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 are 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.

The graphs generated by HybridSE are subsequently utilized for graph similarity in two tasks: malware family classification and Android packer classification. In both tasks, the graphs generated by HybridSE demonstrate high accuracy, achieving an F1 score of 92.67% for the malware classification task and 97.10% for the Android packer classification task.

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

Contents

Abstract

1 Introduction **1**
1.1 Problem statement 1
1.1.1 Cross-environment nature of Android/APK files 1
1.1.2 Security risks posed by native code 2
1.1.3 Why DSE is needed? 4
1.2 Contribution 5
1.3 Outline 7

2 Android framework and Android obfuscation **9**
2.1 Android framework 9
2.2 Obfuscation techniques 11

3 DSE over heterogeneous environments **14**
3.1 Symbolic Execution for instruction sets 14
3.2 DSE implementations in binary code 16
3.3 DSE for cross-environments 18
3.3.1 Calling convention between different environments . . . 18
3.3.2 Handling black box callees 19
3.3.3 Handling whitebox callees 21
3.4 Discussion 22

4 Description of HybridSE **24**
4.1 HybridSE's components 24
4.1.1 DSE for Java Bytecode: Symbolic PathFinder 24
4.1.2 DSE for ARM instruction: CORANA 25
4.1.3 ARM-Linux Kernel call: CORANA/API 26
4.1.4 Java-ARM communication: HybridSE 26
4.2 HybridSE architecture 27
4.2.1 Preprocessing 28

4.2.2 Construction of Cross-environment Control Flow Graph 29
4.2.3 Cross-environment communicator 30
4.3 Taint analysis module 31
4.3.1 How HybridSE detects data leakage? 32
4.3.2 Taint analysis scenarios 34
4.3.3 Cross-environment taint propagation 35
4.4 Discussion 37

5 Evaluation on CFG generation of HybridSE 38
5.1 Experiment datasets 38
5.2 Cross-environment CFG evaluation 39
5.2.1 Native code and obfuscation usage in Android/APK . . 39
5.2.2 HybridSE performance when analyzing cross-environment Android applications 42
5.2.3 What can be shown by HybridSE's CFG? 44
5.3 Graph similarity among CFGs 45
5.3.1 Preliminary classification of Android malware family . 45
5.3.2 Preliminary trial on Android packer identification . . . 46

6 Evaluation on Taint analysis of HybridSE 50
6.1 Experiment datasets 50
6.2 Comparision with static analysis tools 51
6.2.1 Detecting cross-environment data leak 51
6.2.2 Detecting data leaks involving arrays and Java reflection 51
6.3 Data leakage observed from malware dataset 53
6.4 Discussion 55

7 Related works 56
7.1 Symbolic execution for binary code and bytecode 56
7.2 Android taint tools and cross-language analysis 57

8 Conclusions 60

List of Figures

1.1 Simplified Towelroot callgraph with blue node in bytecode and red node in native code. 2
1.2 Control flow graph for *Towelroot* in Listing1.1 4
1.3 Thesis construction 8

2.1 Structure of an APK file 10
2.2 Java Native Interface 10
2.3 DEX encryption mechanism on APKProtect 1 - Packing, 2 - Executing at runtime, 3 - Decrypting, 4 - Unpacking 12

3.1 Environment model 16
3.2 Example of an external call in ARM environment 18
3.3 Call to a black-box platform 20
3.4 Call to a white-box platform 22

4.1 Symbolic PathFinder 25
4.2 Call handling for Android on ARM compared to Windows x86 27
4.3 HybridSE architecture 27
4.4 Example of a inter-environment translation between SPF and corana/api, back and forth 28
4.5 Preprocessing in HybridSE 29
4.6 An overview of HybridSE system 31
4.7 How environment is transfer in Example 4.3.1 33
4.8 Data leakage scenario of Listing 4.3 35

5.1 Distribution on Android native code usages 40
5.2 Distribution of failures and obfuscation 41
5.3 Average CFG size for yearly subsets in the AndroZoo dataset 43
5.4 Deadcode detected in the native function of **towelroot.apk** . 44
5.5 Graph similarity by each packer 48

6.1 Data leakage methods 54

List of Tables

3.1 Java and ARM data type comparison 21

4.1 Captured sources and sinks from detected data leaks 35

5.1 Native code usage over the years 39
5.2 Result on CFG types generated by HybridSE 42
5.3 Relation between graph size and generation time 42
5.4 Average node and edge counts from HybridSE analyzing AndroZoo malware with native code 43
5.5 Graph produced by HybridSE and FlowDroid for various malware families in DREBIN and AMD 46
5.6 Android family classification on DREBIN dataset 46
5.7 Graph generated from PackerGrind dataset 47
5.8 Android packer classification on PackerGrind 48

6.1 NativeDroidBench benchmark result 52

Chapter 1

Introduction

1.1 Problem statement

Android 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 IMEI numbers and personal data, including emails and contact lists. As a result, data compromise is a significant concern in Android security [1]. A substantial amount of research has focused on detecting Android data leaks and malicious behavior using both dynamic [2] and static [3, 4] 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 [5]. Native code can introduce safety and security issues that may be missed by higher-level language analyses [6]. 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

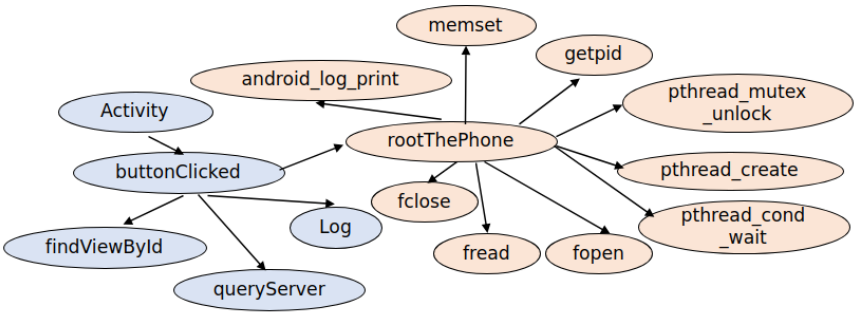

Figure 1.1: Simplified Towelroot callgraph with blue node in bytecode and red node in native code.

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 to 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.

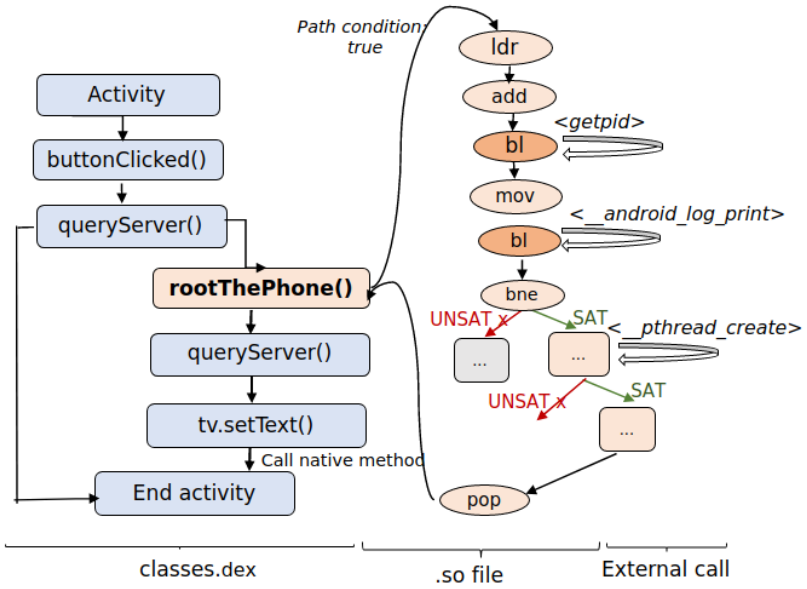

Figure 1.2: Control flow graph for *Towelroot* in Listing1.1

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 [7], JuCify [8], TaintArt [9] and OATs'inside [10]. Dynamic taint analysis tools such as TaintArt [9] and ViaLin [11] 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 [12] and JuCify [8] 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 [13, 14].

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 [15, 16, 12] 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 [17], 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 [16] and CORANA/API [18]. Furthermore, we have enhanced its functionality by adding a taint analysis module.

1.2 Contribution

This thesis 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. This functionality enables HybridSE to generate cross-environment control

[1] figshare.com/s/45b91d138c44e2e55ddd

flow graphs (CFGs) for applications, which is critical for comprehensive analysis.

A primary challenge is managing the calling conventions and ensuring accurate detection of datatype information during pointer value transfers. For visible components, such as native code included within the APK file, we classify these as white-box callees, with path conditions linked by conjunctions. Conversely, for black box callees, we distinguish between eager and lazy concretization of symbolic values, preferring the latter approach to maintain path conditions as thoroughly as possible.

- 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.

- 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 demonstrate high accuracy, achieving an F1 score of 92.67% for the malware classification task and 97.10% for the Android packer classification task.
  - 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 over-approximation. 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 presents the design 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.

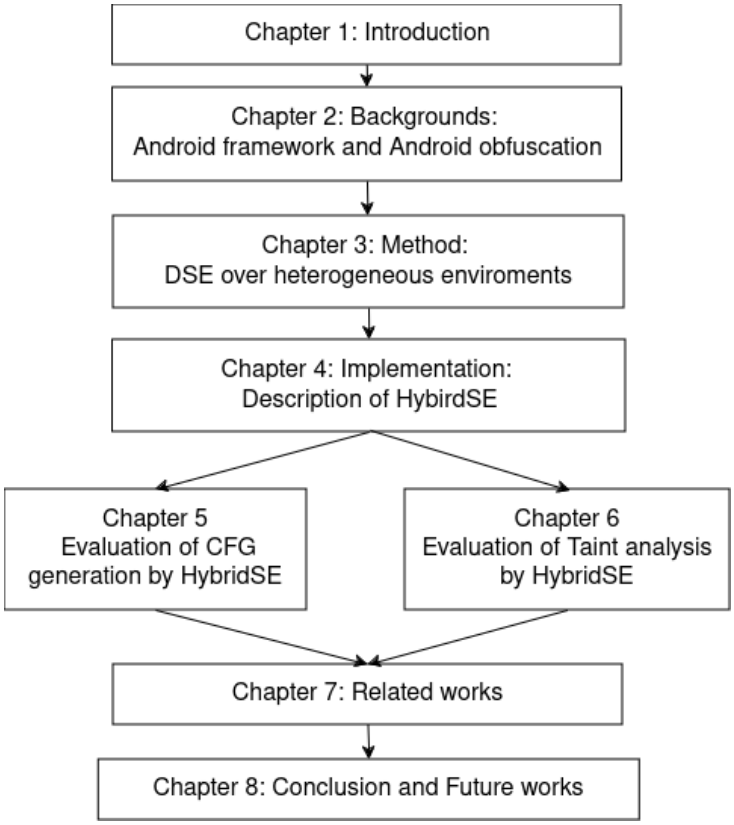

Figure 1.3: Thesis construction

Chapter 5 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 6 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 7 discusses related works, while **Chapter 8** provides concluding remarks for the thesis.

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*

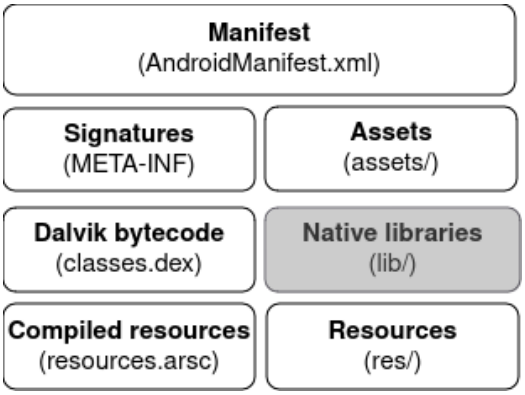

Figure 2.1: Structure of an APK file

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.

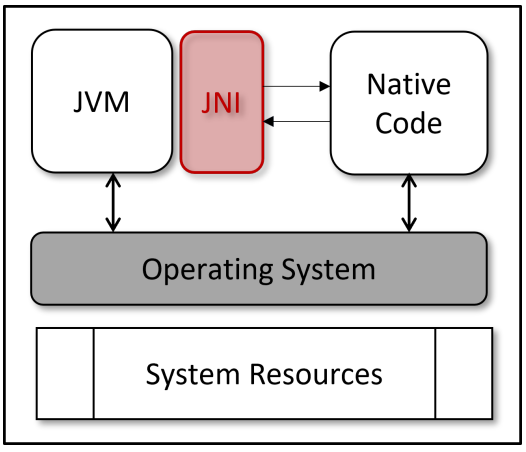

Figure 2.2: Java Native Interface

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, 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.

```
1 public String m3a163f7d(TelephonyManager telephonyManager) {
2     return telephonyManager.getDeviceId();
3 }
4 public C0010p4a8a08f0[] m363b122c() {
5     if ((11 + 19) % 19 > 0) {
6     }
7     return ( C0010p4a8a08f0[]) mc09695e2(f110M, new
          C0010p4a8a08f0[0]);
8 }
```

Listing 2.1: Identifier renaming in an Android malware

String Encryption. Strings are widely used data structures in software development. In an obfuscated app, strings can be encrypted to prevent information leakage. Using cryptographic functions, original plaintexts are replaced by random strings and then restored at runtime. This method effectively hinders static analysis that relies on hard-coded values. 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 an advanced Java feature that allows developers to interact with programs dynamically, such as creating new object instances and invoking methods at runtime.

As an obfuscation technique, reflection is effective for hiding program behaviors because it can transfer control to specific functions implicitly, a challenge for state-of-the-art static analysis tools. Consequently, malware developers often use reflection extensively to conceal malicious actions.

MultiDex. 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, Multidex 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 multidex.

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

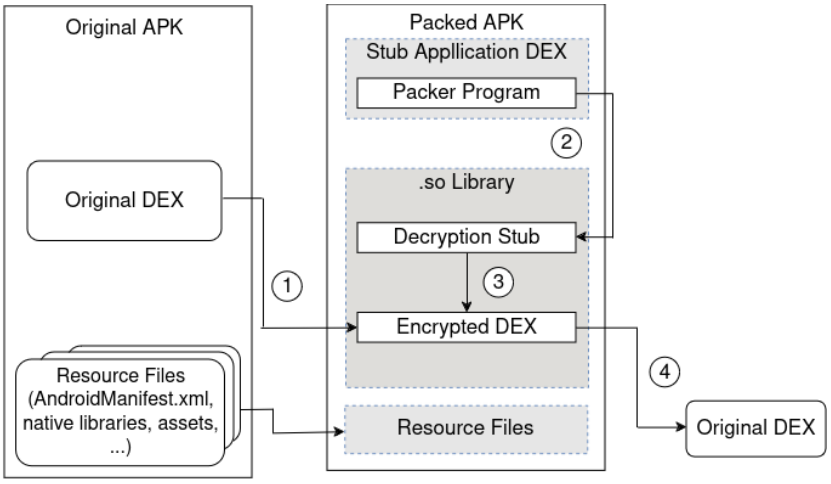

Figure 2.3: DEX encryption mechanism on APKProtect
1 - Packing, 2 - Executing at runtime, 3 - Decrypting, 4 - Unpacking

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 reimplementing 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.

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 the permission of 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) [19] associates formulas to each execution step, obeying the Hoare triple inference rules

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

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* [20] 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*. 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,
- α_i is a path condition (the *pre-condition* of `inst`) at i with $Var(\alpha_i) \subseteq Sym$,
- $Env = \{VarEnv\}$ is a symbolic environment,
  where a variable is $VarEnv : Name \rightarrow Val$ with $Val = \{0,1\}^k \cup Expr(Sym)$
- $CFlow \in (Inst \times Loc)^*$ 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 value stored in the special register $\texttt{pc} \in R$.

$$\frac{Env}{Env'} \texttt{ [inst] if } \psi \;\Rightarrow\; \frac{\langle \alpha_i,\ (CFlow, Env) \rangle}{\langle \alpha_i \wedge \psi,\ (CFlow.(\texttt{inst}, i),\ Env' \rangle} \texttt{ [inst] if } \psi$$

Operational semantics *Hoare logic inference*

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

- *the stack* (e.g., JVM Stack in Java and Stack in x86, ARM) to store local variables and temporary data.

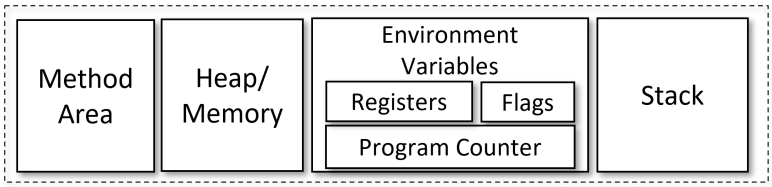

Figure 3.1: Environment model

- *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, the environment model of ARM includes a set of registers, a set of flags, the memory, and the stack. In the environment model, a variable can be either a concrete value or an expression of symbolic values. At the program entry, the value of each variable is initialized with a symbolic value.

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[21], CUTE[22], and SPF [16]). For binary code, McVeto[23] is an early static symbolic execution example, and from around 2015, several dynamic symbolic execution tools have become available, such as MAYHEM[24], KLEE-MC[25], CoDisasm[26], S2E[27], angr[17], BINSEC[28] and BE-PUM[29].

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 syntactic disassembler, e.g., CAPSTONE[2] and IDApro[3], are easily cheated by the obfuscation techniques, especially combined with indirect jumps and

[1]For object-oriented languages, an inter-procedural control flow like a call graph requires a *points-to* analysis [30, 31].

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

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

self-modification to confuse the next control point. Dynamic analyses are also cheated by VM awareness, anti-debugging, and/or trigger-based behavior [32]. Dynamic symbolic execution (DSE) on binary code is considered the most powerful (though heavy) [33, 34].

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[35] for x86, CORANA[36] for ARM, and SyMIPS[37] for MIPS.

As in Chapter 2, an Android APK file consists of Davilk bytecode, native code, and Android library (OS) function calls. There are several formal method tools for Android APK files, such as *JPF-Android* [38], jpf-mobile [39], *SynthesisSE* [12], and ANGR [17]. 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 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`".

Fig. 3.2 shows the calling convention in ARM for the library function *int gettimeofday(struct timeval *tv, struct timezone *tz);*

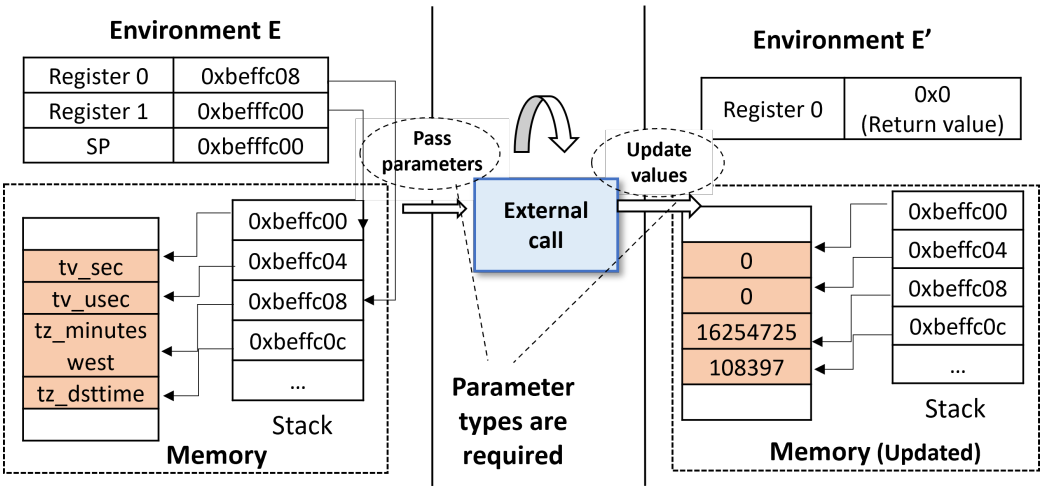

Figure 3.2: Example of an external call in ARM environment

When the caller passes the environment to the callee, there are two choices, *copy* or *share* the environment. A typical choice is the former, espe-

cially 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* [40], 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* [40] is an early example of manually switching such options depending on callees.

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 [15] 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

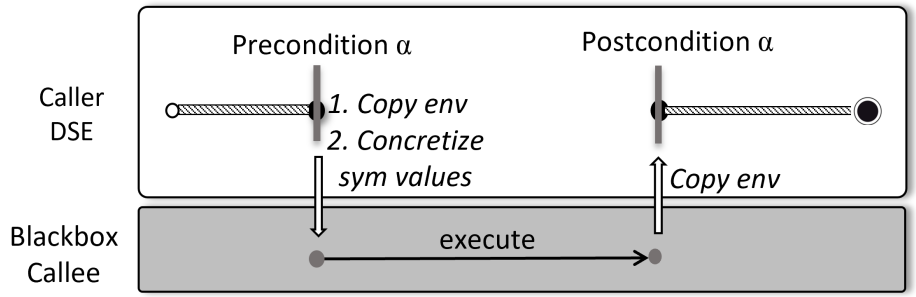

Figure 3.3: Call to a black-box platform

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 [18] follows (2).(b) to keep values symbolic as much as possible.

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 [41], in which the data type information is extracted from Microsoft Developer Network (MSDN).

- For ARM-32 on Android, CORANA/API [18] (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 [41].

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 [18]. 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 [42]. However, this semantics conversion proves to be difficult.

  ANGR [17] 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.

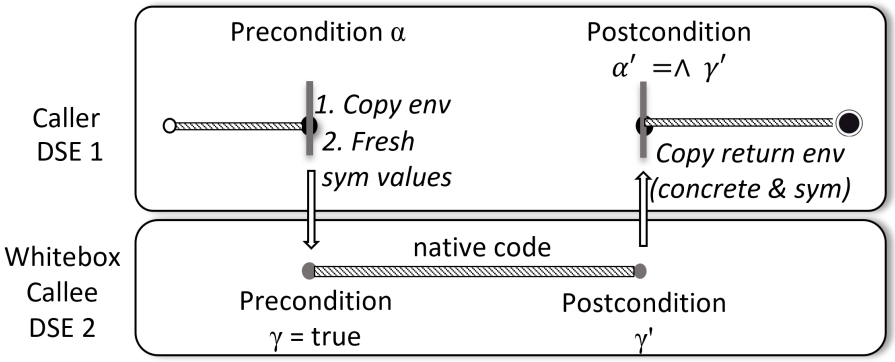

Figure 3.4: Call to a white-box platform

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 [18].

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

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

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 and architecture of HybridSE. This includes the strategy for generating the control flow graph and, atop the DSE engine, an added taint analysis module.

4.1 HybridSE's 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 [43] is an extensible Java analysis tool and its core is a customized JVM that supports multiple analysis strategies. Symbolic PathFinder (SPF) [16] 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 con-

crete 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.

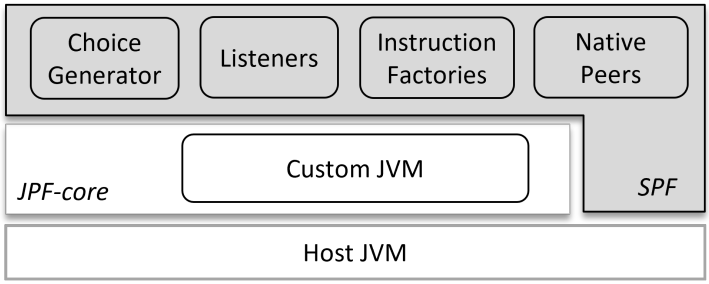

Figure 4.1: Symbolic PathFinder

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) [36] 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 [18]. 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: HybridSE

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 $\Phi'_{java} = \Phi_{java} \wedge \Phi'_{native}$.

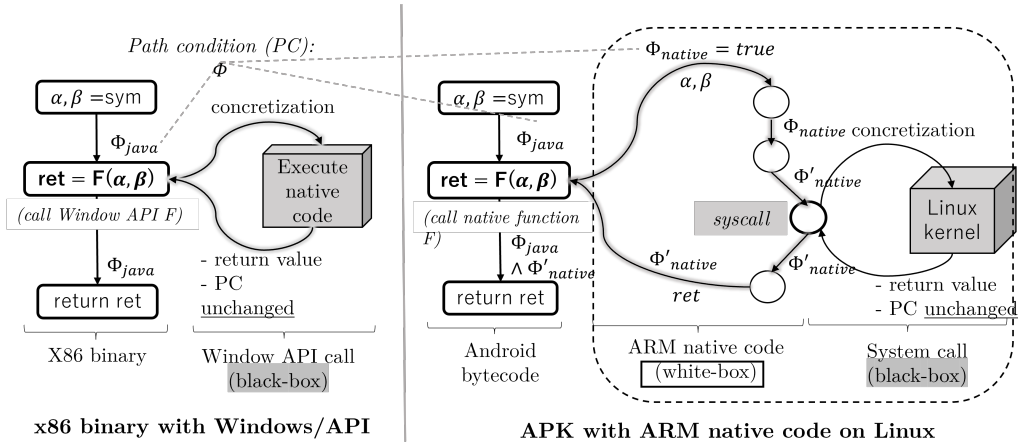

Figure 4.2: Call handling for Android on ARM compared to Windows x86

4.2 HybridSE architecture

We implement a cross-environment analysis tool HybridSE[2] for APK files (Fig. 4.3). Its preliminary goal is to generate control flow graphs (CFGs) and trace data across Java bytecode and native library calls.

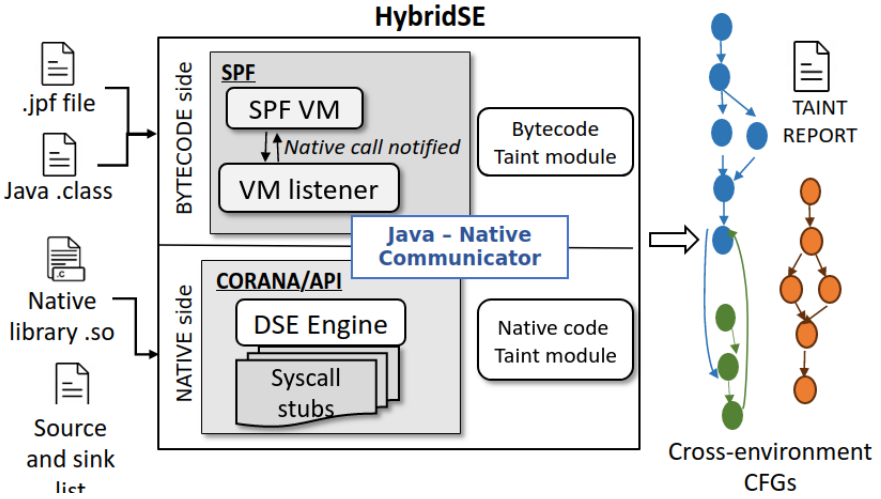

Figure 4.3: HybridSE architecture

HybridSE adopts two DSE tools, SPF for Java bytecode and CORANA/API [18] for ARM natives, which is an extension of CORANA [36] with API stubs to handle system function calls.

[2]figshare.com/s/45b91d138c44e2e55ddd

When the concrete execution crosses platforms, Java Native Interface (JNI) bridges the gap between Java bytecode and the native code (often compiled from C/C++). It adds a communication layer between the JVM and the native code such that (1) JNI maps Java types to equivalent types in C, e.g., *int/jint, long/jlong, Java String/jstring*)[3], and (2) JNI automatically maps native functions in the library.

Similarly, the interfaces between SPF and CORANA/API must facilitate these operations. Our Java-Native Communicator (Fig. 4.3) is implemented following the calling convention, data conversion, and memory handling rules. Fig. 4.4 illustrates an example of such translation.

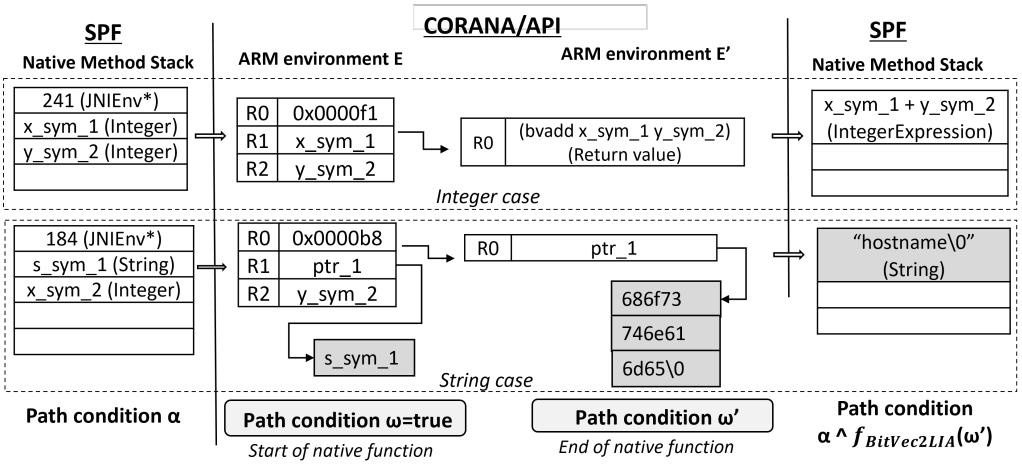

Figure 4.4: Example of a inter-environment translation between SPF and CORANA/API, back and forth

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. 4.5).

- *From AndroidManifest.xml:* Identify potential entry points such as Activities, Services, AsyncTask, and Application, we generate a dummy

[3] docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/types.html

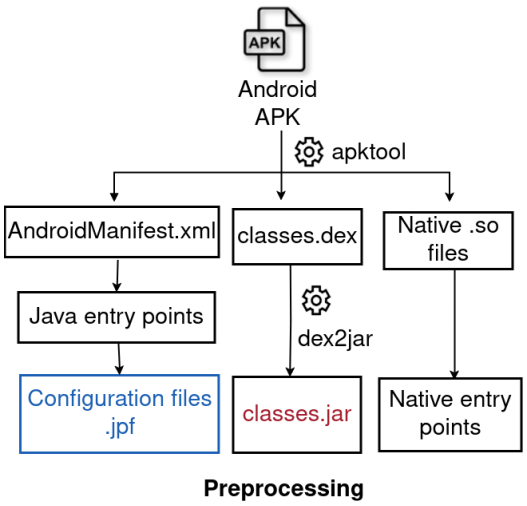

Figure 4.5: Preprocessing in HybridSE

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[4] 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.[5]

4.2.2 Construction of Cross-environment Control Flow Graph

Unlike most static analysis tools [8, 12, 5] 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

[4] `github.com`NationalSecurityAgency/ghidra/releases/tag/Ghidra_10.3.1
[5] docs.oracle.com/en/java/javase/17/docs/specs/jni/design.html

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*, Java classes J, and native code C
Output: A cross-environment CFG G for *jentry*

```
G ← ∅
for insn in SPF.runDFS(jentry) do
    G ← G ∪ {insn}
    if isJNICall(insn) then
        N ← ∅                                          ▷ Native CFG N
        jni_args ← Java2Native.call(SPF.getStack())
        N ← CORANA.runDFS(C, jni_args)
        n_result ←CORANA.getReturn()
        Native2Java.return(n_result)
        G ← G ∪ 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.

4.2.3 Cross-environment communicator

As discussed in Chapter 3, connecting different DSE tools requires interfaces for communication between environments.

It passes the environment and initializes the path condition as $\omega_{BV} =$ `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.

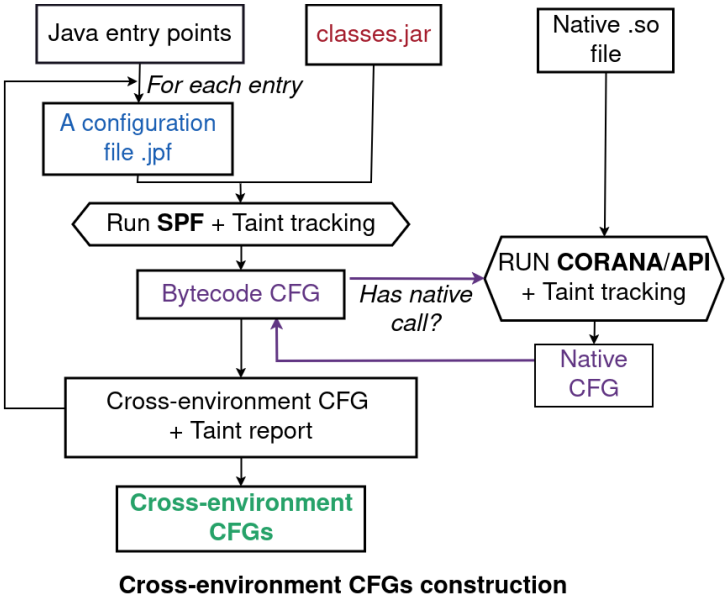

Figure 4.6: An overview of HybridSE system

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. In SPF, listeners allow access and retrieve the information about the VM and the current thread at the occurrence of an event (e.g., *executeInstruction* - before the execution of an instruction, and *instructionExecuted* - after an instruction is executed). Our custom listener interrupts and retrieves the current information of SPF in the event of a native call invocation. SPF listeners provide a way to communicate and configure the execution of the runtime without changing the VM. Then, when a native call is performed, it is caught and processed by CORANA/API. When CORANA/API analysis is finished, results are returned to the listener and the analysis of SPF goes on.

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 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.

```
1 public static native void send(String imei);
2 private void leakImei() {
3     String[] strArr = new String[10];
4     TelephonyManager tel = getSystemService("phone");
5     strArr[1] = tel.getDeviceId();     //strArr[1] is tainted
6             //SOURCE: TelephonyManager.getDeviceId()
7     send(strArr);
8 }
```

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

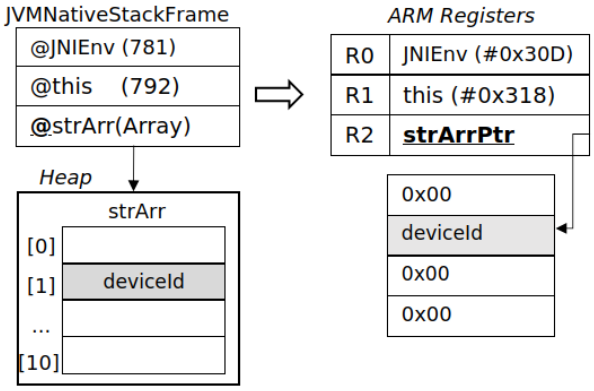

Figure 4.7: How environment is transfer in Example 4.3.1

Android that address both bytecode and native code such as JN-SAF[7] and JuCify[8] 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 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 4.7, 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 [44]. 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. 4.8 illustrates a potential data leak scenario involving JNI callback.

```
1 0x106cc adr r2,[s_getDeviceId_00010754]
2 0x106ce adr r3,[s_()Ljava/lang/String_00010760]
3 0x6d0 ldr  r6,[env,#0x84]
4 0x6d4 mov  r0,r4
5 0x6d6 blx  r6 <JNIEnv_getMethodID()>
6 0x6d8 mov  r2,r0
7 0x6da mov  r0,r4
8 0x6e6 b.w 0x115b0 JNIEnv_callObjectMethod()
9  ;SOURCE: Call getDeviceId()java/lang/String
```

Listing 4.3: Java method is invoked from native code in native_source.apk of Dataset 1 - Chapter 6

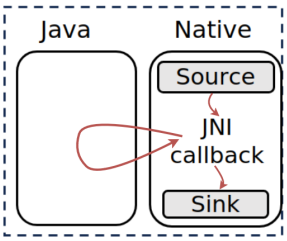

Figure 4.8: Data leakage scenario of Listing 4.3

In either case, we observe an Android malware dataset (Dataset 2 in Chapter 6), 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.

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

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()`)
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 ($\approx$ 230 interfaces)[6]. 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 ...
```

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

```
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:

```
1 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 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

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 5 and Chapter 6 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.

5.1 Experiment 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, 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 [45] 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.

5.2 Cross-environment CFG evaluation

5.2.1 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 5.1 under the category "#w/Native". In the remainder of this study, we concentrate on the samples #w/Native to generate CFGs using HybridSE.

Table 5.1: 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 5.1, the usage of native libraries is notably high across the malware datasets, with a noticeable increase observed from the period spanning 2019 to 2022. .

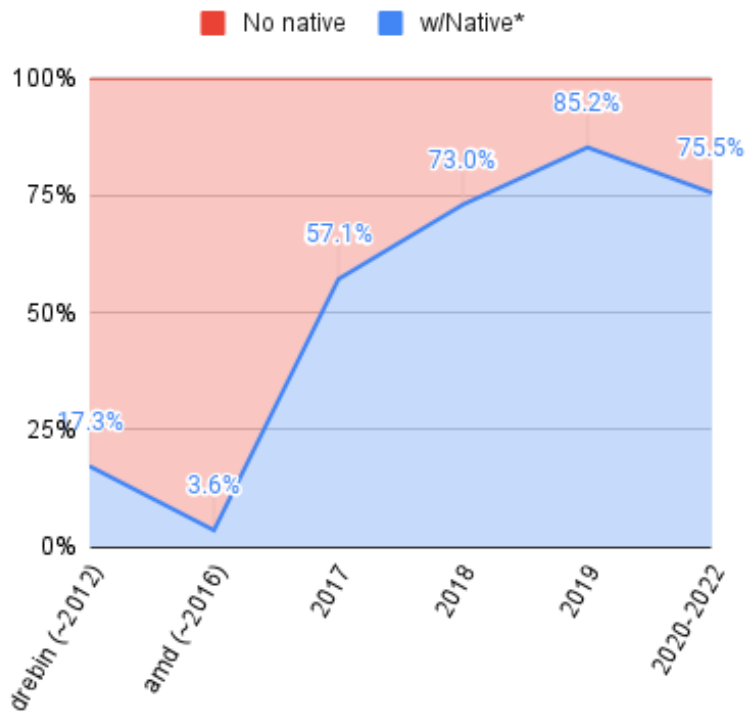

Figure 5.1: Distribution on Android native code usages

We also observe a significant amount of failures and incomplete CFG generation. summerized in Fig. 5.2 and Table 5.2.

We list the main reasons that limit HybridSE (Table 5.2) :

- **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

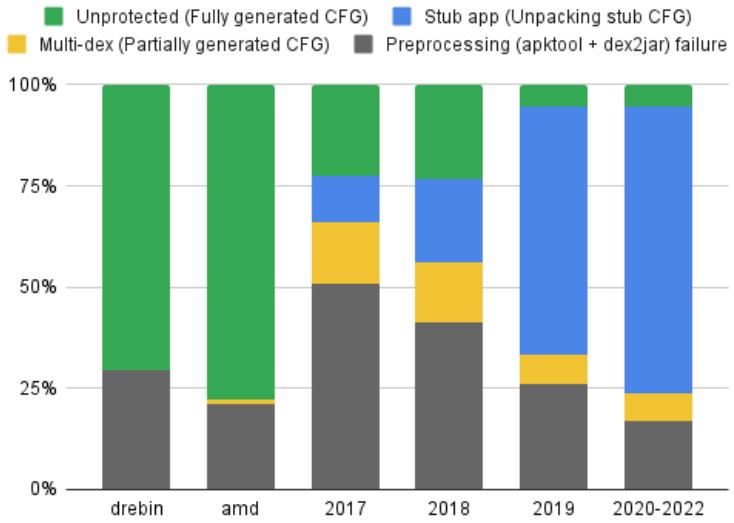

Figure 5.2: Distribution of failures and obfuscation

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 *AndroidManifest.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

Table 5.2: 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

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.

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*.

5.2.2 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 5.3 and 5.4. The average running time is 602.27 seconds; the minimum is 81.64 seconds, whereas the maximum is 96 minutes.

Table 5.3: 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 5.4: Average node and edge counts from HybridSE analyzing AndroZoo malware with native code

Median (± SD)	Bytecode		Native		Time (s)
	# 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)

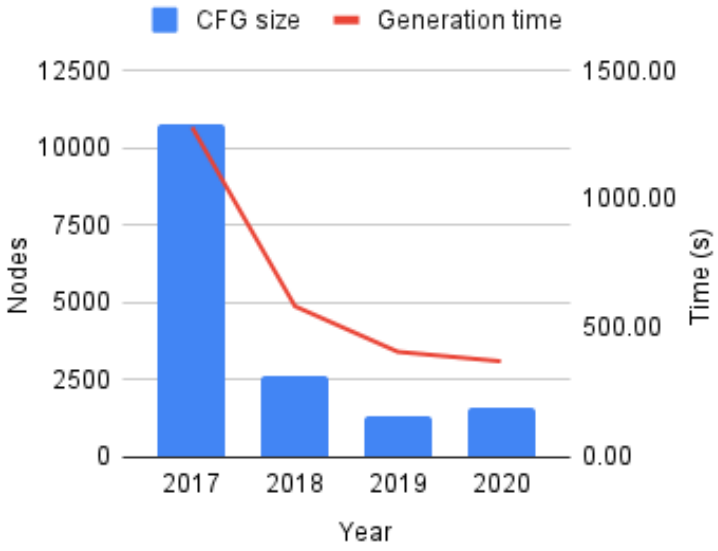

Figure 5.3: Average CFG size for yearly subsets in the AndroZoo dataset
CFG size = Bytecode nodes + Native nodes

Our observation regarding the CFGs of packed and unpacked applications is confirmed by Figure 5.3, which illustrates a decrease in the graph size of samples starting from 2018, coinciding with the increase in the number of packed samples.

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

5.2.3 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

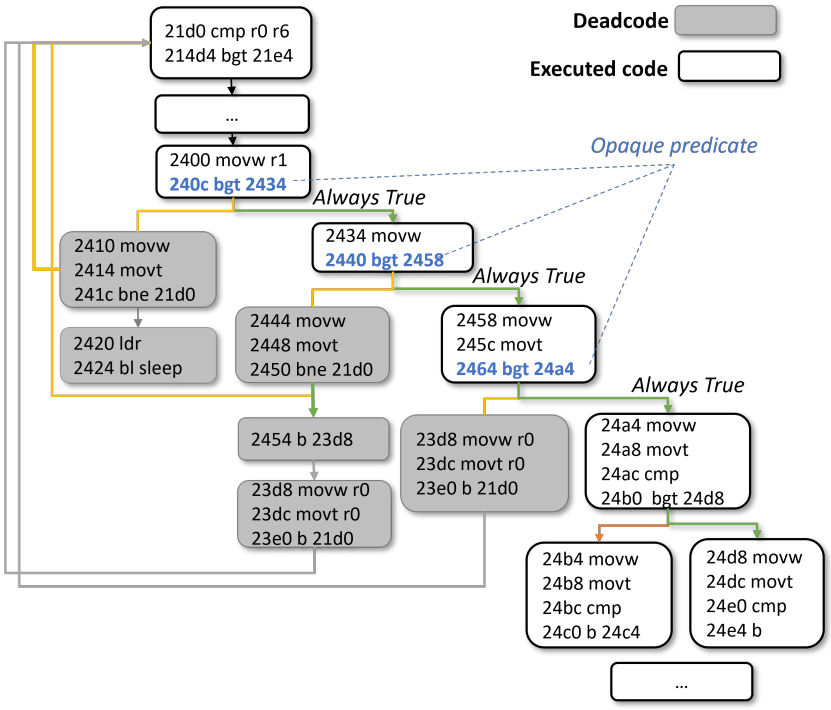

Figure 5.4: Deadcode detected in the native function of **towelroot.apk**

the **Towelroot**[4] shows sequences of `pthread` library calls manipulating 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

[4]gist.github.com/vananhnt/9c9fe78d7a74612d3b5e5363cb76c536
[5]`https://nvd.nist.gov/vuln/detail/CVE-2014-3153`

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.

5.3 Graph similarity among CFGs

To evaluate the quality of the graphs generated by HybridSE and compare them with FlowDroid, we performed graph similarity analysis for two malware analysis tasks: malware family classification and Android packer classification. We employ the SVM classification provided by *scikit-learn* with default settings. For graph embedding, we utilize the Weisfeiler-Lehman graph kernel implementation available in Graph2Vec[6]. Using identical settings, we assess the classification performance by comparing the ability to classify based on graph similarity derived from HybridSE's CFGs and FlowDroid's call graphs.

5.3.1 Preliminary 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.

We employ HybridSE and FlowDroid on the DEBIN and AMD malware datasets, both of which include labeled information regarding malware families. Note that HybridSE generates CFGs across both bytecode and native code, whereas FlowDroid's call graphs contain only bytecode.

From the graphs generated in Table 5.5, we applied graph kernels to obtain feature vector embeddings of the graphs. Subsequently, we classified them into malware families using SVM. The result is reported in Table 5.6. The classification by HybridSE's CFG is carried out in three types. The

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

Table 5.5: Graph produced by HybridSE and FlowDroid for various malware families in DREBIN and AMD

Family	# APK file	HybridSE CFG			FlowDroid Call Graph		
		BN	NN	T(s)	BN	NN	T(s)
DroidKungFu	787	942	387	76.28	3948	0	23.63
Airpush	175	1055	9248	121.43	2689	0	13.637
BaseBridge	123	1133	60	365.71	1939	0	20.86
Dowgin	124	1113	1580	162.71	2558	0	13.18
Lotoor	94	667	645	157.78	262	0	17.58
Youmi	69	1456	4773	220.77	2974	0	22.94
Plankton	55	1727	2013	2777.84	637	0	21.3
Geimini	4	495	358	310.57	1293	0	0.926
Adrd	5	292	904	263.8	937	0	20.13

BN: Bytecode node, NN: Native node, T: Time

combined 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 5.6: Android family classification on DREBIN dataset

	HybridSE+W-L+SVM			FlowDroid+W-L+SVM
	Combined CFG (Bytecode+Native)	Native CFG	Bytecode CFG	Bytecode CLG
Accuracy	76.0 %	**93.91 %**	72.0 %	51.63 %
Precision	70.0 %	**94.84 %**	66.0 %	26.66 %
Recall	76.0 %	**93.91 %**	72.0 %	51.63 %
F1-score	72.80 %	**92.67 %**	68.80 %	35.17 %

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

We recorded an accuracy of 93.91% when classifying the DREBIN malware family using native CFG embedding, compared to 51.65% when using Flowdroid graph embedding.

5.3.2 Preliminary trial on Android packer identification

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

Table 5.7: 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

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[7], 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 5.1: Example output of APKID

Table 5.7 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.

While the number of analyzed samples by HybridSE is lower compared to FlowDroid, the packer identification results using HybridSE's CFGs are

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

significantly higher (Table 5.8). 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.

Table 5.8: Android packer classification on PackerGrind

	HybridSE +W-L+SVM			FlowDroid +W-L+SVM
	Combined CFG (Bytecode+Native)	Native CFG	Bytecode CFG	Bytecode CLG
Accuracy	80.00 %	68.57 %	**97.14 %**	48.89 %
Precision	87.56 %	55.01 %	**97.36 %**	45.34 %
Recall	80.0 %	68.57 %	**97.14 %**	48.89 %
F1-score	74.62 %	59.92 %	**97.10 %**	42.21 %

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

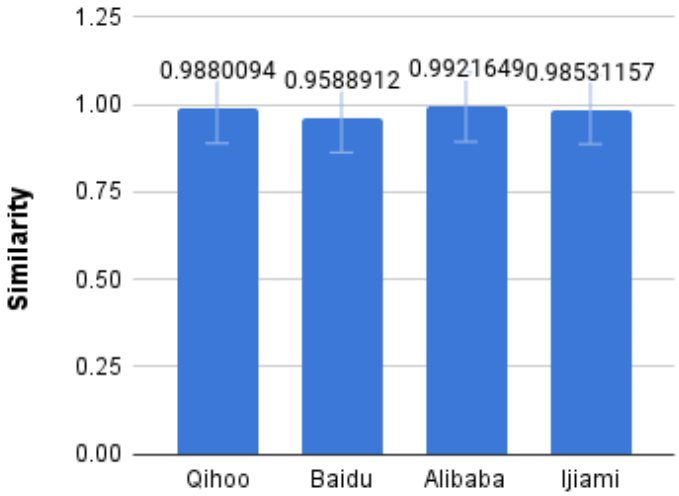

Figure 5.5: Graph similarity by each packer

By leveraging HybridSE and graph kernel similarity, we automate the process for detecting packers. Graph similarity on the bytecode graph of HybridSE's CFG yields the highest accuracy score at 97.14 % F1-score at 97.10 %, compared to only 48.89% and 42.21 % achieved by the call graph on FlowDroid.

We observe that bytecode CFG is better suited for identifying packers than either combined CFG with native code or native code CFG alone. 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 5.5. 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.

Conclusion : Utilizing CFGs generated by HybridSE and applying graph kernels results in high accuracy on classification tasks, achieving an F1 score of 92.67% for malware family classification and 97.10% for packer classification. Our conclusion is that the CFGs generated by HybridSE are well-suited for the characterization of Android applications.

Chapter 6

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.

6.1 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 5. 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

6.2 Comparision with static analysis tools

6.2.1 Detecting cross-environment data leak

Table 6.1 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.

6.2.2 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 6.1. 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 6.1: NativeDroidBench benchmark result

NativeFlowBench - Inter-language dataflow					
	APK file	Ground truth	Hybrid -SE	Flow -Droid	Argus -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 ArgusSAF 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 6.1: Reflection is used to hide source API in JavaRelection1.apk in Dataset 1

Java reflection is handled naturally in SPF [16], 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.

6.3 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 6.1 illustrates the four main data leakage scenarios identified using HybridSE.

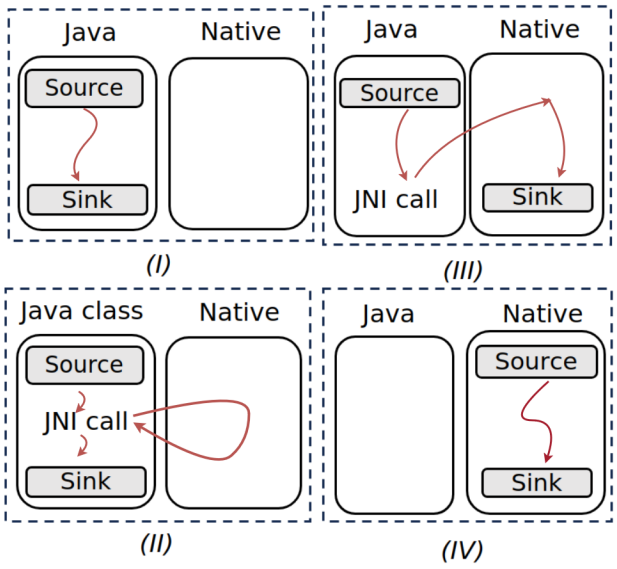

Figure 6.1: Data leakage methods
(I) in *DroidKungFu* and *SimpleLocker*, *Dowgin*, (II) in *Lotoor*, (III) in a *DroidKungFu* variant and (IV) in *Towelroot*

1. *Dataleaks contain only in Java layer (Fig. 6.1-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. 6.1-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. 6.1-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 `getCharFromUTF-String()`, and is shown to the console using the `sprintf()` syscall.

4. *Only in native layer (See Fig. 6.1-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()`.

6.4 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 7

Related works

7.1 Symbolic execution for binary code and bytecode

Symbolic execution (SE) [19] 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 [21] and SPF [16]). Recently, the tools for the symbolic execution of binary code have gradually increased, such as MAYHEM [24], KLEE-MC [25], S^2E [27], ANGR [17], BINSEC [28] and BE-PUM [29].

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[35], ARM as CORANA[36], and MIPS as SyMIPS[37],

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 [21]. However, this is quite a rough approximation and rarely achieves enough accuracy.
- MAYHEM [24] and angr [17] fuse the concrete and the symbolic executions by interleaving the GDB debugger and their symbolic engine.
- BE-PUM [29] 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 [18].

SE for Android/Java bytecode

For Java and Android applications, there are several extensions of Java PathFinder (JPF) [43] that target Android apps. For instance, jpf-mobile [39] uses jpf-nhandler [15] to concretize and run the native code in Host JVM.

SPF [16] and SynthesisSE [12] 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.

7.2 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 [3, 47, 48, 9, 49, 4, 50, 11]. Static taint analysis techniques [3, 48, 49, 4] consider all possible paths that data can flow through without running the

[1] `https://docs.angr.io/advanced-topics/java\_support`

apps. FlowDroid [3] 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[49] extend Flowdroid to handle inter-connection components. Amandroid [4] 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 [47] is arguably the pioneering dynamic taint analysis that traces information flows on the Dalvik Virtual Machine. TaintART[9] adapts dynamic taint analysis for new ART(Ahead of time) runtime. Vialin[11] 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 [51]. 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[52], NativeGuard[53], and TaintArt[9] 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[4] and JuCify [8] 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 8

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 CFGs consisting of feasible paths only.

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.

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, which impacts system correction, is a crucial research question.

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

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

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

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

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

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

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

[38] 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.

[39] 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.

[40] 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.

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

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

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

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

[45] 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.

[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] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

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

[53] 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.