application behavior.

The experiment in Chapter 6 and Chapter 7 are performed in the same testbed. The testbed consists of an AMD EPYC 87, 2.6 GHz, 512 GB of RAM, running on a Linux Ubuntu SMP 5.4.0-66-generic computer. Our preprocessing tools are *apktool 2.4* and *dex2jar 0.9.5*. HybridSE utilizes a customized version of SPF running on JPF for Java 8.

6.1 Datasets

We conduct experiments on two sets below.

Dataset 1. Android malware datasets: DREBIN[1], AMD[2], and An-

[1] sec.tu-bs.de/danarp/drebin/download.html

[2] unb.ca/cic/datasets/maldroid-2020.html

droZoo[3]. DREBIN and AMD are malware datasets released in 2014 and 2017, respectively. The former consists of 5,560 malware samples (collected between August 2010 and October 2012) in 179 malware families. The latter consists of 24,553 malware samples (collected from 2010 to 2016) in 71 malware families.

AndroZoo, launched in 2016, is a repository of Android applications with continuously updated samples. Initially hosting over 3 million apps, it has expanded to over 15 million by mid-2021. AndroZoo draws its content from Google Play, third-party platforms such as the Chinese app markets, and VirusShare. For analyzing native code usage in Android malware, 15,000 malicious apps were selected based on the criteria of being flagged by at least ten antivirus tools.

Dataset 2. Sample packed by Android packers (has ground truth). In 2018, PackerGrind [1] assembled open-source apps sourced from F-Droid and subsequently submitted them to six online commercial packing services (namely, Qihoo, Ali, Bangcle, Tencent, Baidu, and Ijiami). As a result, the dataset contains the ground truth regarding the utilized packer. We retrieve 298 samples from this dataset.

6.2 Survey on malware CFGs

6.2.1 Native code and obfuscation usage in Android/APK

We investigate how widely is the native code utilized in Android malware and examine the generation of CFGs by HybridSE on **Dataset 2**. Native code usage is checked by two steps: (1) At least one .so file in */lib* or */assets* folder and (2) Java native methods are declared. The number of samples meeting both criteria is reported in Table 6.1 under the category "#w/Native". In the remainder of this study, we concentrate on the samples #w/Native to generate CFGs using HybridSE.

Table 6.1: Native code usage over the years

	Year	# down-loaded	# w/Native*
DREBIN	2010-2012	5560	960 (17.26%)
AMD	2010-2016	24553	850 (3.61%)
AndroZoo	2017	5000	2856 (57.12%)
	2018	5000	3651 (73%)
	2019	5000	4259 (85.18%)
	2020-2022	5000	3774 (75.48%)

[3] androzoo.uni.lu

As shown in Table 6.1, the usage of native libraries is notably high across the malware datasets, with a noticeable increase observed from the period spanning 2019 to 2022. .

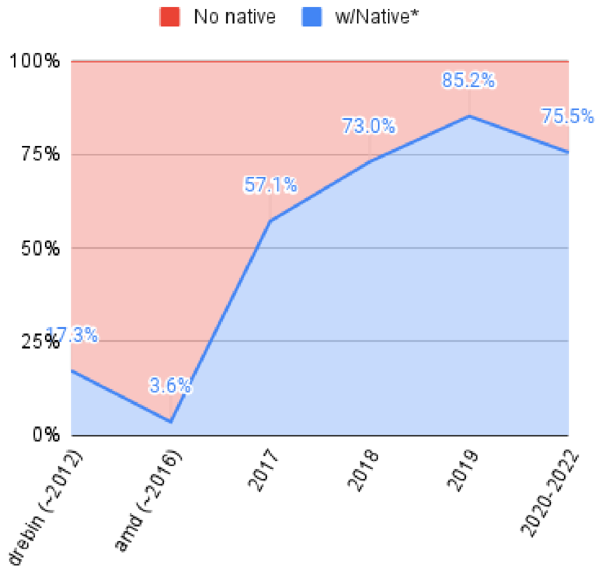


Figure 6.1: Distribution on Android native code usages

We also observe a significant amount of failures and incomplete CFG generation. summerized in Fig. 6.2 and Table 6.2.

We list the main reasons that limit HybridSE (Table 6.2) :

- **Preprocessing failure (apktool, dex2jar or AndroidManifest.xml parsing emits error).**

  In preprocessing, *apktool* decodes apk files into `AnddroidManifest.xml`, `classes.dex`, and others. *dex2jar* (v0.9.5) converts `classes.dex` into JVM bytecode. We observed a high level of translation failure that was collected before 2018.

- **Multi-dex.** Multi-dex (multiple dex files) is supported for applications with more than 64,000 methods. This leads to missing application content in subsequence `classes2.dex`, `classes3.dex`,... `classesN.dex` by *dex2jar*. Consequently, missing certain parts of entries leads to incomplete CFG construction by HybridSE.

- **Packing (Stub application).** Certain numbers of Android APK files are missing the original payload at the entry points specified in the *An-*

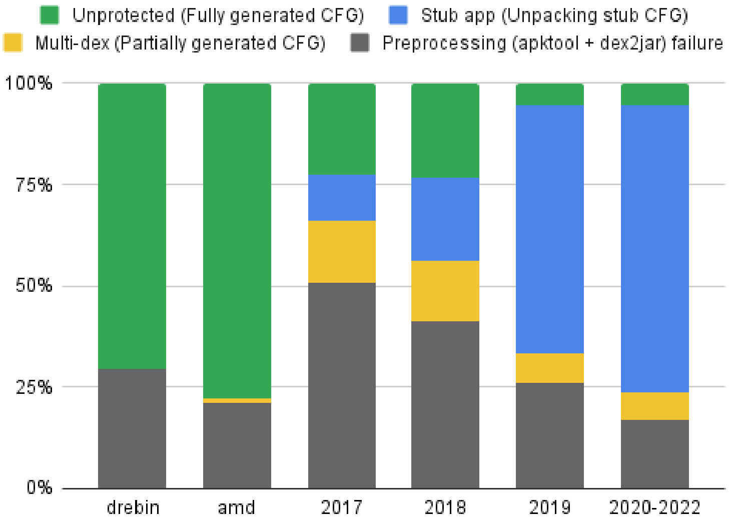


Figure 6.2: Distribution of failures and obfuscation

droidManifest.xml in their *classes.dex*. This absence suggests that the original payload has been concealed, which is typically done by packers [46]. We observe a notable increase in the presence of Android malware exhibiting packing techniques, particularly starting in 2018, and rising exponentially to more than 60% in the 2019 to 2022 period. When executed, these packed malware instances often initiate the process at a wrapper Stub application, and from Java, the application proceeds by calling `AttachBaseContext()` before loading into native code. The native library is concealed within the */assets* directory rather than the default */lib*. Native files may either be fully encrypted or encrypted partially. To generate full CFG of the payload for a packed application (i.e., unpacking packed Android application), dynamic loading is required, which is currently not supported by HybridSE.

Limitation. Our current approach utilizes the same preprocessing tools as several other static analysis tools, including *apktool* and *dex2jar*. This makes our analysis inherit the weaknesses associated with these tools. In the presence of obfuscation techniques like multi-dex and packed apps, additional capabilities are required to generate the full payload of Android applications. This includes parsing multiple dex files and enabling dynamic loading.

Table 6.2: Result on CFG types generated by HybridSE

	# APK w/Native	# dex2jar failure	# generated CFG	Among generated CFG		
				Un-protected	Stub Application	Multi -dex
DREBIN	960	285 (29.68%)	675 (70.32%)	675	0	0
AMD	850	179 (21.05%)	671 (78.96%)	661	0	10
2017	2856	1450 (50.77%)	1406 (49.23%)	644	326	436
2018	3651	1478 (40.48%)	2137 (59.52%)	852	745	540
2019	4259	1120 (26.29%)	3193 (73.71%)	230	2643	321
2020 -2022	3774	633 (16.77%)	3141 (83.23%)	204	2668	269

> **Conclusion :** Native code is widely utilized both for application functionalities and for packing Android applications. HybridSE showcases the ability to generate CFGs (Control Flow Graphs) from these two uses of Android native code, provided that there is successful and complete translation by *dex2jar*.

6.2.2 HybridSE performance when analyzing cross-environment Android applications

The runtime performance of HybridSE with respect to the CFG sizes on Dataset 2 are summarized in Table 6.3 and 6.4. The average running time is 602.27 seconds; the minimum is 81.64 seconds, whereas the maximum is 96 minutes.

Table 6.3: Relation between graph size and generation time

CFG size	Bytecode		Native		Time (s)
	# Nodes	# Edges	# Nodes	# Edges	
Average	3065	4428	268	283	602.27
Median (±SD)	783 (±6504)	1148 (±9258)	172 (±276)	173 (±294)	244.89 (±816.65)
Largest	37166	48997	978	1054	5794.73
Smallest	618	913	124	132	81.64

We observe the average number of nodes and edges for CFGs in two scenarios: the whole CFG of an unprotected apk file, and the stub applications in which HybridSE generates CFGs of only the unpacking stub of a packed APK file.

Table 6.4: Average node and edge counts from HybridSE analyzing AndroZoo malware with native code

Median	Bytecode		Native		Time (s)
(± SD)	# Nodes	# Edges	# Nodes	# Edges	
Un-protected	4654 (±9777)	6588 (±13873)	26 (±238)	25 (±250)	1040.12 (±1171.58)
Stub application	783 (±34)	1148 (±51)	292 (±267)	324 (±286)	214.90 (±253.24)

Conclusion : HybridSE could be a good candidate for cross-environment application analyses, even DSE is still time-costly.

6.2.3 What can be shown by HybridSE's CFG?

We manually observed how the unified CFG can reveal malware behavior intentionally concealed within the native code part. The CFG generated from the **Towelroot**[4] shows sequences of pthread library calls manipulating

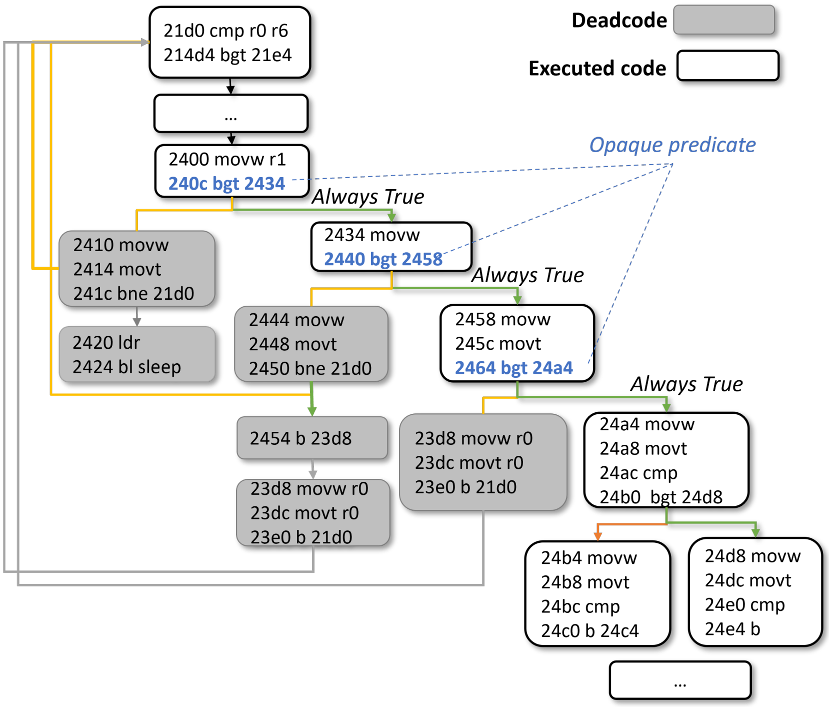


Figure 6.3: Deadcode detected in the native function of **towelroot.apk**

[4] gist.github.com/vananhnt/9c9fe78d7a74612d3b5e5363cb76c536

the mutex queue. This detected sequence is matched with the CVE-2014-315311 vulnerability[5] reported in old Linux kernels to root Android devices, which confuses the waiter structure to give the privilege of control to the user. We also can confirm that Towelroot uses an obfuscator such as O-LLVM in the native binary code to add opaque predicates to insert dead code as additional conditional branches. The CFG shows native function `java` ∗ ∗ ∗ `KernelVersion`() introduces multiple opaque predicates at 0x240c, 0x2440, and 0x2464. HybridSE successfully solves the opaque predicates and identifies 12.27% of instructions in the native functions are dead code.

The Android malware **Lotoor** *(RootKing)* hides the IP address of the external server in the native code. The malware encapsulates its URL within native code, and upon execution, it reads the URL data by invoking the native function `RootUtil.uu()`. HybridSE can retrieve the server address loaded from the data at position #x000018d0 and passed to the function `iks_base64_decode`().

> **Conclusion :** HybridSE can efficiently generate CFGs that represent program behavior, even in the presence of control flow obfuscation. In the realm of Android applications, HybridSE's CFG can offer valuable insights into the inter-language data flow and behavior exhibited by Android malware within native code, aiding in the identification of potential security threats.

6.3 Classification using Graph kernel for CFGs

6.3.1 Feature extraction from HybridSE's CFGs

Definition 6.3.1. Given a graph $G = (V, E)$, the process of graph abstraction using node simplification involves transforming G into a new graph $G' = (V', E')$. The transformation can be described by a mapping function ϕ such that:

$$\phi : V \to V'$$

$$E' = \{(\phi(u), \phi(v)) \mid (u, v) \in E,\ \phi(u) \neq \phi(v)\}$$

Definition 6.3.2. Abstracted Control Flow Graph ACFG.
Given a CFG of HybridSE $G = (V, E)$, where V = $\{< \mathit{Address}, \mathit{Instruction} >\}$, with

$$\mathit{Instruction} = (\text{Opcode}, \text{Operands}, \text{Address})$$

[5] `https://nvd.nist.gov/vuln/detail/CVE-2014-3153`

We define the ACFG as $G' = (V', E')$, where two transformation function $\phi_1 : V \to V'$ and $\phi_2 : V' \to V''$ are applied.

- ϕ_1 (Abstraction) simplifies a node label from the tuple *(Address, Instruction)* into the *Address_Opcode* format.

- ϕ_2 (Relabeling) relabels the nodes to integers starting from 0.

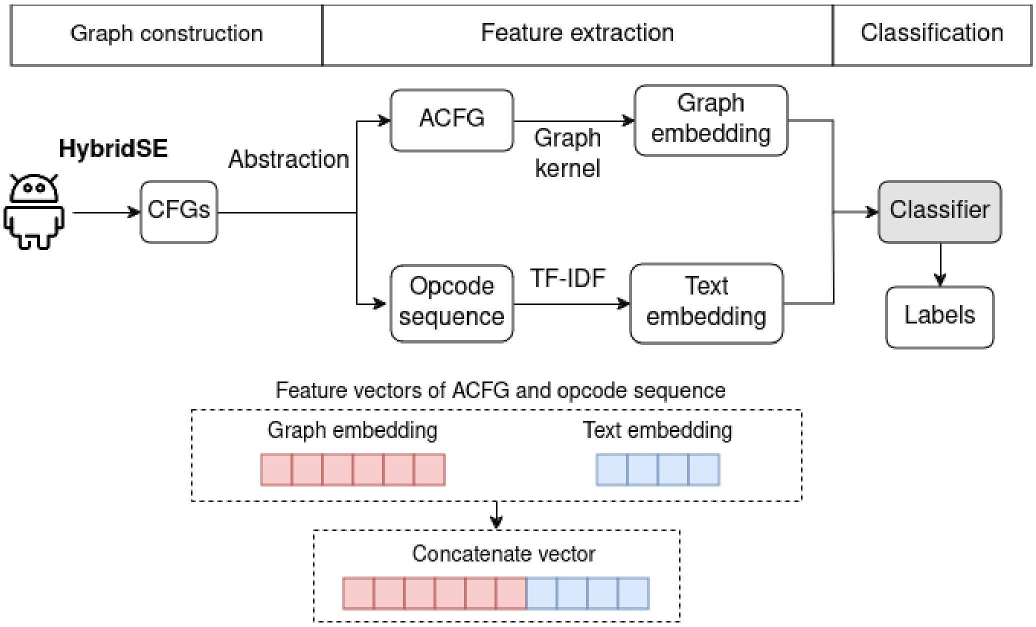


Figure 6.4: Classification workflow

HybridSE's output includes all cross-environment CFGs generated for each entry point of an APK file. To analyze the effects of different APK components, we categorize CFGs into three types: Bytecode graph, Native graph, and Union graph. The Bytecode graph comprises all nodes with Java bytecode instructions from the Android component. Native graphs include nodes with ARM assembly instructions from the Native component. The Union graph is a cross-environment CFG that incorporates both types of instructions.

Graph kernels are methods that measure graph similarity and compute graph embedding vectors for machine learning algorithms. The Weisfeiler-Lehman subgraph kernel [47], for instance, iteratively refines node labels based on the neighborhood of each node to produce a graph feature vector. To represent graph structures as feature vectors for classification and clustering tasks, we use *Graph2Vec*.

In graph analysis, we extract two key pieces of information from graphs: the graph structure itself and textual labels associated with nodes. Graph kernels effectively capture the structural aspects of graphs, yet they often do not account for the textual node information. Specifically within CFGs, node labels represent sequences of operation codes executed, providing crucial insights into the execution flow. These labels offer additional context about how the program unfolds during execution.

6.3.2 Evaluation setup

To evaluate the quality of the graphs generated by HybridSE, we performed graph similarity analysis for two classification tasks: malware family classification and Android packer classification. For each task, we proceed by comparing two criteria: firstly, we assess the structural differences between the CFG of HybridSE and the call graph produced by FlowDroid. This evaluation focuses on their classification capabilities solely based on their respective graph structures (Figure 6.5).

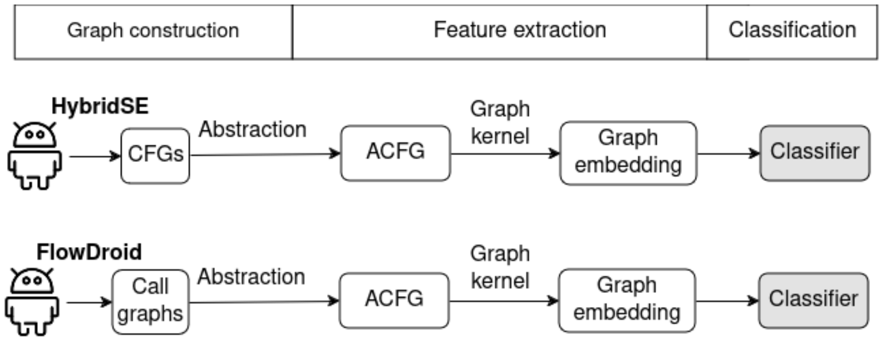


Figure 6.5: Workflow for comparing HybridSE and FlowDroid graphs for classification

Secondly, we analyze how effectively application characteristics are represented using graph components. This includes embedding both the graph structure and node text information. We then benchmark these representations against feature extraction techniques utilized in state-of-the-art malware classifiers (Figure 6.4). The summary of the comparison setup and the dataset used is in Table 6.6, and the evaluation metrics are described in Table 6.5.

We employ the SVM classification provided by *scikit-learn* with default settings. We randomly split the training and testing data for each year in an 80:20 ratio. For graph embedding, we utilize the Weisfeiler-Lehman graph kernel implementation available in *Graph2Vec*[6]. For node label text embedding, we use TF-IDF in *scikit-learn.*

Table 6.5: Metric definition

Metrics	Abbreviation	Definition
True Positive	TP	The Positive label samples are predicted as Positive.
True Negative	TN	The Negative label samples are predicted as Negative.
False Positive	FP	The Negative label samples are predicted as Positive.
False Negative	FN	The Positive label samples are predicted as Negative.
Accuracy	Acc	$Acc = \frac{TP+TN}{TP+FP+TN+FN}$
Precision	P	$P = \frac{TP}{TP+FP}$
Recall	R	$R = \frac{TP}{TP+FN}$
F1-score	F1	$F1 = 2 \times \frac{P \times R}{P+R}$

Table 6.6 shows the malware used for evaluation. HybridSE CFGs were generated from Dataset 1 in Section 6.1, and the AndroZoo dataset includes a subset of 4,000 samples, with 1,000 CFGs randomly selected from each year between 2017 and 2020.

Table 6.6: Summary of comparison experiments

Task	Tool compared	Used features	From dataset	Number of samples
Malware family classification	FlowDroid	Graph embedding	DREBIN AMD	675 671
	Malscan, Drebin	Graph embedding, Label text embedding	AndroZoo DREBIN AMD	4000 675 671
Packer identification	FlowDroid	Graph embedding	PackerGrind	298
	Malscan, Drebin	Graph embedding, Label text embedding	AndroZoo PackerGrind	4000 298

We conduct the experiment in two phases: First, we evaluate which graph structure is better suited for graph embedding by comparing the generated graphs from HybridSE and FlowDroid. Second, we compare HybridSE feature extraction from CFG with state-of-the-art malware classifiers Malscan and Drebin.

- FlowDroid [4]: A static taint analysis tool that performs taint tracking through call graph traversal.

[6] https://karateclub.readthedocs.io/en/latest/

- Malscan [48]: An Android malware detection tool that extracts call graphs and presents malware behaviors by sensitive API calls in the source code. It uses *Androguard*[7] to extract the call graph and identify sensitive APIs.

- Drebin [49]: An Android malware detection tool that extracts numerous features from an APK (e.g., API calls, native library names, entry points, ...) and combines them into a unified vector. Drebin also uses *Androguard* for static feature extraction. Drebin extracts varying numbers of features across different datasets; for example, 49,022 features for Dataset 2019 and 55,856 features for Dataset 2020.

6.3.3 Classification of Android malware family

A malware family refers to a group of malware samples or instances that share common characteristics, such as code structure, behavior, or functionality. Malware families are often classified based on similarities in their source code, propagation methods, or the objectives they aim to achieve. We anticipate that CFGs can reveal the distinctive traits of Android malware.

Labeling dataset

In the malware dataset, DREBIN and AMD already include labels indicating the malware family each sample belongs to, while the AndroZoo labels are obtained from VirusTotal reports. We retrieve all family names reported by different antivirus vendors in AndroZoo and then use majority voting to determine the final label from all the vendors' decisions. Listing 6.1 shows a report from VirusTotal indicating that the family label for the sample is Artemis.

```
{"_id": "022cc5061d2ef8805959085739baa991ec7d148baf8532d104ddb894ce3a6787",
  "_type": "file",
  "last_analysis_results": {
    "Fortinet": {
      "result": "Riskware/PackedTencent!Android"
    },
    "K7GW": {
      "result": "Trojan ( 0054f14b1 )"
    },
    "MAX": {
      "result": "malware (ai score=91)"
```

[7] https://androguard.readthedocs.io/

```
        },
        "McAfee": {
            "result": "Artemis!ABB7534326D3"
        },
        "McAfee-GW-Edition": {
            "result": "Artemis"
        },
        "Trustlook": {
            "result": "Android.PUA.General"
        }
    }}
```

Listing 6.1: A Virus Total report

With the rapid evolution of Android malware, the distribution of popular malware families has changed significantly within two years, with Jiagu[8]—a tracking and packing service in China—emerging prominently. Figure 6.6 shows the distribution of labels retrieved from VirusTotal on AndroZoo samples from 2018 and 2020, within 2-year span.

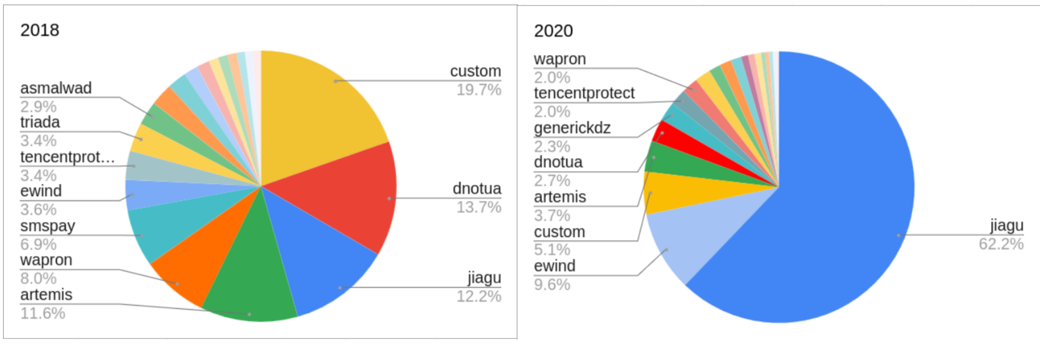


Figure 6.6: Malware label distribution difference between 2018 and 2020

Using identical settings, we assess the classification performance by comparing the classification ability based on graph similarity derived from HybridSE's CFGs and FlowDroid's call graphs.

Comparision with FlowDroid

We employ HybridSE and FlowDroid on the DEBIN and AMD. Note that HybridSE generates CFGs across both bytecode and native code, whereas FlowDroid's call graphs contain only bytecode.

Figure 6.7 and Figure 6.8 illustrate the call graph of FlowDroid and the CFG of HybridSE, respectively, for the same APK file, *native_leak.apk*.

[8] http://dev.360.cn

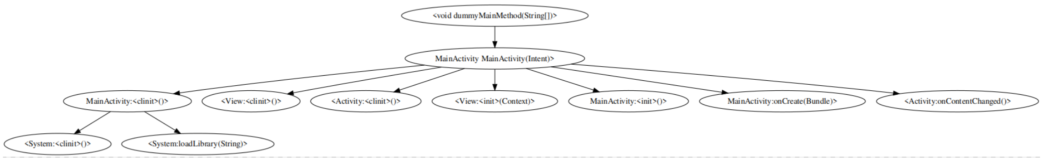

<void dummyMainMethod(String[])>
MainActivity MainActivity(Intent)>
MainActivity:<clinit>()>
<View:<clinit>()>
<Activity:<clinit>()>
<View:<init>(Context)>
MainActivity:<init>()>
MainActivity:onCreate(Bundle)>
<Activity:onContentChanged()>
<System:<clinit>()>
<System:loadLibrary(String)>

Figure 6.7: Structure of *native_leak.apk*'s call graph generated by FlowDroid

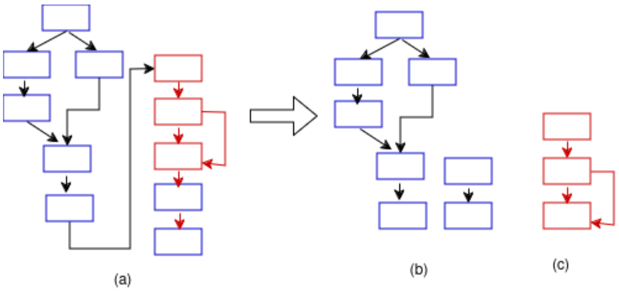


Figure 6.8: Structure of *native_leak.apk*'s CFGs generated by HybridSE
(a) Union CFG, (b) Bytecode CFG, (c) Native CFG

From the graphs generated from HybridSE and FlowDroid, we applied graph kernels to obtain feature vector embeddings and then classified them into malware families using SVM. The result is reported in Table 6.7. The classification by HybridSE's CFG is carried out in three types. The Union CFG involves a cross-environment Control Flow Graph that traverses both bytecode and native code. The Native CFG stage focuses solely on generating a CFG that includes only native code. Finally, the Bytecode CFG is a CFG that encompasses only bytecode. On the other hand, FlowDroid provides a callgraph of Java bytecode instead of CFG.

Table 6.7: Comparision with FlowDroid on malware family classification in DREBIN + AMD dataset

	HybridSE+W-L+SVM			**FlowDroid**+W-L+SVM
	Union CFG (Bytecode+Native)	Native CFG	Bytecode CFG	Bytecode CLG
Accuracy	**0.907**	0.458	0.907	0.5163
Precision	**0.873**	0.209	0.846	0.2666
Recall	**0.907**	0.458	0.907	0.5163
F1-score	**0.884**	0.287	0.871	0.3517

W-L: Weisfeiler-Lehman graph kernel, CFG: Control Flow Graph, CLG: Call Graph

We recorded an accuracy of 90.7% for classifying the DREBIN malware family using native CFG embedding, compared to 51.65% with FlowDroid graph embedding. This result indicates that CFG structures provide a better representation for graph embedding than call graphs.

Comparision with Malscan and Drebin

We evaluate how features extracted from HybridSE's CFG (Figure 6.4) compare with those extracted by other methods from APK files in the task of malware family classification.

In this task, we also report results for three types of graphs in HybridSE: Union, Bytecode, and Native. For each graph type, we extract three levels of information for feature vectors:

- **-G:** Extract only graph kernel embeddings as the feature vector.
- **-O:** Retain node labels, which contain opcodes of instructions as text, and create text embeddings from node label information. The feature vector is the concatenation of graph kernel and node label text embeddings.
- **-C:** Retain class names, which are the entry points of CFG graphs, and encode class name information as text. The feature vector includes information from the graph kernel, node labels, and class names.

Table 6.8: Malware family classification compared with state-of-the-art malware classifier

Dataset		2018		2019		2020		2021		Drebin+AMD	
Metric		Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1
Native	-G	0.533	0.511	0.8333	0.7933	0.7351	0.6825	0.7969	0.7306	0.458	0.287
	-O	0.6635	0.5832	0.8654	0.8589	0.8344	0.8026	0.8750	0.8438	0.458	0.287
	-C	1.000	1.000	1.000	1.000	0.997	0.997	0.893	0.922	0.831	0.773
Byte	-G	0.7404	0.6578	0.8846	0.8361	0.8377	0.7764	0.9219	0.9067	0.907	0.884
	-O	0.8077	0.7412	0.8846	0.8361	0.8609	0.8059	0.9375	0.9235	0.907	0.884
	-C	1.000	1.000	1.000	1.000	0.997	0.997	**0.938**	**0.924**	0.958	0.952
Union	-G	0.7404	0.6606	0.8910	0.8836	0.8411	0.7846	0.9219	0.9077	0.907	0.871
	-O	0.8077	0.7412	0.8910	0.8836	0.8576	0.8027	0.9375	0.9235	0.907	0.871
	-C	**1.000**	**1.000**	**1.000**	**1.000**	0.997	0.997	0.914	0.922	0.958	0.942
Malscan		0.9619	0.9607	0.9158	0.9114	0.9339	0.9076	0.9231	0.9080	**0.978**	**0.973**
Drebin		**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	0.908	0.922	0.903	0.886

-G: graph kernel, -O: graph kernel + opcode, -C: graph kernel + opcode + class name

```
Accuracy: 0.8974358974358975
Precision: 0.8881562881562882
Recall: 0.8974358974358975
F1-score: 0.8854466611819553
                precision    recall  f1-score   support
```

```

       artemis       0.13      1.00      0.24         2
      asmalwad       1.00      1.00      1.00         2
        custom       1.00      0.60      0.75         5
        dnotua       0.57      1.00      0.73        12
         ewind       1.00      1.00      1.00        12
    generickdz       0.00      0.00      0.00         5
         hypay       1.00      1.00      1.00         1
         jiagu       1.00      1.00      1.00       141
         mobby       1.00      1.00      1.00         2
        smspay       1.00      1.00      1.00         2
tencentprotect       0.00      0.00      0.00        10
        triada       0.00      0.00      0.00         2
        wapron       1.00      1.00      1.00         1
         wroba       0.00      0.00      0.00         3

      accuracy                           0.90       200
     macro avg       0.58      0.64      0.58       200
  weighted avg       0.89      0.90      0.89       200
```

Listing 6.2: Detail report of Union-O for 2019

Table 6.3.3 presents the results of HybridSE with different extracted graph features compared to Malscan and Drebin. Compared to Native and Bytecode graphs, the Union graph representation yields the best results. However, using only graph representations, many cases fail to predict the correct label. For instance, in the classification report of Dataset 2019 for Union-O, *'generickdz'*, *'tencent_protect'*, and *'wroba'* are not predicted correctly, likely because HybridSE is unable to produce complete CFGs for these samples. Both Drebin and Union-C extract class names as a feature, indicating that typical malware uses specific naming conventions for functions, which provides a clear indicator for malware classifiers, especially in the period of 2018-2019.

6.3.4 Android packer identification

Labeling dataset

Due to the prevalent issue of plagiarism and repackaging within the Android ecosystem, developers widely embraced Android app packing techniques as an effective safeguard for their applications. Generally, Android packers serve to enhance resilience against static analysis, and dynamic analysis, and deter reverse engineering.

Identifying the used packer is crucial for understanding obfuscation techniques and gaining insights into malware behavior and unpacking methods.

Most importantly, knowing the correct packer enables reverse engineers to apply the appropriate unpacking methods to retrieve the original code for analysis.

A popular tool for packer identification is APKID[9], which utilizes static analysis methods and signature-based methods to identify packers, obfuscators, and anti-analysis measures present within Android application package (APK) files, to retrieve the packer names.

```
[+] APKiD 2.1.5 :: from RedNaga :: rednaga.io
[*] 0E996D263*.apk
[*] 0E996D263*.apk!assets/gdt_plugin/gdtadv2.jar!classes.dex
 |-> anti_vm : Build.FINGERPRINT check, Build.MANUFACTURER
     check, Build.MODEL check, Build.PRODUCT check, possible
     Build.SERIAL check, subscriber ID check
 |-> obfuscator : Obfuscator-LLVM version 9.x, Obfuscator-LLVM
      version unknown (string encryption)
[*] 0E996D263*.apk!assets/libjiagu.so
 — packer : Jiagu
```

Listing 6.3: Example output of APKID

We use APKID to identify the packer name of packed samples. Samples with a "NONE" label, indicating no packer was used, are not considered to be classified and removed from the dataset.

Comparision with FlowDroid

Table 6.9 presents the number of samples for each packer successfully analyzed by HYBRIDSE and FlowDroid. Among them, HYBRIDSE encounters difficulties in generating the CFG for Bangcle and Tencent due to the lack of support for multidex and limited entry point realization.

Table 6.9: Graph generated from PackerGrind dataset

	#Sample	# Partial CFG by HybridSE	# Callgraph by FlowDroid
Baidu	63	63	67
ijiami	71	31	71
Bangcle	47	0	47
qihoo (Jiagu)	39	39	39
alibaba	40	40	40
tencent	34	0	34
Total	298	173	298

[9] https://mas.owasp.org/MASTG/tools/android/MASTG-TOOL-0009/

While the number of analyzed samples by HybridSE is lower compared to FlowDroid, the packer identification results using HybridSE's CFGs are significantly higher (Table 6.10). Current tools like APKID, which utilize database-stored signature matching or rule-based detection, often achieve complete accuracy. However, APKID only offers detection for known packers with predefined rules, necessitating constant updates to its database and detection rules. While APKID can correctly identify Android packers with 100 % accuracy on Dataset 2, currently, the rules for APKID are manually updated with each tool update, making the expansion of rule-based packer detection tools a labor-intensive task. By leveraging HybridSE and graph kernel similarity, we can automate the process for detecting packers.

Classify Android packer on 173 samples that HybridSE's successfully generated, union graph of HybridSE's CFG yields the highest accuracy score at 97.14 % and F1-score at 97.10 %, compared to only 48.89% and 42.21 % achieved by the call graph on FlowDroid.

Table 6.10: Android packer classification on PackerGrind

	HybridSE +W-L+SVM			FlowDroid +W-L+SVM
	Union CFG	Native CFG	Bytecode CFG	Bytecode CLG
Accuracy	**96.67 %**	50.00 %	86.67 %	48.89 %
Precision	**97.11 %**	39.36 %	78.41 %	45.34 %
Recall	**96.67 %**	50.00 %	86.67 %	48.89 %
F1-score	**96.55 %**	41.77 %	81.57 %	42.21 %

W-L: Weisfeiler-Lehman graph kernel, CFG: Control Flow Graph, CLG: Call Graph

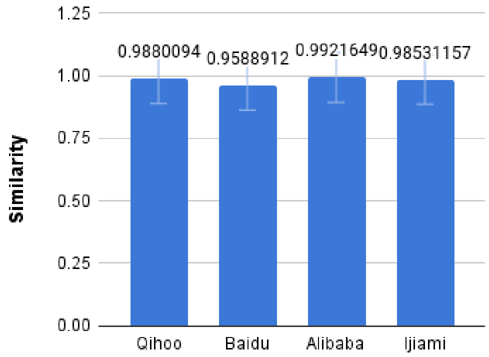


Figure 6.9: Graph similarity of Bytecode CFG by each packer

We observe that Bytecode CFG is better suited for identifying packers than either native code or cross-environment CFG. This is reasonable considering the techniques employed by current packers, which typically begin the application with stub application bytecode. Referring back to Figure 2.3 in Chapter 2, the stub application DEX contains the packer program, which is characteristic for each packer. This observation is supported by Figure 6.9. On the other hand, the native .so library contains both the decrypted stub and the encrypted dex, whose content can vary more across different applications.

Comparision with Malscan and Drebin

We compare the results of Android packer identification across yearly AndroZoo datasets with Malscan and Drebin in Table 6.11. The results suggest that packer identification is relatively straightforward for classifiers, especially when class name information is included, due to the distinct naming conventions used by packers—such as Qihoo's use of *libjiagu.so* for native libraries. Even without class name information, Union-O of HybridSE achieves comparable results to Malscan and Drebin, with Union-O in 2019 and PackerGrind outperforming both Malscan and Drebin. This demonstrates that HybridSE's CFGs effectively capture the characteristics of packer behavior.

Table 6.11: Android packer identification on yearly AndroZoo dataset compared with state-of-the-art malware classifier

Dataset		2018		2019		2020		2021		PackerGrind	
Metric		Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1
Native	-G	0.421	0.250	0.980	0.972	0.923	0.887	0.980	**0.981**	0.500	0.418
	-O	0.667	0.570	0.987	0.981	0.923	0.887	0.980	0.970	0.517	0.431
	-C	1.000	1.000	0.987	0.981	0.923	0.887	0.980	0.970	1.000	1.000
Byte	-G	0.702	0.606	0.953	0.930	0.926	0.890	0.980	0.970	0.867	0.816
	-O	0.702	0.612	0.953	0.930	0.926	0.890	0.980	0.970	0.867	0.816
	-C	1.000	1.000	0.953	0.930	0.926	0.890	0.980	0.970	1.000	1.000
Union	-G	0.702	0.606	0.987	0.981	0.926	0.890	0.980	0.970	0.967	0.965
	-O	0.702	0.612	**0.987**	**0.981**	0.926	0.890	0.980	0.970	**0.967**	**0.965**
	-C	**1.000**	**1.000**	**0.987**	**0.981**	**0.926**	0.890	0.980	0.970	**1.000**	**1.000**
Malscan		0.961	0.966	0.892	0.901	0.926	0.937	**0.989**	0.978	0.815	0.867
Drebin		0.922	0.932	0.919	0.930	**0.930**	**0.950**	0.934	0.956	0.833	0.834

-G: graph kernel, -O: graph kernel + opcode, -C: graph kernel + opcode + class name

The precision of Android malware classification is subject to decline due to evolving malware and packing techniques. While model improvement strategies such as retraining or active learning have been proposed to tackle these challenges, they require substantial human effort for labeling. To examine how model performance deteriorates over time, Table 6.3.4 shows predictions for 2021 samples using models trained on datasets from 2020 and 2019, since the list of the packer labels in AndroZoo datasets for these three years are similar. We aim to track whether changes in the packing strategies of the same packer affect the classification model.

In the 1-year-old and 2-year-old strategies shown in Table 6.3.4, we retain the entire 2021 dataset as the test data and use models trained on data from previous years to make predictions. This simulates a scenario where a new batch of unknown applications needs to be analyzed by the system.

For Drebin, the strategy involves extracting a varying number of features, which prohibit the detection of new malware when using an older model. If new samples are added, retraining the model on the entire dataset is necessary. Malscan achieves the highest accuracy and F1 score when training a model on data from the same year. However, when using a 1-year-old or 2-year-old model, HybridSE's graph structures exhibit a slower rate of decline in accuracy.

Table 6.12: Testing on next year packed sample using the previous dataset

Methods	2021 (same year)		1 year-old model		2 year-old model	
	F1	Acc	F1	Acc	F1	Acc
HybridSE-Union-O	0.980	0.970	0.923	0.941	0.911	0.928
HybridSE-Union-C	0.980	0.970	**0.983**	**0.975**	**0.983**	**0.975**
Malscan	**0.989**	**0.978**	0.925	0.931	0.882	0.898
Drebin	0.934	0.956	-	-	-	-

Conclusion : CFGs generated by HybridSE and analyzed using graph kernels yield comparable results in classification tasks. We conclude that the CFGs produced by HybridSE can effectively characterize Android applications.

6.3.5 Discussion

As packers continuously evolve with new or custom packers emerging, detecting these unknown packers is often challenging using rule-based methods or classification systems based on previous data labels. To address the case where no labels are provided, we investigate whether graph structures can provide insights into the Android application dataset.

We compute the pair-wise cosine similarity of cross-environment CFGs using only graph kernels within the AndroZoo subset described in Table 6.6. The results, shown in Figure 6.10, clearly indicate that some graphs are grouped based on their structural similarity.

Using the k-means clustering algorithm with values of k ranging from 2 to 10, we found that $k = 9$ resulted in the lowest Davies-Bouldin Index (DBI) score[10], indicating better clustering performance. After dividing the dataset

[10] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.davies_bouldin_score.html

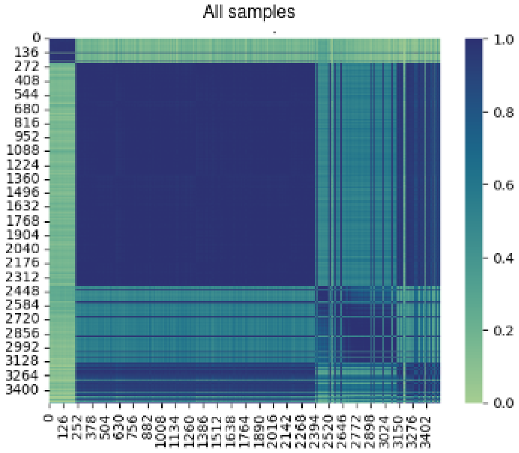


Figure 6.10: Heatmap shown the pair-wise similarity calculated from all packed samples of AndroZoo

into 9 clusters, the average similarity of graphs within each cluster is 0.9868.

Cluster	Packer categorized
1	’jiagu_qihoo’: 1965, ’tencent’: 276, ’dexprotector’: 35, ’bangcle’: 35, ’upx’: 17, ’joker’: 13, ’baidu’: 8, ’apkencryptor’: 7, ’yidun’: 4, ’multidex’: 3, ’apkprotect’: 2, ’secneo’: 2, ’crazydog’: 1
2	’jiagu_qihoo’: 457
3	’joker’: 29
4	’jiagu_qihoo’: 237
5	’unicom’: 4
6	’upx’: 15
7	’jiagu_qihoo’: 155, ’Baidu’: 14, ’upx’: 14, ’multidex’: 2, ’dexprotector’: 2, ’unicom’: 1
8	’jiagu_qihoo’: 224
9	’unicom’: 4

Table 6.13: Packer samples that are categorized into each cluster

Chapter 7

Evaluation on Taint analysis of HybridSE

To take advantage of the capabilities of cross-environment analysis offered by HybridSE, we've integrated a taint module atop the DSE engine. This section evaluates the advantages of employing DSE for taint analysis over static tools. We subsequently applied our taint module to detect data leakage observed in Android malware, focusing on information leakage that occurs through both bytecode and native code.

7.1 Experiment datasets

Dataset1: DroidBench[1]. DroidBench is a popular Android taint analysis benchmark, and NativeFlowBench is its subset that contains native code. We currently focus on part (A) inter-language dataflow in NativeFlowBench and Array and Lists, Reflection in DroidBench. For this benchmark, we use the list of sources and sinks given by Argus-SAF.
Dataset2: Android malware dataset from Chapter 6. From datasets like DREBIN, AMD, and AndroZoo, we successfully generated unprotected CFGs (without multidex and packing) for a total of 3,266 samples. We anticipate that these CFGs accurately represent the payload of the Android malware, and we employ HybridSE to identify potential information leakage vulnerabilities.

[1] github.com/arguslab/NativeFlowBench

7.2 Comparision with static analysis tools

7.2.1 Detecting cross-environment data leak

Table 7.1 demonstrates that HybridSE accurately detects the correct result for 16 out of 18 benchmark items. Most results align with those of Argus-SAF, except for the cases of *native_noleak_array* and *native_multiple_interactions*. Both HybridSE and Argus-SAF produce false positives for *native_complexdata_stringop*.

native_noleak_array. HybridSE produced a correct result for this test case, which was specifically designed to address the issue of false positive in taint analysis tools. HybridSE manages taint tags for array elements individually, instead of over-tainting like Argus-SAF or FlowDroid, which improves precision when handling array elements individually.

native_multiple_interactions. HybridSE fails to provide accurate results for this test, since currently corana checks each process sequentially. As a result, we do not consider cases where processes are concurrently running and interacting with each other.

native_complexdata_stringop. In this scenario, HybridSE experienced a false positive, similar to Argus-SAF. Unlike arrays, which HybridSE can manage element-wise, deciding on how complex data is transferred from one environment to another poses challenges due to the varied nature of complex data structures. In such cases, we track the whole complex data as a taint object, rather than on an element-wise basis as with arrays.

7.2.2 Detecting data leaks involving arrays and Java reflection

In tasks involving array manipulation and Java reflection, the DSE engine facilitates the handling of data that requires runtime resolution, such as dynamically invoked classes via Java reflection, with ease. Both FlowDroid and Argus-SAF face challenges due to their static nature.

An example of Java reflection used to hide source API is shown in Listing 7.1. In this example, the IMEI is saved in the `foo()` method of the `ConcreteClass` on line 9. When the `onCreate()` function of `MainActivity` starts on line 3, it registers the `ConcreteClass` using reflection instead of directly invoking `ConcreteClass.foo()`. Later, on line 6, `bc.foo()` is called, which, during real execution, invokes `ConcreteClass.foo()` and returns the IMEI. This, in turn, leads to the IMEI being leaked through `sendTextMessage`.

Table 7.1: NativeDroidBench benchmark result

NativeFlowBench - Inter-language dataflow					
	APK file	Ground	Hybrid	Flow	Argus
		truth	-SE	-Droid	-SAF
1	n_source	O	O	X	O
2	n_nosource	X	X	X	X
3	n_source_clean	X	X	O	X
4	n_leak	O	O	X	O
5	n_leak_dynamic_reg	O	O	X	O
6	n_dynamic_reg_multiple	O	O	X	O
7	n_noleak	X	X	X	X
8	*n_noleak_array*	X	**X**	X	**O**
9	n_leak_array	O	O	X	O
10	n_method_overloading	X	X	X	X
11	*n_multiple_interactions*	O	**X**	X	**O**
12	n_multiple_libraries	O	O	X	O
13	n_complexdata	O	O	X	O
14	*n_complexdata_stringop*	X	**O**	X	**O**
15	n_leak_heap_modify	O	O	X	O
16	n_set_field_fm_native	OO	OO	XX	OO
17	n_set_field_fm_arg	OO	OO	XX	OO
18	n_set_field_fm_arg_field	OO	OO	XX	OO
Arrays and Lists					
19	ArrayAccess1	X	X	O	O
20	ArrayAccess2	X	X	O	O
21	ArrayCopy1	O	O	O	X
22	ArrayToString1	O	O	X	X
23	HashMapAccess1	X	X	O	O
24	ListAccess1	X	X	O	O
25	MultidimensionalArray1	O	O	X	X
Java reflection					
26	Reflection1	O	O	X	X
27	Reflection2	O	O	X	X
28	Reflection3	O	O	X	X
29	Reflection4	O	O	X	X
	Accuracy		93.10 %	20.68 %	55.17 %

O^n = Contain n data leaks, X^n = No n data leak, prefix native_ in APK file name is shortened as n_ due to space limitation

Static analysis tools face difficulties in resolving `bc.foo()` to `ConcreteClass.foo()` since the reflection information is only available at runtime. This causes static taint analysis tools like FlowDroid and Argus-SAF to miss the data leak at line 6.

```
1  public class MainActivity extends Activity {
2   protected void onCreate() {
3     BaseClass bc = (BaseClass)
          Class.forName("de.ecspride.ConcreteClass").newInstance();
4     // Registering 'ConcreteClass' using Java reflection
5     SmsManager sms = SmsManager.getDefault();
6     sms.sendTextMessage("+49 1234", null, bc.foo());
7  }
8  public class ConcreteClass extends BaseClass {
9    public String foo() {
10    TelephonyManager tM = getSystemService("phone");
11    imei = tM.getDeviceId();
12    return imei;
13 }}
```

Listing 7.1: Reflection is used to hide source API in JavaRelection1.apk in Dataset 1

Java reflection is handled naturally in SPF [17], and arrays are managed element-wise. This allows HybridSE to accurately resolve reflected calls and track taint tags within array elements, resulting in more precise leak detection.

7.3 Data leakage observed from malware dataset

From 3,266 CFGs from Dataset 2, HybridSE identified 139 apps containing leaks. Specifically, these comprised 24 from DREBIN, 47 from AMD, and 68 from AndroZoo.

To verify the validity of HybridSE, we manually checked the results obtained. Below, we summarize the typical observations we made.

First, we noticed that the most frequently utilized sources include APIs that gather device information, such as `BUILD.model`, `getDeviceId()`, and `getLine1Number()`. Meanwhile, the predominant sinks observed are HttpPost-related APIs or print statements used to publish sensitive information (see TABLE 4.1)

On the DREBIN and AMD datasets, HybridSE detects several common data leak scenarios within multiple malware families, e.g., *DroidKungFu*, *Lotoor*, *Dowgin*, and *Towelroot*.

Figure 7.1 illustrates the four main data leakage scenarios identified using HybridSE.

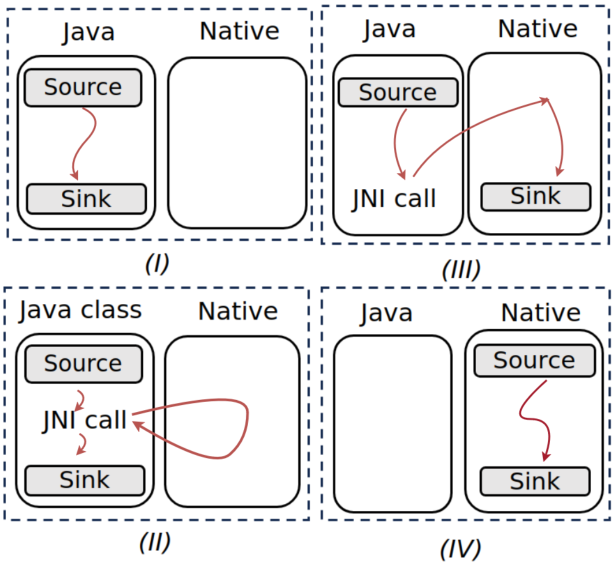


Figure 7.1: Data leakage methods
(I) in *DroidKungFu* and *SimpleLocker*, *Dowgin*, (II) in *Lotoor*, (III) in a *DroidKungFu* variant and (IV) in *Towelroot*

1. *Dataleaks contain only in Java layer (Fig. 7.1-I).*

   (a) *Data is published via HTTP connections.*
   The *DroidKungFu* and *SimpleLocker* families record device information via `getDeviceID()`, `getLine1Number()`, or `BUILD.-VERSION` in the Java layer. Then the information is added to a `HTTPPost.setEntity()` to send the information via `HttpPost.-openConnection()`.

   (b) *Leakage of device information through logging and writing.*
   Dowgin collects IMEI, network operator, and Build.MODEL, and displays in the log. Other data-flow recorded from a different *Dowgin* variation passes the IMEI to a JSONObject, which is later written to a file using `fileOutputStream.write()`.

2. *Dataleak that traverses through native code (Fig. 7.1-II).*
   HybridSE found in *Lotoor* family the data leakage runs through the native code. *Lotoor* retrieves multiple pieces of information from the Java layer and then concatenates all this information into a single

string. It is passed to the native function `cs()`, which includes MD5 hash operations, and later published using `HttpPost` like *SimpleLocker*.

Furthermore, we noticed that *Lotoor* integrates its C&C server URL within native code. When executed, it fetches the URL data using a native function named `uu()`.

3. *Information is transferred from Java to native and published via kernel syscall, i.e. across Java and native code layers (Fig. 7.1-III).*
   A variant of *DroidKungFu* obtains the package name and device's IMEI in the Java layer and passes them to the native method DataInit. IMEI is translated to Characters via the JNI function `getCharFromUTFString()`, and is shown to the console using the `sprintf()` syscall.

4. *Only in native layer (See Fig. 7.1-IV).*
   Towelroot retrieves the process ID using the `getpid()` function and accesses the kernel version by calling `fopen("/proc/version", "rb")`. After obtaining the information, the kernel number is exposed by `__android_log_print()`.

7.4 Discussion

HybridSE performs well compared to other static analysis tools on samples specifically designed for taint analysis. In particular, HybridSE is more precise, and as a result, it avoids false alarms, as exemplified in the *'native_noleak_array'* case mentioned above.

As mentioned in Section 4.3.3, for the native to Java bytecode direction, HybridSE currently employs an over-approximation method to taint the outcomes of JNI callbacks. In the future, we hope to support more features of Android such as inter-component communication and native activity by additional implementation, particularly in mapping specific Java functions invoked within native code.

Chapter 8

Related works

8.1 Symbolic execution for binary code and bytecode

Symbolic execution (SE) [21] is a classical method in software engineering, aiming for the test data generation for the control flow coverage. There are lots of tools for high-level programming languages, such as C/C++ and Java are developed (e.g., KLEE [23] and SPF [17]). Recently, the tools for the symbolic execution of binary code have gradually increased, such as MAYHEM [26], KLEE-MC [27], S^2E [29], ANGR [18], BINSEC [30] and BE-PUM [31].

SE tools for binary code

Most SE tools for binary use existing disassemblers or binary lifters to translate binary code to an intermediate assembly language (IAL), such as LLVM in KLEE-MC, VEX in angr, and BAP in MAYHEM. This approach ensures the symbolic execution tools can analyze binaries of multiple architectures (e.g., x86-64, x86, ARM, MIPS) without preparing execution engines for individual architectures. However, this method does not perform well in the presence of obfuscated code, such as indirect jumps, self-modifying code, and overlapping instructions.

To overcome this limitation, some works have directly interpreted binary as a step-wise disassembler. This method requires a huge effort to implement the binary emulator, which requires defining the formal semantics of each instruction set. Therefore, a method to automatically extract the formal semantics of binary instructions is desired. We have tried for x86 as an extension of BE-PUM[37], ARM as CORANA[20], and MIPS as SyMIPS[38], respectively.

Binary code, including malware, often uses API functions (and/or system functions prepared in OS). Based on the instruction-level DSE tools (which work only in the uniform context), we need to extend DSE to handle external function calls, which are executed in different contexts. There are three approaches.

- KLEE-MC abstracts the environment as a model [23]. However, this is quite a rough approximation and rarely achieves enough accuracy.
- MAYHEM [26] and angr [18] fuse the concrete and the symbolic executions by interleaving the GDB debugger and their symbolic engine.
- BE-PUM [31] prepare the *API Stub* to execute a system call in real Windows OS to obtain an exact snapshot of the environment update.

We use the last approach for CORANA/API [19].

SE for Android/Java bytecode

For Java and Android applications, there are several extensions of Java PathFinder (JPF) [44] that target Android apps. For instance, jpf-mobile [40] uses jpf-nhandler [16] to concretize and run the native code in Host JVM.

SPF [17] and SynthesisSE [13] are SE tools built on JPF and JDART, respectively, which reduce all calls (including the native code) as concrete execution. SPF requires manual modeling of native components, the latter resolves all callees by the concrete execution.

Currently, the only DSE tool that supports analysis of native code in Android applications is an experiment version of ANGR[1]. They leverage both Java/DEX bytecode and native code to SootIR. However, the intermediate code translation shares the weakness for the obfuscation.

8.2 Android taint tools and cross-language analysis

Static analysis has been used widely to assess Android application's security such as detecting sensitive data leaks or checking malicious behavior. There is a large body of work on Android taint analysis, both dynamic and static [4, 50, 51, 10, 52, 5, 53, 12]. Static taint analysis techniques [4, 51, 52, 5] consider all possible paths that data can flow through without running the

[1] https://docs.angr.io/advanced-topics/java_support

apps. FlowDroid [4] employs CHA (class hierarchy analysis) and a flow and context-sensitive IFDS algorithm to perform taint detection. It avoids the handling of native method invocation and implements a thorough model for native method calls. IccTA [52] extend Flowdroid to handle inter-connection components. Amandroid [5] is another flow and context-sensitive dataflow analysis framework. It creates an environment model for every Android component and employs a component-based analysis algorithm. Like FlowDroid, Amandroid also does not take into account native code. The only exception is JN-SAF, which is extended based on Amandroid and includes extensive methods for managing native method calls and inter-language data flows. All mentioned tools encounter the challenges of the statical approach including Java reflection and dynamic class loading in the Java environment.

Dynamic taint analysis is a practical approach that allows access to runtime information. TaintDroid [50] is arguably the pioneering dynamic taint analysis that traces information flows on the Dalvik Virtual Machine. TaintART [10] adapts dynamic taint analysis for new ART (Ahead of time) runtime. Vialin [12] also proposed an optimized dynamic approach for runtime performance and efficiency to track taint flows on Dalvik bytecode. These tools suffer from the drawbacks of dynamic tools, including the inability to reason about behaviors not activated at runtime (by anti-debugging or VM awareness) and execution runtime overhead.

To evade the disadvantages of current static and dynamic approaches, dynamic taint analysis based on forward symbolic execution has been proposed [54]. The two analyses are used in conjunction to guarantee that tainted data is in actual feasible paths.

Cross-language analysis for Android APK file. Most Android static analysis tools (FlowDroid, Amadroid, DroidSafe, IccTA) avoid handling native methods and focus only on bytecode. Native methods are often modeled or treated as black boxes when performing taint analysis, i.e., call arguments and return values become tainted if a parameter is tainted. However, there has been more and more attention on native code analysis due to the introduction of new vulnerabilities and security issues previously overlooked by Android-only analysis tools. NDroid [55], NativeGuard [56], and TaintArt [10] use dynamic analysis to track information flowing on the bytecode and native sides. NDroid uses TaintDroit to track information at the point of transferring to a native call, without actually tracking data within native code. NativeGuard uses a sandbox to isolate native libraries from other components in Android applications. TaintArt compiles the whole Android application to ART and then performs taint analysis on binary code.

JN-SAF [5] and JuCify [9] enhance the capabilities of static Android analysis tools (Amandroid and FlowDroid, respectively) by integrating them with a binary symbolic execution tool (ANGR) to conduct cross-language analysis. Both tools translate native code into Jimple and apply taint analysis on the Jimple representation of both bytecode and native code. JN-SAF conducts separate analyses on bytecode and native code, later merging the outputs. In contrast, JuCity merges the call graph of bytecode and native code into a unified model before performing taint analysis.

Both tools follow static analysis and inherit path explosion issues from ANGR.

Chapter 9

Conclusions

Malware and spyware on Android devices are major concerns. For instance, despite Android 13's security policies, some malware has circumvented these protections to exfiltrate user interactions, capture audio with the device's microphone, and track the device's location.

As system architectures become more complex, it is essential to develop tools for reverse engineering applications and analyze them in this ongoing arms race against threat actors.

This thesis presents HybridSE, a cross-environment dynamic symbolic execution tool equipped with a taint analysis module, for analyzing Android/apk files on ARM. HybridSE seamlessly combines symbolic execution combining SPF on Java bytecode and corana/api on ARM 32bits instruction set and generates only CFGs consisting of feasible paths.

However, HybridSE encounters difficulties with highly obfuscated applications, like multi-dex and packed samples. Currently, HybridSE is unable to handle several features, including inter-component communication. Addressing these issues will be crucial for improving HybridSE in the future.

Several updates are necessary to address the current limitations. First, multi-dex handling should be improved by upgrading dex2jar from version 0.9.5 to 2.x. For dynamic loading in SPF, execution can either be initiated via a virtual machine until the loading process is complete, or manual stubs can be implemented for the loading functions. Additionally, the realization of entry points for inter-component communication should be refined. By leveraging Control Flow Graph structures, malware analysis tasks such as classification, vulnerability detection, and clone detection can be more effectively supported.

Nevertheless, we successfully applied HybridSE to approximately 10,000 applications and demonstrated its ability to detect data leakages with fewer false positive alarms than other tools. A final note on false alarms: Given the vast number of samples analyzed, false alarms significantly slow down the work of security experts. Thus, addressing the false alarm rate impacting system correction is a crucial research question.

Bibliography

[1] L. Xue, H. Zhou, X. Luo, L. Yu, D. Wu, Y. Zhou, and X. Ma, "Packergrind: An adaptive unpacking system for android apps," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 551–570, 2022.

[2] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman, "50 ways to leak your data: An exploration of apps' circumvention of the android permissions system," in *28th USENIX Security Symposium (USENIX Security 19).* Santa Clara, CA: USENIX Association, Aug. 2019, pp. 603–620.

[3] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in *Network and Distributed System Security Symposium*, 2015.

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

[5] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," vol. 21, no. 3, 2018, pp. 1–32.

[6] C. Li, X. Chen, R. Sun, M. Xue, S. Wen, M. E. Ahmed, S. Camtepe, and Y. Xiang, "Cross-language android permission specification," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. ACM, 2022, p. 772–783.

[7] C.-A. Staicu, S. Rahaman, Á. Kiss, and M. Backes, "Bilingual problems: Studying the security risks incurred by native extensions in scripting languages," in *32nd USENIX Security Symposium (USENIX Security 23).* Anaheim, CA: USENIX Association, Aug. 2023, pp.

6133–6150. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/staicu

[8] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "JN-SAF: precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code," in *CCS 2018*, 2018, pp. 1137–1150.

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

[10] M. Sun, T. Wei, and J. C. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. Association for Computing Machinery, 2016, p. 331–342.

[11] P. Graux, J.-F. Lalande, V. Viet Triem Tong, and P. Wilke, "Oats'inside: Retrieving object behaviors from native-based obfuscated android applications," *Digital Threats: Research and Practice*, vol. 4, no. 2, 2023.

[12] K. Ahmed, Y. Wang, M. Lis, and J. Rubin, "Vialin: Path-aware dynamic taint analysis for android," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023, 2023, p. 1598–1610.

[13] X. Gao, S. H. Tan, Z. Dong, and A. Roychoudhury, "Android testing via synthetic symbolic execution," ser. ASE 2018, 2018, p. 419–429.

[14] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2007, pp. 421–430.

[15] X. Sun, X. Chen, L. Li, H. Cai, J. Grundy, J. Samhi, T. Bissyandé, and J. Klein, "Demystifying hidden sensitive operations in android apps," vol. 32, no. 2, mar 2023.

[16] N. Shafiei and F. van Breugel, "Automatic handling of native methods in java pathfinder," in *SPIN*, 2014, pp. 97–100.

[17] C. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic pathfinder: Integrating symbolic execution with model checking for java bytecode analysis," *ASE*, vol. 20, pp. 391–425, 2013.

[18] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *SP*, 2016, pp. 138–157.

[19] A. T. V. Nguyen and M. Ogawa, "Automatic stub generation for dynamic symbolic execution of arm binary," ser. SoICT '22. ACM, 2022, p. 352–359.

[20] A. Vu and M. Ogawa, "Formal semantics extraction from natural language specifications for arm," in *FM*, ser. LNCS, vol. 11800, 2019, pp. 465–483.

[21] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, p. 385–394, 1976.

[22] H. Attiya, G. Ramalingam, and N. Rinetzky, "Sequential verification of serializability," *SIGPLAN Not.*, vol. 45, no. 1, p. 31–42, Jan. 2010.

[23] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008.

[24] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *ESEC/FSE-13.* New York, NY, USA: Association for Computing Machinery, 2005, p. 263–272.

[25] A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps, "Directed proof generation for machine code," 07 2010, pp. 288–305.

[26] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *SP*, 2012, pp. 380–394.

[27] A. Romano, "Methods for binary symbolic execution," PhD Dissertation, Stanford University, 2014.

[28] G. Bonfante, J. M. Fernandez, J. Marion, B. Rouxel, F. Sabatier, and A. Thierry, “Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions,” in *CCS*, 2015, pp. 745–756.

[29] V. Chipounov, V. Kuznetsov, and G. Candea, “S2e: A platform for in-vivo multi-path analysis of software systems,” *SIGARCH Comput. Archit. News*, no. 1, p. 265–278, 2011.

[30] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion, “BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis,” in *SANER*, 2016, pp. 653–656.

[31] N. M. Hai, M. Ogawa, and Q. T. Tho, “Obfuscation code localization based on cfg generation of malware,” in *FPS*, ser. LNCS, vol. 9482, 2015, pp. 229–247.

[32] G. Xu and A. Rountev, “Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis,” in *International Symposium on Software Testing and Analysis, ISSTA 2008*. ACM, 2008, pp. 225–236.

[33] X. Li and M. Ogawa, “Stacking-based context-sensitive points-to analysis for java,” in *Hardware and Software: Verification and Testing - 5th International Haifa Verification Conference, HVC 2009*, ser. LNCS, vol. 6405. Springer, 2009, pp. 133–149.

[34] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, “Automatically identifying trigger-based behavior in malware,” in *Book chapter in Botnet Analysis and Defense, Editors Wenke Lee*, 2008, pp. 65–88.

[35] M. Nguyen, M. Ogawa, and T. Quan, “Packer identification based on metadata signature,” in *The 7th Software Security, Protection, and Reverse Engineering Workshop (SSPREW-7), ACM*, 2017.

[36] J. Salwan, S. Bardin, and M. Potet, “Symbolic deobfuscation: From virtualized code back to the original,” in *DIMVA*, vol. 10885. Springer, 2018, pp. 372–392.

[37] N. L. H. Yen, “Automatic extraction of x86 formal semantics from its natural language description,” March, 2018.

[38] Q. T. Trac and M. Ogawa, "Formal semantics extraction from MIPS instruction manual," in *FTSCS*. Springer, 2019, pp. 133–140.

[39] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, p. 1–5, nov 2012.

[40] A. Kohan, M. Yamamoto, C. Artho, Y. Yamagata, L. Ma, M. Hagiya, and Y. Tanabe, "Java pathfinder on android devices," *SIGSOFT Softw. Eng. Notes*, vol. 41, no. 6, p. 1–5, 2017.

[41] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. X. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection*, ser. ADIS, 2008, vol. 36, pp. 65–88.

[42] L. Vinh, "Automatic stub generation from natural language description," August, 2016.

[43] P. Kesseli, "Semantic refactorings," PhD Dissertation, University of Oxford, 2017.

[44] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *ASE*, vol. 10, no. 2, p. 203–232, 2003.

[45] L. Luo, E. Bodden, and J. Späth, "A qualitative analysis of android taint-analysis results," *ASE*, pp. 102–114, 2019.

[46] B. Li, Y. Zhang, J. Li, W. Yang, and D. Gu, "Appspear: Automating the hidden-code extraction and reassembling of packed android malware," *Journal of Systems and Software*, vol. 140, pp. 3–16, 02 2018.

[47] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," *CoRR*, vol. abs/1707.05005, 2017.

[48] Y. Wu, X. Li, D. Zou, W. Yang, X. Zhang, and H. Jin, "Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis," in *ASE'19*, 2019, pp. 139–150.

[49] R. Kumar, Z. Xiaosong, R. U. Khan, J. Kumar, and I. Ahad, "Effective and explainable detection of android malware based on machine learning algorithms," ser. ICCAI '18, 2018.

[50] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones,” in *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, 2014, pp. 1–29.

[51] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, “Information flow analysis of android applications in droidsafe.” in *NDSS*, vol. 15, no. 201, 2015, p. 110.

[52] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, “Iccta: Detecting inter-component privacy leaks in android apps,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 280–291.

[53] C. Wang, R. Ko, Y. Zhang, Y. Yang, and Z. Lin, “Taintmini: Detecting flow of sensitive data in mini-programs with static taint analysis,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 932–944.

[54] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 317–331.

[55] C. Qian, X. Luo, Y. Shao, and A. Chan, “On tracking information flows through jni in android applications,” in *DSN*, 2014, pp. 180–191.

[56] M. Sun and G. Tan, “Nativeguard: protecting android applications from third-party native libraries,” in *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, ser. WiSec ’14, 2014, p. 165–176.