

國立臺灣大學電機資訊學院資訊工程學研究所

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

在 Android 惡意程式中定位模擬器檢測的程式碼區段

Locating Anti-Emulator Code in Android Malware

林永濬

Yung-Chun Lin

指導教授：蕭旭君 博士

Advisor: Hsu-Chun Hsiao, Ph.D.

中華民國 112 年 3 月

March, 2023





摘要

惡意程式開發與分析偵測一直都是攻擊方和防禦方的軍備競賽，主流的分析方法會結合動態與靜態分析來試圖截長補短。對於惡意程式開發者而言，如何去規避分析不讓自己的惡意行為被輕易逆向分析出來，就成為一個相當重要的課題。對於分析人員來說，動態分析對於要瞭解程式行為是不可或缺的一環，然而去針對動態分析的規避行為也相當普遍，其中最常見的就是去檢測大部分動態分析方案皆需要的模擬器環境，這導致在做大規模程式分析時的效果不彰。因此，能夠定位出這些針對動態分析環境的規避行為，對於徹底解析惡意程式有實質上的幫助。

有鑑於模擬器檢測手法一直推陳出新，傳統基於靜態特徵來找出模擬器檢測的方法經常不再堪用。因此，我們開發了名叫 SADroid 的系統，透過靜態輔助動態 (Static-Aided Dynamic) 的分析方法，試圖找到模擬器檢測 (反模擬器) 在程式碼中的哪些片段，並找到這些在近期版本的模擬器上的確能夠改變程式行為的檢測方法。這些檢測方法可作為惡意程式開發者的利器。

關鍵字：惡意程式、動態分析、模擬器檢測、規避行為





Abstract

Malware development and analysis is an ongoing arms race between attackers and defenders. Mainstream analysis designs often combine dynamic and static analysis in an attempt to complement each other's weaknesses. For malware developers, how to evade analysis and not let their malicious behavior be easily reversed and analyzed becomes a critical issue. While dynamic analysis is essential for analysts trying to understand program behavior, evasion of dynamic analysis is also common, with one of the most common designs being the detection of emulator environments which are required for many dynamic analysis approaches. This leads to poor performance when conducting large-scale program analysis. Therefore, the ability to identify these evasion behaviors targeted at dynamic analysis environments is substantially helpful in thoroughly analyzing malware.

Given that techniques for detecting emulator environments are constantly evolving, traditional static feature-based approaches for detecting emulator environments are often no longer effective. As such, we have developed a system called SADroid that uses static-

aided dynamic analysis to identify segments of code in programs that are used for detecting emulator environments (anti-emulation) and finding techniques that can alter program behavior on latest emulator versions, which can be used as tools by malware developers.

Keywords: Malware, Dynamic Analysis, Emulator Detection, Evasion



Contents

	Page
摘要	iii
Abstract	v
Contents	vii
List of Figures	xi
List of Tables	xiii
Chapter 1 Introduction	1
Chapter 2 Related Work	5
Chapter 3 Preliminaries	9
3.1 Dalvik Bytecode and Smali Syntax	9
3.1.1 Locals/Parameters Register	9
3.1.2 Register Range of Dalvik Bytecode Instructions	10
3.1.3 Method Invocation Types And Return Value	10
3.1.4 Tags for jumps	11
3.1.5 Wide Type Register Usage	11
3.1.6 Calling Android APIs in Smali Bytecode	11
3.1.7 Android Support Libraries in Smali Directories	12
3.1.8 App Obfuscation Everywhere	12



3.2	Android App Protection / Malware Evasion	12
3.3	Sensitive APIs List (Target APIs List)	13
3.4	”adb” And ”adb logcat” And ”adb devices”	14
3.5	Android Lifecycle Entry Points And New Process/Thread Entry Points	14
3.6	Bytecode Instrumentation	15
Chapter 4	SADroid Approach Overview	17
4.1	Research Questions	17
4.2	Definitions	17
4.3	Features of Our Approach	18
4.4	Model Architecture	19
Chapter 5	Implementation Details	23
5.1	Bytecode Instrumentation Tool	23
5.1.1	Special Logging Methods: Passing Arguments Without Any Register	24
5.1.2	Registers Modification And Side Effect Handling	26
5.1.3	Runtime Method-Scoped Random ID Passing Mechanism	27
5.1.4	Virtual Method Handling	28
5.1.5	Logging Methods Implementation	29
5.1.6	Multidex Support	29
5.2	Two-Device UI Fuzzer	30
5.3	Log Sequence Analyzer	31
5.4	Dynamic Control Flow Passing Counter	33
Chapter 6	Evaluation	35
6.1	Experiment Environments And Dataset Collection	35

6.2	Sensitive evasion in Our Dataset	36
6.3	Discovery of New Evasion Types	37
6.4	Actual Sensitive Behavior	39
6.5	Bytecode Instrumentation Failure	40
6.6	Case Studies	40
6.6.1	True Positive And True Negative	40
6.6.2	"Try-Catch" as An Evasion	42
6.6.3	Weird Failed Instrumentation Sample	42
Chapter 7	Discussion And Limitation:	45
7.1	What Cause The Instrumentation Fault	45
7.2	Main Cause of The Exception Line During Log Sequence Analysis .	46
7.3	Static Uncertainty: Virtual Methods Issues And Reflection	48
7.4	All Evasion Candidates Results And Cause of Anti-Emulator FNs . .	49
7.5	Cause of Anti-Emulator FPs: UI/Phone Style Difference And Neutral Environmental Checks And Unknowns	50
7.6	Limitations - Code Coverage / Not Supported Apps	51
Chapter 8	Future Work	55
Chapter 9	Conclusion	57
References		59



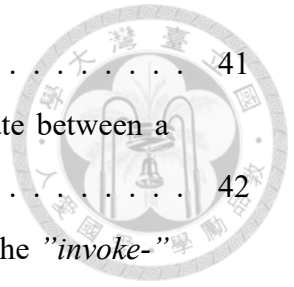




List of Figures

4.1	The overall architecture of the SADroid model, where the input is a dataset of APK files and the output is the evasion candidates and sensitive evasions calculated by the model.	20
4.2	Bytecode Instrumentation Tool	20
4.3	Two-Device UI Fuzzer	20
4.4	Log Sequence Analyzer	21
4.5	Dynamic Control Flow Passing Counter	21
5.1	The Process of Bytecode Instrumentation	25
5.2	Calling a method(<i>callLog()</i>) that prints the caller→callee relationship without using parameter registers	26
5.3	We see different behaviors in whether threads are opened on emulators and physical devices, and the execution order is random	27
5.4	Illustrates how the random ID is stored within the method space to avoid being overwritten by any intermediate call sequence	28
5.5	SADroid’s actual logging method for monitoring branch instructions, which calls a function immediately after the label where the branch instruction jumps. There are also corresponding logging methods for other monitoring points.	30
6.1	A breakdown of the sensitive behaviors observed in the sensitive evasions detected by SADroid in our dataset.	39
6.2	An actual behavior of a sensitive evasion, shown as a pseudo-code in a decompilation tool	39

6.3	A sample of True Positive and True Negative in code.	41
6.4	A special case of using a <i>try-catch</i> block to differentiate between a physical device and an emulator	42
6.5	Rare Smali syntax in which tags are interspersed between the <i>invoke-</i> and <i>move-result-</i> instructions, causing existing bytecode instrumentation algorithms to fail	43





List of Tables

2.1	We compare three static analysis approaches and one dynamic analysis approach. The static analysis approaches can locate the position of anti-emulator code without disrupting the system environment, while the dynamic analysis approach does not require complex code analysis or pre-definition of anti-emulator features. SADroid is designed to incorporate these advantages.	7
6.1	Results from the entire dataset	36
6.2	Results from the logic bomb labels	37
6.3	Conduct actual testing on these attributes on both physical device and emulator	38





Chapter 1 Introduction

Reversing an Android app is relatively easier compared to reversing a PC program, due to the characteristics of languages such as Java and the architecture of the Dalvik Virtual Machine (DVM), which compiles high-level languages into bytecode that runs on the DVM. This makes the syntax more readable to humans compared to assembly language that runs on physical machines [4]. This has led app developers to adopt various protective measures to prevent their apps from being easily hacked, especially for those with malicious intentions. To prevent their intentions from being easily analyzed, various processes are used to resist static and dynamic analysis [20], which will be discussed more in Chapter 3-2.

Static protections can make it difficult for reverse engineers to understand the true intention of a program in the short term, however the actual behavior of the code will always manifest during runtime. Researchers can capture this behavior through dynamic analysis techniques. Therefore, the prevalence of dynamic protection has increased. Among the various processes of dynamic protection, one of the simplest strategies is to detect the environment of dynamic analysis (primarily emulator environments). Most dynamic analysis tools that have been proposed in the past have been detected due to the unique characteristics of these emulator environments compared to physical machines [17]. Therefore, the most commonly used mechanism of dynamic protection is emulator detection (anti-

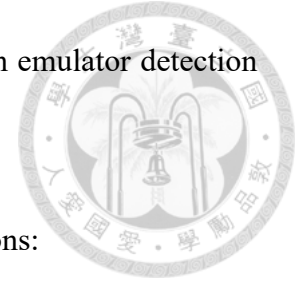
emulator).

Developers have many ways for an app to know if it is in an analysis environment, which can generally be divided into three categories: (1) static device characteristics, (2) dynamic device characteristics, and (3) hypervisor characteristics. Static device characteristics refer to fixed attributes within the phone, such as device identification, device information, call information, and hardware attributes. Dynamic device characteristics refer to attributes that change over time or with use during the app's execution, such as location, network, gestures, light sensors, and CPU status. Hypervisor characteristics refer to the state change of QEMU executing VM machine code.

Through the differences between these characteristics in physical phones and emulators, apps can distinguish whether they are running in an analysis environment and subsequently change their dynamic behavior, such as choosing not to perform certain malicious actions or terminating processes. This significantly affects the effectiveness of dynamic analysis, as malware are less likely to trigger sensitive actions in analysis environments [12]. From the perspective of analysts, if they can find code segments that perform dynamic environment checks, they can bypass such checks through modifying bytecode or using dynamic hooking techniques [22], breaking through the obstacles encountered in dynamic analysis.

In this study, we developed a static-aided dynamic analysis system, SADroid, to identify potential emulator detection-based evasion (referred to as evasion) in malicious apps. Firstly, we used static injection to hook all the necessary monitoring points, and then used a customized UI input generator, droidbot [15], to drive the app to automatically execute. After outputting the dynamic execution results, we compared the dynamic analysis traces

from different devices to calculate which basic blocks might contain emulator detection measures, and thus locate the positions of these checks in the code.

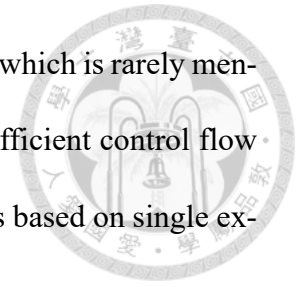


In summary, this research makes the following main contributions:

- **Fine-grained bytecode instrumentation:** The development of a fully automated bytecode instrumentation tool for Smali bytecode syntax, which can monitor the beginning and the end of bytecode methods' definition, as well as branching and jumping points, allowing analysts to understand the dynamic behavior and detailed execution traces of an app at the bytecode level. This leads to the proposal of a static-aided dynamic analysis approach and POC for locating emulator detection without modifying the system itself.
- **Multi sequences problem:** The discussion and resolution of the issue of interleaved log output from multiple threads during the implementation of parsing the output of dynamic analysis, which is rarely mentioned in previous dynamic analysis studies.
- **Evasion location:** The application of an efficient control flow passing counter design to calculate the location of *evasion candidates* based on *single execution sequences* (as defined in Chapter 4-2), thereby identifying potential locations for anti-emulator evasion.
- **New anti-emulator patterns:** The discovery of 8 previously unreported techniques for emulator detection.

The ability to identify potentially emulator-detection bytecode basic blocks in the apps being studied, achieving a precision of about 0.875 on a labeled dataset of malware. The discussion and resolution of the issue of interleaved log output from multiple threads

during the implementation of parsing the output of dynamic analysis, which is rarely mentioned in previous dynamic analysis studies. The application of an efficient control flow passing counter design to calculate the location of evasion candidates based on single execution sequences (as defined in Chapter 4-2).

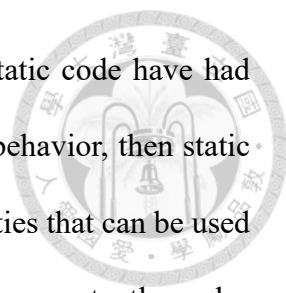




Chapter 2 Related Work

In the implementation of emulator detection, one or more features are typically selected [5], and based on the differences between these features on physical machines and emulators, a conditional statement is used to determine and distinguish the subsequent behavior of the code. For instance, in the Android Studio emulator, the `android.os.Build.DEVICE` property is usually a string starting with "generic", while on a physical phone it is related to the device name. This type of technique can be used to determine whether the code is running on an emulator.

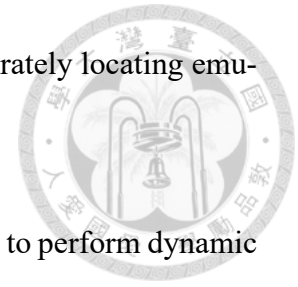
The most intuitive design for finding evasions such as emulator detection is to statically scan the entire program for features used in emulator detection [2, 6, 17, 20]. However, many of the features previously mentioned in the literature for detecting emulators are no longer applicable to current versions of the API, which means developers need to find features that are still present in updated versions of the emulator in order to detect it successfully. We can also further confirm the source of the evasion through code analysis systems [13, 19, 22], which may require parsing the program's call graph (CG) and control flow graph (CFG). For example, by parsing the dependency of a certain branch, we can try to determine whether the boolean value of the conditional statement is related to the attributes of the emulator or the analysis environment.



In the past, methods of scanning static features or analyzing static code have had some obvious drawbacks. If we are looking for emulator detection behavior, then static methods must list all possibilities for checking. Listing all the properties that can be used to check the emulator is impossible, as emulator detection techniques are constantly evolving. Furthermore, as emulators are updated, in previous preliminary experiments, it has been found that many of the emulator detection methods listed in previous papers [21] are no longer able to distinguish emulators. This past knowledge needs to be re-examined. Furthermore, complete static analysis of the program's graphs to identify the presence of emulator detection is computationally expensive and can easily result in a high false positive rate [22]. Because emulator detection is a dynamic means of evasion, we decided to use dynamic systems to look for differences in behavior caused by emulator detection and believed that this approach was more suitable for identifying actual evasion behavior.

Using dynamic analysis to detect emulator evasion is a more feasible approach. For instance, by comparing the dynamic execution trace (API call sequence), it is possible to find differences in behavior between real devices and emulators, and thus infer the existence of evasion [10]. However, pure dynamic analysis design usually requires control over the analysis environment, such as modifying the OS or at least requiring root privileges [9, 10] in order to monitor dynamic execution. This is a time-consuming and labor-intensive design that causes inconvenience in deployment and unexpected risks after modifying the system, and also affects the normal use of other apps. In general, without modifying the system, even with root privileges, it is only possible to monitor the API level, such as tracing system calls through strace or monitoring bytecode function calls through hook frameworks such as Frida and Xposed. However, it is not possible to truly know the actual branch information executed within a function, which limits the research

on the execution status of program branches and is essential for accurately locating emulator detection locations.



Therefore, SADroid uses a special static-aided dynamic analysis to perform dynamic analysis without modifying the system or elevating privileges. In addition to being able to deploy the dynamic analysis environment in a lighter weight way, it can also monitor more fine-grained levels to avoid the drawbacks of traditional dynamic or static systems, as shown in Table 2.1. Chapter 4 will explain the architecture design of SADroid.

Table 2.1: We compare three static analysis approaches and one dynamic analysis approach. The static analysis approaches can locate the position of anti-emulator code without disrupting the system environment, while the dynamic analysis approach does not require complex code analysis or pre-definition of anti-emulator features. SADroid is designed to incorporate these advantages.

(Comparing Related Work)	T	A	D	L	S (Our work)
Locate Anti Emulator Code	♠	♠	♠		♠
No Modify The Device	♠	♠	♠		♠
No Program Analysis				♠	♠
No Predefined Anti Emulator Patterns				♠	♠

(T: TriggerScope [14](Static, Program Analysis), D: Difuzer [19](Static, Anonymous Detection), A: Droid-AntiRM [22](Static, Program Analysis, L: Lumus [10](Dynamic, API Behavior Difference), S: SADroid(Dynamic, Control Flow Behavior Difference).)





Chapter 3 Preliminaries

3.1 Dalvik Bytecode and Smali Syntax

In Android reverse analysis, the main source of code that analysts read is the Dalvik Bytecode in the APK file. Dalvik Bytecode is a register-based intermediate language that is different from the stack-based native Java language. Dalvik Bytecode can be converted to a more human-readable Smali format using the Smali tool [7] in APKTool [3]. Each original code class is converted to a single Smali file that records the definition of the class, including its internal methods and variables. There are some special key syntaxes in Dalvik Bytecode and Smali that we must use, which are described in detail below.

3.1.1 Locals/Parameters Register

In each method of Dalvik bytecode, registers are divided into two categories: locals and parameters. Locals are placed before parameters. For example, if there are 5 locals and 3 parameters, the entire method register in Smali syntax will be $v0, v1, v2, v3, v4, p0, p1, p2$ or $v0, v1, v2, v3, v4, v5, v6, v7$. The number of locals plus the number of parameters is the maximum number of registers that can be used in this method. If the method is not a static method, $p0$ is the default instance object of the class to which the method belongs. In this

case, $p1$ and $p2$ are the parameters of the method.



3.1.2 Register Range of Dalvik Bytecode Instructions

In Dalvik Bytecode, the range of registers that can be used by each instruction also varies, roughly divided into 16, 256, and 65536 registers. Most instructions can only use registers $v0-v15$, while a few instructions can use a wider range of registers. When modifying Bytecode, the most common cause of app crashes is the misuse of registers by manually modified instructions that should not be used. For instance, the "*invoke-*" instruction when calling a function will cause a syntax error and cause the app to crash when it is opened if it is changed to use a parameter above $v16$. This is also a common problem we encounter when modifying bytecode instructions later.

3.1.3 Method Invocation Types And Return Value

Corresponding to the method types in Java, Dalvik bytecode also has five ways of calling methods: "*invoke-virtual*", "*invoke-direct*", "*invoke-static*", "*invoke-super*", and "*invoke-interface*". Among these, "*invoke-virtual*" and "*invoke-interface*" are often unable to be known which instance is called at the static stage, so it is difficult to trace back to the callee's method bytecode. This characteristic will cause some restrictions when doing Bytecode Instrumentation later. In addition, when the parameters are more than six, "*invoke-xxx/range*" will be used instead of "*invoke-xxx*" and consecutive registers must be used. These will be allocated automatically when compiled into bytecode, but when analysts need to modify bytecode, there are many restrictions.



3.1.4 Tags for jumps

In the bytecode of the method definition part, information about all branches and basic blocks is also included. Except for the first block, each basic block is distinguished by various tags that mark the line at which the block begins in the bytecode. This means that all instructions related to branching, such as *"if-*", *"goto-*", *"try-catch"*, and *"switch"*, are identified by different tags to indicate the position to jump to, which is also needed to know for fine-grained dynamic trace tracking.

3.1.5 Wide Type Register Usage

In Dalvik Bytecode, the storage length of registers is 32 bits, but for special data types such as long and double, the length is 64 bits. Therefore, it is not possible to store such data in a single register, but two consecutive registers are used to store it. Such as, if the *"move-*" instructions moves double data to register *v3*, both *v3* and *v4* will be written at the same time and cannot be separated, otherwise the app will crash. Instructions that operate on long data types have the suffix *"-wide"*.

3.1.6 Calling Android APIs in Smali Bytecode

In Android, most functions related to the phone environment and interface require calling Android APIs to achieve, but the source code for these APIs is not included in the apk, but is instead stored internally on the phone system. This means that we will not find the source code in the Smali file, and if we want to monitor the call of these APIs through bytecode instrumentation, we can only operate at the caller's location. The specific API list will change with Android version updates and can be referred to the

official documentation [1].



3.1.7 Android Support Libraries in Smali Directories

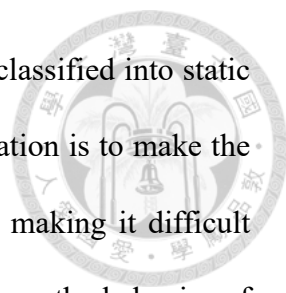
In the data folder decompiled by APKTool, there are still some Android-related libraries, such as the Android Support Library, which is generated by the compiler for compatibility with some systems. These codes are basically unrelated to the functions written by the app developer and are not called, so they can be excluded from static analysis without the need for instrumentation modification.

3.1.8 App Obfuscation Everywhere

In Android app development, it is common practice to use obfuscation tools to protect the user-defined class names, method names, and variable names from being easily understood by reverse engineers. These tools often result in meaningless class and method names, such as a.a.a.a.a(), which can make it difficult for analysts to determine the function of the code during manual reverse engineering. Obfuscation is a major obstacle for reverse engineers attempting to understand the functionality of an app.

3.2 Android App Protection / Malware Evasion

In Android Apps, various protection measures are commonly applied, such as obfuscation, reflection, dynamic loading, encryption, shell, anti-debugging, anti-rooting, anti-hooking, anti-tampering, and anti-emulator, to prevent the program from being easily reversed and analyzed, thereby completely mastering all program logic. In malware, these protection measures are used to hide malicious behavior characteristics, which is known



as evasion. There are many types of evasion, which can be broadly classified into static obfuscation and dynamic checks. The main purpose of static obfuscation is to make the program code difficult to read, whether manually or automatically, making it difficult to understand the meaning. Dynamic checks, on the other hand, change the behavior of the program when certain conditions are met, either by directly terminating the program or hiding malicious behavior. This type of check for hiding malicious behavior is often referred to as a logic bomb [18].

In dynamic checks, checking whether an app is running on an emulator or analysis environment is the most common and intuitive method. Therefore, understanding the location of these emulator checks is also a starting point for analyzing malware, which is the main focus of this study. For convenience, the evasions mentioned in this paper refer to anti-emulator based or analysis environment detection evasions.

3.3 Sensitive APIs List (Target APIs List)

To further understand the activities of these apps with evasion techniques, we collected a target API list from several sources, totaling approximately 16350 API calls. This list mainly includes Android APIs which require permissions to use, as well as some data transfer or dynamic loading functions. These are considered sensitive behaviors and are likely to be used when malware executes their real functions. Therefore, it is common in android malware analysis studies [15, 19, 22, 23] to collect such an API list to focus on the target of the analysis. The relationship between the invocation of these APIs and evasion may seem intuitive, e.g. , adding an evasion technique to control this branch before calling these APIs in order to avoid being easily detected by analysts. Therefore, we

believe that this background knowledge is essential.

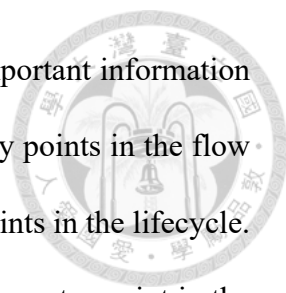


3.4 "adb" And "adb logcat" And "adb devices"

To communicate between an Android device and a PC, the Android Debug Bridge (adb) must be used. Through adb commands, we can control an Android device connected to an adb server from a PC, including both physical phones and emulators. Using adb commands, we can install/uninstall apps, send UI operation commands to the phone, start or stop any app, and anything else that can be done on the phone interface. The Logcat tool within adb is a built-in tool for viewing and filtering logs generated by app execution. There are several types of log functions in Android, including Log.v(), Log.d(), Log.i(), Log.w(), and Log.e(), which differ only in the level of alert or notification. Developers can choose which to use, and these can be used to monitor the dynamic execution track. The adb devices command lists all Android devices currently connected to the computer, including physical phones connected via USB and emulators within the computer. Only devices connected to the PC's adb server can be operated through adb commands.

3.5 Android Lifecycle Entry Points And New Process/Thread Entry Points

In the process of running an Android app, the app is typically run as a component. There are four main components in Android: activity, service, content provider, and broadcast receiver. The user sees each screen of the app as an activity. All the functions of the Android app operate on top of these components, which is why the stages



of their creation and termination, known as the lifecycle, are very important information when analyzing an app. These lifecycle stages are often used as entry points in the flow of an Android app, meaning that the app's functions begin at these points in the lifecycle. In addition to the lifecycle, the creation of threads/processes is also an entry point in the flow of a program. The use of threads is very common in modern programming and is a characteristic of almost all apps.

3.6 Bytecode Instrumentation

Bytecode Instrumentation [16] is a technique used in dynamic analysis that involves injecting code into target locations to track or record program behavior. For Android app APK files, instrumentation using Smali bytecode is an effective form for analysts because the Smali tool is easy to use and has higher readability for humans. In order to achieve the goals of this study, a comprehensive understanding of bytecode syntax is necessary, as even a small error in modification can cause the entire app to crash during execution. After bytecode instrumentation, the original APK file is repackaged with the injected bytecode to create a new APK, which is then installed on the target machine for execution.





Chapter 4 SADroid Approach

Overview

4.1 Research Questions

The research questions we want to understand are:

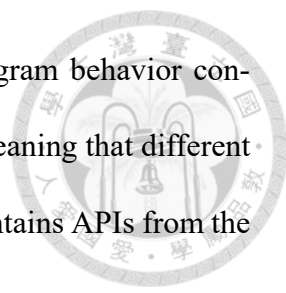
RQ1: Is there evasion of emulator detection in Android malware, and if so, which code is causing the evasion?

RQ2: Are there any types of evasion that have not been discovered in previous studies?

RQ3: What are these anti-analysis evasions being used to conceal in terms of app behavior?

4.2 Definitions

Evading Points Candidate(Evasion Candidate): In dynamic analysis, if a basic block causes behavioral differences between a physical device and an emulator, the block is referred to as an evasion candidate. Behavioral differences refer to the fact that, after executing the evasion candidate block, the two devices execute different basic blocks.



Hidden Behavior And Sensitive Hidden Behavior: The program behavior controlled by a particular evasion candidate is called hidden behavior, meaning that different child blocks are executed on a physical machine and emulator if it contains APIs from the target API list, which are considered to have sensitive hidden behavior.

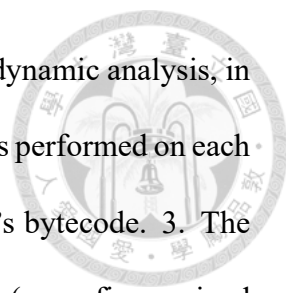
Sensitive Evasion: Within evasion candidates, if it has sensitive hidden behavior, it is referred to as a sensitive evasion. Of course, it is not easy to determine whether the developer was aware of this use or intended to detect the presence of an emulator. We only examine the functionality of the code itself.

Single Execution Sequence: The logs of all the running tracks generated by a single entry point are grouped into the same sequence, called a single execution sequence. When an app is running, different threads/call sequences often execute simultaneously, which results in log output lines that are often not on the same call sequence. We need to split the original log output lines into multiple single execution sequences to truly understand the actual behavior of the program.

Free Register(s): In bytecode, a register that can be freely assigned without affecting the function of the original code is called a free register. For example, at the beginning of a method, the free registers contain all the locals registers, and before the return instruction, all the locals minus the register containing the return value, as well as all the parameters registers.

4.3 Features of Our Approach

SADroid has the following features: 1. Non-invasive dynamic analysis environment, which means no root permissions are required and the system itself is not modified, and has



no impact on the system or other apps in use. 2. Static assistance for dynamic analysis, in order to achieve the effectiveness of dynamic analysis, static analysis is performed on each app beforehand, and only needed modifications are made to the app's bytecode. 3. The ability to track app execution and jump tracks at the basic block level (more fine-grained than the API level). 4. No source code is required.

4.4 Model Architecture

The model consists of four main components, as shown in Figure 4.1 . These are the Bytecode Instrumentation Tool (Figure 4.2), the Two-Device UI Fuzzer (Figure 4.3), the Log Sequence Analyzer (Figure 4.4), and the Dynamic Control Flow Passing Counter (Figure 4.5). The Bytecode Instrumentation Tool decompiles the APK using APKTool [3] and then inserts specialized logging functions and other support functions at all monitoring points, before repackaging the modified APK. The Two-Device UI Fuzzer drives the app to execute on both physical devices and emulators using the same input search strategy, generating UI events for both. The Log Sequence Analyzer filters and thoroughly analyzes the app's special logs to piece together the original execution flow of the program. The Dynamic Control Flow Passing Counter, after knowing the dynamic execution track of the program, counts and compares the execution times of each basic block on two platforms, and outputs the possible evasion basic block info.

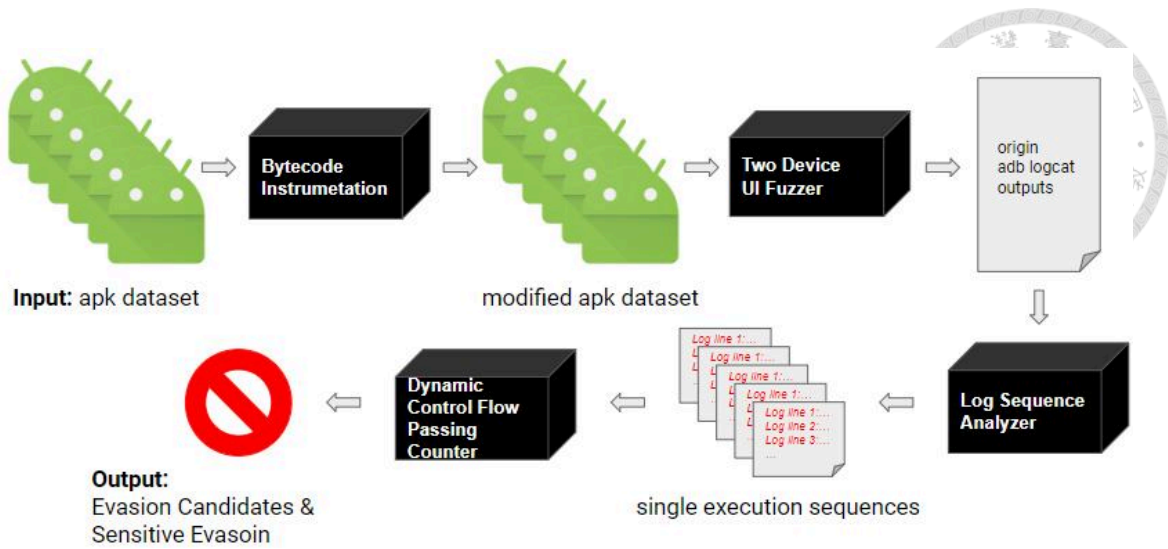


Figure 4.1: The overall architecture of the SADroid model, where the input is a dataset of APK files and the output is the evasion candidates and sensitive evasions calculated by the model.

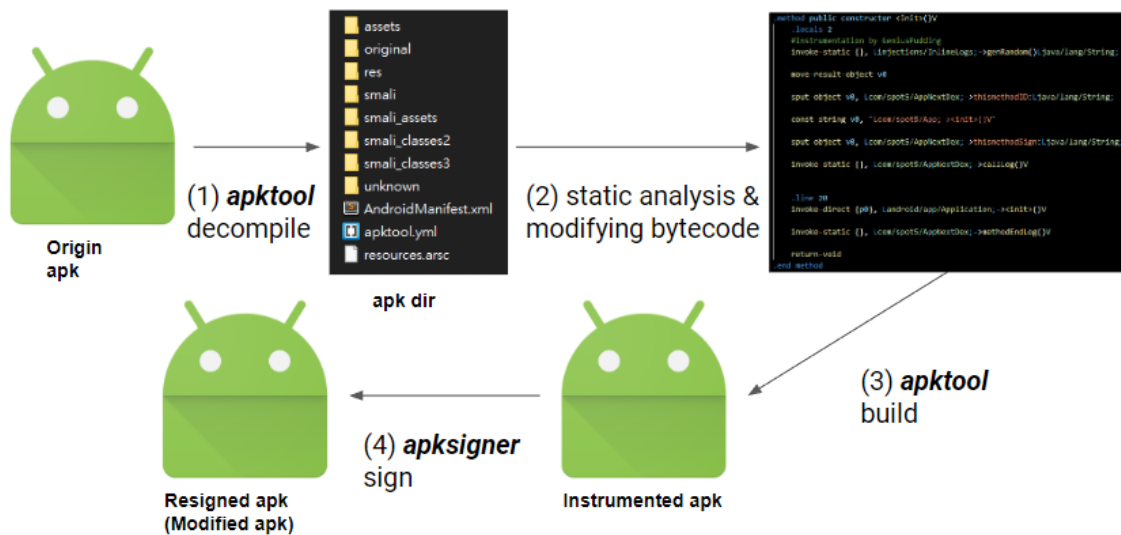


Figure 4.2: Bytecode Instrumentation Tool

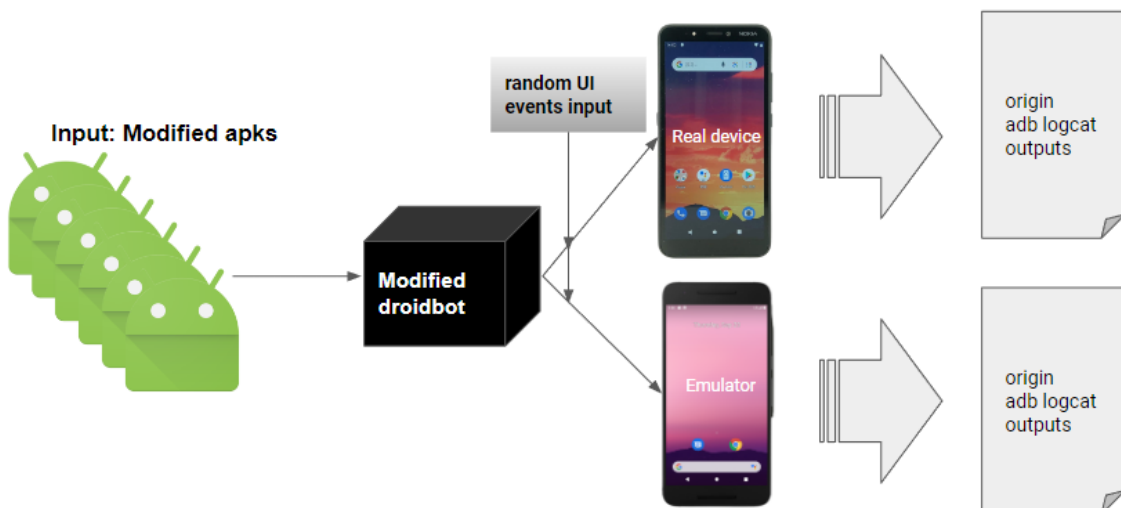


Figure 4.3: Two-Device UI Fuzzer

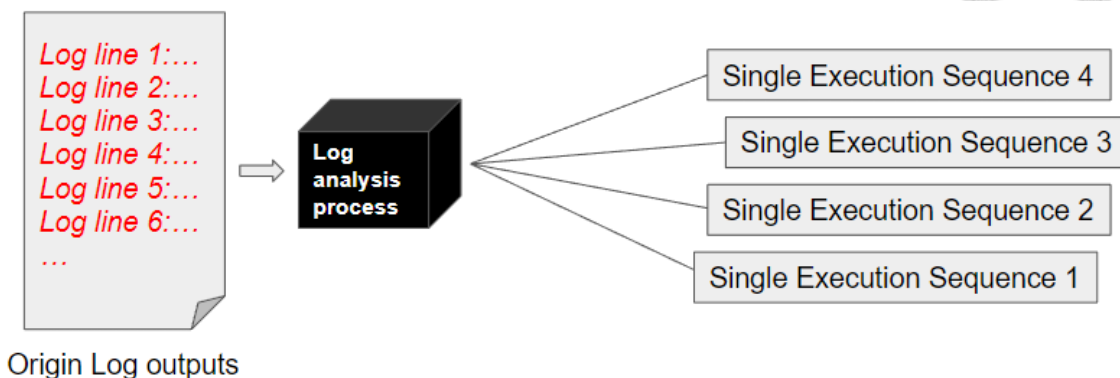


Figure 4.4: Log Sequence Analyzer

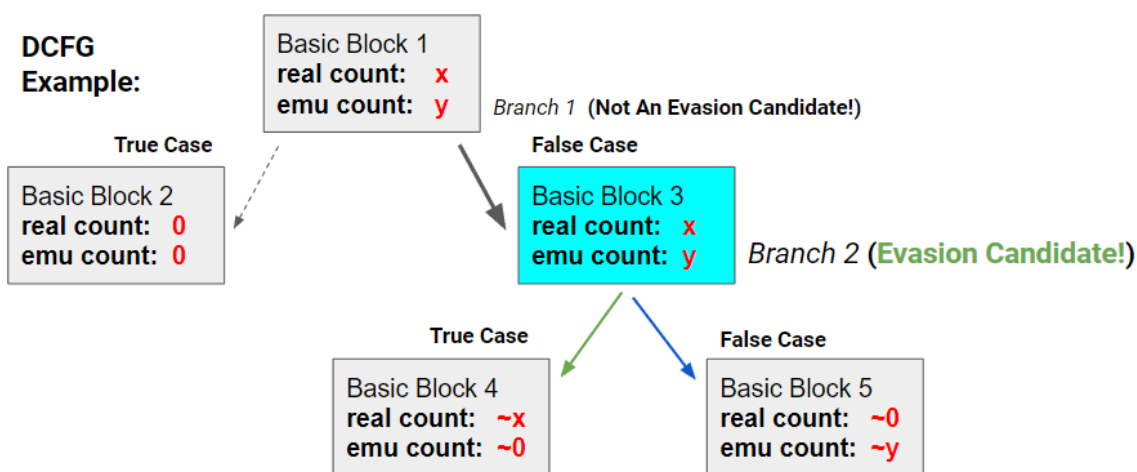


Figure 4.5: Dynamic Control Flow Passing Counter





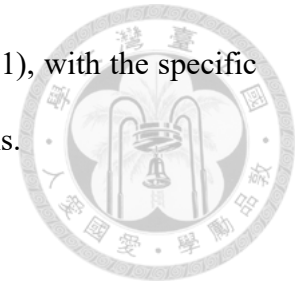
Chapter 5 Implementation Details

5.1 Bytecode Instrumentation Tool

To meet the requirements of subsequent dynamic analysis, static analysis is performed on the bytecode of the entire APK file, followed by a comprehensive rewrite. The general process is as follows: 1. Use APKTool to convert the dex file into a folder containing Smali files. 2. For the subfolder of Smali files under the directory, exclude the Android Official Libraries and then iterate through each developer-defined class file (Smali file). 3. For each Smali file, read the bytecode in each method, and then inject the required bytecode code through program code analysis at all targeted points where monitoring is desired, using an inline hook to inject all necessary logging functions. The targeted points include the beginning and return before each user-defined method, the call and return of the target API, before and after branch instructions, the start and end of "try-catch", or exception handling. 4. Re-sign and package the rewritten APK.

The above process is fully automated. In this process, code analysis is a determinative step, the main purpose of which is to obtain a freely usable free register at all points where we need to add features. Therefore, we need a comprehensive understanding of the side effects that these changes to the bytecode have on the original code function, and any change will cause a cost in register usage that should be minimized as much as possible.

The following is the process of Bytecode Instrumentation (Figure 5.1), with the specific code analysis and injection details explained in the following sections.



5.1.1 Special Logging Methods: Passing Arguments Without Any Register

To assist dynamic analysis through static analysis, we need to pass necessary information to the android log functions. The android Log functions all have two parameters, a tag and a message, to print out information. In SADroid, we use a specific tag to allow the Logcat tool to filter out SADroid's output logs directly, without seeing logs generated by other apps on the device. However, no matter what function is called, only registers up to number 16 can be used, and it is not possible to ensure that a free register within 16 can be found at any point in the code, so we do not want to pass information to the logging functions through parameters.

Therefore, SADroid creates a class relevant extra-space (C-space) corresponding to each class, and uses the C-space to pass the required data to avoid the trouble of passing parameters when calling logging methods (*"invoke-"* instructions can only use registers up to number 16). The specific implementation is to open a new Dex in the original APK and define a corresponding class inside, defining all significant static variables and logging methods. Any method called will temporarily borrow the variables in the C-space corresponding to its class to access the required information, as shown in Figure 5.1.

This design is because we actually cannot accurately know which of the registers within 16 is free to store the required argument when the program reaches a certain line of bytecode, which eliminates a lot of unnecessary troubles that cannot be avoided at any

```

.method public final e()Ljava/lang/String;
.locals 3

iget-object v0, p0, Lco441/ronash/pushe/c; ->b:Landroid/content/Context;

invoke-static {v0}, Lco441/ronash/pushe/e/a/b; ->a(Landroid/content/Context;)Lco441/ronash/pushe/e/a/b;

move-result-object v0

const-string v1, "$instance_id"

const/4 v2, 0x0

invoke-virtual {v0, v1, v2}, Lco441/ronash/pushe/e/a/b; ->a(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;

move-result-object v0

return-object v0
.end method

```

(a) Before Bytecode Instrumentation

```

.method public final e()Ljava/lang/String;
.locals 4
↳ add 1 locals
↳ instrumentation by GeniusPudding
↳ generate random ID
↳ log call relation
↳ store random ID
↳ get random ID
↳ log method return

```

```

.invoke-static {}, LInjections/zalinelogs; ->genRandom()Ljava/lang/string;

move-result-object v0

sput-object v0, Lco441/ronash/pushe/cwextdex; ->thisMethodID:Ljava/lang/String;

const-string v0, "Lco441/ronash/pushe/c; ->e()Ljava/lang/String;"

sput-object v0, Lco441/ronash/pushe/cwextdex; ->thisMethodSign:Ljava/lang/String;

invoke-static {}, Lco441/ronash/pushe/cwextdex; ->callLog()V

iget-object v0, p0, Lco441/ronash/pushe/c; ->b:Landroid/content/Context;

sget-object v3, Lco441/ronash/pushe/cwextdex; ->thisMethodID:Ljava/lang/String;

sput-object v3, Lco441/ronash/pushe/e/a/bwextdex; ->callerID:Ljava/lang/String;

invoke-static {}, Lco441/ronash/pushe/cwextdex; ->concat()V

sget-object v3, Lco441/ronash/pushe/cwextdex; ->tmp:Ljava/lang/String;

invoke-static {v0}, Lco441/ronash/pushe/e/a/b; ->a(Landroid/content/Context;)Lco441/ronash/pushe/e/a/b;

move-result-object v0

sput-object v3, Lco441/ronash/pushe/cwextdex; ->tmp:Ljava/lang/String;

invoke-static {}, Lco441/ronash/pushe/cwextdex; ->split()V

const-string v1, "$instance_id"

const/4 v2, 0x0

sget-object v3, Lco441/ronash/pushe/cwextdex; ->thisMethodID:Ljava/lang/String;

sput-object v3, Lco441/ronash/pushe/e/a/bwextdex; ->callerID:Ljava/lang/String;

invoke-static {}, Lco441/ronash/pushe/cwextdex; ->concat()V

sget-object v3, Lco441/ronash/pushe/cwextdex; ->tmp:Ljava/lang/String;

invoke-virtual {v0, v2, v3}, Lco441/ronash/pushe/e/a/b; ->a(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;

move-result-object v0

sput-object v3, Lco441/ronash/pushe/cwextdex; ->tmp:Ljava/lang/String;

invoke-static {}, Lco441/ronash/pushe/cwextdex; ->split()V

invoke-static {}, Lco441/ronash/pushe/cwextdex; ->methodEndLog()V

return-object v0
.end method

```

(b) After Bytecode Instrumentation

Figure 5.1: The Process of Bytecode Instrumentation



point that requires additional invocation.

```

4731 .method public show(Ljava/lang/String;Lcom/startapp/sdk/adsbase/adlisteners/AdDisplayListener;)Z
4732     .locals 40
4733     #Instrumentation by GeniusPudding
4734     invoke-static {}, Linjections/InlineLogs;->genRandom()Ljava/lang/String;
4735
4736     move-result-object v0
4737     "sput" command can use 1~256th registers
4738     sput-object v0, Lcom/startapp/sdk/adsbase/StartAppAdNextDex;->thisMethodID:Ljava/lang/String;
4739
4740     const-string v0, "Lcom/startapp/sdk/adsbase/StartAppAd;->show(Ljava/lang/String;Lcom/startapp/sdk/adsbase/adlisteners/AdDisplayListener;)Z"
4741
4742     sput-object v0, Lcom/startapp/sdk/adsbase/StartAppAdNextDex;->thisMethodSign:Ljava/lang/String;
4743
4744     invoke-static {}, Lcom/startapp/sdk/adsbase/StartAppAdNextDex;->callLog()V ⇐ log call relation without argument registers

```

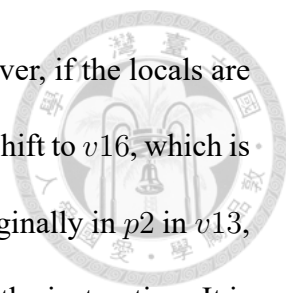
Figure 5.2: Calling a method(*callLog()*) that prints the caller→callee relationship without using parameter registers

5.1.2 Registers Modification And Side Effect Handling

At least one register needs to be available for storing arguments or temporary data when passing this necessary information. Therefore, the most intuitive approach is to increase the number of local registers by one. In theory, as long as the number of locals declared after the method declaration is increased by one, a free register can be obtained. We name this register v_{last} .

However, nothing in life is free. When the number and order of registers change, the most direct impact is that the originally compiled instructions may happen to exceed the available range, causing the program to crash. In fact, it is almost nonexistent for more than 256 registers to be used in a Dalvik bytecode method (only one was seen in the entire dataset), and it is usually just over 16 registers. Therefore, we named the side effects caused by increasing the number of local registers as the *v16* problem (*vx* is the register name in the Smali syntax).

To handle the *v16* problem, we parse each Dalvik instruction and use a pre-established register rule table to determine if there are any instructions that are exactly at the limit of the available registers. For instance, if the "invoke-" instruction originally has a parameter *p2* (the third parameter register) and there are originally 13 locals, then *p2* would originally be



equivalent to $v15$ (which can be directly replaced in the code). However, if the locals are increased to 14, with the additional register being $v13$, then $p2$ would shift to $v16$, which is exactly at the limit. In this case, we can temporarily store the data originally in $p2$ in $v13$, rewrite the original $p2$ instruction to $v13$, and then move it back after the instruction. It is important to be careful here as "invoke-" instructions may have a return value, followed by a "move-result-" instruction, which cannot be inserted with other instructions.

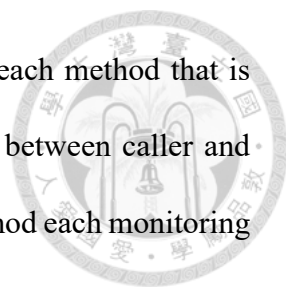
Although this approach seems perfect, there are still cases of $v16$ problems that cannot be solved. We will discuss this in Section 7-1.

5.1.3 Runtime Method-Scoped Random ID Passing Mechanism

During the execution of an app, there is often not just one entry point for the program, but rather multiple single execution sequences that interweave and output. This makes it difficult for us to directly read the original Logcat output as the order and trace of code execution, and we must first split it apart. In the output from both physical devices and emulators, we see the same code, but the behavior of the two devices when creating an execution sequence is different (Figure 5.3), making it difficult for us to distinguish between single execution sequences using just the thread ID.

Logcat Output (Left)	Logcat Output (Right)
34 Method END: Lcom/appbrain/c/b;:set(Lcom/appbrain/c/b;: (PID_0,TID_0)	34 Method END: Lcom/appbrain/c/b;:set(Lcom/appbrain/c/b;: (PID_0,TID_0)
35 Method START: Lcom/appbrain/c/g;:<<()I (PID_0,TID_0)	35 Method START: Lcom/appbrain/c/g;:<<()I (PID_0,TID_0)
36 Method END: Lcom/appbrain/c/g;:<<()I (PID_0,TID_0)	36 Method END: Lcom/appbrain/c/g;:<<()I (PID_0,TID_0)
37 Method START: Lcom/appbrain/c/g;:set(Ljava/lang/Runnable;:V (PID_0,TID_0)	37 Method START: Lcom/appbrain/c/g;:set(Ljava/lang/Runnable;:V (PID_0,TID_0)
38 Method START: Lcom/appbrain/c/g\$1;:newThread(Ljava/lang/Runnable;:Ljava/lang/Thread;: (PID_0,TID_0)	38 Method START: Lcom/appbrain/c/g\$1;:newThread(Ljava/lang/Runnable;:Ljava/lang/Thread;: (PID_0,TID_0)
39 Method END: Lcom/appbrain/c/g\$1;:newThread(Ljava/lang/Runnable;:Ljava/lang/Thread;: (PID_0,TID_0)	39 Method END: Lcom/appbrain/c/g\$1;:newThread(Ljava/lang/Runnable;:Ljava/lang/Thread;: (PID_0,TID_0)
40 Method END: Lcom/appbrain/c/g;:set(Ljava/lang/Runnable;:V (PID_0,TID_0)	40 Method END: Lcom/appbrain/c/g;:set(Ljava/lang/Runnable;:V (PID_0,TID_0)
41 Method END: Lcom/appbrain/c/f;:set(V (PID_0,TID_0)	41 Method END: Lcom/appbrain/c/f;:set(V (PID_0,TID_0)
42 Method START: Lcom/appbrain/c/g;:set(V (PID_0,TID_0)	42 Method START: Lcom/appbrain/c/g;:set(V (PID_0,TID_0)
43 Method START: Lcom/appbrain/c/f;:set(Landroid/content/Context;: (PID_0,TID_0)	43 Method START: Lcom/appbrain/c/f;:set(Landroid/content/Context;: (PID_0,TID_0)
44 Branch: Lcom/appbrain/f/i;:set(Landroid/content/Context;:>if-egg v0, :cond_0 (PID_0,TID_0)	44 Branch: Lcom/appbrain/f/i;:set(Landroid/content/Context;:>if-egg v0, :cond_0 (PID_0,TID_0)
45 Case: False (PID_0,TID_0)	45 Case: False (PID_0,TID_0)
46 Method END: Lcom/appbrain/c/f;:set(Landroid/content/Context;: (PID_0,TID_0)	46 Method START: Lcom/appbrain/c/g\$1;:run(V (PID_0,TID_0)
47 Method START: Lcom/appbrain/c/g;:set(Landroid/os/Handler;: (PID_0,TID_0)	47 Method END: Lcom/appbrain/c/f;:set(Landroid/content/Context;:>if-egg v0, :cond_0 (PID_0,TID_0)
48 Try: Start (PID_0,TID_0)	48 Method START: Lcom/appbrain/c/g;:set(Landroid/os/Handler;: (PID_0,TID_0)
49 Method START: Lcom/appbrain/c/f;:set(Landroid/content/Context;: (PID_0,TID_0)	49 Try: Start (PID_0,TID_0)
50 Branch: Lcom/appbrain/f/i;:set(Landroid/content/Context;:>if-egg v0, :cond_0 (PID_0,TID_0)	50 Method START: Lcom/appbrain/c/f;:set(Landroid/content/Context;: (PID_0,TID_0)
51 Case: False (PID_0,TID_0)	51 Branch: Lcom/appbrain/f/i;:set(Landroid/content/Context;:>if-egg v0, :cond_0 (PID_0,TID_0)
52 Method END: Lcom/appbrain/c/f;:set(Landroid/content/Context;: (PID_0,TID_0)	52 Case: False (PID_0,TID_0)
53 Method START: Lcom/appbrain/c/g\$1;:run(V (PID_0,TID_0)	53 Method END: Lcom/appbrain/c/f;:set(Landroid/content/Context;: (PID_0,TID_0)
54 Method START: Lcom/appbrain/c/f;:set(Lcom/appbrain/c/f;:Lcom/appbrain/c/g\$1;: (PID_0,TID_0)	54 Try: Done (PID_0,TID_0)
55 Method END: Lcom/appbrain/c/f;:set(Lcom/appbrain/c/f;:Lcom/appbrain/c/g\$1;: (PID_0,TID_0)	55 Method END: Lcom/appbrain/c/g;:set(Landroid/os/Handler;: (PID_0,TID_0)
56 Method START: Lcom/appbrain/c/g;:set(Landroid/content/Context;: (PID_0,TID_0)	56 Method END: Lcom/appbrain/c/g;:set(Landroid/os/Handler;: (PID_0,TID_0)
57 Method START: Lcom/appbrain/c/f;:set(Landroid/content/Context;: (PID_0,TID_0)	57 Method START: Lcom/appbrain/c/g;:set(Ljava/lang/String;:Ljava/lang/String;: (PID_0,TID_0)
58 Branch: Lcom/appbrain/f/i;:set(Landroid/content/Context;:>if-egg v0, :cond_0 (PID_0,TID_0)	58 Method END: Lcom/appbrain/c/f;:set(Landroid/content/Context;: (PID_0,TID_0)
59 Case: False (PID_0,TID_0)	59 Try: Done (PID_0,TID_0)
60 Try: Done (PID_0,TID_0)	60 Method START: Lcom/appbrain/c/f;:set(Landroid/content/Context;: (PID_0,TID_0)
61 Method END: Lcom/appbrain/c/f;:set(Landroid/os/Handler;: (PID_0,TID_0)	61 Method END: Lcom/appbrain/c/g;:set(Ljava/lang/String;:Ljava/lang/String;: (PID_0,TID_0)
62 Method END: Lcom/appbrain/c/f;:set(Landroid/content/Context;: (PID_0,TID_0)	62 Method END: Lcom/appbrain/c/g;:set(Ljava/lang/String;:Ljava/lang/String;: (PID_0,TID_0)
63 Method START: Lcom/appbrain/c/g;:set(Ljava/lang/String;:Ljava/lang/String;: (PID_0,TID_0)	63 Method START: Lcom/appbrain/c/f;:set(Landroid/content/Context;: (PID_0,TID_0)

Figure 5.3: We see different behaviors in whether threads are opened on emulators and physical devices, and the execution order is random



Therefore, during execution, we generate a random ID within each method that is reached and use it as a dynamic identification. By passing this ID between caller and callee, we can determine the dynamic call relationship and which method each monitoring point belongs to.

In order to minimize the cost of accessing registers, the method-scoped random ID is stored in a static variable within a class. This means that if two methods within the same class are used at the same time, a collision may occur. To avoid recursive calls or deep calls to methods within the same class in the same sequence, we store the random ID and method signature of the method in the free register within the original method scope before calling any function, and then write it back after the call is complete to avoid space overlap (Figure 5.4).

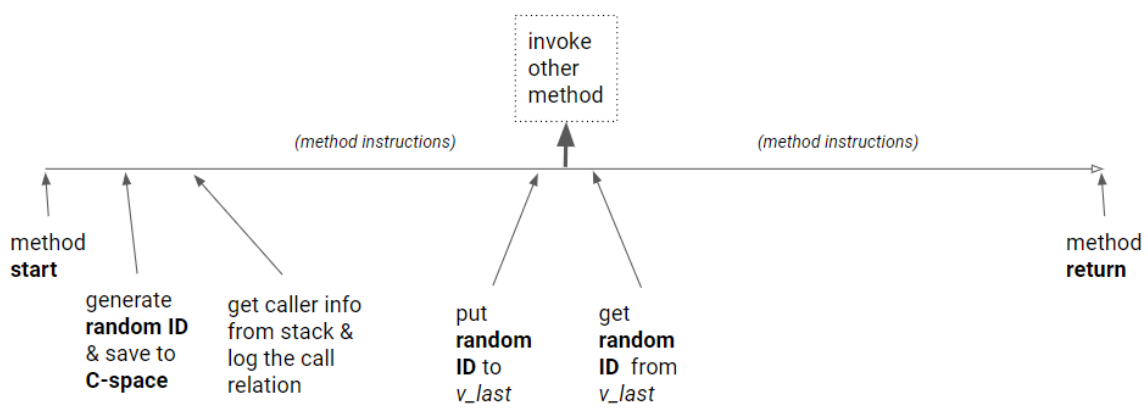
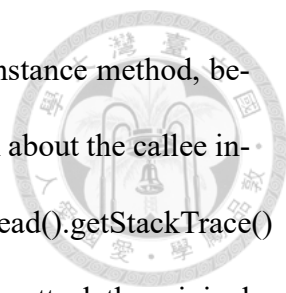


Figure 5.4: Illustrates how the random ID is stored within the method space to avoid being overwritten by any intermediate call sequence

5.1.4 Virtual Method Handling

Since we track inter-method call relationships by rewriting the caller and callee bytecode during static periods, this method has an inherent limitation, which is that we cannot accurately know all the callees at static periods, which will inevitably happen when calling virtual methods.



Although we cannot know in advance which class defines the instance method, because every method has been rewritten, it will still output information about the callee instance being called. At this point, we can use Java's `Thread.currentThread().getStackTrace()` to get the names of the called functions. Through this method, we can reattach the original output sequences that were broken due to calling virtual methods.

5.1.5 Logging Methods Implementation

In order to minimize the number of "invoke-" instruction's parameters as much as possible, we use specific wrapper functions for each point to be monitored (such as method start/end, "try-catch", "if-else", etc.), which contain the necessary information parsing and android's native "Log" function. All of these wrapper functions do not require any parameters (Figure 5.5).

Whenever a branch jumps, this wrapper function is called immediately after the branch tag label. Its function is to concatenate all the information used to identify this position, including the output type, label content, and random ID, and print it with a specific SADroid tag in the Android Log. This is the core method used in static-aided dynamic analysis. In SADroid, there are a total of 12 such wrapper functions, each of which is used to monitor different types of code positions, and these methods are defined in the C-space corresponding to each class.

5.1.6 Multidex Support

Dex (Dalvik Executable Format) is the executable file format for Android, and the number of methods in a dex file is limited to 65536. For some large-scale apps, they have

```

.method public static declared-synchronized branchTrueLog()V
.locals 2

new-instance v1, Ljava/lang/StringBuilder;

invoke-direct {v1}, Ljava/lang/StringBuilder;-><init>()V

const-string v0, "- Case True: "

invoke-virtual {v1, v0}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

move-result-object v1

sget-object v0, Lc/a/a/a/a/cNextDex;->branchTag:Ljava/lang/String;

invoke-virtual {v1, v0}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

move-result-object v1

sget-object v0, Lc/a/a/a/a/cNextDex;->thisMethodID:Ljava/lang/String;

invoke-virtual {v1, v0}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

move-result-object v1

invoke-virtual {v1}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;

move-result-object v1

const-string v0, "[SADroid TAG]"

invoke-static {v0, v1}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I

return-void
.end method

```

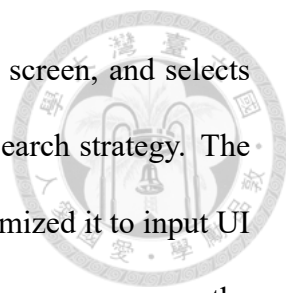
← get tag sign from C-space

Figure 5.5: SADroid’s actual logging method for monitoring branch instructions, which calls a function immediately after the label where the branch instruction jumps. There are also corresponding logging methods for other monitoring points.

to split the file into more than one dex file. As mentioned earlier, during static analysis, for each original class, we create a new C-space containing many new wrapper functions and other required functions, which may not be supported by the original dex capacity of the app. Therefore, we place all these additional C-spaces in a new dex file, and if one is not enough, we open a second one.

5.2 Two-Device UI Fuzzer

Droidbot [15] is a python-based, extensible, and cross-platform Android app UI input generator (UI fuzzer) that can automatically test UI for Android apps. It reads the screen



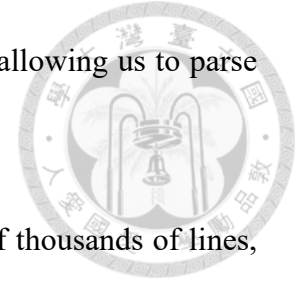
content of each device, examines the available UI operations on the screen, and selects the UI operation commands (UI events) to be sent based on the set search strategy. The original Droidbot could only be used to test a single app, so we customized it to input UI events on different ADB devices using the same search strategy. This way, we can use the same graph search strategy to execute an app on both a real device and an emulator, and compare the execution tracks. Then, we filter out all the logs generated by the app being tested through ADB Logcat's filter tool and output them to a file for later use.

We chose Droidbot for its ease of use, which allows for quick and easy deployment and start of automated UI testing, and because it allows users to avoid the low-level details of interacting with the phone and provides better dynamic analysis performance than the traditional Android built-in monkey tool.

5.3 Log Sequence Analyzer

During dynamic analysis, SADroid collects the execution traces of the app, which are output as lines of log. However, not all logs are part of the same call sequence. In addition to the almost universal presence of multi-threaded execution in apps, there are also cases where different call sequences are interleaved within a single thread due to the initialization of various classes and objects. Therefore, it is necessary to split the original log output into multiple single execution sequences, where each line of the sequence is part of the same call sequence, for further parsing. The main basis for this splitting is the dynamic random ID that we embed in the code beforehand. A random ID is generated at the runtime of a method and maintained until the method returns. When a method calls another method, in addition to the callee generating its own random ID, the relationship

between the two random IDs is also output through a log function, allowing us to parse the dynamic call sequence smoothly in the future.



Log data generated by each app may range from tens to tens of thousands of lines, where each line is in the format of (location type, location signature, random ID). Location type includes the start of the first line of instruction before a method, before each return in a method, before a branch instruction and after a jump, the normal start and end of a "try-catch" and after jumping to the exception segment, and the point of loop jump. The location signature includes the information needed to identify the location, such as the caller and callee method random IDs in the log function at the start of the method, and the information on whether it is the true case or the false case in the log function after a branch instruction. Each line of log must include the corresponding method signature of this method, which is a string concatenated by the (method name, method parameter type, method return type). The method signature can uniquely determine a specific method. The random ID is generated before printing the call relationship at the start of the method and maintained during the execution of the method (refer to Chapter 5-1-3). It allows all log functions generated within this method to share it, so each random ID corresponds to a specific method signature.

While reading the original log data line by line, a calling set is maintained to collect the methods that are currently being called (stored using random IDs). Whenever a call relationship is encountered, the calling set is traversed to check if the caller's ID exists. If it does, the ID is replaced with the callee's ID in the set. If not, it indicates the start of a new single execution sequence, and the callee's ID is added to the calling set. This happens when the execution reaches the entry points. Whenever a method return occurs, the ID is removed from the calling set. Other types of logs are for monitoring the execution trace

within the method and do not affect the log analysis process in principle. They just need to be assigned to different single execution sequences according to the ID.

The efficiency of the SADroid algorithm is quite high, with a capability of parsing over $400K \sim 500K$ lines of original log outputs into various single execution sequences per second, while also handling exceptions. Exception cases will be discussed further in chapter 7-2.

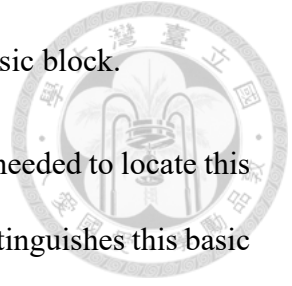
5.4 Dynamic Control Flow Passing Counter

After obtaining a large number of single execution sequences that reflect the dynamic execution of the app, each sequence can be used to depict the corresponding Dynamic Control Flow Graph (DCFG) because it contains information on method call relationships and control flow execution traces. Since each single execution sequence includes the signatures of all the basic blocks that were executed, we can use this information to count the number of times each control flow path in the CFG of a method is executed. This is the main task of the Dynamic Control Flow Passing Counter (DCFPC).

Intuitively, if a basic block is an evasion candidate, there should be a branch divergence after this block is executed on a physical device and emulator, meaning that the physical device tends to follow one side of the basic blocks while the emulator follows the other side, causing a significant difference in the number of times the next block is executed on different devices. In the DCFPC, we use 0.99 as a proportionate threshold. If a device has more than 99% of its execution times going through one side of the DCFG and the other side is similar, we consider this to be a significant difference. Therefore, as long as we observe such a basic block that meets these conditions on the DCFG, we

consider it an evasion candidate and output the information of this basic block.

The information of the evasion candidate includes the signature needed to locate this basic block, which includes the method it belongs to, the label that distinguishes this basic block, and the child blocks pointed to after the branch. According to the definition of the evasion candidate, the physical phone and emulator will execute one of the child blocks respectively, which will contain information on whether the target API exists, helping us to understand what an app is doing with evasion.





Chapter 6 Evaluation

We will answer three research questions, including some samples to illustrate the findings.

6.1 Experiment Environments And Dataset Collection

The main experimental devices are a SAMSUNG Galaxy J6+ with Android API 29 and the emulator (API 29-30) in Android Studio, which are connected to a Win11 host through adb. The data was downloaded from the official website of AndroZoo [11], which is currently the largest online app database and is continuously collected. Almost all the data used in previous papers can be downloaded from AndroZoo through the sha256 of the APK.

Based on the need for the host environment to run x86 emulators, we selected 373 out of 404 apps from TriggerZoo [18] that can run on the x86 platform for experiments. TriggerZoo is a dataset containing various logic bombs, which were written into the apps through automated probing by the authors of the paper. The types of logic bombs include time, location, text message, network, android.os.BUILD properties, camera, music, and screen on/off. Each logic bomb includes a specific type of behavior, some of which are classified as malicious. We expect SADroid to execute some of these logic bombs and

output those that can be used for emulator detection. TriggerZoo was selected because it is one of the few open-source datasets with tags related to emulator detection and includes malicious tags. In addition, there is a small annotated dataset, evadroid [13], consisting of 22 evasive apps, but since each app has only one main function with an evasion and no normal functionality, we believe that the results obtained from such data are insufficient to extend to general apps and do not need to be used.

6.2 Sensitive evasion in Our Dataset

In response to our research question, **RQ1**: Is there evasion of emulator detection in Android malware, and if so, which code is causing the evasion? In our dynamic analysis, we examined the TriggerZoo dataset and found a total of about 100 evasion candidates, of which 24 were determined to be sensitive evasions after manual review (Table 6.1). Out of these sensitive evasions, we discovered 2 samples that may not be related to emulator detection, as they are based on the values of the method’s parameters to determine the direction of execution. It is difficult to determine whether these blocks are based on emulator detection results.

Table 6.1: Results from the entire dataset

# of total evasion candidates	# of sensitive evasion	Actual anti-emulator ratio by manual verification
~ 100	24	87.5% (21)

In regard to labeling, 123 labels out of 349 apps were successfully executed during dynamic analysis, with 10 of them being evasion candidates, including 9 sensitive evasions (Table 6.2). One sample was found to be an emulator detection chill block with no code included. As we know the content of these labels to be emulator environment detection, the precision of the labeled dataset is 100%. In addition to the labeled sensitive evasions,

the rest were discovered through additional exploration within the dataset by SADroid, demonstrating its ability to find unknown evasions.



Table 6.2: Results from the logic bomb labels

# of total triggered label	# of evasion candidates	# of sensitive evasion	Actual anti-emulator ratio
123	10	9	100%

(Table 4: Statistics of output related to labels)

Regarding labels that are not triggered after multiple rounds of automated testing, our interpretation is that these may belong to code sections that are not easily or will not be executed, even if these sections contain dynamic analysis protection, they may not have much opportunity to play, so as an analyzer, it is not necessary to dig them out one by one, shallow evasions have a greater impact on automated dynamic analysis.

Although dynamic analysis based on random UI testing cannot guarantee that all code sections will be executed, as the number of executed events increases, we observe the opportunity for increasing the number of unique evasion candidates. We believe that given enough time for repeated fuzzing, we will inevitably find more and more evasion candidates and sensitive evasions.

6.3 Discovery of New Evasion Types

In response to our research question, **RQ2**: Are there any types of evasions that have not been discovered in previous studies? In past research, emulator detection was often identified through the examination of device properties such as Android.os.Build series, trace-related attributes such as contact lists, and the presence of the QEMU emulator. In addition, we found that there are still many different types of evasion features that are

easily ignored by traditional static methods, such as: (1) Built-in date format. (2) Mobile network and telecommunications information. (3) Determining whether music is in an active state. (4) Screen lock usually only occurs in physical mobile phone environments, and mobile phones can still perform app functions while in lock screen state. (5) Different network connection states, generally speaking, emulators are always connected to the PC's virtual network, while the network of physical mobile phones may be disconnected. (6) Obtaining the maximum frame rate of the camera device. (7) Battery changes, emulators do not produce battery changes, but physical mobile phones have this feature. (8) The binary file existence of the "su" instruction. Refer to Table 6.3 for a comparison of the differences between our physical device and emulator in these attributes.

Table 6.3: Conduct actual testing on these attributes on both physical device and emulator

<i>Anti-Emulator Patterns</i>	<i>On Our Real Device</i>	<i>On Android Studio Emulators</i>
API: <code>is24HourFormat()</code>	Always True	Always False(NULL)
API: <code>getNetworkOperatorName()</code>	Always NULL	Always "Android"
API: <code>isMusicActive()</code>	Always False	Always True
API: <code>isScreenOn()</code>	sometimes False	Always True
API: <code>getActiveNetworkInfo()</code>	sometimes NULL	Always False(NULL)
API: <code>getOutputMinFrameDuration()</code>	Always 0	Always !=0
Intent: BATTERY_CHANGED	Always Status!=2	Always Status=2
File: "su" binary existence	Always False	Always True

It is foreseeable that as we delve deeper, the likelihood of finding evasions increases and these possibilities are not easily identified through black list based mechanisms. Therefore, we believe that SADroid can help analysts understand more evasion behaviors in programs.



6.4 Actual Sensitive Behavior

In response to our research question, **RQ3**: What are these anti-analysis evasions being used to conceal in terms of app behavior? We classified the sensitive evasions we found according to the sensitive behavior they contained (Figure 6.1).

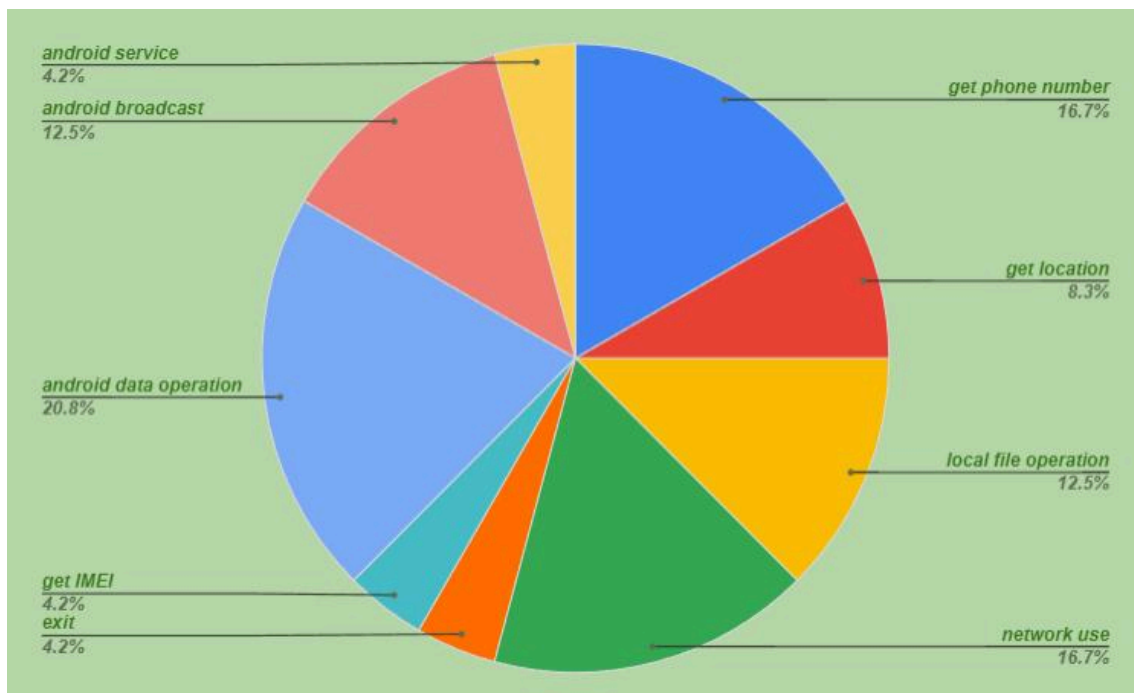


Figure 6.1: A breakdown of the sensitive behaviors observed in the sensitive evasions detected by SADroid in our dataset.

For example, in the "get IMEI" category, the actual code is shown in Figure 6.2:

```
if (((PowerManager)object.getSystemService("power")).isScreenOn()) return;
object = ((TelephonyManager)object.getSystemService("phone")).getImei();
SmsManager.getDefault().sendTextMessage("+33639980000", null, (String)object, null, null);
```

Figure 6.2: An actual behavior of a sensitive evasion, shown as a pseudo-code in a de-compilation tool

This code functions to send the IMEI number through a text message if the screen is locked. The IMEI number is a unique identification number for a mobile device and can be used to identify a specific mobile device. Because emulators do not lock their screens, checking the emulator environment and then executing a sensitive behavior that

sends personal information out is a standard sensitive evasion and is what we aim to find as an anti-emulator based evasion.



6.5 Bytecode Instrumentation Failure

During the process of bytecode instrumentation, on average it takes around one minute to complete the process for each APK file and generate a fully rewritten APK file. However, among the 373 APK files in the dataset, 27 APK files crashed upon execution after bytecode instrumentation. Upon further investigation, we found that 24 of these were APK files that were originally unable to run properly, a problem that was present in the TriggerZoo dataset. The remaining 3 crashed due to errors introduced by the bytecode instrumentation process. As mentioned earlier, there is a small possibility of encountering syntax errors, which is an inevitable result of automated modification. However, considering the high level of convenience in usage, the error rate is within an acceptable range.

6.6 Case Studies

6.6.1 True Positive And True Negative

Let's give an sample to illustrate the appearance of True Positive and True Negative in code, as shown in Figure 6.3:

SADroid output the decision statement in line 948, which checks for Bluetooth devices, as a sensitive evasion, causing a divergence in behavior between physical and emulated devices. This is a True Positive as the physical and emulated devices often have significant differences in hardware status, making it a clear anti-analysis evasion.

```
com.mobilmob/mncconfig/TncConfig.class x
FernFlowr Decompiler
onCreate
931
932 public void onAudioOutputDialogTwistLevelChanged(AudioOutputFragment var1) {
933     int var2 = var1.getOutputTwist();
934     this.mOutputTwist = var2;
935     this.mTncService.outputTwist(var2);
936 }
937
938 protected void onCreate(Bundle var1) {
939     super.onCreate(var1);
940     if (VERSION.SDK_INT >= 11) {
941         this.getWindow().requestFeature(8);
942     }
943
944     this.setContentView(2131296286);
945     Log.e("TncConfig", "+++ ON CREATE +++");
946     BluetoothAdapter var2 = BluetoothAdapter.getDefaultAdapter();
947     this.mBluetoothAdapter = var2;
948     if (var2 == null) {
949         Toast.makeText(this, 2131427472, 1).show();
950         this.finish();
951     } else {
952         this.mBluetoothDeviceView = (TextView)this.findViewById(2131165227);
953         this.mFirmwareVersionView = (TextView)this.findViewById(2131165266);
954     }
955 }
```

Figure 6.3: A sample of True Positive and True Negative in code.

However, in line 940, the program actually executes a decision statement that does not produce any divergence. After reverse engineering, it is easy to see that it checks the android.os.BUILD version's SDK_INT to determine the execution environment and attempts to filter out certain environments. This check is ineffective for recent versions of emulators, making it a True Negative for SADroid.

In apps with known environment attribute checks, many such cases are found where the attributes checked are no longer able to distinguish between emulators and physical devices on updated versions of Android Studio emulators. Therefore, SADroid does not output these as evasion candidates and only discovers these True Negatives when we already know that the labels have been executed.



6.6.2 "Try-Catch" as An Evasion

In addition to using branching to evade detection, in some cases it is also possible to use "try-catch" blocks to achieve evasion. For example, in the case shown in the Figure 6.4, a try block is used to catch other packages that may exist on the phone in order to execute a function of a certain app. These packages may not exist in the emulator, and by using a "try-catch" block in this way, it is possible to evade the emulator and jump to the exception handling branch, resulting in a difference in log sequences between the emulator and the physical phone.

```
private String t() {  
    try {  
        String installerPackageName = l.a().getPackageManager().getInstallerPackageName(this.o);  
        return installerPackageName != null ? installerPackageName : "";  
    } catch (Exception e) {  
        return "";  
    }  
}
```

Figure 6.4: A special case of using a "try-catch" block to differentiate between a physical device and an emulator

This technique of evasion has not been mentioned in previous research, thus it is not possible to find such evasion through static code analysis.

6.6.3 Weird Failed Instrumentation Sample

There are a small number of samples in which it is not possible to successfully perform automated rewriting and repackaging of bytecode instrumentation into a runnable APK. This is usually due to the inclusion of rare syntax, as shown in the Figure 6.5 below:

Normally, if a return value is expected after an "invoke-" instruction, a "move-result-" instruction must follow. This case is unusual in that tags are interspersed between these instructions. Tags do not belong to the instructions themselves, but are generated during


```
55005 invoke-virtual {p0}, Lmph/dexprotect/a/a;->getContentResolver()Landroid/content/ContentResolver;
55006
55007 move-result-object v0
55008
55009 const-string v1, "android_id"
55010
55011 invoke-static {v0, v1}, Landroid/provider/Settings$Secure;->getString(Landroid/content/ContentResolver;Ljava/lang/String;)Ljava/lang/String;
55012
55013 :try_end_0
55014 .catch Ljava/lang/Exception; {:try_start_0 .. :try_end_0} :catch_0
55015
55016 move-result-object v0
```

↳ **Unsupported syntax in SADroid**

Figure 6.5: Rare Smali syntax in which tags are interspersed between the "invoke-" and "move-result-" instructions, causing existing bytecode instrumentation algorithms to fail

the process of compiling to Smali in order to distinguish between code segments. However, when SADroid encounters these tags during the bytecode instrumentation stage, it generates the log function required for monitoring, breaking the connection between the "invoke-" and "move-result-" instructions and causing the packaging to fail.



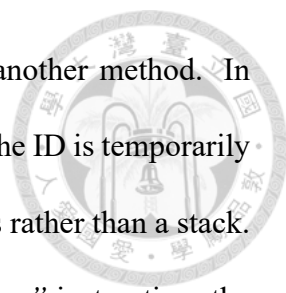


Chapter 7 Discussion And Limitation:

7.1 What Cause The Instrumentation Fault

Recalling the actions taken by the Instrumentation process, a local register was added to every method as a free register for general use. The most direct approach is to increase the number of locals defined in the method by 1, in which case the free register will appear at the end of the locals register, which we will refer to as v_{last} . This approach requires the least amount of code modification.

However, this change causes the parameters registers that were originally numbered after v_{last} to shift by one number. It is easy to see that the most direct result of this is that the original register number 16 ($v15$) may be moved to number 17 ($v16$) in a few cases (if the added number is before 16). Most $v16$ problems can be solved using the design described in Chapter 5-1-2, but there are still a few cases that are difficult to handle. One of these is when $v14-v15$ are wide types and need to be checked for invalid instructions. In cases where only $v15$ is invalid, as described in Chapter 5-1-2, we can use the free register as a temporary storage area to swap the data and rewrite the instructions. However, when $v14-v15$ are bound together by wide type data, it is almost impossible to move them, and the invalid instructions resulting from this will cause the app to crash after repackaging.



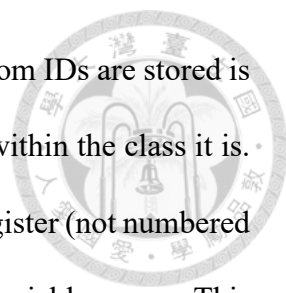
Another issue is when a self-defined method is called within another method. In order to maintain the original method ID within the original method, the ID is temporarily stored in v_{last} before the call, as Dalvik bytecode is based on registers rather than a stack. However, when a parameter of v_{16} is encountered during an *invoke-* instruction, the data in v_{16} must be moved to a free register numbered 16 or lower in order to perform the *invoke-* instruction. It is usually the case that the free register is v_{last} , which can lead to conflicts.

By the way, when an *invoke-range* includes the range of parameters registers following the SADroid register in the locals register, adding an extra locals register will directly break the parameter arrangement of the *invoke-range* and require the addition of a large amount of data movement instructions. This type of case is not particularly rare. While this problem can be solved by moving the parameter data extensively, it incurs a significant performance cost in terms of additional instructions.

The author of Smali has provided a response and discussion [8] on these issues. The conclusion is that it is difficult and not guaranteed to always have a free register available, and the difficulty and complexity of the modification increases exponentially if more than one free register is desired.

7.2 Main Cause of The Exception Line During Log Sequence Analysis

During the Log Sequence Analysis stage, we assign each line of log output to a different single execution sequence based on its random ID. However, it is not always possible to do this accurately, as some logs cannot be identified as belonging to a specific sequence,



known as exception lines. This is because the space where these random IDs are stored is a specific static variable within a class, regardless of which method within the class it is. This is also done to save on the use of free registers, as any one free register (not numbered 16 or lower) can be used to retrieve the random ID from the fixed variable space. This means that if, at the same time point, methods within this class are being invoked concurrently, the spaces allocated for their respective IDs will overlap. In fact, this phenomenon is quite common in practice and can be observed in most apps. When this happens, we consider these logs to be exception lines and filter them out.

There may be potential solutions to address issues with multi-sequence overlapping (where it is not allowed to simultaneously read or write variables within the same class), such as the one described. For example, we could leverage data structures like HashTable to set up exclusive variable spaces for each sequence (or random ID). However, methods like these require two free registers, which is another challenge. Therefore, in order to minimize the negative side effects of changes, we will reserve the improvement of this issue for future work.

Overall, although the existing methods exhibit very high computational efficiency, they still have the potential for some errors. After weighing the trade-offs, although it may not be possible to achieve 100% accurate capture of all dynamic traces, we have struck a balance in terms of implementation complexity and efficiency, and are still able to output a considerable number of evasion candidates and sensitive evasions. We believe that we have preliminarily achieved our research objectives.

7.3 Static Uncertainty: Virtual Methods Issues And Reflection



There are also inherent issues in the SADroid method. What we are actually concerned with is the dynamic analysis trace and results, and in order to capture this information, we try to set up monitoring points using static analysis. However, during static analysis, it is not possible to fully grasp all the crucial information and we need to pass the caller's random ID to the callee method's C-space when calling the method. However, when invoking a virtual/interface method, it is not possible to directly identify the instance of the virtual/interface method from the caller's invocation information, which leads to the inability to successfully pass on the information to the callee, who is only identified in the dynamic phase. As described in Chapter 5-1-4, in such cases, we use the stack information to deduce the call relationship of the origin log sequence. In theory, this can perfectly solve the problem of virtual methods, but in reality, `Thread.currentThread().getStackTrace()` can only obtain the function name and not the parameter format, which may cause difficulty in determining which of the overloaded methods is being called when multiple sequences call them simultaneously. Such cases were also observed in the experimental process.

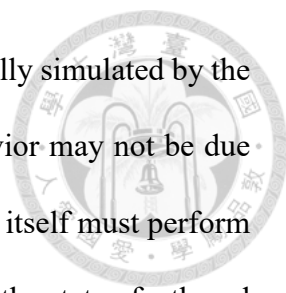
Reflection is a language feature in Java that allows for the determination of which class's method to call at runtime, which can lead to issues in determining the call relationship during static analysis, similar to the problem of virtual methods mentioned earlier. These are also potential causes for exceptions in log analysis.

7.4 All Evasion Candidates Results And Cause of Anti-Emulator FNs



In addition to being concerned with the results of sensitive evasions, we were also curious about whether other evasion candidates included emulator detection. Based on manual reverse engineering, we classified the evasion candidates into three categories: anti-emulator, not anti-emulator, and unknown. Anti-emulator means that the evasion candidate is indeed based on attributes that can differentiate between a physical device and an emulator environment, and can identify the emulator environment and lead to a difference in execution branches. These are False Negative Cases output by SADroid. There are generally two reasons for the existence of such cases. One is Delayed Sensitive Behavior, which means that after the anti-emulator check is executed, the program does not immediately perform some critical behavior, but rather performs it deeper down (such as calling a self-defined method containing malicious behavior). This causes SADroid to be unable to classify the evasion candidate as a sensitive evasion. The other is Not Listed Sensitive Behavior, which means that our sensitive API list is based on a collection of target API lists [15, 19, 23] and may not always be fully updated in the API version, resulting in the failure to detect the execution of sensitive behavior.

Not anti-emulator means that the evasion candidate is not designed to detect the emulator environment, and may simply perform different functions based on the configuration or state of the phone. It is possible to occur on both physical devices and emulators, but may be wrongly classified due to insufficient statistical quantity or coincidence. Some differences in behavior may occur due to the differences in display or current settings on the two devices, but the true cause may have nothing to do with the emulator environment.

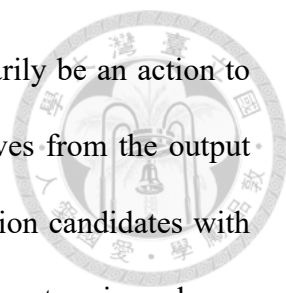


For example, the Samsung phone used in the experiment cannot be fully simulated by the commonly used emulator on the market, and the difference in behavior may not be due to emulator detection. In addition, sometimes the program's function itself must perform different functions based on the specific environmental state, such as the state of a thread synchronization lock or the arrival of network packet contents, which may cause differences in branch results in dynamic analysis. These checks on the environment that do not determine the existence of an emulator are called Neutral Environmental Checks.

Unknown means that it is not possible to determine whether it is related to evasion at the moment, and the cause of the difference in execution may be based on the value of certain class variables or static variables, or the parameters of the method. In the vast majority of cases where app code is obfuscated, it is often not possible to simply analyze the method through static analysis and deduce the values of variables and parameters. In such cases, we can only rely on dynamic analysis to understand the function of the method and the cause of the behavior divergence. There are many such cases in the experimental results, and it is difficult to accurately classify them.

7.5 Cause of Anti-Emulator FPs: UI/Phone Style Difference And Neutral Environmental Checks And Unknowns

In the process of manually reviewing the output of the model, it was discovered that a small number of sensitive evasions may not have been intended to be used for emulator detection. It is possible that the same "Not anti-emulator" or "Unknown" situations described earlier may also occur. The reason for the presence of such situations in sensitive evasions may be that the sensitive behavior list includes some neutral APIs, causing




SADroid to classify it as a sensitive evasion, but it may not necessarily be an action to detect an emulator. To further filter out these potential false positives from the output of sensitive evasions, one feasible approach is to test the same evasion candidates with more physical mobile devices and emulators, and to conduct experiments using a larger number of devices to determine whether each case will indeed result in differences between physical devices and emulators. However, this would lead to a substantial increase in experimental costs.

To further filter out potential false positives from the output of sensitive evasions, a feasible approach is to test the same evasion candidates on a larger number of physical devices and emulators to demonstrate that each case actually causes a difference between physical devices and emulators. However, this increases the exponential cost of the experiment.

7.6 Limitations - Code Coverage / Not Supported Apps

Overall, while SADroid is able to address the main issues of traditional static and dynamic approaches, both false positives and false negatives still have some difficult-to-completely-resolved cases, some of which can be further refined and others that are limitations of the methodology or tool.

Regarding the dynamic analysis component, SADroid uses automated droidbot to test and does not provide much human guidance. The primary concern with this dynamic analysis approach is whether it is possible to reach the location of the evasion. In previous experiments, anti-emulator code that was reached was able to fully distinguish between physical devices and emulator environments during dynamic execution, but was unable



to do so for unreached locations. This problem is more pronounced for apps that require heavy user interaction, such as games or those requiring login. This issue is reserved for future work, and can potentially be addressed by replacing droidbot or further customizing it to increase its exploration ability, or by using a target input generator process to try to trigger more areas that need analysis.

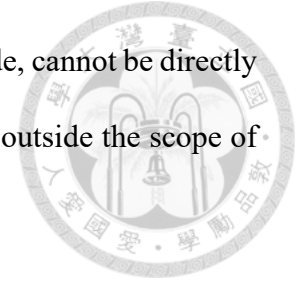
During the dynamic analysis experiments, it was also observed that some apps produced very few log lines, partly because there are many apps that require heavy user interaction to function properly, including registration, login, and filling out forms. These types of apps are very difficult to understand through automated analysis.

Dynamic analysis using the Droidbot open-source project utilizes the minicap tool, which only supports Android API 30 (Android 11). Subsequently, if you want to run it on a current version of Android, you will need to use a different tool. Additionally, the current testing environment is on an x86 PC, which makes it difficult to run an ARM-based emulator. This means that about 5% of apps that cannot run on an x86 emulator are currently excluded. Similar limitations with the tools used can also cause some apps to be unable to be analyzed by SADroid.

In static analysis, the full automation required inevitably sacrifices some success rate due to syntax limitations in the bytecode. For instance, the switch-case multiple branch syntax is currently not supported. This is because the position of code jumps after bytecode instrumentation is not easily calculated based on the offset in the bytecode, making it more difficult to inject code.

In addition, SADroid's dynamic analysis relies on static analysis and modification of the bytecode to be successfully executed. As a result, APKs that have been obfuscated or

tampered with, as well as those with all functions hidden in native code, cannot be directly analyzed using SADroid and require other countermeasures that are outside the scope of this research.



Due to these inherent limitations of the SADroid method, as well as implementation challenges, it may be unable to analyze some cases and may cause unknown false negatives (FN). In theory, we cannot guarantee complete analysis and confirmation of a specific result for any program, so we cannot actually calculate the rate of FN, which is a common limitation of code analysis.





Chapter 8 Future Work

In the dynamic analysis part, it would be worthwhile to investigate whether using a different dynamic analysis engine can effectively increase code coverage. For example, fastbot, recently released by Bytedance, claims to significantly outperform older tools such as droidbot and monkey in terms of analysis results. For input screens that automated testing cannot pass, we can try combining manual input or further reversing to bypass these difficult screens and directly reach the core functions.

In the static part, We consider it is essential to have a deeper understanding of the Dalvik bytecode syntax and the syntax compiled by the Smali tool, and there may still be opportunities to find an implementation that is both more complete and less modifying. Currently, SADroid does not fully support all Dalvik bytecode and dex syntax, so we can seek a more stable and accurate insertion modification algorithm to try to avoid the possibility of log analysis errors. In general, there is still a lot of room for improvement in the static analysis part.

Currently, the number of apps and the average size of apps in the TriggerZoo dataset are relatively small (about 3-15KB). A key to scalability to larger datasets is still performance. Both bytecode instrumentation and UI fuzzing have a lot of room for optimization, and this will inevitably be a challenge if we want to apply SADroid to a larger dataset of

apps in the future.





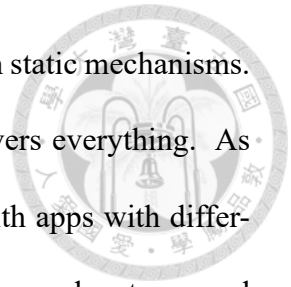
Chapter 9 Conclusion

In the research of malware, various evasions are a major obstacle for reverse analysts. In order to assist them in better understanding the behavior of malware, we attempt to develop an approach for anti-emulator-based evasion to provide new insights into analyzing these heavily protected programs. We propose the SADroid model, which quickly analyzes APK files and outputs evasion candidates and sensitive evasions found in the code to facilitate further research by reverse analysts.

To achieve the goal of comprehensive and detailed monitoring of program execution while not affecting the system, SADroid uses a static-aided dynamic analysis approach. By deeply understanding the syntax of dalvik/smali bytecode and addressing the output issues that must be faced in dynamic analysis, it aims to find emulator detection that is traditionally difficult to discover and locate. To the best of our knowledge, the completeness of the bytecode instrumentation function is a trade-off with the complexity of its modification, and the more modification there is, the greater the side effects, resulting in a higher probability of the rewritten app crashing directly. Currently, there are no known concise and efficient bytecode instrumentation algorithms, making some concessions in both areas seems to be more practical.

In the experimental results, we also noticed that the types of evasion found by this

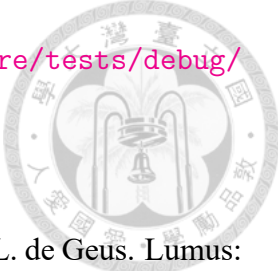
approach differ somewhat from those found by traditional mainstream static mechanisms. For research problems like this, there is no perfect solution that covers everything. As an analyst, you will need to use different analysis designs to deal with apps with different protection measures, and different designs can each have their own advantages and disadvantages, complementing each other.

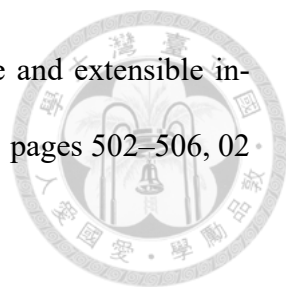




References

- [1] androiddeveloper. <https://developer.android.com/reference>.
- [2] Anti android emulator detection final report. <https://i.cs.hku.hk/fyp/2018/fyp18033/data/final.pdf>.
- [3] apktool. <https://ibotpeaches.github.io/Apktool/>.
- [4] Dalvik bytecode. <https://source.android.com/docs/core/runtime/Dalvik-bytecode>.
- [5] detectandroidevasion. https://hitcon.org/2014/downloads/P1_12_%E8%83%A1%E6%96%87%E5%90%9B%20-%20Guess%20Where%20I%20am-Android%E6%A8%A1%E6%8B%9F%E5%99%A8%E8%BA%B2%E9%81%BF%E7%9A%84%E6%A3%80%E6%B5%8B%E4%B8%8E%E5%BA%94%E5%AF%B9.pdf.
- [6] Mobile security testing guide. <https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05j-testing-resiliency-against-reverse-engineering>.
- [7] Smali. <https://github.com/JesusFreke/Smali>.
- [8] smaliregisters. <https://stackoverflow.com/questions/27341565/Smali-increase-number-of-registers>.

- 
- [9] strace. <https://source.android.google.cn/docs/core/tests/debug/strace>.
- [10] V. Afonso, A. Kalysch, T. Müller, D. Oliveira, A. Grégio, and P. L. de Geus. Lumus: Dynamically uncovering evasive android applications. In L. Chen, M. Manulis, and S. Schneider, editors, Information Security, pages 47–66, Cham, 2018. Springer International Publishing.
- [11] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon. Androzoo: Collecting millions of android apps for the research community. In 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pages 468–471, 2016.
- [12] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer. Emulator vs real phone: Android malware detection using machine learning. In Proceedings of the 3rd ACM on International Workshop on Security And Privacy Analytics, IWSPA '17, page 65–72, New York, NY, USA, 2017. Association for Computing Machinery.
- [13] L. Bello and M. Pistoia. Ares: Triggering payload of evasive android malware. In 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pages 2–12, 2018.
- [14] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. pages 377–396, 05 2016.
- [15] Y. Li, Z. Yang, Y. Guo, and X. Chen. Droidbot: a lightweight ui-guided test input generator for android. In 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pages 23–26, 2017.

- 
- [16] J. Liu, T. Wu, X. Deng, J. Yan, and J. Zhang. Insdal: A safe and extensible instrumentation tool on dalvik byte-code for android applications. pages 502–506, 02 2017.
- [17] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of android malware. In Proceedings of the Seventh European Workshop on System Security, EuroSec '14, New York, NY, USA, 2014. Association for Computing Machinery.
- [18] J. Samhi, T. F. Bissyande, and J. Klein. Triggerzoo: A dataset of android applications automatically infected with logic bombs. In 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), pages 459–463, Los Alamitos, CA, USA, may 2022. IEEE Computer Society.
- [19] J. Samhi, L. Li, T. F. Bissyande, and J. Klein. Difuzer: Uncovering suspicious hidden sensitive operations in android apps. In 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), pages 723–735, Los Alamitos, CA, USA, may 2022. IEEE Computer Society.
- [20] V. Sihag, M. Vardhan, and P. Singh. A survey of android application and malware hardening. Computer Science Review, 39:100365, 2021.
- [21] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, page 447 – 458, New York, NY, USA, 2014. Association for Computing Machinery.
- [22] X. Wang, S. Zhu, D. Zhou, and Y. Yang. Droid-antirm: Taming control flow anti-analysis to support automated dynamic analysis of android malware. In Proceedings

of the 33rd Annual Computer Security Applications Conference, ACSAC '17, page
350–361, New York, NY, USA, 2017. Association for Computing Machinery.

- [23] M. Wong and D. Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. 01 2016.

