A STUDY OF SELECTED ISSUES IN ANDROID SECURITY

A Dissertation in
Computer Science and Engineering
by
Chuangang Ren

© 2016 Chuangang Ren

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2016
The dissertation of Chuangang Ren was reviewed and approved* by the following:

Peng Liu  
Professor of College of Information Sciences and Technology  
Co-Chair of Committee

Sencun Zhu  
Associate Professor of Computer Science and Engineering  
Dissertation Advisor, Co-Chair of Committee

Wang-Chien Lee  
Associate Professor of Computer Science and Engineering

Minghui Zhu  
Assistant Professor of Electrical Engineering

Chitaranjan Das  
Professor of Computer Science and Engineering  
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.
Abstract

Mobile devices such as smartphones have become an integral part of society today, shaping people’s daily life, changing the landscape of how business operate and how industries are powered today. However, the unprecedented popularity of mobile devices introduces a concerning side effect, a dramatically increasing number of security threats is posing serious risks to the security of mobile systems and the applications. Notably, one of the most successful mobile systems, Android, has exposed a plethora of vulnerabilities which are actively exploited by a large number of potentially harmful apps (malware, adwares, risk-wares, etc.), most of which are distributed in under-scrutinized third-party Android markets.

Mitigating the security threats to Android is non-trivial. There has been considerable efforts in securing Android to achieve the following two complimentary goals: (1) Market-scale detection and identification of problematic apps in an efficient manner, and (2) discovery, analysis of the Android system vulnerabilities, and defense measures against the enabled attacks.

In this dissertation, we propose techniques and approaches to solve specific problems in the above two aspects respectively, providing the step stones towards finally achieving these two goals. Specifically, we first present a novel software watermarking scheme, namely Droidmarking, that can efficiently and effectively impede the prevalent software plagiarism (a.k.a app repackaging) problem in the Android markets. Second, we systematically study and propose a new prevalent Android system vulnerability, which, once exploited by an attacker, can lead to serious security breaches of integrity, confidentiality and availability of the graphic user interface (GUI) on an Android device. Finally, we devise a comprehensive and practical solution to protect the GUI sub-system in Android. The defense is able to defeat all know GUI attacks while preserving the original user experience of Android. We plan to further explore Android system and app security towards a more secure ecosystem for Android.
Table of Contents

List of Figures viii

List of Tables x

Chapter 1
Introduction 1
1.1 Motivation ............................................. 1
1.2 Contributions ......................................... 3
1.3 Outline .............................................. 5

Chapter 2
Android Security Background 7
2.1 System Model .......................................... 7
2.2 Android Security Models ............................... 9
   2.2.1 Android Application Sandbox .................... 10
   2.2.2 Android Permission System ....................... 10
2.3 Android Markets ..................................... 11

Chapter 3
Related Work 13
3.1 Software Plagarism and App Repackaging .............. 13
3.2 Android Malware ...................................... 14
3.3 GUI System Security .................................. 14
3.4 Android System Vulnerability ......................... 16

Chapter 4
Droidmarking: Resilient Software Watermarking for Impeding
   Android Application Repackaging 17
4.1 Introduction ........................................... 17
4.2 Watermarking Background ............................ 20
4.2.1 Traditional Watermarking ........................................... 20
4.2.2 Software Watermarking ............................................. 21
4.3 Overview ................................................................. 22
  4.3.1 Problem Statement and Assumptions ............................. 23
  4.3.2 Architecture ......................................................... 24
4.4 Design and Implementation ............................................. 25
  4.4.1 Self-decrypting Code ............................................ 25
  4.4.2 DM Embedder ...................................................... 27
    4.4.2.1 Watermark Code Snippet Generation ..................... 27
    4.4.2.2 Finding Candidate Branches .............................. 28
    4.4.2.3 Watermark Embedding ................................... 30
  4.4.3 DM Recognizer .................................................... 32
  4.4.4 DM Scanner ....................................................... 33
4.5 Security Analysis ........................................................ 34
  4.5.1 Steal Attack ....................................................... 34
  4.5.2 Change Attack .................................................... 34
4.6 Evaluation ............................................................... 36
  4.6.1 Feasibility ........................................................ 36
  4.6.2 How Soon to Recover a Watermark? ............................ 38
  4.6.3 Resilience ......................................................... 39
    4.6.3.1 Resilience to Static Analysis ............................ 39
    4.6.3.2 Resilience to Dynamic Analysis ......................... 40
  4.6.4 Performance ...................................................... 42
4.7 Discussion and Conclusion ............................................. 43

Chapter 5
Towards Discovering and Understanding Task Hijacking in Android 45
5.1 Introduction ........................................................... 45
5.2 Background ........................................................... 47
5.3 Android Tasks State Transition Model ............................... 49
  5.3.1 Task and Back Stack ............................................ 49
  5.3.2 A Tasks State Transition Model ............................... 49
  5.3.3 A Task State Transition Example ............................. 50
  5.3.4 Android Implementation ....................................... 52
5.4 Task Hijacking in Android ........................................... 52
  5.4.1 Adversary Model ................................................ 52
  5.4.2 Hijacking State Transition ................................... 53
  5.4.3 The Causes of HSTs ............................................ 53
  5.4.4 Exploiting Conditions ....................................... 56
5.4.4.1 Activity Attributes ........................................ 56
5.4.4.2 Intent Flags .................................................. 58
5.4.5 Exploiting Events .............................................. 59
  5.4.5.1 Callback Function ......................................... 59
  5.4.5.2 Framework API ............................................ 59
5.5 Task Hijacking Attack Examples ................................ 60
  5.5.1 Breaching UI Integrity ....................................... 60
    5.5.1.1 Spoofing Attack ......................................... 60
    5.5.1.2 Phishing Attack - “Back Hijacking” ....................... 61
  5.5.2 Breaching UI Availability ................................... 64
    5.5.2.1 Preventing Apps from Being Uninstalled ................. 64
    5.5.2.2 Ransomware .............................................. 66
  5.5.3 Breaching UI Confidentiality ................................ 67
5.6 Defense Discussion ............................................. 67
  5.6.1 Detection in Application Review ............................. 68
  5.6.2 Secure Task Management .................................... 68
5.7 Related Work .................................................. 69

Chapter 6
Systematic Protection of Android GUI System 71
6.1 Introduction .................................................... 71
6.2 Android GUI System ............................................ 74
  6.2.1 Activity and Window ........................................ 75
  6.2.2 GUI Architecture Overview ................................ 75
    6.2.2.1 Activity Management .................................... 77
    6.2.2.2 Window Management ..................................... 78
  6.2.3 Security Mechanisms ....................................... 80
  6.2.4 Security Risks .............................................. 81
6.3 Android Window Integrity ..................................... 83
  6.3.1 Display Owner and Activity Session ......................... 84
  6.3.2 System State Transition ................................... 86
  6.3.3 System State Legitimacy ................................... 87
6.4 WindowGuard .................................................. 90
  6.4.1 AWI Model Implementation ................................ 90
  6.4.2 Security of App Navigation “Hubs” ........................ 93
  6.4.3 Preserving User Experience ................................. 94
6.5 Evaluation .................................................... 95
  6.5.1 Effectiveness ............................................... 95
## Chapter 7
### Directions for Android Security

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Security Overhaul of Android System Services</td>
<td>99</td>
</tr>
<tr>
<td>7.2 Dynamic Analysis of Harmful Applications</td>
<td>100</td>
</tr>
<tr>
<td>7.3 Android System Vulnerability Patching</td>
<td>101</td>
</tr>
</tbody>
</table>

### Bibliography

102
# List of Figures

2.1 Android system architecture [1] .......................... 7

4.1 Deployment model of Droidmarking system .................. 24
4.2 Examples of candidate branches. ............................ 25
4.3 Example of self-decrypting code (SDC) segment. ............. 26
4.4 Work flow of Droidmarking Embedder. ........................ 27
4.5 Watermark code snippet. ..................................... 28
4.6 Branch conditions transformation .............................. 29
4.7 Embedding watermark code snippet into a candidate branch. .... 30
4.8 Process of SDC segment generation in Dalvik bytecode segment. . 32
4.9 Average # of candidate branches per 1000 LOC (the number above each bar is the average # of candidate branches per app) ........... 37
4.10 Watermark recovery by human testers. Show three open source apps: Zirco Browser (ZB), Tomdroid Notes (TN), and Dudo Game (DG). ... 39
4.11 (a) Percentage of watermarks removed by attacker using Monkey. (b) Percentage of surviving watermarks later recovered by human testers. ... 41

5.1 A simple task state transition example. .......................... 50
5.2 Data structures of tasks, activities and back stacks in the Activity Manager Service. ............................. 52
5.3 (a) A sub-graph of the over-sized task state transition graph for a simulated system with three apps. (b) Visualization of task states of all nodes in figure (a) .......................................................... 55
5.4 Tasks state transition diagram of the spoofing attack ............. 60
5.5 “Back Hijacking” phishing attack to a well-known bank app. (a) the main activity of the bank app, with a tutorial video link for new users; (b) A system dialog showing available video players after clicking the video link; (c) video player activity, user clicks “back” button to quit the video and “goes back” to the original main activity; (d)-(e) the back button directs the user to the phishing UIs, which steal users’ bank account information, then quit after user clicks “Sign On”; (f) The original main activity is resumed, with a log-in failure error message left by the malware on screen.

5.6 Tasks state transition diagrams of “Back Hijacking” attacks. Figure (a) and (b) shows method I and II respectively.

5.7 Tasks state transition diagram of application uninstall prevention attack.

6.1 Overview of Android GUI System Architecture
6.2 A snapshot of system states in AMS and WMS
6.3 Multiple activity sessions in the system.
6.4 An activity session transition example.
6.5 System state and activity session in a back hijacking attack. A: victim app; M: malware; U: legitimate utility app.
6.6 Screenshot of (a) Admin privilege confirmation window (b) tapjacking attack window - “please update the app to latest version” (c) Security alert dialog created by WindowGuard.
List of Tables

4.1 Open source Android apps from 9 categories .......................... 36
4.2 Surviving watermarks of each app ........................................ 40
4.3 Application performance overhead ........................................ 43

5.1 Types of task hijacking attacks presented in this paper (all system versions considered - Android 3.x, 4.x, 5.0.x) .................................. 47
5.2 Task control knobs - configurable task state transition conditions and events provided by Android ............................................. 53
5.3 Detailed information of the HSTs in Figure 5.3. E.g., condition “M1:taskAffinity=B2” indicates that the taskAffinity attribute of activity M1 is set to that of B2; Event “launcher:startActivity(A1)” means that activity A1 is started by the launcher ........................................ 55

6.1 Existing known GUI attack vectors in prior work. The consequences are: A - sensitive data stolen; B - user input eavesdropping; C - user spoofing; D - loss of availability; E - malware infection; F - user privacy infringement .......................... 83
Chapter 1
Introduction

The increasingly popular mobile devices, especially the smartphones and tablets, have made profound impact to shape people’s daily life, change how business operate and affect how industries and economies are powered today.

In particular, Google’s Android system has become one of the most popular operating systems for mobile devices. The recent years has witnessed the breathtaking growth of Android devices and the booming of Android applications markets. By 2014, the number of Android users has grown to 1.1 billion and the number of Android devices has reached 1.9 billion. At the mean time, enterprises are also actively moving to Android-based Bring Your Own Device (BYOD) solutions. For instance, in Intel’s BYOD program, there are over 20,000 Android devices across over 800 combinations of Android versions and hardware configurations [2].

1.1 Motivation

Due to the extreme popularity of mobile devices, the traditional PC threats have migrated to mobile space. In particular, Android has become one of the most targeted mobile OSes for attackers. As a result, Android has come with great security risks, by virtue of the careless security design of the system, and the rise of loosely-supervised third-party Android markets.

Although Android enforces various security mechanisms in different components in the system (e.g. kernel-level app sandboxing, app permission model, etc.), the disparities between the assumptions of security models and the real implementations
can lead to serious breaches of the security guarantees (the confidentiality, integrity and availability) of Android system and apps. The attacks, as a result of exploiting these vulnerabilities and design flaws, range from the local attacks (e.g., through locally installed malware), and remote attacks (e.g., via exploits of system or app vulnerabilities). For example, Android malwares, which are widely distributed in third-party app markets, can cause serious security consequences once they are installed on user device. Alternatively, even without using a malware, attackers can remotely launch powerful attacks to vulnerable apps from the most strictly-scrutinized Android markets with relatively less malwares (like Google Play).

Based on OSVDB [3], an open-sourced database of security vulnerabilities, there are approximately more than seventy open system vulnerabilities affecting the latest Android versions (>4.0), not to mention the unknown system vulnerabilities and the numerous number of vulnerable apps. It is estimated that, 98% Android devices are vulnerable to at least two system vulnerabilities, 97% devices can be rooted, and at least 43% devices can be exploited remotely. To this end, we have a pressing need for a much more secure Android system in order to tackle the emerging threats. The assumptions of Android’s security models have to be re-examined; the security of Android system components requires to be fully investigated.

Moreover, unlike the relatively closed ecosystem and app market of Apple’s mobile system, iOS [4], the openness of Android ecosystem allows Android apps on user devices come from alternative sources other than Google Play. This bring about unique challenges in Android, since the alternative sources, i.e., various third-party Android app markets, are troubled with a large number of malwares, adwares and risk-wares. Most of these malicious or risky apps masquerade as repackaged apps, i.e. apps plagiarized from legitimate apps and distributed separately, in order to attract user installs by stealing useful functions/features from legitimate apps. In addition to the potential security threats to the users, the repackaged apps further cause intellectual property infringement and revenue losses to original app developers, and in turn can do real damage to the health of the Android markets. Despite these serious issues, many third-party Android apps markets cannot provide security assurance on the apps they hold. However, it is important to assure the sanity of the apps in these markets in order to prevent the security threats from propagating.

To this end, there has been considerable number of efforts in securing Android
to achieve the following two complimentary goals: (1) Market-scale detection and identification of malware, risk-ware, and repackaged apps in an efficient manner; (2) discovery, analysis of the Android system vulnerabilities, and defense measures against the enabled attacks.

1.2 Contributions

We develop techniques to solve specific problems in achieving the above two goals respectively. We make the following specific research contributions.

First, in order to effectively detect and prevent app repackaging in the Android markets, we devise a novel dynamic software watermarking technique - Droidmarking. Droidmarking can fully overcome the difficulties of existing app repackaging detection techniques, and provide three properties: (1) copyright ownership assertion for developers, (2) real-time app repackaging detection on user devices, and (3) resilience to evading attacks. Droidmarking combines the joint efforts of all stakeholders (users, developers and app markets) in defeating app repackaging threats. Notably, these properties effectively enable normal users to reliably recover and verify watermark copyright information, without requiring a confidential watermark recognizer like in traditional watermarking techniques. Based on our evaluation, we show that Droidmarking is a feasible and robust technique to effectively impede app repackaging with relatively small performance overhead. We summarize the contributions in this work below:

- We propose a new “non-stealthy” software watermarking approach that allows users to freely recover and verify watermarks without requiring a confidential watermark recognizer yet is still resilient to de-watermarking attacks.

- Our “non-stealthy” watermarking is based on a primitive called self-decrypting code (SDC). We implement our watermarking approach to a prototype - Droidmarking - for Android applications.

- We systematically evaluate the efficacy of Droidmarking on its feasibility, resilience and performance.

Second, we systematically study an important feature and the behind supporting component of the Android system - Android multitasking. Specifically, Android
multitasking provides rich features to enhance user experience and offers great flexibility for app developers to promote app personalization. However, the security implication of Android multitasking remains under-investigated. With a systematic study of the complex tasks dynamics, we find design flaws of Android multitasking which make all recent versions of Android vulnerable to task hijacking attacks. We demonstrate proof-of-concept examples utilizing the task hijacking attack surface to implement UI spoofing, denial-of-service and user monitoring attacks. Attackers may steal login credentials, implement ransomware and spy on user’s activities. We have collected and analyzed over 6.8 million apps from various Android markets. Our analysis shows that the task hijacking risk is prevalent. Since many apps depend on the current multitasking design, defeating task hijacking is not easy. We have notified the Android team about these issues and we discuss possible mitigation techniques in this paper. We summarize our contributions in this work below:

- To our best knowledge, we are the first to systematically study the security implications of Android multitasking and the Activity Manager Service design.

- We discover a wide open attack surface in Android multitasking design that poses severe threat the security of Android system and applications.

- Based on our vulnerability analysis over 6 million apps, we find that this problem is prevalent and can be used by attackers to cause a variety of serious consequences.

- We provide mitigation suggestions towards a more secure Android multitasking sub-system.

Lastly, we propose a new security model called Android window integrity, which defines the principles of how the system should be kept valid from one state to the next and specifies the capabilities of various principles in the system. The model is designed based on Android app model and the norm of app navigation from user’s perspective. This largely preserve the original user experience and the enforcement on security violators becomes natural. We implement the our security model, and the prototype, WindowGuard, can comprehensively protect the GUI system and is able to defeat all known GUI attacks, a much broader range of attacks than previous work. When WindowGuard is deployed on user device, the user is not bothered at
all until a suspicious event is detected. WindowGuard takes a light-weight response by briefing the user about the event and ask for the final decision from user, who is inherently more capable of making the best decision for him/herself based on the context. More importantly, this design makes WindowGuard more practical and suitable for the diverse needs of app developers and users in the Android ecosystem. WindowGuard is implemented as a Xposed module, making it realistic to be quickly deployed by a large number of user devices and provide immediate protection. We make the following contributions:

- **New understanding of the Android GUI security.** To the best of our knowledge, we are the first to systematically overhaul the security implication of Android GUI system. Composed of two key system services, activity manager service and window manager service, it is the single most important and sophisticated subsystem in Android. This new understanding can further inspire follow-up research on mobile GUI system security.

- **Novel GUI security model in mobile environment.** We define a novel security model - Android window integrity - for the GUI subsystem in a unique mobile environment like Android. By holding a set of security criteria under this model, the system could comprehensively and automatically protect itself against a wide spectrum of GUI attacks, ranging from user deception attacks, denial of service attack to undesirable pop-up window. More importantly, the new security model also considerably raise the bar for future attacks. Now new attacks can be put under the test of our defense before they make real threats.

- **Implementation and evaluation.** We develop WindowGuard based on our proposed model. Compared to existing solutions, our solution is effective maximally preserves original GUI features and user experience. Our evaluation demonstrates its efficacy in effectiveness, usability and performance.

### 1.3 Outline

The remainder of this report is organized as follows. Android background is first introduced in Chapter 2. We first present the Android system model and then
introduce the Android ecosystem. We next discuss the related work in Chapter 3. In Chapter 4, we introduced the Droidmarking scheme for impeding Android app repackaging. In Chapter 5, we investigate the security implications of Android multitasking and present a new attack - task hijacking - for the Android system. In Chapter 6, we introduce the new defense mechanism that protects the GUI system in Android. Finally, we summarize our work and discuss future directions in Chapter 7.
Chapter 2  
Android Security Background

2.1 System Model

As shown in Figure 2.1, Android platform is essentially a stack of software components, supported by the underlying Linux kernel. The Linux kernel provides the fundamental OS functionality such as process and memory management, networking, security sandboxing, power management, and contains a rich spectrum of device drivers that support various hardware (e.g. cellular radio, various sensors, etc.) for the mobile device.

Figure 2.1. Android system architecture [1]
On top of the Linux kernel, there are a set of libraries that are used by the software in the upper levels in the software stack, including a standard C library, Bionic, and other libraries supporting for audio, graphics, database, web browser core, to name a few. A key component in this level of software stack is the runtime environment for the apps and system framework to run within this environment. Until version 5.0, Android uses Dalvik as the default runtime. Dalvik is a process virtual machine with just-in-time (JIT) compilation to run Dalvik bytecode, which is translated from the Java bytecode. Android 4.4 introduces Android Runtime (ART) as a new runtime environment, which uses ahead-of-time (AOT) to compile the bytecode upon installation of an app in order to achieve high runtime performance.

Based on the native libraries, the Application Framework provides high-level services to the apps. In runtime, Application Framework exists mainly in the form of system services, a set of services that supervise the dynamics of all apps components in the system and run in a privileged process, namely System Server. The system services govern the system in all aspects, ranging from controlling the apps life cycles, facilitating app IPC interactions, high-level process, memory and data management, to providing multitasking features, controlling windows systems, app package management notification of messages, etc. The interactions and communications between app processes and the system services are implemented by an IPC mechanism called binder and are transparent to app developers. From user’s point of view, Android Framework is the heart of the Android system, since it implements the system features and functionality offered to users, and essentially defines the uniqueness of Android compared to other OSes. Android Framework provides APIs for app developers to create app components and interact with the system services.

At the top of the software stack are applications, including pre-installed system apps, e.g., phone dialer, address book, etc., and third-party apps downloaded and installed from various Android app markets. An Android app is composed of a collection of the following components.

- **Activities** provide user interfaces. Activities are started with Intents, and they can return data to their invoking components upon completion.

- **Services** run in the background and do not interact with the user. Other components can bind to a service, which builds a continual connection with
the target service.

- **Broadcast Receivers** receive Intents sent to multiple apps. Receivers are triggered by the receipt of an appropriate Intent and then run in the background to handle the event. Android defines many standard action strings corresponding to system events (e.g., system booted). Developers often define their own action strings.

- A **Content provider** component is a database-like scheme for sharing data with other apps. The interface does not use intent messages, but rather is addressed via a content URI. The content provider interface includes standard SQL-like queries and IO streams for reading and writing file content.

Every app package includes a manifest file, which must specifies all components in an app. An Android app is developed mainly in Java and is then packaged with other app resources to a APK file, a package ready to be published and installed on an Android device. Apps use the well-defined APIs provided by the Application Framework to ask for system resources and services, fulfilling the apps’ functionality and their design goals.

The Android API framework is composed of two parts: a library that resides in each app’s virtual machine and an implementation of the API that runs in the system process(es). The API library runs with the same permissions (will discuss shortly) as the application it accompanies, whereas the API implementation in the system process has no permission restrictions. API calls are handled in three steps. First, the application invokes the public API in the library. Second, the library invokes a private interface, also in the library. The private interface is an RPC stub. Third, the RPC stub initiates an RPC request with the system process that asks a system service to perform the desired operation.

### 2.2 Android Security Models

Android security model differs significantly from the standard desktop security model. Android applications are treated as mutually distrusting principals; they are isolated from each other and do not have access to each others’ private data.

The Android Market contains a wide array of third-party applications, and a user may install applications with varying trust levels. Users install applications from
unknown developers alongside trusted applications that handle private information such as financial data and personal photographs. For example, a user might install both a highly trusted banking application and a free game application. The game should not be able to obtain access to the user’s bank account information.

2.2.1 Android Application Sandbox

Under the Android security model, all applications are treated as potentially malicious. The underlying Linux kernel enforces the Linux-user based protection and process isolation, building a sandbox for each app. Specifically, the Android system assigns a unique user ID (UID) to each Android app upon installation and runs it in a separate process. Android apps are built on top of app components, and it means that components from different apps run in separate processes. One exception is that different apps can run in one process only if they are from the same developer (granted the same UID), and the developer explicitly specifies this in the manifest file. This Linux sandbox provides the foundation for app security in Android. The kernel enforces security between apps and the system at the process level through standard Linux facilities, such as user and group IDs that are assigned to apps. By default, apps cannot interact with each other and apps have limited access to the operating system.

2.2.2 Android Permission System

In addition to the app sandbox, Android restricts access to the system API with permissions, and applications must request the appropriate permissions in their manifests to gain access to protected API calls. There are four types of permissions depending on the protection levels:

- **Normal** permissions are granted automatically.

- **Dangerous** permissions can be granted by the user during installation. If the permission is denied, then the app is not allowed to be installed.

- **Signature** permissions are only granted if the requesting app is signed by the same developer that defines the permission. Signature permissions are useful for restricting component access to a small set of apps trusted and controlled by the developer.
• *SignatureOrSystem* permissions are granted if the app meets the Signature requirement or if the app is pre-installed as the system app.

To enforce permissions, various parts of the system invoke a permission validation mechanism to check whether a given application has a specified permission. The permission validation mechanism is implemented as part of the trusted system services, and invocations of the permission validation mechanism are spread throughout the API. There is no centralized policy for checking permissions when an API is called. Rather, mediation is contingent on the correct placement of permission validation calls.

Permission checks are placed in the API implementation in the system process. When necessary, the API implementation calls the permission validation mechanism to check that the invoking application has the necessary permissions. In some cases, the API library may also redundantly check these permissions, but such checks cannot be relied upon: applications can circumvent them by directly communicating with the system process via the RPC stubs. Permission checks therefore should not occur in the API library. Instead, the API implementation in the system process should invoke the permission validation mechanism.

A small number of permissions are enforced by Unix groups, rather than the Android permission validation mechanism. In particular, when an application is installed with the INTERNET, WRITE_EXTERNAL_STORAGE, or BLUEETOOTH permissions, it is assigned to a Linux group that has access to the pertinent sockets and files. Thus, the Linux kernel enforces the access control policy for these permissions. The API library (which runs with the same rights as the application) can accordingly operate directly on these sockets and files, without needing to invoke the API implementation in the system process.

### 2.3 Android Markets

An Android app market, also known as app store, is a central point for distribution of apps, e.g., Google’s official Android app market, Google Play. The app markets facilitate the sales, distribution and discovery of new apps for users. The market is responsible for accepting payment from users, compensating the app developers and hosting the apps such that users can discover the existence of the apps.
The app market client usually runs on the Android device. It provides a central location for the users to search the existing apps on particular markets. On some mobile platforms such as iOS from Apple, there is only one "legitimate" market, from which the apps are allowed to be installed on user devices. On Android, except the Google Play, there are multiple third-party markets from which users can download and install the apps.

However, the fragmented Android markets and the different security guarantees on the apps in these markets let the third-party Android markets become a major source of security threats including malwares, adwares and risk-wares. Although there exists certain app vetting process, the limits of accuracy and scalability make it difficult to deploy complete security checks on the hosted apps in all the third-party markets. Take one of the rising threats for example, software plagiarism (also known as app repackaging) is posing prevalent and serious risks to the health of the Android markets. Existing approaches perform centralized repackaging detection, usually based on similarity mining from a large basis of apps, falls short in detection efficiency in large scale and its resilience to circumventing techniques, e.g., code obfuscation. We have a pressing need for new effective and efficient techniques to mitigate the potential threats in the current Android markets.
Chapter 3  
Related Work

3.1 Software Plagarism and App Repackaging

Software watermark and birthmark: There has been extensive work on both static [5–7] and dynamic [7–12] software watermarking techniques. Most recently, AppInk [13] has adopted traditional graph-based dynamic watermarking technique and implements it on Android apps. A software birthmark is a characteristic that can uniquely identify a program. The characteristic is selected either statically or dynamically from the whole program paths [14] or system call graphs [15]. Droidmarking differs from all existing watermark and birthmark techniques in that our approach is “non-stealthy”, allowing normal users to readily verify watermarks without requiring any secret or confidential watermark recognizer. The “inter-dependency” built between watermarks and carrier code insures its resilience to various evading attacks.

Android app repackaging detection: The app repackaging problem has drawn recent efforts from both industry (e.g., Google Android app licensing [16], Amazon DMS [17], etc.) and academia. App repackaging detection techniques have been proposed mainly focusing on similarity mining from a large basis of apps. These techniques differ in the features and methods used for app similarity comparison. For instance, [18, 19] use hashing of app instruction sequence, [20] and [21] use program syntactic and semantic fingerprints respectively, and [22–24] use program dependency graph as features. These works either employ pair-wise similarity comparison [18, 20, 22, 23, 25] or more efficient algorithms like nearest neighbor searching or clustering [19, 21, 24] to improve scalability. Our work differs from
existing techniques in that we devise a new watermarking technique that aims to quickly and accurately detect app repackaging threats and provide solid proof of original copyright ownership of an app.

3.2 Android Malware

Many prior efforts focus on large-scale detection of malicious or high-risk Android apps [26,27], e.g., fingerprinting or heuristic-based methods [28–31], malware classification based on machine learning techniques [32–34], and in-depth dynamic analysis for app behaviors [34–44] and static analysis for information flow analysis [45,46]. The attack surface discovered in this paper can be easily employed by attackers to create a wide spectrum of new malwares, as discussed in Section 5.5. In task hijacking work, we report our threat assessment based on over 6 million market apps and provide defense suggestions in order to prevent the outburst of task hijacking threats in advance.

To evade detection, malware usually employs code obfuscation techniques. Self-decrypting code is first proposed by Sharif et al., referred to as conditional code obfuscation in their work [47]. They apply it to malware programs in order to conceal trigger-based malicious behaviors from state-of-the-art malware analyzers. The fundamental difference in our work is that we use self-decrypting code for designing and implementing a new dynamic watermarking approach, targeting at a totally different problem - Android app repackaging detection. In our design, self-decrypting code is used “non-stealthily” as the primitive of our watermarking scheme. The purpose is not to conceal watermarks but instead to build the “inter-dependency” between watermarks and the carrier code in order to improve its resilience to evading attacks.

3.3 GUI System Security

GUI security has been well studied in traditional desktop environments [48–52]. However, the unique mobile environment has raised unique challenges.

GUI confidentiality attacks and defenses. Previous research proves that the confidentiality of GUI can be broken through side channels such as shared-memory side channel [53], peeking sensor information [54,55]. Sensitive GUI information can
also be disclosed by taking screen shots because of adb flaws [56], or via embedded malicious UIs [57,58]. One the other hand, GUI information disclosure by analyzing memory images can be put into good use for forensics analysis [59,60]. A few approaches are proposed to protect GUI confidentiality, e.g. impeding the runtime information gathering [57,61,62], preventing shoulder surfing [63], or introducing security enhancement to the access control model [64], which help confine the attack vectors for confidentiality breaches. However, how to comprehensively protect GUI confidentiality from a large amount of attack vectors remains to be an open question. Our work instead focuses on the integrity and availability of Android GUI, which are seriously threatened by emerging GUI attacks.

**GUI integrity and availability attacks and defenses.** Previous research shows the possibility to launch phishing [53,65,66] or tapjacking attacks [67,68] in Android by overlaying a window of attacker’s control on top of the victim app’s window. It is also feasible to manipulate the activity browsing history to launch a variety of task hijacking attack [69]. Denial of service attacks [69,70] and adware [71–73] are also posing increasing threat to the GUI availability. Roesner et. al. [57] systematically study and design secure embedded user interfaces such as ad lib and third-party plug-ins for browsers and smartphones system. To protect Android GUI integrity, Bianchi et. al. [65] propose a two-layer defense towards defending against GUI confusion attack, an important type of GUI attack. In spite of the novelty of their findings and two-layer solution, a static analysis vetting process and a on-device defense, the proposed defense mechanisms has some obvious drawbacks, as mentioned in Section 6.1. Compared with the ad-hoc approaches, we propose a new security model to systematically protect the integrity and availability of the GUI system. The implementation, WindowGuard, can defeat all known GUI attacks, a much broader range of attacks in terms of attack types and vectors. In addition, the security model preserves the original user experience. The user is not bothered at all until a suspicious event happens, after which WindowGuard takes a light-weight response by briefing the user and asking for the final decision. The implementation is implemented as a Xposed module compatible with most latest Android versions, which makes our solution practical in reality.

**Integrity of program execution.** Control flow integrity [74] defends against subverted machine-code execution such as return-oriented programming [75] and return-to-libc [76] attacks. One of the well-known approaches is to save the state
of the program (e.g. the native return address) in a shadow stack [\cite{57,77}]. When the program state is resumed (e.g. function return), the resumed program state is compared with the saved copy on the shadow stack. Similar idea is applied to the legitimacy check of activity session in our work, but under the Android GUI protection context. Analogous to shadow stack, the previous-visited activities are saved in a activity session. To prevent back hijacking, when the user returns, a sanity check is performed to ensure the user return to the original activity. To defeat all task hijacking attacks, the integrity of the foreground activity session is additionally scrutinized whenever an activity obtains focus. Since the GUI window transition occurs much less frequently than function returns, WindowGuard does not suffer the performance issue in most shadow stack schemes \cite{78}.

### 3.4 Android System Vulnerability

The security threats in the inter-component communication (ICC) has been widely studied and a rich set of static analysis approaches are proposed to identify vulnerable apps or detect malicious ICC behaviors \cite{79–86}. Moreover, there has been considerable prior work on emerging Android vulnerabilities and their mitigation measures in many aspects in private data leakage \cite{87–91}, webview vulnerabilities \cite{58,92}, Android system customization \cite{93}, dynamic malicious code loading \cite{94}, crypto uses \cite{95} and vulnerabilities in security utility programs \cite{96–99}. In addition, third-party libraries that are widely used in Android apps has posed serious threats to the Android ecosystem \cite{100,101}. However, the critical Android multitasking mechanism and the feature provider, the AMS, have not been deeply studied before. This paper fills in this gap by systematically studying the Android multitasking and the security implications of this design.
Chapter 4
Droidmarking: Resilient Software Watermarking for Impeding Android Application Repackaging

4.1 Introduction

The past years have witnessed the unprecedented popularity of Android devices and the booming of Android application markets. Software plagiarism in Android markets, however, which is also known as app repackaging, is posing prevalent and severe threats to the health of the Android ecosystem. For instance, App repackaging has caused intellectual property infringement and enormous advertising revenue losses for original app developers [25]. A significant portion of Android malwares use repackaged apps as a “free” vehicle to carry malicious payloads, which have posed serious risks to user privacy and financial security [26, 102]. While there have been considerable recent efforts from both academia and industry to impede the app repackaging acts [13, 16, 18, 19, 21, 23, 24], the existing methods have limitations that largely undermine their effectiveness in practice.

Most existing app repackaging detection approaches are “offline” solutions, performing centralized repackaging detection regularly or on purpose by authorities like Android markets (e.g., similarity mining from a large basis of app samples [18, 19, 21, 23, 24]). The primary drawback of these “offline” solutions is that they allow
for repackaged apps to be widely distributed (depending on how often repackaging detection is performed) and to cause extensive harm to users and original app developers before they are detected. Moreover, most existing repackaging detection approaches can be easily evaded by various obfuscation techniques as illustrated by [103], which further encourages repackaged app developers (attackers) to submit more repackaged apps to the markets, hoping to make as much illegal profits as possible before they are detected and removed from the markets.

To this end, we have a pressing need for a new robust anti-repackaging scheme that can combine the efforts of other major stakeholders - users and app developers - and can instantly thwart the distribution of repackaged apps before massive damages are caused. We thus have the following design goals for the new anti-repackaging scheme:

1. Accurately and quickly detect app repackaging threats with help from users and original developers

2. Provide undeniable and solid proof of copyright ownership to protect app developers

3. Offer strong resilience to various evading attacks

Toward all these goals, we are intrigued by traditional watermarking techniques, which are widely employed by printing industries to fight against piracy acts. They provide three noteworthy properties: (1) copyright assertion, (2) real-time copyright verification by users and (3) resilience to evading attacks. These properties are highly desirable for those who want to overcome the difficulties in impeding app repackaging in today’s Android market.

Software watermarking is not new. By using a custom tool, namely \textit{watermark embedder}, a software developer can insert a copyright notice into a program. An concerned authorized party can later recover and verify the software’s copyright ownership using a tool called \textit{watermark recognizer}, which keeps some form of secrets (e.g., password, special input) to reverse the watermarking embedding process and recover the copyright notice from the program.

To prevent attackers removing the watermarks, existing software watermarking approaches rely heavily on the stealth of watermark (i.e., embed watermarks in a way to make them locally indistinguishable from the carrier code). However, this
“stealthy” design principle causes two consequences that prevent existing software watermarking techniques from achieving the properties we desire in traditional watermarking. First, the watermark recognizer has to be kept confidential from the public and be held by a small number of authorized parties (e.g., developer or trusted authority) as it contains the secret to disclose the locations of watermarks in a program. As a result, there is no way for normal users to verify watermarks in real-time like they can on, for example, a book page. Second, it is difficult to ensure absolute “stealth” as we will see in Section 6.2. Therefore, simply deploying existing software watermarking techniques on Android apps would fail to achieve our goals.

It is exactly these challenges that this paper seeks to address by presenting a new “non-stealthy” software watermarking scheme. Contrary to the design criteria of existing software watermarking approaches, our design is based on the insight that “non-stealthy”, i.e. do not conceal the locations of watermarks on purpose, is an appealing property for software watermarking due to two advantages: (1) when the locations of watermarks in a program is open to the public, normal users (and also attackers) are free to recover watermarks without confidential tools needed. Of course, users still need assistance from certain tools to recover watermarks, but these tools contain no secret for watermark recovery, and, thus are no longer exclusive to authorized parties. (2) It circumvents the design challenge of “how to make watermarks stealthy”, which is difficult based on the experience of previous work [8, 104]. On the other hand, “non-stealthy” makes watermarks vulnerable to evading attacks. Our solution is to make the resilience of our watermarking rely on “inter-dependency” between the embedded watermarks and the carrier program. It means that not only do watermarks depend upon a carrier program to reside, but watermarks also are embedded in a way that the correct and complete functionality of the original program also relies on the integrity of original watermarks. Simply de-coupling the two would result in miss- or non-functional code.

Based on these new criteria, our watermarking scheme - Droidmarking - lets a developer embed watermarks into an app and later allows a normal user who installs this app on his/her device to quickly recover and verify the developer’s watermark from the app by simply playing the app for a short period of time without any confidential watermark recognizer needed. Based on the recovered watermark information, a Droidmarking facility on a user device can automatically
detect potential repackaging threat and immediately report it to user and/or
the Android market. Droidmarking is based on a primitive called *Self-decrypting
Code* (SDC) segment, which is first introduced by Sharif et al. and referred to as *conditional code obfuscation* in their work [47]. A SDC segment is an encrypted
code block (containing both the original functional code and embedded watermark
code snippet) in a special type of branch in the program. The encryption key comes
from the constant value in the special branch condition. The branch condition is
re-written in a semantic equivalent way that this particular constant/key is removed
from the static code and can only be dynamically recovered upon branch condition
satisfaction at runtime. By this means, SDC helps build the “inter-dependency”
between watermark and original carrier code by combining and encrypting the
two into a single SDC segment. Furthermore, by creating a large number of
watermark-carrying SDC segments, it is difficult for attackers to de-couple all
embedded watermarks from the original functional code without significant efforts.
We summarize our contributions below:

- We propose a new “non-stealthy” software watermarking approach that allows
  users to freely recover and verify watermarks without requiring a confidential
  watermark recognizer yet is still resilient to de-watermarking attacks.

- Our “non-stealthy” watermarking is based on a primitive called self-decrypting
code (SDC). We implement our watermarking approach to a prototype -
Droidmarking - for Android applications.

- We systematically evaluate the efficacy of Droidmarking on its feasibility,
  resilience and performance.

### 4.2 Watermarking Background

#### 4.2.1 Traditional Watermarking

Traditional watermarking embeds a watermark message into the carrier media to
declare copyright ownership. Take watermarking in printing industry for instance,
a valid watermark (usually an image or a pattern in paper) in a printed book
carries unique authorship and publisher information that discloses precise copyright
ownership of the book. A concerned reader can readily verify these information
immediately when getting a hard copy. An effective watermarking technique is
designed to make sure that (1) watermark is readily verifiable by people who possess
the object, and (2) it is infeasible to copy or counterfeit a watermark. The first
objective allows users be able to easily verify the authenticity and originality of
the object by themselves immediately. The second objective protects the integrity
of watermarks against forgery attempts. As a result, there is no easy way for an
attacker to forge a watermarked object without being noticed by concerned users
who always verify the watermark first.

4.2.2 Software Watermarking

Software watermarking [8] does not prevent software copying but instead discourages
software piracy by embedding copyright information into software code and allows
one to prove copyright ownership when plagiarism has occurred.

Software watermarking embeds a watermark $W$ into a program $P$, such that
$W$ can later be reliably located and recognized from $P$, even after $P$ is subject
to code transformations. Software watermarking comes with two flavors - static
and dynamic. Static watermarks are stored in the code or data of the executable,
whereas dynamic watermarks are usually built at runtime and contained in the
dynamic state of the program, e.g. data structure, execution trace, etc. Software
watermarking must be able to defend against two types of attacks in general - steal
attack and change attack. In a steal attack, attackers wish to disguise plagiarized
software as original by copying/forging the original watermark, making users believe
that their copy comes from a trusted original software vendor. In a change attack,
attackers attempt to disable the original watermark by changing the program. More
specifically, change attacks fall into the following three categories: (1) A subtractive
attack tries to completely or partially remove the watermarks from the program.
(2) A distortive attack applies a sequence of code transformations, attempting to
obstruct the recovery of original watermarks. (3) An additive attack add attacker’s
own watermarks to the software and claim attackers’ ownership of the software.
In practice, attackers can opt for a combination of the above attacks. On the
defense side, one natural step towards defeating subtractive and distortive attacks
in existing work [5,10–12] is to conceal the watermark in a stealthy way, such that
it is difficult to locate/remove/distort all watermarks in the program, or the quality
of the distorted program is significantly degraded such that it is no longer of any value to the attacker.

However, the principle of stealth is inherently flawed and in turn causes two consequences. First, the watermark recovery tool - watermark recognizer - that contains some forms of secret information (e.g., password, special inputs) used for watermark recovery is kept confidential from the public. It means that unlike traditional watermarking, existing software watermarking approaches cannot enable normal users to perform watermark verification; instead, it solely relies on a small number of authorities who have access to the confidential watermark recognizer. Although software watermarking techniques have been developed for many years, in practice, no such authorities actually exist. We believe that Android markets do not have strong enough incentives to become such a liable authority. Nevertheless, the originality of Android apps is a serious concern for developers and users. Second, previous experience in watermarking design has shown us that it is non-trivial to ensure absolute stealth [9,105]. Once a program resides in an attackers’ machine, attackers can use any conceivable techniques to (approximately) locate the watermarks. For example, it is well-known that static watermarking is highly susceptible to semantics preserving transformation attacks. Dynamic watermarking, while yielding a certain degree of resilience to semantics preserving transformation attacks, relies on the stealth of runtime states (e.g., bogus graph data structure [5]). Once a subset of these runtime states are detected and scrambled by the attacker, the watermark is ruined and rendered useless.

4.3 Overview

To overcome the limitations of existing watermarking methods, Droidmarking is designed to be a “non-stealthy” watermarking technique; in other words, the purpose is not to hide watermarks. In fact, it is fairly easy to locate the watermarks in an Android app program. To defend against watermarking evading attacks, Droidmarking is based on a primitive called Self-Decrypting Code (SDC), which makes it difficult to de-couple the two without significant efforts and costs for attackers.
4.3.1 Problem Statement and Assumptions

In watermark embedding, our scheme takes as input an app program $A$ and the original developer’s watermark instance $w$, and outputs a watermarked app program $A'$ such that $A'$ is semantically equivalent to $A$, i.e. $A' = E(A, w)$, where $E(\cdot, \cdot)$ is the watermark embedding function. In the watermark recognition phase, a set of watermark instances (may consists of multiple $w$s), denoted by $\Sigma$, is recovered from program $A'$ and collected by a watermark recognizer, i.e. $\Sigma = D(A', \alpha)$. $\alpha$ represents a series of user inputs (and interactions) to app $A'$. $D(\cdot, \cdot)$ is watermark extraction function called by $A'$ in response to user inputs $\alpha$. Let $T(A')$ denotes the app program after code transformations performed by an attacker, and $\Sigma'$ denote the set of watermark instances recovered from $T(A')$, i.e. $\Sigma' = D(T(A'), \alpha)$. We say the original watermark instance $w$ is successfully recovered even subject to code transformations, iff:

$$\Sigma' \neq \emptyset \text{ and } w \in \Sigma'.$$

$\emptyset$ denotes an empty set. In other words, successful watermark recovery ensures that at least one watermark instance $w$ from original developer be recovered and collected by the watermark recognizer. In addition, our goal also requires $w$ to be successfully recovered within an anticipated short period of time $t$. Otherwise, we call it “failure to recover” original watermark $w$.

Now we consider a set of watermarks $\hat{\Sigma}$ recovered from an unknown app $\hat{A}$, i.e. $\hat{\Sigma} = D(\hat{A}, \alpha)$, a watermark conflict occurs if: $\exists w_1, w_2 \in \hat{\Sigma}$ and $w_1 \neq w_2$. A watermark conflict unquestionably indicates that $\hat{A}$ includes different copyright information and hence is a repackaged app.

We assume that an original app developer has the app source code. Developers are concerned about their intellectual property and financial interests and, thus, are motivated to apply protection schemes against potential piracy acts. Android users, however, are divided into two classes with respect to their care and caution to the originality of the apps: concerned users and causal users. Concerned users are prudent about their privacy and cyber security and are willing to build trust with newly installed apps first, whereas casual users do not bother to care. Droidmarking draws the joint efforts from original developers, concerned users and Android markets to defend against app repackaging threats.

We assume that attackers can use any conceivable static or dynamic analysis
tools to examine the compiled app program (bytecode and native binary), line-by-line, and make any code transformations necessary aiming to remove original watermarks, render watermarks infeasible to be recovered, or even add attackers’ own watermarks to the app. A determined attacker may also carefully infer the program semantics and re-write part of the app program, or in the extreme cases, the whole app program. However, the cost of re-writing the app may outweigh the value of the repackaged app itself. We assume cash/interests-driven attackers are interested in repackaging an app only if the cost of repackaging is less than the value it adds. In that case, an attacker may even be willing to sacrifice certain functionality and value of the original app as long as the cost-efficacy still holds. The plagiarized app can be distributed by the attacker either in the same market as the original app, or across different markets.

4.3.2 Architecture

Figure 4.1 shows the overall deployment model and architecture of the Droidmarking system. Droidmarking is comprised of three major components (as highlighted in Figure 4.1) for the three stakeholders, respectively: Droidmark embedder (DM embedder) used by developers, Droidmark recognizer (DM Recognizer) installed on user devices, and Droidmark scanner (DM Scanner) employed by Android market. An app developer feeds app source code $A$ as input to DM embedder. DM embedder embeds watermarks into the code and finally outputs a dex file (executable for Dalvik virtual machine on Android system). The dex file is then packaged with libSDC (a native library used in our watermarking scheme) and other app resources to an apk file for publication (application package). Before publishing the app packages to the app store, Android market conducts a quick static scan to the dex file to insure the valid and correct use of the DM embedder. Those who fail DM scanning are subject to misuse or evading attacks (see Section 4.5) and are
if ( uri.startswith("smtp") ) {
    Code
}

if ( weight == 75 && height > 180 ) {
    Code
}

Figure 4.2. Examples of candidate branches.

rejected by the market. When users install the app on their devices and use the app, the embedded watermarks are automatically recovered. DM recognizer (residing on user devices) is responsible for harvesting the recovered watermark instances from the app and alerting the user and/or Android market immediately when an repackaging threat is detected. In the next section, we will discuss in detail the design and implementation of these three components, respectively.

4.4 Design and Implementation

4.4.1 Self-decrypting Code

Droidmarking is based on a primitive called Self-decrypting Code (SDC), which was first proposed by Sharif et al. [47] for malware obfuscation and has nothing to do with watermarking. A SDC segment is created from a special type of branch from the app code. Figure 4.2 shows two examples of these types of branches in Android apps. The two common properties in the examples are that (1) both examples use equality test (such as \texttt{==} operator, \texttt{startswith()}, \texttt{endswith()} or \texttt{equals()} routines) as one of the conditions to enter the branch; (2) one of the equality test operands is a constant numerical/string value. We call the branches that have these branch conditions candidate branches.

An SDC segment is created from a candidate branch as illustrated in Figure 4.3. In this example, the original branch condition \( \texttt{Var==Const} \) in Figure 4.3(a) is transformed to an equivalent condition \( \texttt{V==M} \) in Figure 4.3(b). The operands \( \texttt{V} \) and \( \texttt{M} \) of the equality test are created by performing encryption on the same randomly generated string \( \texttt{S} \). One encryption is performed at runtime at line 2, i.e.
\[ S = "\text{Rand}\_\text{String}" \]

\[ V = \text{encrypt}(S, \text{Var}) \]

if ( \( V == M \) ) {
  decrypt(EC, \text{Var})
}

\[ \text{EC} = \text{encrypt}(C, \text{Const}) \]

Where, \( M = \text{encrypt}(S, \text{Const}) \), \( \text{EC} = \text{encrypt}(C, \text{Const}) \)

Figure 4.3. Example of self-decrypting code (SDC) segment.

\( \text{V} = \text{encrypt}(S, \text{Var}) \), where \( \text{Var} \) is used as the key for this encryption operation. Another encryption is pre-computed offline, i.e. \( M = \text{encrypt}(S, \text{Const}) \), where \( \text{Const} \) is the key. As a result, \( V == M \) holds only when \( \text{Var} == \text{Const} \). \( \text{Const} \) is also used as the key to encrypt the original candidate branch code block \( C \) to a cipher code block \( \text{EC} \). Note that by using the equivalent condition \( V == M \) the constant \( \text{Const} \) (also the key to encrypt \( C \) ) has been removed from the code. As shown in Figure 4.3(b), we define the new branch condition as well as the (cipher) code inside the branch as a SDC segment. During runtime, \( \text{EC} \) is only reached when the equivalent condition is satisfied upon \( \text{Var} == \text{Const} \). When this happens, \( \text{Var} \) is immediately used as the key to decrypt \( \text{EC} \) (line 4). \( \text{EC} \) is then replaced by the resulting decrypted branch code \( C \) such that the execution of code \( C \) proceeds normally.

In Droidmarking, we embed watermark code snippets in branch code \( C \) of all candidate branches and create SDC segments accordingly. By this means, we build the “inter-dependency” between watermarks and original branch code such that it is infeasible to de-couple the two within a cipher code block unless one first decrypts it.

Knowledge Asymmetry Property of SDC: This property provided by SDC says that, the original developer knows both the constant value \( (\text{Const}) \) of the original branch condition and the enclosed branch code \( (\text{C}) \) that this constant value could “lead” to. In contrast, an attacker does not know this at all until the dynamic execution of the app program “happens to” go into the SDC branch. This property is the fundamental reason why our software watermarks can be “inter-dependent”
with original branch code and become non-stealthy yet still resilient to evading attacks.

4.4.2 DM Embedder

In the watermark embedding phase, the DM embedder helps embed watermarks into an app program for developers. The DM embedder takes app source code as input and takes three steps to output a watermarked dex file: (1) watermark code snippet generation, (2) finding SDC candidate branches, and (3) embedding watermark code snippet into SDC segments. Figure 4.4 shows the work flow and modules of the DM embedder. Module WM initiator handles step 1 and 2 and outputs intermediate (IM) source code. The following compiling process of IM source code is intervened and a modified dx compiler (a tool used for compiling Java bytecode to Dalvik bytecode) takes care of step 3. We now present the design and implementation of these three steps respectively.

4.4.2.1 Watermark Code Snippet Generation

The most important piece of information in a watermark instance is a unique developer identifier. On the Android platform, every installed app must be self-signed by the developer with a certificate whose private key is held by the app developer. The whole purpose of app signing is to allow the Android system to uniquely identify the developer of an app, which is used for authenticating app updates and building trust among apps from the same developer. The public key certificate is distributed with apk file (contained in self-signed X.506 certificate “META-INF/CERT.RSA”) for signature verification (signature file “META-INF/CERT.SF”) by the Android system upon app installation. To this end, we use public key certificates as the key information for a watermark, as it is uniquely distinguishable for developer identity.
WM initiator offline generates a watermark code snippet (as shown in Figure 4.5) that can dynamically create a watermark instance at runtime. As shown in the code snippet, a watermark instance is an intent object (line 1), an Java object developer uses to send messages between app components on the Android system. Multiple pieces of watermark information are stored in the intent object as key-value pairs (line 2 to 7) - “cer”: developer's public key certificate, “pid”: PID of running app, “pck”: app package name, “iss”: certificate issuer (developer), and “tim”: time of watermark assertion. It is important that the public key certificate be the same one whose private key will be used next for signing the app. The watermark code snippet then explicitly sets an service (“com.dmrecognizer.dmService” service in DM recognizer) to receive and handle the intent (line 8-9). This intent/watermark instance is then sent to the DM recognizer by calling startService() function (line 10). We will discuss DM recognizer in more detail in Section 4.4.3.

This watermark code snippet will be embedded into every candidate branch found in Section 4.4.2.2.

### 4.4.2.2 Finding Candidate Branches

In this step, WM initiator aims to find all candidate branches in the app source code that can be used to carry watermark code snippet. WM initiator statically analyzes the source code and determines the following types of if candidate branch conditions: (1) equal to operator ==, (2) string comparison routines such as startswith(), endswith(), and equals(), (3) not equal operator !=, and (4) multiple simple conditions combined by && and || operators. Type 1 and

```java
android.content.Intent wm = new android.content.Intent();
wm.putExtra("cer", <public key certificate>);
int pid = android.os.Process.myPid();
wm.putExtra("pid", pid);
wm.putExtra("pck", "com.package.appname");
wm.putExtra("iss", "CN=Android Debug, O=Android, C=US");
wm.putExtra("tim", "04/02/2014 10:06:59");
wm.setComponent(new android.content.ComponentName("com.dmrecognizer", "com.dmrecognizer.dmService");
context.startService(wm);
```
2 conditions (we called *simple conditions*) can be directly transformed to SDC segments, as described in Section 4.4.2.3. Type 3 and 4 conditions must first be transformed to simple conditions.

For type 3 and 4 conditions however, we need to transform them to simple conditions first. As shown in Figure 4.6(a), we handle type 3 conditions by replacing `!=` with `==` operator and switching the code (Code1 and Code2) in *if* and *else* branches. If there is no *else* branch as in Figure 4.6(b), we first craft a corresponding *else* branch and fill in bogus code that do not change the semantics of the original program. Type 4 condition may involve more than one above candidate branch conditions. If so, and in case of `&&` operator as shown in Figure 4.6(c), we split the long condition into nested single conditions. In case of a `||` operator as shown in Figure 4.6(d), we create two separate single conditions with duplicated branch code under each condition.
4.4.2.3 Watermark Embedding

In this step, the watermark code snippet generated in Section 4.4.2.1 is embedded into all candidate branches found in Section 4.4.2.2. These watermark-carrying candidate branches are then transformed to SDC segments.

**Watermark code snippet embedding.** Figure 4.7 shows how WM initiator embeds the watermark code snippet into a candidate branch and generates the intermediate (IM) code A’. WM initiator appends watermark code snippet (WM) right before the branch code (C), and inserts special bogus expressions at line 6 and 10 respectively in Figure 4.7(b) as the begin and end marker of this combined code block. As we will see, these markers will be used for SDC segment generation and kept in the final executable. As we have already seen in Section 4.4.1, we replace condition \( \text{Var} == \text{Const} \) with semantic equivalent condition \( V == M \), where \( M = \text{encrypt}(S, \text{Const}) \), such that the branch code encryption key \( \text{Const} \) in Figure 4.7(a) is removed from the final executable. For now, the encryption key \( \text{Const} \) is temporarily retained in IM code right before the begin marker and will be removed after being used for code encryption. We also insert `sdc_decrypt()` function call at the beginning of branch code in line 4. `sdc_decrypt()` is a Java function call to our native library `libSDC` via Java Native Interface (JNI). This library is responsible for decrypting the cipher code block once the branch condition is satisfied. So far, WM initiator has finished all its jobs and has generated “watermarked” IM source code, which will be compiled by javac and then passed to our modified dx compiler for SDC segments generation.
**SDC segment generation.** This is the final stage of DM embedding process, where the transformed candidate branches in IM code are encrypted and transformed by dx compiler to SDC segments in the output dex executable. dx compiler is a tool used to convert Java class files to a dex file, which contains the code, data and references to all the classes and methods of Java classes. Our modification to dx occurs after it has already generated the Dalvik bytecode segments and before it assembles these segments to an output dex file. Figure 4.8 shows the layout of the bytecode block in a candidate branch (after compilation from IM source code) observed by dx. It also shows how the branch bytecode $C$ is “reshaped”, and the beginning part of it ($C_1$) together with the watermark code snippet ($WM$) are finally encrypted to Cipher Block. For illustration purposes, the code block is depicted with a width of 8 bytes, to be consistent with the block size (8 bytes) of DES block cipher.

In Figure 4.8(a), we first let dx find the transformed candidate branches by searching for the special bogus instructions indicating `<begin>` and `<end>` markers in all code segments. The code in between the markers is watermark code snippet $WM$ followed by original code $C = C_1 + C_2$. It is important to note that, in order to prevent misuse or evading attacks of SDC segments (will be further discussed in Section 4.4.4 DM scanner), we mandatorily require the resulting cipher code block to have a fixed size, i.e. a constant number $L$ of 8-byte blocks in Figure 4.8. In one scenario in Figure 4.8(a) where the size of branch code is larger than $L \times 8$ bytes, we can only encrypt part of the branch code, $C_1$, together with watermark code snippet $WM$ to the cipher block.

As shown in Figure 4.8(b), once $C_1$ is identified, we pad $WM+C_1$ with $0x0$ (NOP ops) to generate a full $L \times 8$ bytes code block (including $WM$, $C_1$ and padding $0x0$s). The `<end>` marker and $C_2$ are relocated as shown in the figure. Recall that $Const$ is retained in code transformation step and remains in the bytecode before `<begin>` marker (Figure 4.7(b)). As we proceed in to from Figure 4.8(b) to Figure 4.8(c), $Const$ is used as the key to encrypt the code block using DES algorithm and the original code block is overwritten by the resulting cipher block (same size with original code block). Note that `<begin>` and `<end>` markers must be retained to bound the cipher code block. Finally, key $Const$ is removed from the code. So far, a SDC segment has been generated. Note that $L$ should be at least larger than the watermark code snippet size, otherwise we cannot include any
functional code in the SDC. $L$ should not be too large either, to avoid too much wasted space for padding. We choose $L$ to be 16 (128 bytes cipher block) in our prototype.

### 4.4.3 DM Recognizer

As we have seen, the watermark code snippets are first executed after the cipher code blocks in SDC segments are decrypted. This is achieved by calling function `sdc_decrypt()` to native library `libSDC` via JNI, which is able to modify bytecode during runtime. `sdc_decrypt()` decrypts cipher code block using the key procured at runtime when the branch condition is satisfied. The decrypted code (including watermark code snippet and original branch code) overrides cipher block so the program proceeds normally: the watermark code snippet is executed first and the watermark instance is created and sent to DM recognizer. For performance reasons, `libSDC` only decrypts a particular cipher code block for once, i.e. when the branch code is decrypted, it will be not be encrypted again.

We implement DM recognizer as a stand-alone application. It is responsible for managing the watermarks received from different apps on the device and for reporting repackaging threats based on the collected watermark information. A repackaged app is determined based on the public key certificate enclosed in watermark instances, if either one of the following two events occurs: (1) Watermark conflict, i.e., different public key certificates are discovered from the watermarks sent from the same app. (2) No watermark conflict, but the public key certificate
retrieved from a watermark is different from its counterpart in the self-signed certificate (in file “META-INF/CERT.RSA”) of the running app. The first case indicates potential change attack (i.e., attempts to disable original watermarks by changing the code), and the second indicates possible steal attack (i.e., forging original developer’s watermark).

4.4.4 DM Scanner

Before a new app is published to the app store, Android market performs a lightweight static scan to the dex file in the app package to insure the valid and correct use of SDC segments. DM scanner can be used as part of existing vetting process performed by Android markets, such as Bouncer [106]. Those who fail the scanning are subject to misuse or evading attacks and are rejected by the market. In particular, DM scanner is deployed by Android markets to defend against library libSDC replacement attack. In other words, attacker’s goal is to modify the decryption behavior (invoked by function call sdc_decrypt() and performed by native library libSDC) such that the watermark code snippet is removed right after the cipher code block is decrypted. Toward this goal, attackers can implement this malicious decryption behavior to be a library namely libAttacker.

To defend against this attack, the rules for DM scanner to check are straightforward: (1) DM scanner makes integrity check to both libSDC and sdc_decrypt() function call site. It prevents library or function call replacement attacks (positioning of SDC segments is ensured by complying to rule 4 below); (2) DM scanner examines that sdc_decrypt() function must be invoked at the very beginning of the candidate branch code. This rule makes sure that no function call to libAttacker is added before sdc_decrypt(); within the branch scope (in which the key to the cipher code block has already been recovered). (3) The cipher code blocks must be guided by begin and end markers. Any unrecognized instructions outside markers’ guide are regarded as violation of this rule. This rule makes sure that all cipher blocks can be precisely located and no “hidden” cipher blocks in dex file can survive the scanning; and (4) Each cipher code block must have a fixed size. This rule is designed to defeat an attack in which an attacker generates an “outer” cipher code block that stealthily incorporates/encrypts both the original SDC segment (which forms nested cipher blocks) and a malicious
function call to \texttt{libAttacker} (which is “hidden” from DM scanner in the “outer” cipher code block). Specifying a fixed size cipher block prevents attackers from adding encrypted malicious instructions that disable the watermark.

4.5 Security Analysis

4.5.1 Steal Attack

In a steal attack, attackers wish to disguise a repackaged app as original by directly copying and forging the original watermarks, making users believe that the app comes from the original trusted app developer. This attack can be readily frustrated by Droidmarking and Android app signing mechanism. Recall that a watermark instance takes the public key certificate as the vital information to identify different developers. In the watermark recognition phase, the DM recognizer detects an app’s originality by comparing the public key certificates obtained from the recovered watermarks with its counterpart in the certificate in the app package (in “META-INF/CERT.RSA”), which is used for certificate verification by Android system upon app installation and has to be signed with a legitimate private key. Therefore, unless the attacker can sign the app package with the private key of original developer, leaving the original watermarks intact in the app program is always subject to being flagged as a repackaging threat, even though the app is plagiarized across different Android markets. As a result, attackers are forced to deal with watermarks and resort to more sophisticated change attacks.

4.5.2 Change Attack

\textbf{Subtractive Attack:} The goal of subtractive attack is to remove watermarks from an app without significantly degrading the value of the app. Since watermarks are embedded in non-stealthy SDC segments (containing cipher block and special begin/end markers) in the app code, it is easy for an attacker to locate all these SDC segments by static analysis. However, blindly removing SDC segments not only removes the watermarks but also the functional code contained in the SDC segments. Considering the large number of SDC segments in a program, this attack approach leads to mis- or non-functional apps, which significantly degrades their
value to attackers. Attackers may resort to existing program path exploration
techniques, for example forced conditional execution of branch paths or symbolic
execution to automatically resolve the keys to the branches in SDC segments.
However, all these techniques would fail because it is either impossible to forcibly
execute encrypted code or infeasible for these path exploration tools to resolve
the branch constraints calculated by non-linear encryption operations. In order to
recover the SDC segments, two approaches are promising for attackers: brute-force
attack and dynamic analysis. We will justify the security offered by Droidmarking
against these attacks in Section 4.6.3.

Additive Attack: An additive attack augments the app by inserting the attacker’s
own watermark, which either completely overrides the original watermark or causes
copyright ownership confusion upon watermark conflict, i.e., temporally infeasible
to distinguish the copyright violator and original developer solely based on recovered
watermark information. In our case, once the cipher block in a SDC segment is
cracked by either brute-force or dynamic analysis, an attacker can override the
original watermark code snippet with the attacker’s own watermark and re-generate
the SDC segment (assuming that attacker knows the watermark embedding process
and DM scanner rules). As we will see in evaluation, although it is possible to break
a subset of SDC segments in an app, it is tricky to crack all the SDC segments.
Moreover, there is no copyright ownership confusion upon watermark conflict for
Droidmarking, since the app signing mechanism guarantees that a watermark
conflict undoubtedly indicates repackaging practice by the app publisher (whose
public key certificate is contained in the app package).

Distortive Attack: An effective distortive attack applies a sequence of semantics-
preserving code transformations that leads to failure to recover original watermarks.
Since watermarks are encrypted with functional code and thus cannot be modified
by attackers directly without first decrypting the cipher blocks, attackers may
instead focus on modifying the normal decryption process (performed by library
libSDC) invoked by sdc_decrypt(). DM scanner can effectively impede this
attack by enforcing stringent rules, as we have discussed in Section 4.4.4.
Table 4.1. Open source Android apps from 9 categories

<table>
<thead>
<tr>
<th>Category</th>
<th># of Apps</th>
<th>Avg LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication</td>
<td>13</td>
<td>36,436</td>
</tr>
<tr>
<td>Development</td>
<td>27</td>
<td>10,513</td>
</tr>
<tr>
<td>Game</td>
<td>35</td>
<td>6,970</td>
</tr>
<tr>
<td>Multimedia</td>
<td>38</td>
<td>13,456</td>
</tr>
<tr>
<td>Navigation</td>
<td>37</td>
<td>9,298</td>
</tr>
<tr>
<td>Office</td>
<td>44</td>
<td>9,359</td>
</tr>
<tr>
<td>Reading</td>
<td>21</td>
<td>15,347</td>
</tr>
<tr>
<td>Security</td>
<td>9</td>
<td>5,993</td>
</tr>
<tr>
<td>Web Browser</td>
<td>4</td>
<td>9,903</td>
</tr>
</tbody>
</table>

4.6 Evaluation

We now proceed to the empirical evaluation of the efficacy of Droidmarking in the following facets: feasibility, resilience and performance. We consider a data set of open source Android apps from F-Droid [107], an catalogue of open source apps for the Android platform. We collected totally 228 apps from nine categories as shown in Table 4.1. Note that the lines of code (LOC) includes the Java source code of both the app and the libraries it includes in the apk package. We conduct our experiments on Samsung Nexus 4 emulator with Android 4.0 system. As proof-of-concept, our implementation of Droidmarking requires Dalvik virtual machine (DVM) to run in non-optimization mode, in order to prevent DVM from making changes to the format of SDC segments by doing bytecode optimization. Droidmarking is applicable to DVM optimization mode and needs DVM to be “Droidmarking-aware”, we leave to future work. We choose to use emulator in our experiments as it is flexible to configure and it does not affect our results/conclusions compared with using real devices in our study. Droidmarking is general enough to be applied to apps running on other OS versions and real devices.

4.6.1 Feasibility

We begin by studying the feasibility of Droidmarking system. In particular, we seek to understand if there are sufficient candidate branches in a Android app program for watermark embedding.

Figure 4.9 shows the population of candidate branches in the apps from different
categories. To even further understand the crypto strength of the keys used for SDC segments, we classify all candidate branches into three groups based on the type of constant values in the branch conditions: (1) weak - boolean or predefined values/strings, (2) medium - large numeric value, and (3) strong - string constant. We find that the density of candidate branches offered by an app source code varies depending on the nature of different categories of apps. For instance, a game app has almost 10 candidate branches per 1000 LOC on average, whereas a development app has less than half of this number. In general, most apps have a decent number of candidate branches either because of high candidate branch density (e.g., games) or large LOC (a communication app has more than 30,000 LOC and a total of 279 candidate branches on average). Development and security categories, however, have less than 50 candidate branches per app. We also find that weak and medium candidate branches count for roughly two-thirds of the candidate branch population. This is not good news since these constants are vulnerable to brute-force attack (Section 4.6.3.1).

**Key insights:** (1) The number of candidate branches varies considerably across different app categories. (2) There are sufficient candidate branches for watermark embedding in most applications.
4.6.2 How Soon to Recover a Watermark?

In this section, we seek to understand how quickly users can recover a watermark and verify the copyright information. In principle, a quick answer is that, it depends on the frequency and possibility that the program execution can be lead into the SDC segments at runtime.

For clarity of the following presentations, we show the results of three selective sample apps that are representative of different categories of apps having different ranges of candidate populations: Zirco Browser (ZB), Tomdroid Notes (TN) and Dudo Game (DG). Other apps have shown similar statistics and dynamic behaviors in watermark recovery and performance. We embedded watermarks into these apps and let human testers play each app for one hour. Figure 4.10(a) depicts the percentage of watermarks triggered in a time series. We find that all these apps can only recover less than 40 percent of all watermarks. DG recovers less than 10 percent. We also show in Figure 4.10(b) the average time to recover one watermark instance (multiple instances may be generated from the same watermark code snippet in an app program). The result is encouraging as it implies that users could expect relatively short waiting time (most likely less than a minute) before receiving and reviewing the first notice of copyright information from DM recognizer.

We also observe that the average time to recover a watermark varies significantly, depending on which part of program logic the SDC segments reside in from different apps. Take Tomdroid Note (TN) for instance, we find that one SDC segment is located inside a while loop which searches for a user input string by repeatedly scanning through all note contents. This program logic triggers this particular SDC segment repeatedly once the same string is found during runtime and generates a considerably large number of watermark instances in a short period of time. This is an important observation from the performance point of view (which we will discuss in more detail in Section 4.6.4), as we could effectively reduce overhead once we know where it comes from.

**Key insight:** Users could expect a reasonably short period of time to recover a watermark instance.
4.6.3 Resilience

4.6.3.1 Resilience to Static Analysis

Attackers may statically analyze the app program and launch brute-force attacks to the encryption keys of the cipher code blocks in SDC segments.

More specifically, recall that in watermark embedding phase (Section 4.4.2.3), DM embedder replaces original branch condition \( \text{Var} == \text{Const} \) with an equivalent condition \( \text{V} == \text{M} \), where \( \text{V} = \text{encrypt(S,Var)} \) and \( \text{M} = \text{encrypt(S,Const)} \). An attacker can brute force all possibilities of \( \text{Var} \) and determine the key when the resulting cipher text \( \text{V} \) is equal to \( \text{M} \). Therefore, the strength of a single SDC segment against brute-force attack depends on the bit length \( l \) of variable \( \text{Var} \). Since every SDC segment uses a different random string \( S \), it requires \( 2^l \) attempts at the maximum to get one key.

As presented in Section 4.6.1, although a considerable portion of weak and medium candidate branches are vulnerable to brute force attack, considering a large number of lengthy string values being the key in some categories of apps (e.g. communication and reading), brute-forcing all the keys in these apps may still be a formidable task for attackers.
<table>
<thead>
<tr>
<th>Apps</th>
<th>Total # of Surviving WMs</th>
<th>Avg Time to Recover a WM Inst.(min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZB</td>
<td>48 (77%)</td>
<td>28.2</td>
</tr>
<tr>
<td>TN</td>
<td>61 (68%)</td>
<td>5.1</td>
</tr>
<tr>
<td>DG</td>
<td>131 (94%)</td>
<td>20.2</td>
</tr>
</tbody>
</table>

Table 4.2. Surviving watermarks of each app

4.6.3.2 Resilience to Dynamic Analysis

An alternative attack is to dynamically execute the app. Once the branch condition of a SDC segment is satisfied, the value of the key is recorded and can later be used by an attacker to decrypt the corresponding cipher block in SDC segment. An attacker may even instrument the app and automate the app execution process, hoping that the keys to all cipher code blocks could be recovered automatically. There are many tools that attackers can use for this purpose, e.g., Android app stress testing tool like Monkey [108], or smarter unit test generator like Randoop [109] or Evosuite [110].

To this end, we would like to understand whether users can still be able to recover the original watermarks in a simple dynamic analysis attack. We employ Monkey to generate pseudo-random streams of user events to each app continuously for 12 hours and flag the watermarks recovered as “removed” by attackers. For those surviving watermarks that are not “removed”, we assume they are kept intact in the program. We then let human testers to manually practice each app for one hour and take down the surviving watermarks recovered during this time period. Figure 4.11(a) depicts the percentage of original watermarks removed from each app by a attacker in a time series, and Figure 4.11(b) shows the percentage of surviving watermarks that are later recovered from the “repackaged app” by human testers.

We make several important observations. First, although Monkey is slightly better than human testers, still has difficulty recovering all watermarks, only achieving less than 40 percent as shown in Figure 4.11(a). Second, Monkey removes most of the original watermarks that the human testers should have recovered when they are practicing the app Third, human testers can still trigger a small subset of surviving watermarks that Monkey does not trigger, even though human testers
only run the apps for one hour (compared to 12 hours of testing by Monkey) and feed much slower events to the apps than Monkey does. We re-do the experiments using Monkey but guided by human testers (to avoid redundant testing) for one hour and have found similar results.

By carefully analyzing the surviving watermarks triggered by the humans, we summarize several reasons of the above observations: (1) Some watermark-carrying SDC segments are executed only if they are given the “right” input. Take the text searching functionality of Tomdroid Notes for instance, the SDC segment can be triggered only if the note content contains the exact string of user input. (2) Some SDC segments cannot be easily triggered without human intelligence involved. We find this effect is especially pervasive in game apps like DG, in which certain SDC segments require problem-solving logic to be triggered. (3) Some SDC segments are only reached under very specific software/hardware or environment conditions, which vary considerably case-by-case. For example, the size of the screen and the orientation of the phone affects Zirco Browser’s execution logic.

An attacker may resort to smarter unit test generators, such as Randoop or Evosuite (the former uses feed-back directed and the latter uses mutation-driven test generation). However, these techniques cannot benefit from performing code analysis to the encrypted SDC segments, which weakens their capability of test generation for the SDC segments. Moreover, although these widely-used tools and have proved to be capable of finding software defects and achieving high code coverage for non-user-interactive traditional Java programs, the event-driven
Android applications have raised unique challenges to the effectiveness of these tools. These tools are not good at (or designed for) mimicking a diversity of user interactions, events, and software/hardware environments on an Android system. For these reasons, it is non-trivial to recover all SDC segments in an Android app even using state-of-the-art unit test generators.

Meanwhile, recall that it is sufficient for Droidmarking to flag an app as repackaged (with zero false-positive rate) as long as one original watermark in an SDC segment is recovered from one user device. Considering the large user base, a repackaged app cannot become fully “safe” from being detected unless the attacker manages to remove (or disable) all original watermarks. While it is possible for a dedicated attacker to achieve this by using a hybrid of attacks (sophisticated static and dynamic analysis) and/or directly removing the un-recovered SDC code segments (losing program functionality), the resulting cost may no longer be attractive for an attacker.

**Key insights:** (1) Droidmarking provides different levels of defense against brute force attack, depending on the number and types of candidate branch conditions in an app. (2) In a Monkey-based dynamic analysis attack, users are still able to successfully recover original watermarks. For smarter testing tools, the SDC segments in Android apps have raised unique challenges to their effectiveness. (3) To completely remove all watermarks, an attack may resort to a combination of various attacks, which increases the cost of app repackaging.

### 4.6.4 Performance

The application performance overhead introduced by Droidmarking comes from two major types of computations: (1) the encryption operation (in bytecode) computing the equivalent branch condition before each SDC segment, and (2) the decryption operation of the cipher code block performed by libSDC (in native code) when the above condition satisfies for the first time. To evaluate the overhead, we use Monkey to generate the same sequence of 2,000 user events to both the apps with and without watermarks embedded. We measure the run time $T$ and $T_w$ for original and watermarked apps respectively. The overhead is calculated by $(T_w - T)/T \times 100\%$. Table 4.3 shows the performance overhead of Droidmarking for the three apps. We find that the performance overhead of app DG is relatively low (around 3 percent).
although it is a game app and has higher candidate branch density. However, the overheads of ZB and TN are particularly higher, even though the ZB’s watermark recovery frequency is the lowest (see Figure 4.10). In fact, this is not surprising. ZB and TN represent a large group of apps, in which a few candidate branches reside in the “popular” loops in these apps, e.g., the program logic of text search in TN app, as we have discussed in Section 4.6.2. Although these branches are not necessarily entered every time, the branch conditions (encryption) are always calculated. This is the reason why ZB has high overhead in spite of low watermark recovery frequency.

To improve performance, one could opt for a smarter watermark embedding strategy. For instance, in the watermark embedding phase, one could skip those candidate branches in the scopes of loop statements. To be more precise, one can potentially build the program dependency graph of the app and choose to “filter out” those candidate branches in the “popular” execution path of the program. In this way, one can even quantitatively evaluate the trade-off among performance, speed of watermark recovery, and resilience of Droidmarking scheme based on one’s demand. We leave this for future work.

**Key insight:** (1) The overhead is relatively small on some apps but higher on others, mainly depending on whether branch conditions reside in frequent execution paths of the app program. (2) Optimized watermark embedding strategies can be applied to reduce performance overhead.

### 4.7 Discussion and Conclusion

There are several limitations of Droidmarking. First, the types of candidate branches that our watermark can be embedded to is still limited, (i.e., the conditions of candidate branches must be equality test). Operators such as $<$, $>$ are not applicable. Second, the encryption strength relies on the number of possible values

<table>
<thead>
<tr>
<th>Apps</th>
<th>$T_{w/o}$ WM(sec)</th>
<th>$T_w$ with WM(sec)</th>
<th>App Perf. Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZB</td>
<td>142</td>
<td>150</td>
<td>5.1%</td>
</tr>
<tr>
<td>TN</td>
<td>196</td>
<td>212</td>
<td>8.2%</td>
</tr>
<tr>
<td>DG</td>
<td>177</td>
<td>182</td>
<td>2.8%</td>
</tr>
</tbody>
</table>

Table 4.3. Application performance overhead.
and length of a numerical/string constant that is used in a candidate branch. As we have presented in Figure 4.9, roughly two-thirds of the candidate branch conditions use predefined values, which are in some cases not difficult to be enumerated by attackers. Moreover, for some string constants, understanding the semantics of the branch conditions is helpful for an attacker to guess their values. Take the candidate branch in Figure 4.2(a) for instance, once an attacker knows that variable \texttt{url} refers to a web resource identifier, it becomes substantially easier than brute force to guess the string constant. Third, the introduction of cipher code blocks may be used by malware writers for code obfuscation although stringent rules are enforced by the DM scanner.

In conclusion, we have designed a new “non-stealthy” dynamic software watermarking technique - Droidmarking - based on SDC for Android apps. We evaluate Droidmarking’s efficacy in feasibility, resilience and performance on real-world open source apps. Our results prove that Droidmarking is able to achieve all our goals in impeding app repackaging in today’s Android markets.
Chapter 5  
Towards Discovering and Understanding Task Hijacking in Android

5.1 Introduction

In the PC world, computer multitasking means multiple processes are running at the same period of time. In Android systems, however, multitasking is a unique and very different concept, as defined in Android documentation: “A task is a collection of activities that users interact with when performing a certain job” [111]. In other words, a task contains activities [112] (UI components) that may belong to multiple apps, and each app can run in one or multiple processes. The unique design of Android multitasking helps users to organize the user sessions through tasks and provides rich features such as the handy application switching, background app state maintenance, smooth task history navigation using the “back” button, etc. By further exposing task control to app developers, Android tasks have substantially enhanced user experience of the system and promoted personalized features for app design.

Despite the merits, we find that the Android task management mechanism is plagued by severe security risks. When abused, these convenient multitasking features can backfire and trigger a wide spectrum of task hijacking attacks. For instance, whenever the user launches an app, the attacker can condition the system to display to the user a spoofed UI under attacker’s control instead of the real UI.
from the original app, without user’s awareness. All apps on the user’s device are vulnerable, including the privileged system apps. In another attack, the malware can be crafted as one type of ransomware, which can effectively “lock” the tasks that any apps belong to on the device (including system apps or packages like “Settings” or “Package Installer”), i.e. restricting user access to the app UIs and thus disabling the functionality of the target apps; and there is no easy way for a normal user to remove the ransomware from the system. Moreover, Android multitasking features can also be abused to create a number of other attacks, such as phishing and spyware. These attacks can lead to real harms, such as sensitive information stolen, denial-of-service of the device, and user privacy infringement, etc.

However, the Android multitasking mechanism and the underlying feature provider, the Activity Manager Service (AMS), haven’t been studied thoroughly before. In this paper, we take a first step to investigate the security implications behind the multitasking design and the AMS in Android. At the heart of the problem, although the Android security model renders different apps sandboxed and isolated from one another, Android allows the UI components (i.e., activities) from different apps to co-reside in the same task. Given the complexity of task dynamics, as well as the vagaries of additional task controls available to developers, the attacker can play tricky maneuvers to let malware reside side by side with the victim apps in the same task and hijack the user sessions of these apps. We call this task hijacking.

It becomes important to study Android multitasking behaviors in a systematic way. We approach this topic by projecting the task behaviors into a state transition model and systematically study the security hazards originated from the the discrepancies between the design assumptions and implementations of Android tasks. We find that there is a plethora of opportunities of task hijacking exploitable to create a wide spectrum of attacks. To showcase a subset of the attack scenarios and their consequences, we implement and present a set of proof-of-concept attacks as shown in Table 5.1.

We do vulnerability assessment to the task hijacking threats and discover that all recent Android versions, including Android 5, can be affected by these threats, and all apps (including all privileged system apps) are vulnerable to most of our proof-on-concept attacks on a vulnerable system. By investigating the employment
Table 5.1. Types of task hijacking attacks presented in this paper (all system versions considered - Android 3.x, 4.x, 5.0.x).

<table>
<thead>
<tr>
<th>Attacks Types</th>
<th>Consequences</th>
<th>Vulnerable system &amp; apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spoofing</td>
<td>Sensitive info stolen</td>
<td>all; some vulnerable apps</td>
</tr>
<tr>
<td>Denial-of-service</td>
<td>Restriction of use access to apps on device</td>
<td>all; all</td>
</tr>
<tr>
<td>Monitoring</td>
<td>User privacy infringement</td>
<td>Android 5.0.x; all</td>
</tr>
</tbody>
</table>

of task control features by app developers based on 6.8 million apps in various Android markets, we find that despite the serious security risks, the “security-sensitive” task control features are popular with developers and users. We have reported our findings to the Android security team, who responded to take a serious look into the issue. We summarize our contributions below:

- To our best knowledge, we are the first to systematically study the security implications of Android multitasking and the Activity Manager Service design.

- We discover a wide open attack surface in Android multitasking design that poses severe threat the security of Android system and applications.

- Base on our vulnerability analysis over 6 million apps, we find that this problem is prevalent and can be used by attackers to cause a variety of serious consequences.

- We provide mitigation suggestions towards a more secure Android multitasking sub-system.

5.2 Background

Android Application Sandbox: The Android security model treats third-party apps as untrusted and isolates them from one another. The underlying Linux kernel enforces the Linux-user based protection and process isolation, building a sandbox for each app. By default, the components of one app run in the same Linux process with an unique UID. Components from different apps run in separate processes. One exception is that different apps can run in one process only if they are from the same developer (granted the same UID), and the developer explicitly specifies this
in the manifest file. This Linux sandbox provides the foundation for app security in
Android. In addition, Android provides a permission model \cite{66,113} to extend app
privileges based on user agreement, and offers an inter-component communication
scheme guarded by permissions for inter-app communications.

Activity: Activity is a type of app component. An activity instance provides a
graphic UI on screen. An app typically has more than one activities for different
user interactions such as dialing phone numbers and reading a contact list. All
activitys must be defined in the app’s manifest file.

Intent: To cross process boundaries and enable communications between app
components, Android provides an inter-component communication (ICC) scheme
supported by an efficient underlying IPC mechanism called binder. To perform
ICC with other components, an component use intent, an abstract description of
the operations to be performed. An intent object is the message carrier object used
to request an action from another component, e.g., starting an activity instance
by calling startActivity() function. Intent comes in two flavors. Explicit
intent specifies the component to start explicitly by name. Implicit intent instead
encapsulates a general type of action, category or data for a component to take.
The system will launch a component “capable” of handling this intent. If more
than one possible target activities exist, the user is prompted to choose a preferred
one.

Activity Manager Service (AMS): AMS is an Android system service that
supervises all the activity instances running in the system and controls their life
cycles (creation, pause, resume, and destroy). The interaction and communication
protocols between activities and the AMS are implemented by the Android frame-
work code and is transparent to app developers, leaving developers focusing on
the app functionality. AMS organizes all the activities in the system into tasks. It
is responsible for task management and supporting the multitasking features, as
described in Section 5.3.

In addition, AMS is in charge of supervising service components, intent routing,
broadcasting, content providers accesses, app process management, etc., making
itself one of the most critical system services in the Android system.
5.3 Android Tasks State Transition Model

5.3.1 Task and Back Stack

In Android, a task [111] is a collection of activities that users have visited in a particular job. Activities in the task are kept in a stack, namely back stack, ordered by the time the activities are visited, such that clicking the “back” button would navigate the user back to the most recent activity in the current task. The activities in the back stack may be from the same or different apps.

The activity displayed on the screen is a foreground activity (on the top of the back stack) and the task associated with it is a foreground task. Therefore, there is only one foreground task at a time and all other tasks are background tasks. When switched to the background, all activities in a task stop, and remain intact in the back stack of the task, such that when the users return they can pick up from where they left off. This is the fundamental feature that multitasking has to offer to users.

5.3.2 A Tasks State Transition Model

The status of tasks in a system keeps changing as a result of user interaction or app program behaviors. To understand the complex task dynamics and its behind security implications, we view the task transitions through time as a state transition model. The model is described by \((S, E, \Lambda, \rightarrow)\), where \(S\) denotes a set of task states; \(E\) and \(\Lambda\) are sets of events and conditions respectively; and \(\rightarrow\) indicates a set of feasible transactions allowed by the system under proper events and conditions.

1. **Task state** \((s \in S)\): represents the state of all tasks (specifically, the back stacks) in the system and their foreground/background statuses. In other words, the tasks in the system remain in one state iff the activity entries and their orders in the back stacks stay the same, and the foreground task remains to be the same task.

2. **Event** \((e \in E)\): denotes the event(s) it takes to trigger the state transition, for example, pressing the “back” button or calling `startActivity()` function.

3. **Condition** \((\lambda \in \Lambda)\): the prerequisites or configurations (usually default) that enable a state transition under certain events. We denote \(\lambda_{\text{default}}\) as the
(a) Figure 5.1. A simple task state transition example.

system default conditions in this paper.

4. **Transition** (→): stands for a feasible state transition. Not all task transitions are feasible, e.g., the order of activities in back stack cannot be changed arbitrarily (only push and pop are viable operations over the stack). A viable transaction is also represented as \( s_1 \rightarrow s_2 \), or \( (s_1, s_2, e, \lambda) \), where \( s_1, s_2 \in S \).

### 5.3.3 A Task State Transition Example

Given the state transition model, we depict a simple task state transition example in Figure 6.4. The figure shows three task states, and the state transitions reflect the process in which the user first launches an app from the home screen \( (s_0 \rightarrow s_1) \), visits an additional activity UI in the app \( (s_1 \rightarrow s_2) \) and return to the home screen by pressing the “back” button twice \( (s_2 \rightarrow s_1 \rightarrow s_0) \).

In each task state, we show all existing tasks and their back stacks. For example, \( s_0 \) is a task state in which no task, except the launcher task, is running in the system. The launcher task has only one activity in its back stack - the home screen from which users can launch other apps.

In \( (s_0, s_1, e^{\text{start}}, \lambda^{\text{newtask}}) \), a new app task is created and brought to the foreground in the resulting state \( s_1 \). \( e^{\text{start}} \) represents the event that \( \text{startActivity}() \) is called by the home activity in the launcher task. This event could happen when
the user clicks the app’s icon on the home screen. $\lambda^{\text{newtask}}$ specifies a special condition, i.e., the FLAG_ACTIVITY_NEW_TASK flag is set to the input intent object to $\text{startActivity()}$ function. This flag notifies the AMS the intention of creating a new task to host the new activity. Note that in this example most state transitions are under default conditions, indicated by $\lambda^{\text{default}}$, while here $s_0 \rightarrow s_1$ is an exception because the launcher app customizes the condition ($\lambda^{\text{newtask}}$) for its own valid design purpose: when the user launches a new app, start the app in a brand new task. This is an example where app developers can customize certain configurable conditions to implement helpful app features. However, condition like $\lambda^{\text{newtask}}$ can be abused in a task hijacking attack, as discussed in Section 5.4.

Next, $(s_1, s_2, e^{\text{start}}, \lambda^{\text{default}})$ is triggered by event $e^{\text{start}}$ again (this time called by activity A instead) but under the default condition. By default, AMS pushes the new activity instance B on top of the current back stack as shown in $s_2$. The previous activity A is stopped and its state is retained. In $(s_2, s_1, e^{\text{back}}, \lambda^{\text{default}})$, $e^{\text{back}}$ represents the event of user pressing the “back” button. As expected by the user, the next activity A on stack is brought back to the screen, and its original state is resumed. Activity B is popped from the back stack and destroyed by the system. The initial state $s_0$ is finally restored through $(s_1, s_0, e^{\text{back}}, \lambda^{\text{default}})$ when the user presses “back” button again. The app’s task is destroyed because when the popped activity is the last activity in the back stack, the activity is destroyed together with the “empty” task.

Note that activities from different apps can co-reside in the same task (e.g. activity A and B in this example). In other words, although activities from different apps are isolated and protected within their own process sandboxes, Android allows different apps to co-exist in a common task. This creates opportunities for malicious activities to interfere with other activities once they are placed in the same task, and the system passes the program control to the malicious activities.

In reality, the amount of possible task states in a system is big, and the state transitions can be complex, e.g., each state may again have numerous incoming and outgoing transitions connecting with other states. In Section 5.4, we discuss what may go wrong during the complex task state transitions.
5.3.4 Android Implementation

AMS maintains Android tasks and activities in a hierarchy shown in Figure 5.2. AMS uses TaskRecord and ActivityRecord objects to represent tasks and activities in the system respectively. A TaskRecord maintains a stack of ActivityRecord instances, which is the back stack of that task. Similar to the activities in a back stack, tasks are organized in a stack as well, maintained by a ActivityStack object, such that when a task is destroyed, the next task on stack is resumed and brought to the foreground. There are usually two ActivityStack containers in the system - one containing only the launcher’s tasks and the other holding all remaining app tasks.

5.4 Task Hijacking in Android

5.4.1 Adversary Model

We assume the user’s Android device already has a malware installed. The malware pretends to seem harmless, requiring only a minimum set of widely-requested permissions such as INTERNET permission. The attacker’s goal is clear: blend the malicious activities with the target app’s activities in one task, and intercept the normal user operations to achieve malicious purposes.
Table 5.2. Task control knobs - configurable task state transition conditions and events provided by Android.

5.4.2 Hijacking State Transition

A hijacked task state is a desirable state to attackers, in which at least one back stack (or task) in the system contains both malicious activities (from malware) and benign activities (from the victim app). In an attack, the attacker needs at least one hijacked state to succeed. A hijacking state transition (HST) is a state transition which turns the tasks in the system to a dangerous hijacked task state. Conceptually, there are two types of HSTs:

1. The malicious activity gets pushed onto the victim task’s back stack (malware⇒victim);

2. The victim app activity is “tricked” by malware and pushed on the malware’s back stack (victim⇒malware).

5.4.3 The Causes of HSTs

Android provides a rich set of task control features, i.e., task state transition conditions and events. We call these features as task control knobs. The task control knobs has granted app developers with broad flexibility in controlling the launch of new activities, the relocation of existing activity to another task, “back” button behaviors, even the visibility of a task in the recent task list (a.k.a overview
screen), etc. Table 5.2 lists such conditions and events in four categories: activity attribute, intent flags, call-back functions, and framework APIs. All these control flexibility further complicates task state transitions. Due to HST’s potential threats to app and system security, understanding the extent of HSTs in the complicated task state transitions becomes important.

To achieve this, we simulate the tasks in a Android system based on our state transition model with the purpose to capture all possible task states (including hijacked task states) and state transitions (especially the HSTs) in the system.

In theory, there is a huge number of possible task states, since each app may have a number of activities, and an activity can be instantiated for multiple times. We confine the number of task states to more interesting cases by adding two constraints: (1) each app only has two activities - the main activity and another public exported activity (can be invoked by other apps), and (2) each activity can only be instantiated once. In the simulation, we specify three apps in the system - namely, Alice, Bob and Mallory (the malware) - as it covers most HST cases.

Given the task states, the task state transition graph is constructed by connecting pairs of states with directed edges. For instance, state 1 \((s_1)\) and 2 \((s_2)\) are connected only if \(\exists e \in E, \lambda \in \Lambda\), such that \((s_1, s_2, e, \lambda)\) or \((s_2, s_1, e, \lambda)\) are valid transitions, where \(E\) denotes all feasible events and \(\Lambda\) represents all possible conditions in Table 5.2. After this, all hijacked states and HSTs are found and highlighted in the graph. Figure 5.3(a) shows a sub-graph of the over-sized task state transition graph for a simulated system with three apps. The sub-graph shows the typical cases of HSTs (red edges) and the resulting hijacked task states (red nodes). \(s_0\) represents the initial state, i.e., no tasks except the launcher task exists in the system. Figure 5.3(b) visualizes the task states of all nodes in figure (a). A, B and M represent the activities from Alice, Bob and Mallory (the malware) respectively. We skip showing the launcher task in the task states. Hijacked states are highlighted as red boxes. F and B denote foreground and background tasks respectively.

For clarity of the presentation, we only show the interesting branches of the over-sized graph and have skipped many duplicated HST cases. Moreover, we zoom in each of the HSTs and show their detailed information in Table 5.3, in particular, their task state transition conditions and events. We manually verify all presented HSTs on real systems and these HSTs are proven to be exploitable to launch real attacks (the last column in Table 5.3) as we will see in Section 5.5.
Figure 5.3. (a) A sub-graph of the over-sized task state transition graph for a simulated system with three apps. (b) Visualization of task states of all nodes in figure (a).

<table>
<thead>
<tr>
<th>#</th>
<th>HST Type</th>
<th>Conditions</th>
<th>Events</th>
<th>Attacks in Section 5.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>malware⇒victim</td>
<td>M1:taskAffinity=B2; NEW_TASK intent flag set</td>
<td>A1: startActivity(M1)</td>
<td>phishing I</td>
</tr>
<tr>
<td>2</td>
<td>victim⇒malware</td>
<td>M1:taskAffinity=B2; NEW_TASK intent flag set or B2:launchMode=&quot;singleTask&quot;</td>
<td>A2: startActivity(B2)</td>
<td>phishing II</td>
</tr>
<tr>
<td>3</td>
<td>malware⇒victim</td>
<td>M2:taskAffinity=A1; M2:allowTaskReparenting=&quot;true&quot;; NEW_TASK intent flag set</td>
<td>launcher: startActivity(A1)</td>
<td>spoofing</td>
</tr>
<tr>
<td>4</td>
<td>victim⇒malware</td>
<td>M1:taskAffinity=A1; NEW_TASK intent flag set</td>
<td>launcher: startActivity(A1)</td>
<td>denial-of-use; ransomware; spyware</td>
</tr>
<tr>
<td>5</td>
<td>victim⇒malware</td>
<td>M1:taskAffinity=B2; B2:allowTaskReparenting=&quot;true&quot;</td>
<td>startActivities([M1, M2]) or use TaskStackBuilder</td>
<td>phishing III</td>
</tr>
<tr>
<td>6</td>
<td>malware⇒victim</td>
<td>M2:taskAffinity=A1; NEW_TASK intent flag set</td>
<td>M1: startActivity(M2)</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.3. Detailed information of the HSTs in Figure 5.3. E.g., condition “M1:taskAffinity=B2” indicates that the taskAffinity attribute of activity M1 is set to that of B2; Event “launcher:startActivity(A1)” means that activity A1 is started by the launcher.
We make two important observations from our analysis. First, once exploited, the hijacked states shown in Figure 5.3(a) could result in serious security hazards. For example, in \( s_6 \) and \( s_{14} \), the malware activities M1 and M2 are able to intrude into Alice’s task and are brought to the foreground. The screen is thus under the control of Mallory and it can launch potential attacks such as user spoofing. On the other hand, in \( s_9 \) and \( s_{18} \), the benign activities A1 and B2 are “deceived” and put into Mallory’s task upon creation. This can lead to phishing attack, or cause GUI confidentiality breaches.

Second, compared with the HST triggered by the default conditions and events (HST #1 in Table 5.3), more HSTs scenarios (HST #2-6) are produced under the configurable conditions and events. In other words, although HST can occur even under default conditions and events, e.g., a user accidentally starts a malware activity in the current task like in HST #1, the flexibility and impact of these “passive” HSTs are quite limited. On the other hand, By abusing the flexible task control “knobs” readily offered by the Android system, the attacker can actively create a plethora of HSTs targeting any apps. In Figure 5.3(a), we only show several typical HST cases, yet there are much more HST instances of these types in the complete state transition graph.

The HST cases and their conditions/events summarized in Table 5.3 may now look mysterious. We demystify these conditions and events for the rest of this section.

### 5.4.4 Exploiting Conditions

In Table 5.3, HSTs #2, #4, #6 are similar in terms of their state transition conditions, i.e. all three HSTs occur by virtue of customized activity launch mode (by setting `launchMode` attribute or NEW_TASK intent flag). HSTs #3, #5 are similar as they both use `allowTaskReparenting` attribute to enable activity re-parenting.

#### 5.4.4.1 Activity Attributes

One can define the attributes [114] of an activity in the `<activity>` element in manifest file. The attributes not explicitly defined are set to default values.

**Task Affinity**: Task affinity declares what task an activity prefers to belong to.
It is a hard-coded string defined as `<android:taskAffinity="affinity">`, where `affinity` is the task affinity string that can be defined arbitrarily. By explicitly declaring a task affinity, an activity is able to actively “choose” a preferable task upon start-up or during its life cycle. If not specified, the task affinity of an activity is by default set to the package name of the app, such that all activities in an app prefer to reside in the same task by default. The affinity of a task is determined by the task affinity of the task’s root activity (the activity on the bottom of back stack).

Task affinity is a crucial condition used in most of the HSTs in Table 5.3. There are two occasions in which an activity can “choose” its preferred host task: (1) when an activity attempts to be started as a new task (i.e., “singleTask” launch mode or NEW_TASK intent flag as in HST #2, #4, #6), and (2) if the allowTaskReparenting activity attribute is set to true, and another task with the same task affinity is brought to the foreground (as in HST #3, #5). We explain the above two cases in detail in the following paragraphs.

**Launch Mode**: Activity launch mode defines how an activity should be started by the system. Based on the launch mode, the system determines: (1) if a new activity instance needs to be created, and (2) if yes, what task should the new instance be associated with. The launch mode can be either statically declared by specifying `<android:launchMode="value">` in the manifest file or dynamically defined using intent flags discussed in Section 5.4.4.2.

By default, `launchMode="standard"`. In this mode, the AMS would create a new activity instance and put it on top of the back stack on which it is started. It’s possible to create multiple instances of the same activity and those instances may or may not belong to the same task. With `launchMode="singleTask"`, the decision-making of activity start-up is more complex. An investigation into Android source code reveals three major steps the AMS takes towards starting an activity. First, if the activity instance already exists, Android resumes the existing instance instead of creating a new one. It means that there is at most one activity instance in the system under this mode. Second, if creating a new activity instance is necessary, the AMS selects a task to host the newly created instance by finding a “matching” one in all existing tasks. An activity “matches” a task if they have the same task affinity. After finding such a “matching” task, the AMS puts the new instance into the “matching” task. This explains why in HST #2 and #6, the
newly-started and foreground activities (B2 and M2) are put on other “matching” tasks (with the same task affinity) instead of the tasks who start them. Third, without finding a “matching” task, the AMS creates a new task and makes the new activity instance the root activity of the newly created task.

**Task Re-parenting:** By default, once an activity starts and gets associated with a task, such association preserves for the activity’s entire life span. However, setting `allowTaskReparenting` to true breaks this restriction, allowing an existing activity (residing on an “alien” task) to be re-parented to a newly created “native” task, i.e., a task having the same task affinity as the activity.

For example, in HST #3, M2 is supposed to stay on Mallory’s task for its entire life. However, M2 has its `allowTaskReparenting` set to true, and `taskAffinity` set to Alice’s package name, such that when Alice’s task is started (A1 as the root activity) by the launcher, M2 is re-parented to Alice’s new task and the user sees M2 on screen instead of A1. In fact, A1 is never shown on screen at all in this case. Likewise, HST #5 happens due to similar reason, except that this time the benign activity B2 (with its `allowTaskReparenting` set to true) is re-parented to the malware task.

The above activity attributes give flexibility to attackers. The attackers can put their malicious activities to a preferred hosting tasks under certain events, e.g., `singleTask` launch mode during an activity start-up and `allowTaskReparenting` during a new task creation. Furthermore, an activity is free to choose any app as their preferred task owner (including the privileged system apps) by specifying the target app’s package name as their task affinity. These conditions lead to a good number of HSTs in the simulation, and these HSTs can be very useful in powerful task hijacking attacks presented in Section 5.5.

### 5.4.4.2 Intent Flags

Before sending an intent to start an activity, one could set intent flags to control how the activity should be started and maintained in the system by calling `intent.setFlags(flags)`. `intent` is the intent object to be sent, and `flags` is an `int` value, each bit of which indicates to the AMS about a particular flag (or configuration).

Noticably, the `FLAG_ACTIVITY_NEW_TASK` intent flag, if set, lets an activity be
started as if its `launchMode="singleTask"`, i.e. the system goes through the same procedures as explained in launch mode to find a “matching” task or create a new task for the new activity instance. This is the dynamic way of setting activity’s launch mode. Launcher app always uses this flag to start an app in a new task as in HST #4.

5.4.5 Exploiting Events

5.4.5.1 Callback Function

Android framework provides a variety of callback functions for activities to customize their behaviors under particular events, e.g., activity life cycle events (start, pause, resume or stop), key pressing events, system events, etc.

`onBackPressed()` is a callback function defined in `Activity` class, and is invoked upon user pressing the “back” button. The default implementation in framework code simply stops and destroys the current activity and resumes the next activity on top of the current back stack, as we have seen in Section 5.3.3. However, an attacker can override this callback function for its malicious activity and arbitrarily define a new behavior upon “back” button pressing, or simply disable the “back” button by providing an empty function. As a result, once the malicious activity is brought to the foreground, pressing the “back” button triggers the code of attacker’s control.

5.4.5.2 Framework API

Android framework provides APIs to create new tasks with established back stacks. For example, `TaskStackBuilder` is a utility class that allows an app developer to construct a back stack with specified activities, and to start the back stack as a brand new task in the system at a later time (e.g. using a PendingIntent). Similarly, `startActivities()` in `Activity` class achieves the same thing except that it builds and starts the tasks in one API function call. These framework APIs are helpful for attackers to build and launch new tasks containing designated back stacks without explicitly displaying all activities in the back stacks on screen.
5.5 Task Hijacking Attack Examples

In this section, we demonstrate attack examples utilizing exploitable HSTs we find in Section 5.4 to breach the integrity, availability and confidentiality of victim apps’ UIs respectively. We have tested these attacks on Android 3.x, 4.x and 5.0.x.

5.5.1 Breaching UI Integrity

We clarify that the attack examples in this section breaches the “origin/source integrity” of the victim app’s activities, instead of the “data integrity”. That is, instead of modifying the original activities of the victim app, attackers deceive the user by spoofing UIs, which can prevent the original UIs from being displayed on screen.

5.5.1.1 Spoofing Attack

In this attack, when the user launches a victim app from the home screen, a spoofing activity with a UI masquerading the victim app’s main activity shows up. The user is not able to recognize the UI as spoofing since the spoofing activity has already become part of the victim app’s task and the original activity has no chance to show itself on screen at all.

Figure 5.4 shows the HST of this attack, which is the similar case as HST#3 in Table 5.3. $s_1$ depicts a tasks state in which the home screen is shown to the user while a malware task lurks in the background. The taskAffinity of the malware activity “mal-main” is intentionally set to match the victim app and its allowTaskReparenting is set as “true”. When the user clicks an app icon on the
the home screen, a new task of the victim app is created and its main activity is at the root in the back stack as shown in \( s_2 \). “mal-main” is then reparented from the background malware task to the top of the victim app’s back stack, becoming the foreground activity displayed to the user. From the very beginning of the victim app’s launch, the app task has already been smoothly hijacked and all user interactions are now under attacker’s control.

In this way, the attacker can make spoofing activities disguised as any app that the attacker specifies, including all third-party and privileged system apps (e.g. Settings). The attacker can even target multiple victim apps on user device at one time, as long as he/she has started the malware tasks in advance, and each of the targeted victim activities is arranged in a separate task.

**Stealthiness:** Since the spoofing activity is part of victim app’s task, the overview screen shows the spoofing activity in the task of the victim app, rising no suspicion to the user. The attacker can also make its background malware tasks absent from the overview screen by setting the activity attribute `excludeFromRecents` to true. By accidentally clicking the “back” button from spoofing activity (“mal-main”), the user may resume the original root activity (“main”) in victim app’s task. To prevent this from happening, the attacker can override `onBackPressed()` of “mal-main” activity, such that the screen is re-directed to the home screen, giving the user an illusion that pressing “back” button is in coherence with the system’s default behavior.

### 5.5.1.2 Phishing Attack - “Back Hijacking”

The back button is popular with users to navigate back through the history of activities they have worked with. However, attackers may abuse the back button to mislead the user into a phishing activity.

We devise three phishing attack methods and demonstrate two of them that target at the same banking app. Figure 5.5 shows the phishing attack process. The phishing UIs show up when the user returns from the video playing, and the user unwittingly believes that he/she has returned to the original bank activity.

Figure 5.6 shows the state transition diagrams of two attack methods. The second attack differs from the first attack slightly. In the first attack, user chooses a malicious video player. While in the second one, even though the user selects a
Figure 5.5. “Back Hijacking” phishing attack to a well-known bank app. (a) the main activity of the bank app, with a tutorial video link for new users; (b) A system dialog showing available video players after clicking the video link; (c) video player activity, user clicks “back” button to quit the video and “goes back” to the original main activity; (d)-(e) the back button directs the user to the phishing UIs, which steal users’ bank account information, then quit after user clicks “Sign On”; (f) The original main activity is resumed, with a log-in failure error message left by the malware on screen.

benign player, the bank task is still susceptible to be hijacked. We now discuss the two attack methods separately.

**Method I**: Figure 5.6(a) shows the state transition diagram of the first attack method. We skip the unrelated task(s) (e.g. launcher) in the system and only show tasks of interest. In $s_1$, the bank app task contains activities A and B, in which B is the login activity. The HST occurs in $s_1 \rightarrow s_2$, triggered by the event that the user clicks the tutorial video from the login UI, sending out a implicit intent to look for an exported activity in the system capable of playing the tutorial video. Unfortunately,
the user selects the malicious video player activity “mal-player” from the system pop-up and this results in the hijacked state $s_2$. After user finished watching the video, $s_2 \rightarrow s_3$ is triggered by user pressing the “back” button. However, the “back”-pressing event is modified by overriding `onBackPressed()` in “mal-player” activity: instead of resuming activity B, a new malicious task is created using `TaskStackBuilder`, and brought to front in $s_3$ after the “back”-pressing. As can be seen, the HST takes place under default conditions as in HST#1.

From now one, the user session is hijacked by the malicious player activity to the malware task, which contains “mal-A” and the foreground “mal-B” phishing activities that resemble the UI of original bank activities A and B. Note that in this attack, the malware need to camouflage as a useful app (e.g. a video player in this case) that users are willing to use.

**Method II:** As shown in Figure 5.6(b), the same phishing attack can succeed even when the user selects a benign video player for the tutorial. In $s_1$, a malware task with two phishing activities lurks in the background. Similarly, HST occurs in $s_1 \rightarrow s_2$, when the user launches a benign video player. However, as shown in the resulting state $s_2$, instead of joining the banking task, the new video player activity
is pushed in the malware task’s back stack, such that pressing the “back” button after the video play resumes the phishing activity “mal-B”.

This HST is similar to HST#2 in Table 5.3 in the sense that the benign video player attempts to be started as a new task, either because of the NEW_TASK flag set in the intent by the bank activity, or the “singleTask” launch mode set by the video player. Furthermore, the existing malware task has its taskAffinity maliciously set to the benign video player.

Stealthiness: We employ similar methods in the spoofing attack to ensure the stealthiness of the background malicious tasks in both attack methods, i.e. to make malicious tasks invisible and unreachable from user. Moreover, we disable the animation of task switchings, producing an illusion to the user that the screen transition is within the same task/app.

5.5.2 Breaching UI Availability

Task hijacking can also be leveraged to restrict the availability of an app’s UI components, or in other words, to prohibit user access to part or all functionality of an victim app.

5.5.2.1 Preventing Apps from Being Uninstalled

In this example, the attacker is able to completely prevent apps from being uninstalled.

Ways to Uninstall An App: There are generally three ways for a user to uninstall an app from the device: (1) uninstall from the system Settings app; (2) dragging the app icon to the “trash bin” on home screen; or (3) uninstall with the help of a third-party app, e.g. an anti-virus app. In these scenarios, the Settings, Launcher, and the third-party apps will respectively generate an request to uninstall the app. Such a request eventually reaches the system package installer, which has the exclusive privilege to install/uninstall apps. Upon receiving the request, package installer pops up a dialog for the user to confirm. The dialog itself is an activity (namely uninstaller activity) from the system package installer and is pushed in the back stack of whoever is making the request (e.g. $s_4$ in Figure 5.7). No app can be uninstalled without user confirmation on the uninstaller activity.

Attack Method: The attacker can prevent app uninstallation by restricting user
access to the uninstaller activity when it shows up on screen. In this attack, once the uninstaller is found to be in the foreground, a malicious activity is immediately pushed on top of the uninstaller activity in the same back stack, such that the uninstaller is “blocked” and can even be destroyed by the foreground malicious activity.

Figure 5.7 shows the state transition diagram of this attack targeting Settings app. Similar methods can be easily adopted to block app uninstallation from the launcher or the anti-virus apps (e.g. when malware is detected).

In s₁, a task with only one root activity (“mal-root”) from the malware is hiding in the background, with its taskAffinity set to the Settings app (com.android.settings). The HST occurs in s₁ → s₂, triggered when the user opens up the Settings from the home screen (we skip Launcher task in the figure). In s₂, instead of hosting the newly-created “setting” activity in a new task, “setting” activity is pushed on top of the malware’s back stack because the it is started by the Launcher with a NEW_TASK-flagged intent. As a result, upon start-up, the
privileged Settings app is unwittingly “sitting” on a task owned by the malware. This is similar to HST#4 in Table 5.3.

The user then goes through a few more sub-setting menus to find the app (as shown in $s_3$) and clicks the uninstall button, after which the uninstaller activity shows up for user confirmation (as shown in $s_4$). Once this happens, a malicious activity namely “mal-blocker” is immediately (even without user awareness of the uninstaller dialog) launched by a malicious background service, which keeps monitoring the foreground activity. The “mal-blocker” activity, started by a NEW_TASK flagged intent and with the same task affinity as the Settings app, is thus pushed in the same task, and effectively blocks the uninstaller as shown in $s_5$. The “mal-blocker” activity has its “back” button disabled, such that the user has no way to access the uninstaller activity right below it in the back stack whatsoever, and thus cannot confirm the app uninstallation operation.

In fact, the “back” button of “mal-blocker” is not only disabled, but also augmented with a new event that triggers $s_5 \rightarrow s_6$: invoking (call startActivity()) the “mal-root” activity with an intent having CLEAR_TOP flag set, which results in killing of the uninstaller and Settings activities in the task.

Preventing Uninstallation from adb: An advanced user may resort to Android Debug Bridge (adb), a client-server program used to connect Android devices from a computer, and uninstall the malware from adb. However, in order to use adb, the user needs to first enable USB debugging in the Settings. The malware can block it in the Settings using similar technique and prevent the use of adb, as long as the USB debugging is not enabled before the attack (which is the case for most normal users).

5.5.2.2 Ransomware

Ransomware blackmails people for money in exchange of their data, and it has recently hit Android in a large scale [70]. The attackers may use UI hijacking to implement ransomware.

The malicious background service mentioned above takes the following two responsibilities and is difficult to be completely stopped. (1) Assure the malicious root activity (“mal-root”) is alive: it re-creates a new root activity once the activity is found to be destroyed; and (2) monitor the foreground activity: if the target activity shows up, it immediately starts “mal-blocker” to block user access to the
target activity, as we have seen in $s_4 \rightarrow s_5$. To prevent itself from being killed, the service registers itself in the system alarm service, who fires a pending intent in every given fixed time interval, re-launching the service if it is found to be killed.

By this mean, the ransomware is able to restrict user access to any target apps on user device at attacker’s will, and can potentially render the Android device completely useless.

5.5.3 Breaching UI Confidentiality

The attack method in Section 5.5.2 can also be deployed to devise a new spyware, namely “TaskSpy” capable of monitoring the activities within any tasks in the newest Android 5.0.x systems (API 21), without requiring any permissions.

In Android, the system regards the owner of the root activity in a back stack to be the owner of the corresponding task. Android 5.0 allows an app to get the information of the caller app’s own tasks (including the activities in the tasks) without requiring any permission. It means that, if a spyware can “own” the tasks of all the apps it intends to spy on, it is able to get the information of these tasks that in fact contain the victim apps’ activities. Task hijacking is especially useful to “TaskSpy” in this case. In other words, “TaskSpy” can use the HST presented in Section 5.5.2 to “own” the tasks of any victim apps and thus stealthily spy on their activities without using any permission. Chen et. al. have achieved the same goal in their work [53] by monitoring and interpreting the shared VM information via public side channels. Compared with their attack, task hijacking can do this in a more direct and reliable way on Android 5.0.x.

5.6 Defense Discussion

Given the popular use of the security-sensative task control features, simply disabling these features would greatly hurt app functions and user experience. Mitigating the task hijacking threats become a trade-off between app security and multitasking features.
5.6.1 Detection in Application Review

Existing app vetting processes such as Bouncer [106] may conduct a inspection over the “sensitive” task control knobs, a light-weight defense strategy without significantly affecting existing multitasking features.

However, specifying a guideline balancing the security/feature trade-off is non-trivial. For example, a tentative guideline could be: taskAffinity attribute should be specified in a strict format, e.g., with app package name followed by developer-defined affinity name (now task affinity can be any string); and the task affinity should not contain any other app’s package name, except that the two apps are from the same developer. This effectively eliminates a big portion of hijacking state transitions where a malicious activity specifies the victim app as its preferred affinity. However, this rule also restricts useful features and contradicts with an important principle of Android multitasking design - give an activity the freedom to live in its preferred task even though they are from different apps. This contradiction cannot be solved by app review alone in this case. We need system support together with app review to achieve a good balance of security/feature trade-off.

Moreover, detecting problematic events can be sometimes difficult for the app review. For instance, one could confine the behaviors in onBackPressed(), preventing it from generating potential hijacking transition event. However, discovering all possible program behaviors using static analysis is an undecidable problem. A skillful attacker can replace class methods (onBackPressed() method in Activity class) with another method by changing Dalvik internals using native code during runtime, and static analysis does not know this by simply looking at the original onBackPressed() method. Dynamic analysis is of little help as well since this behavior can be triggered only after passing the app review.

As a result, completely mitigating task hijacking risks and without affecting existing features in app review remains challenging.

5.6.2 Secure Task Management

An alternative approach involves security enhancement to the task management mechanism of Android system.

A more secure task management could introduce additional security guides or
logic, which draws developers’ awareness of the security risk and limits the attacker surface. Take the above task affinity for example, an additional boolean attribute can be introduced for each app to decide if it allows the activities from other apps to have the same affinity as the app. If the boolean is “false” (also by default), the system would not unconditionally relocate the “alien” activities to the app’s task or vice versa, even though the “alien” activities declare to have the same task affinity as the app. Likewise, a finer-grained boolean attribute can be further employed for allowParentReparenting attribute - determining if to allow “alien” activities to be re-parented to the app’s task (even though defining the same task affinity is permitted). For other “security-sensative” features, we suggest first consider the same approach. Considering the serious security hazards that can be prevented, it is well worth of making such changes. At the very least, enhanced security scheme like this has to be applied to assure the security of the most privileged system apps.

Completely defeating task hijacking is not easy. As we have discussed in the last section, it is difficult to identify the exact behavior of pressing “back” in an activity during app review phase. For these popular and security-sensitive features, more powerful runtime monitoring mechanism is required to fully mitigate task hijacking threats.

In summary, we advocate future support for security guidance and/or mechanism, which can protect Android apps from task hijacking threats and bring along a both secure and feature-rich multitasking environment for Android users and developers.

5.7 Related Work

GUI security : GUI security has been extensively studied in traditional desktop and browser environments [48,49], e.g., UI spoofing [50], clickjacking [51,52], etc. Android, on the other hand, is unique in the design of its GUI sub-systems. It has been shown that the GUI confidentiality in Android can be breached by stealthily taking screenshots due to adb flaws [56], via embedded malicious UIs [57,58], or through side channels, e.g. shared-memory side channel [53] or reading device sensors information [54,55]. In contrast to existing work, this paper focuses on the fundamental design flaws of the task management mechanism (supported by the AMS), the control center that organizes and manages all existing UI components in the Android system,
Android Vulnerability: The security threats in the inter-component communication (ICC) has been widely studied [79–83]. Moreover, there has been considerable prior work on emerging Android vulnerabilities and their mitigation measures in many aspects [87–89, 92–94, 96–98]. However, the critical Android multitasking mechanism and the feature provider, the AMS, have been largely neglected before. This paper fills in this gap by systematically studying the Android multitasking and the security implications of this design.

Android Malware: Many prior efforts focus on large-scale detection of malicious or high-risk Android apps [26], e.g., fingerprinting or heuristic-based methods [28–30], malware classification based on machine learning techniques [33,34], and in-depth data flow analysis for app behaviors [34–37]. The attack surface discovered in this paper can be easily employed by attackers to create a wide spectrum of new malwares, as discussed in Section 5.5. We report our threat assessment based on over 6 million market apps and provide defense suggestions in order to prevent the outburst of task hijacking threats in advance.
Chapter 6  
Systematic Protection of Android GUI System

6.1 Introduction

Android’s graphic user interface (GUI) has greatly promoted user experience and gained massive popularity to the Android system. Despite the merits, Android has been recently plagued by increasing and various GUI attacks.

Android GUI attack refers to any harmful behaviors that alter the contents on mobile display in an attempt to either maliciously deceive the user into taking undesirable actions (a.k.a. GUI confusion attack), or forcefully push unwanted content to the user. For example, Chen et al. [53] shows it is possible to accurately infer sensitive user activities (e.g. logging into bank account) via side channels, and trick user input secrets (e.g. bank login information) by immediately launch a phishing GUI to the display once a target event is detected. Previous research shows that apps with certain permission (e.g. SYSTEM_ALERT_WINDOW ) can launch similar phishing or tapjacking attacks [65,67,68] without user awareness. Notably, more dreadful attacks can be conducted even without any permission. In [69], by manipulating activity browsing history in a task, an attacker can perform task hijacking to launch a broad range of stealthy attacks, including GUI confusion attack, denial-of-service, and user activity monitoring attacks. The attacker, even without acquiring any permission, can now target any apps installed on the infected device, including the privileged system apps and UIs.

To be more concerning, the GUI attacks is growing at an alarming rate and
have posed serious real-world threat. For instance, one malware family use GUI confusion as an attack vector to steal credit card information [7]. An emerging threat, ransomware, which locks device screen and data until a ransom money is paid, has made their way to the Android platform [7,115], and affected more than 900 thousand Android devices within two years [7]. Ad-ware, which repeatedly present unwanted (and sometimes 'unclosable') advertisements pop-up window to the user, is not only annoying for normal use, but also make the user prone to new malware infection [72,73]. Given the threats, we have a pressing need for a defense mechanism to mitigate the GUI attacks.

**Challenges:** A recent defense solution has been proposed by Bianchi et al. [65], which involves a two-layer defense: an static analysis vetting process in app markets, and a runtime defense on device. The static analysis searches for uses of GUI-related platform APIs and permissions, and flag those with suspicious behaviors as potentially malicious. A fundamental challenge is that it is difficult for code analysis to rule out the apps who use the same set of suspicious APIs/permissions for legitimate purpose. For instance, a legitimate screen locker app could behave similarly to a ransomware. As they both use the same set of GUI features, it is finally up to user’s decision on if a GUI change is desired or not, base on the context at runtime (e.g. whether a screen locker or a ransomware lock screen). The on-device solution of [65] takes this approach, By putting a reliable identity indicator in the navigation bar, which is supported by HTTPS EV certification, an app can constantly inform the user of its identity as long as the navigation bar is shown, and hence preventing user from providing sensitive information to the wrong entity in a GUI confusion attack. However, the passive defense solely relies on the correct judgment of savvy users and thus requires users’ continuous attention to the indicator on the navigation bar. This not only significantly affects user experience, but also largely undermines its effectiveness, i.e. reportedly only 76% detection rate at best in a user study. Moreover, this defense strategy is only helpful in a GUI confusion attack, and cannot deal with other types of GUI attacks, e.g. the emerging denial-of-service attack. In addition to these challenges, the requirements of modifying both existing apps (implementing EV certification) and the system make it impractical to be adopted by a significant portion of apps in any Android markets.

**WindowGuard:** It is exactly these challenges that this paper seeks to address by
proposing a new solution: GUI-Guard. At the heart of the problem, although the Android security model renders different apps sandboxed and isolated from one another, they share the same mobile display, a few inches screen that usually only shows one app at a time. A GUI attack takes place only when a GUI under attacker’s control (may be visible and/or capable of receiving user input) can interrupts the normal user session, and takes over part or all of the display, regardless of whether the user realizes it or not. Given the limited size of the display, and the fact that there exists a plethora of APIs that can affect the contents on screen, it is much easier for an attacker to take control of the display, compared to its counterpart machines in desktop environments.

To better understand the problem in mobile platform, we take systems approach and take the first step to investigate the security implications behind Android GUI system. We find that the problem is largely due to the lack of access control of the GUIs to be displayed, and the neglection of suspicious system state that clearly indicate an ongoing GUI attacks. For example, although Android provides “official” ways to inform user of interesting events, e.g. notifications, an app still suddenly overlays a full screen window on top of others even though the user is clearly in the middle of using other apps. Therefore, it is the system’s responsibility to scrutinize the state of the GUI system, ensure its integrity based on a GUI integrity model, and enforce access control once an integrity violation has occurred.

We propose a new security model called Android window integrity, which defines the principles of how the system should be kept valid from one state to the next and specifies the capabilities of various principles in the system. The model is designed based on Android app model and the norm of app navigation from user’s perspective. This largely preserve the original user experience and the enforcement on security violators becomes natural. We implement the our security model, and the prototype, WindowGuard, can comprehensively protect the GUI system and is able to defeat all known GUI attacks, a much broader range of attacks than previous work. When WindowGuard is deployed on user device, the user is not bothered at all until a suspicious event is detected. WindowGuard takes a light-weight response by briefing the user about the event and ask for the final decision from user, who is inherently more capable of making the best decision for him/herself based on the context. More importantly, this design makes WindowGuard more practical and suitable for the diverse needs of app developers and users in the Android ecosystem.
WindowGuard is implemented as a Xposed module, making it realistic to be quickly deployed by a large number of user devices and provide immediate protection. In summary, we make the following contributions:

- **New understanding of the Android GUI security.** To the best of our knowledge, we are the first to systematically overhaul the security implication of Android GUI system. Composed of two key system services, activity manager service and window manager service, it is the single most important and sophisticated subsystem in Android. This new understanding can further inspire follow-up research on mobile GUI system security.

- **Novel GUI security model in mobile environment.** We define a novel security model - Android window integrity - for the GUI subsystem in a unique mobile environment like Android. By holding a set of security criteria under this model, the system could comprehensively and automatically protect itself against a wide spectrum of GUI attacks, ranging from user deception attacks, denial of service attack to undesirable pop-up window. More importantly, the new security model also considerably raise the bar for future attacks. Now new attacks can be put under the test of our defense before they make real threats.

- **Implementation and evaluation.** We develop WindowGuard based on our proposed model. Compared to existing solutions, our solution is effective maximally preserves original GUI features and user experience. Our evaluation demonstrates its efficacy in effectiveness, usability and performance.

### 6.2 Android GUI System

As the most sophisticated sub-system in Android, the proper function of GUI system requires a close collaboration of various system services and components. Among them, the most important ones are Activity Manager Service (AMS) and Window Manager Service (WMS). In this section, we introduce how the Android GUI system works and the security risks that come with such a strategy.
6.2.1 Activity and Window

**Activity:** Activity is a type of app component that provides one or more windows to the user. Activity and window are closely related: each activity must have a window instance, which contains the GUI contents to be shown on screen. The GUI contents are specified by the app developer, and if not explicitly specified, a default one will be populated to the activity’s window. In Android, activity is designed to be the building block of window/GUI browsing experience in the application model. For example, a user may open a system setting app having the first activity that shows the main menu, the second one to manage installed apps, and the third one to uninstall a particular app.

**Window:** Conceptually, window is a visual area on screen that shows the GUI of the program it belongs to. More specifically, in an activity (the client side), a window instance is a container that holds a view hierarchy, i.e. hierarchical GUI contents to be displayed on screen. On the other hand, in window manager service, window refers to WindowState instance. Each WindowState instance represents a window instance from an app or the system, and it contains all the meta data about the window. From user’s perspective, there are usually multiple windows on display, composed by the system based on each window’s size, location, transparency, and z-order etc. Android defines three classes of windows: (1) App window: the top-level windows that are associated with activities (as discussed); (2) Sub-window: the windows that are attached to other windows, e.g. a dialog window; (3) System window: special types of windows used by system for specific purposes, e.g. status bar, navigation bar, and key guard etc. Note that system window does not necessarily belong to the system UI. For example, method input or wall paper windows can be from third-party packages. These windows are registered in the system, which later launches them in appropriate occasions.

6.2.2 GUI Architecture Overview

Figure 6.1 shows the architecture of Android GUI system and also illustrates how an app’s activity manages to display its window on screen. The process involves the following steps from a high level: (1) AMS launches a new activity (added to the activity history called tasks). (2) The new activity then creates a window for itself during initialization and registers the window to the WMS. To ease the
management of windows, WMS adds the new window to a window stack, which contains all available windows and their meta data in the system. (3) Requested by the app, WMS then asks the SurfaceFlinger to create a drawing surface, i.e. a buffer of graphical data, for the window. The drawing surface, also known as layer by SurfaceFlinger (its primary component is called BufferQueue), is in turn shared with the app by passing a handle back to it. Meanwhile, WMS also passes the window parameters over to SurfaceFlinger (e.g. window size, location, transparency and z-order etc.), such that the latter can later use this information to compose the final frame. (4) Once the app receives the shared surface, its framework code draws the window’s entire view hierarchy on it and signals SurfaceFlinger when the drawing is completed. This drawing process is synchronized with SurfaceFlinger and can happen as fast as 60 frames/second. (5) SurfaceFlinger keeps multiple layers of different windows in a layer stack. When all the visible layers are ready, SurfaceFlinger composes them and displays the final graphic data to the screen, with the help from Hardware Composer (HWC).

In this architecture, although SurfaceFlinger has direct control to the display hardware, it strictly carries out the commands from its “supervisor”. The “supervisor”, i.e. AMS and WMS, not only controls the how the windows should be displayed graphically, e.g. size, location, transparency, z-order, but also decides
what windows should or should not be displayed. In other words, they are the heart of Android GUI system and the focus of this paper. We now try to understand the mechanism by looking at an example in Figure 6.2.

6.2.2.1 Activity Management

Activity management is performed by AMS, one of the most important system services in Android. Activities are started by AMS upon client requests through \textit{intents}, an abstract description of the activity to be started. As shown in the left part of Figure 6.2, every app activity has a corresponding \textit{ActivityRecord} instance in AMS. For convenience, we refer to an ActivityRecord instance in AMS simply as activity for the rest of this paper.

AMS organizes all activities in \textit{tasks} [111]. Each task includes a stack of activities, namely \textit{back stack}. Activities in a back stack are ordered by the time that they are visited, such that the user can go back to the most recent activity. There is only one activity running in the system at a time, namely \textit{focused activity}. The task that contains the focused activity is \textit{focused task}; all other tasks are in the background. Figure 6.2 (in AMS) shows multiple tasks: two for app A and B respectively, and another task for the launcher. When the user clicks an app icon in the launcher, the app’s existing task comes to the foreground. If no task exists for the app, a new task is created and the main activity of the app becomes the root activity in the task. When another activity is later started, it is by default pushed on top of the task who launches it. For instance, activity A2 is started by A1 and is thus put on top of task A. Activity A1 is stopped but remains in the task, whereas A2 gets the focus and is shown on display. When the user later presses the back button on navigation bar, the top activity is popped from the task and destroyed. The previous activity A1 on stack is then resumed and become the focused activity again.

Since activity is the building block in Android’s app model, AMS essentially controls the GUI user experience in Android, e.g. driving the GUI work flow in an app, dividing multiple work flows in separate task, and keeping GUI browsing history for user re-visits later in time.
### 6.2.2.2 Window Management

As briefly introduced in Section 6.2.2, WMS is responsible for a variety of tasks. One of the most important one is to managed all windows in the system, update their parameters and pass them to SurfaceFlinger when needed. Specifically, each window is represented as a `WindowState` instance in WMS. For convenience, we refer to a `WindowState` instance simply as a window for the rest of this paper.

All windows in a display (usually only one for mobile device) are kept in one stack, namely window stack. Window parameters (including z-order and visibility) are tuned by WMS based on windows’ runtime status. Once changes happen to any window or the window stack, e.g. window re-sizing, activity launch, etc., WMS would walk through the window stack, calculate the new parameters for all windows, and pass the new information to SurfaceFlinger, which would compose all updated layers and finally reflect the changes on screen.

As shown in Figure 6.2, window stack ranks windows by a numeric value called z-order. A window with higher z-order resides higher in the stack, which means that the window will appear on top of others if window overlap occurs. A window’s z-order is determined in two steps. First, a reference z-order value is statically assigned according to window types. For example, app windows have lower z-order values while system windows (such as status and navigation bar) have higher values and hence are shown on top of screen. The final z-order is further calculated based on a variety of factors, such as the window functionality, order of creation, etc. For example, the z-order of an input method window is always set to be a bit higher than the app window (B1 window) that requests the input keyboard, but no higher than the window that is originally above the app window.
A window can either be visible or hidden, and there are usually multiple visible windows simultaneously. Figure 6.2 exhibits the visible windows in WMS and their window areas on screen. Here, visibility does not refer to being visible or not by naked eye. Instead, visibility is a state in each window. If “visible”, it indicates that the window is ready for display. Nevertheless, whether the window will finally be shown on screen depends on the location, transparency and z-order of other “visible” windows. That is, if the window area is completely overlayed by other windows higher in the window stack, the window is in fact hidden “behind” the foreground windows even if the state is “visible”. For example, although the visibility of launcher’s wallpaper window is always set to be “visible”, it is completely overlayed by other visible windows (because it resides at the bottom of the window stack) and is in fact invisible by naked eye. This is intentionally designed by the wallpaper service to fill in any void screen areas (in case there are any) with the wallpaper background. On the other hand, if visibility is set to “hidden”, a window will not be shown at all, regardless of where it resides in the window stack (e.g. the Keyguard window).

Having a window stack is convenient for window management, but it is even better when WMS can quickly and correctly identify the owner of each window. That is where WindowToken and AppWindowToken come in. As its name implies, a WindowToken is a type of binder token (will be discussed in Section 6.2.3) that is used to uniquely identify a group of related windows in the system. AppWindowToken is similar to WindowToken, except that all windows in the group are all associated with the same activity. In other words, an AppWindowToken is a representation of its corresponding activity instance in AMS. Given these two types of tokens, all windows are classified into two categories:

1. Activity window: a window that is associated with a particular activity. An activity may have more than one windows (e.g. top level app windows, or sub-windows that are attached to app windows), and the windows are grouped into a list under the corresponding AppWindowToken, which uniquely identify the activity in AMS.

2. Free window: a window that are not associated with any activity. A free window is either a system window or a sub-window attached to the system window.
Figure 6.2 shows all available WindowTokens, and AppWindowTokens including their one-to-one mappings to ActivityRecord instances in AMS, and activity instances in apps. Notably, two activity windows are visible and both belong to token A2: one sub window in the foreground (a progress dialog “A2 Sub”), and another app window ('A2 Win') beneath it. They are emplaced above all other activity windows by WMS when activity A2 becomes the focused activity.

An activity window is always started by an activity. On the other hand, although free windows (mostly system windows) are often used by the system, a normal app can also directly launch them by making an `addView()` API call to WMS. Take the example in Figure 6.2, app A is free to start a toast window on top of others from any of its components, i.e. activity, service, or broadcast receiver.

### 6.2.3 Security Mechanisms

There are three security mechanisms that play critical roles to ensure GUI security in Android: app sandboxing, binder token, and permissions.

In Android, every app is given a unique Linux UID and runs in a separate process by default, which effectively isolates one app from the others. From GUI’s perspective, sandboxing guarantees the security of the graphic information in each app window, e.g., preventing an app from modifying the drawing surface in the address space of another app, given that the system itself is not compromised.

In reality, an app has to cross process boundaries and contact with system services to enable its proper functionality. Android provide an IPC mechanism called **binder**. In this client-service communication, it is crucial for the system services to securely identify who send the request, in order to prevent security problems such as spoofing the system to remove other apps’ windows. It turns out that a binder object has a unique property: supported by the **binder driver**, each binder object maintains a globally unique identity across all processes. It is exactly this property that makes a binder object ideal for this security use: a security token shared between client and service. Specifically, AMS and WMS create app/window tokens (e.g. AppWindowToken and WindowToken) and shared them with a client app. When an app (or its enclosed activity) asks for service, the system services require the client to identify itself by including the given token. This authentication scheme is a cornerstone of Android security architecture. By confirming the client’s
identity, system services can prevent being spoofed by attacker, e.g., to mistakenly modify the window states of another app, or to deliver user inputs tapped on a particular window to the wrong principle.

Moreover, Android provides finer-grained security enforcement based on permission mechanism. Apps must explicitly declare the permission they need for additional capacities. For example, an app can launch certain system windows only if it has obtained the `SYSTEM_ALERT_WINDOW` permission. Permission can also be declared to protect activity. During activity launch, AMS enforces access control by consulting Package Manager Service, which checks the permission requirement of the callee activity (if there is any) and returns the check result to AMS.

### 6.2.4 Security Risks

Although the current security model characterized by app sandboxing works especially well in many aspects, e.g. protecting an app’s window graphics from being accessed and tampered by attackers, it performs poorly in protecting the GUI system against existing attacks.

The fundamental problem is that, in the GUI system, it is also the user session - a list of GUIs that an user has visited in a time series when doing a particular job - that requires security guarantees and protection. The security of app sandboxing is only enforced on process/app boundaries, leaving the user sessions vulnerable to being arbitrarily interrupted or manipulated by attacker, e.g. popping up a phishing window, or redirecting the user to a different task during app navigation.

Unfortunately, this problem is further exaggerated by the uniqueness of GUI in mobile environment. First, unlike the desktop machines, the screen of a mobile device is relatively small and usually only show one app at a time. To save space, there is generally no app indicator on screen, e.g. a task bar or a window title bar like in desktop OSes. Because of this design, it is cumbersome for a user to constantly verify the real identity of the current GUI. Although a user can resort to the recent button for the name of the current task, the displayed information is untrustworthy as the recent task list is subject to manipulation by attacker [69]. Hence, reliably identifying the current GUI for a normal user becomes infeasible. Secondly, the design of GUI and the established user habit in the mobile environment makes attacks more likely to be successful. For example, because the user gets used
to the app navigation user experience, one might be easily spoofed if the default app navigation behavior is maliciously altered. Due to a lack of user control to the screen, it is troublesome for the user to escape a lock screen (e.g. a system window) in a denial-of-service attack. Thirdly, Android provides apps with great flexibility to control the window states and the user session without being strictly disciplined. For example, a normal app can freely launch new activity, add high z-order system windows on screen, and change the window parameters from any of its app components at any time. Moreover, it can also change its task’s back stacks and app navigation behaviors at will, and can affect those of other apps if it wants to without requiring any permission. Originally intending to promote the platform features for app developers, these features however inadvertently enrich the GUI attack vectors and encourage apps’ mal-behavior. In fact, Google has long realized the problem - the security issues of the over-flexible GUI features - and has taken steps to remedy the problems in newer Android releases, e.g. adding security attributes to GUI components, requiring explicit user consent, deploying runtime permission, etc. Nevertheless, it soon becomes a dilemma: many security features are barely used by unwitting developers (even Google apps themselves), and even they are fully deployed, the ad-hoc protection cannot systematically mitigates all attacks. On the other hand, removing or modifying the longstanding GUI features will break a bulk of existing apps.

As a result, the Android GUI system becomes particularly vulnerable to a variety of serious attacks that could be easily launched and lead to severe consequences. This is proven by both the prior research findings and the rapidly growing real-world threats. Table 6.1 shows a list of these known GUI attacks. Depending on attack vectors, all attacks are classified into two categories: window overlay and task hijacking. Window overlay is the act of rendering a window on top of screen, partially or completely overlaying other windows. Task hijacking refers to a set of malicious behaviors that trick the system to modify the app navigation behavior and the tasks (which are originally designed to capture user sessions) in the system. Both categories of attacks can cause serious consequences as shown in the last column of Table 6.1, seriously affecting the integrity, availability and confidentiality of Android GUI. In summary, we consider a threat model as follows:

**Threat Model:** We consider a harmful app has been installed on the user’s Android device. Like most real-world malware, the harmful app does not has
Table 6.1. Existing known GUI attack vectors in prior work. The consequences are: A - sensitive data stolen; B - user input eavesdropping; C - user spoofing; D - loss of availability; E - malware infection; F - user privacy infringement

system privilege, e.g. running in system UID. Instead, it may seem harmless, requiring the minimal set of permission needed for the malicious purpose. We assume that the system itself is un-compromised and trustworthy. We also assume that one window involves one principal. In the cases when a window is composed of elements from different principles, e.g. the app itself and the embedded ad library, we consider the principle to be the one who owns the window. In order to achieve its malicious purpose, the attacker’s goal is clear: take over part of or all the display using window(s) under attacker’s control.

6.3 Android Window Integrity

As demonstrated by the prior research, the existing security model is not designed to cope with GUI attacks. The fundamental limitation is that, GUI attacks target the sessions of windows, which is beyond the scope of app sandboxing protection. Motivated by the challenges and serious threats, we take systematic approach and propose a new security model - Android Window Integrity (AWI) - to fill in this important gap. AWI model redefines a session as an activity chain, describes the principles of how the sessions and windows in the system should be kept valid from one state of the system to the next, and specifies the capabilities of various
principals in the system.

### 6.3.1 Display Owner and Activity Session

The key principle of AWI is that, no application, by default, has permission to perform any operations that would adversely affect the user session of other apps or the system UI. At the center of the model are display owner and activity session, which are the basic entities to be protected in AWI, like a process in the app sandboxing.

As discussed in Section 6.2.4, mobile display is a unique time-sharing resource that is shared by different apps at different times. We introduce display owner, a principle who “owns” and has control of the mobile display. There is only one display owner at a time. In AWI, we specify the display owner to be the app of the currently focused activity. It means that the app of the focused activity is more privileged than others in terms of GUI-related operations and its windows are prioritized to be shown on display (although the display owner is still disciplined by the existing security confinement).

An activity session is a sequence of activities that lead the user to the currently focused activity. It stems from the use pattern based on the design of Android app model. That is, activity is the building block of Android app and constitutes the basic GUI unit. A user starts a job by opening an app from the launcher activity, the primary navigation “hub”. The state of previous activities in a job is saved and can be resumed by clicking back button. Once the user finishes the current job, one can go back to the launcher activity by pressing the home button, or view a recent task list by pressing the recent button. From there, one can start or switch to another job.

Activity session preserves the norm of app navigation from user’s perspective. An activity session always starts from the launcher activity, which is the first focused activity being brought up on screen after system boots up. When a new activity is launched and obtains the focus, e.g., when the user clicks an app icon in launcher to display the app’s main activity window, the new activity is chained to the end of the activity session. There can be multiple activity sessions at the same time, i.e. the user may go back to the launcher and start another app; and the state of the original activity session is retained. Although an activity session
looks similar to a task in AMS, the two have major differences as we will see in Section 6.3.3. Figure 6.3 shows a diagram that depicts an example of multiple activity sessions in the system. As shown in the figure, each node indicates an activity. Activities are chained by the order that their corresponding windows are created and displayed on screen. All the activity sessions forms a tree: the launcher activity is the root and the only joining points of all activity session branches. The focused activity (display owner) is always at the tail of a session, and that session is called focused activity session (e.g. launcher $\rightarrow$ B1 $\rightarrow$ B2).

An activity session transition occurs when the currently focused activity transits to an existing activity in another activity session. For instance, as illustrated in the diagram of Figure 6.3, focused activity B2 may transit to A2 in three different routes (the dashed line). In 1 and 2, the user can either go to the launcher or the recent task activity, and then routes to A2. In particular, the recent task activity belongs to system UI and is only started upon user pressing the recent button. Since its role is to be a “hub” that facilities the transition of different tasks in the system, we regard the recent task activity itself as a separate activity session that does not overlap with any other sessions. In this sense, the transition from B2 to the recent task activity itself is a activity session transition. In route 3, a focused activity transition can also occur directly from B2 to A2. This could happen in many circumstances, e.g. launching an existing activity with special intent flag, using MoveTaskToFront API to switch to another task, etc.
Figure 6.4. An activity session transition example.

Note that although the activities in an activity session are viewed by the user in a timely order, they may form multiple tasks/back stacks in AMS. Figure 6.4(a) is such an example, depicting both the activity session and corresponding system state. A2 was launched by A1 but was put into a new task in AMS (e.g. by specifying `singleTask` activity attribute). Since A1 and A2 are displayed in a continued sequence, they are still regarded to be in the same activity session. In this case, if the focused activity transits from B2 to A1 because of any of the above reasons, the original activity session in fact breaks into two sessions, as shown in Figure 6.4(b).

Given display owner and activity session, we introduce the three aspects of integrity that AWI attempts to achieve for the display owner and foreground activity session in Section 6.3.3. To better understand these principles, we first simplify the complex GUI system and formalize it in a state transition model.

### 6.3.2 System State Transition

The state transition of the GUI system is described by \((S, \Lambda, \rightarrow)\), where \(S = \{S_{AMS}, WS\}\). \(S\) denotes the set of system states, in which \(S_{AMS}\) indicates the set of system states in AMS and \(WS\) stands for the set of window stack states in WMS. \(\Lambda\) indicates a set of events and conditions; and \(\rightarrow\) denotes a set of viable
transactions under certain events or conditions. As discussed in Section 6.2, window state transition occurs frequently in response to a variety of events and conditions, such as launching an activity, removing a window, and change of window parameter, etc.

Specifically, given a system state $s_A \in S_{AMS}$ and a window stack state $ws \in WS$, they have the following definition:

- $s_A = \{a_{focus}, BS\}$, where $BS = \{bs_1, bs_2, ..., bs_n\}$, and $bs = (a_1, a_2, ..., a_n)$. $a_{focus}$ denotes the current focused activity. $BS$ is a set of all $n$ available back stacks in the system. $bs$ indicates a back stack, each of which includes an ordered list of activities, $(a_1, a_2, ..., a_n)$.

- $ws = (w_1, w_2, ..., w_m)$. $ws$ represents the window stack containing a total of $m$ windows. $w_1$ and $w_n$ represent the bottom and the top windows in the stack respectively. Each window $w_i$ ($0 < i < m$), includes a handful of parameters such as visibility, size, transparency, etc.

6.3.3 System State Legitimacy

Android window integrity is composed of three legitimacy: the legitimacy of past activity session, the current visible windows, and the future windows to be displayed. Instead of being a rigid security model, AWI can dynamically adapt to the diversity of user needs and app behaviors with legitimate intention in the Android ecosystem. Once a integrity violation is detected, it is up to the user to make the final decision. The user choices are incorporated in the model. This makes AWI both useful and practical in reality.

**Legitimacy of activity session.** A activity session looks similarly to a task’s back stack at the first glance, e.g. both keep a record of previous activities. However, activity session is not a simple duplication of task. A task is a container that keeps existing activities instances. Although the order of activities in a back stack is typically reserved and follows the order of activity launch most of the time, it is not always true. In fact, Ren et. al. [69] demonstrates that the back stacks can be manipulated outrageously, e.g., the activity order can be changed; an activity could be relocated to other task; a full back stack of activities can be created without user’s awareness. These flexibility however contradicts with user’s common sense,
e.g. the back stack saves a series of activity windows one has previously seen; clicking an app icon from the launcher would start the app window, (which may not be the case) and so on. By taking advantage of these pitfalls, the most dreadful task hijacking attacks could be launched, as listed in Table 6.1.

Activity session is designed to simulate the user’s visual observation by capturing the sequence of activity windows that the user actually sees when doing a task. Every time the focused activity changes, the foreground activity session is used to safeguard the integrity of back stacks in the system by comparing the difference between itself and back stacks. Any disparity between the two may indicate a suspicious system state. More specifically, the model considers the following statement as a proper system state:

$$\exists \{bs_1^*, bs_2^*, ..., bs_n^*\} \subseteq BS : s_{fg} = (bs_1^* \parallel bs_2^* \parallel ..., \parallel bs_n^*)$$

where $BS$ indicates the set of all back stacks in the system; $bs_i^*$ is one of the back stacks; $s_{fg}$ denotes the foreground activity session, composed of an ordered list of activities. In other words, it checks if there exists a subset of back stacks, such that the concatenation of their ordered activity list is the same as the ordered activities in the foreground activity session, we considers it to be a valid system state.

Here, the reason to consider a concatenation of multiple back stacks is that, activities may be launched in new tasks (e.g. using NEW_TASK intent flag) while the user remains to be in the same activity session.

**Legitimacy of current visible windows** After an activity gets focus and become the display owner, other than its own activity window(s), there are usually other visible windows at the present. To assure that no other windows are disturbing the appearance of the focused activity window(s), an overhaul of the window stack ($ws$) is necessary. Specifically, the model ensures that no other visible windows, except the windows belonging to the display owner app and a set of white-listed windows, should overlay on top of the focused activity window in the window stack. To put it formally, the model have the following guarantees:

$$\neg \exists w_i \in ws : w_i.visible = true, i > k,$$
where $w_i$ is the $i^{th}$ window from the bottom of window stack; $k$ represents the index of the top focused activity window in the window stack; and $L$ denotes a white list of windows. The white list typically includes system windows and other windows explicitly specified by the user, as we will discuss in Section 6.4.

**Legitimacy of future windows.** There is a plethora of ways to start a variety of windows, but all in two categories: free window and activity window. The principle is, given the current display owner app A, and a window W to be started or resumed, the entity that initiates the launch of W must be from app A’s process(es). This principle holds for both free window and activity window, with a few exceptions. We now explain these two categories of windows respectively.

Although third-party app or package can launch free windows, such as toast window (without any permission), or other system windows (requires `SYSTEM_ALERT_WINDOW` permission), many free windows are typically launched by the system or system apps. The model considers a white list of UIDs of the system processes (e.g. system server) and system packages (e.g. system UI) trustworthy, and allow their windows being displayed freely. Moreover, another exceptions are two special types of windows that the system treats differently: input method or dialog window (input window), and wallpaper window. An input window is registered in Input Manager Service, and is placed on top of the target client window that requires a input method (e.g. a keyboard) on window stack. Wallpaper window is similar, except that it is started by a wallpaper service and is placed under the client window on window stack. The target client window to which the input or wallpaper windows provide service is specified by WMS. To prevent blocking an useful input or wallpaper window, the model checks whether their client window belongs to the display owner app.

An activity window is launched in two scenarios: (1) the focused activity remains the same, such as launching a sub-window, or another app window associated with the focused activity; (2) the focused activity changes, and the window is the first app window associated with the new focused activity. The model does not worry about the first case, as the display owner does not change and the activity window must be started by the display owner (thanks to app window tokens; it is impossible to spoof the system in creating an activity window for another activity). In the second scenario, the change of focused activity implies a possible change of display
owner. This may happen during a activity session transition. For example, in Figure 6.3, the focus can be taken over by activity B2 from C2 in another activity session. Change of display owner may also occur within the same activity session. For instance, activity C2 may launch a new activity belonging to app A, say activity A2. Activity A2 becomes the new focused activity and is chained to the same activity session. No matter what the case is, the principle retains, the model requires the change of focused app to be initiated by an entity that belongs to the original display owner. In the above example, the model allows activity C to voluntarily yield the display ownership to app A, giving up its privilege assured by the Window Integrity model. By this means, the model assures that the display ownership cannot be arbitrarily transferred to other apps without being suggested by the original display owner.

6.4 WindowGuard

We implement the AWI as a module for the Xposed framework, a popular code-injection framework for Android. The implementation, namely WindowGuard, is tested on Google Nexus 5 phone and can be used in Android 4.4, Android 5.x and Android 6.0 with minor changes. An Xposed module can be installed like an app. The module can hook arbitrary functions of the system or apps at runtime and change their behaviors without modifying the system or apps themselves. WindowGuard can be used on all Android device brands that the Xposed framework supports. These features makes WindowGuard practical to be distributed to a large number of Android devices and provide immediate protection.

6.4.1 AWI Model Implementation

WindowGuard implements AWI by hooking 21 functions of AMS, WMS, Package Manager Service (WMS), and system UI in a total of 2300 lines of code.

Activity session integrity. As previously depicted in Figure 6.3, activity sessions are implemented as a tree data structure maintained in the system server process (a privileged process hosting all system services implemented in Java). Each node represents an activity, which links to its predecessor and successor activities. Activity sessions share the same root, the launcher activity. An activity node is
only added to the current foreground activity session only if a new activity obtains the focus (its windows become visible), and it perishes when the corresponding activity is destroyed.

The activity session integrity check occurs when the focused activity changes. The focused activity session is then compared with the back stacks as described in formula xxx. If they are same, the integrity check passes. If not, the mismatch may due to several reasons. It may be caused by relocation of the focused activity to another existing back stack by the system, as a result of the use of various activity attributes [114], e.g., `allowTaskReparenting`, `taskAffinity`, etc. It is also possible that the back stack change is inherently inconsistent with user observation during certain operations. For example, an activity may launch a stack of multiple activities by calling `startActivities` or using the `TaskStackBuilder` class. In this case, the whole activity stack is pushed on top of the current back stack, yet only the top activity gets the focus and added in the activity session, and the user is unaware of the other activities at all.

The failure of integrity check in this case may indicate an possible attack, but it is no surprising if it comes from legitimate operations. The distinction lies in an important premise of task hijacking: a task hijacking attack happens only if the malicious and legitimate activities co-exist in the same task. WindowGuard iterates the activities in the back stacks that are involved in the focused activity session. If they are from the same app, then WindowGuard regards it as legitimate operation. Otherwise, a notification is created to alert the user about the event and possible security hazard.

**Access control of free windows** Given a display owner, access control is enforced on the free windows that are about to be displayed or resumed based on the discussion in the legitimacy of future windows. For those existing visible windows that violate the window legitimacy principle, free windows are made invisible, and activity windows (if there is any) are reordered under the focused activity windows in the window stack. To achieve this, WindowGuard hooks a handful of functions in WMS with the functionality such as adding windows to the window stack, window stack reordering behaviors, and window visibility control. Access control of free window helps prevent window overlay attacks such as user spoofing or a ransomware denial-of-service attack.
Safeguarding focused activity transition

Focused activity transition happens frequently during the use of device in practice. It happens either during activity launch or resumption and may result in a new display owner. In principle, in either cases, the focused activity transition must be initiated by the display owner app, as discussed in Section 6.3.3. The complex app navigation features and logic requires comprehensive monitoring and protection of focused activity transitions, which should be carefully implemented and designed.

A new activity get focus when it is launched by one of the `startActivity` function calls from an app. The origin of caller app is examined by the hooking function in AMS. If the caller UID is not the current display owner or one of the white-listed principles, An alert notification with detailed information is prompted to the user for confirmation to proceed. This effectively prevents attacker from overlaying a malicious activity window on top of a victim app. On the other hand, activity resumption can happen either implicitly or explicitly. It happens implicitly when AMS resumes the next activity on the back stack, while the original focused activity is destroyed either voluntarily by itself (e.g. activity quits) or by the system (e.g. pressing the back button). Since the activity destroy is led by the display owner or the system, it is perfectly fine in this case to let the display owner yield its ownership. Activity resumption can also be requested explicitly by invoking a set of APIs such as `startActivity`, `moveTaskToFront/Back`, or `moveTaskForward/Backward` APIs. `startActivity` may also resume an existing activity under certain conditions, e.g., the activity launch mode is set to `singleInstance`. The latter two APIs can arbitrarily move a task to the foreground or background, which can modify the display owner at will. Likewise, WindowGuard hooks the internal functions for each of these app APIs to check the caller origin. If the caller app is not the display owner, a confirmation request is prompt to the user.

One can navigate backward to the previous screens one has worked with by pressing the back button, making the back button the most popular user control on the Android platform. By default, pressing the back button ends the focused activity and resume the previous activity on back stack. Meanwhile, back button behavior can also be overridden by the `onBackPressed` function of the focused activity to implement customized back behavior, e.g. back navigation for activity fragments or webviews. However, the back behavior customization feature can be abused to launch a phishing attack, e.g., redirecting the user to attacker’s activity.
and spoof the users in believing one has already went back to the previous screen. WindowGuard keeps a close eye on the system state after a back button pressing. For example, if the new focused activity is neither the next activity on back stack nor the original activity, a message is then sent to notify the user.

6.4.2 Security of App Navigation “Hubs”

The launcher and recent task list play a critical role and act as “hubs” in app navigation. The proper function of these critical components have to be assured by WindowGuard.

Launcher: Launcher is the first app activity to be started. Other than the default Android launcher, third-party launchers can also be installed and used. The first goal is to securely start the launcher activity of user’s choice. After system boot-up, AMS queries PMS about the information of packages that can serve as the launcher. If multiple launcher activities are returned, a dialog is prompted for user to make a choice. At this stage, WindowGuard is able to prevent an attacker from affecting user’s choice by manipulating windows on screen, as the display owner is system UI (the owner of the focused dialog activity). WindowGuard trusts the user’s choice and regards the chosen launcher activity as the only root of all activity sessions in this system launch.

The second goal is to assure that an app is reliably started when the user clicks its icon in the launcher. We are the first to discover a security issue in the app launch process, and it affects all Android and launcher versions. When an app icon is clicked, an intent with SINGLE_TASK and ACTIVITY_BROUGHT_TO_FRONT is sent to start the corresponding app. The combined use of these intent flags creates a new task to host the app’s new activity. If the app’s task already exists, the task will simply be brought to the foreground. However, in this operation, AMS considers the task owner to be the package name described in the taskAffinity attribute of the root activity (the bottom activity in the back stack), instead of the app of the root activity itself. Although the two are by default the same, the taskAffinity attribute can be configured arbitrarily to some other app’s package names without restriction. Therefore, a malicious activity can specify a victim app’s package name as its taskAffinity and start the activity in a new task. The task is believed by the system to belong to the victim app, but in fact, is controlled by the attacker. The
problem occurs when the user clicks on the victim app, yet the malware task is started instead of the victim app itself. To remedy this problem, WindowGuard monitors the requests of activity launch in AMS. If it comes from the launcher, WindowGuard marks the app that the launcher intends to start, and later verifies that either the new focused activity or the root activity of the foreground task belongs to the app.

**Recent Task List:** The recent tasks screen contains a list of all recently accessed tasks, showing the task owner’s name/icon and the task’s last screenshot. The user can browse through the list and choose a task to resume. However, recent task activity suffers the same problem as in app launch process, i.e. it regards the owner of a task to be the app described in the `taskAffinity` attribute of the task’s root activity. As a result, the user could be easily spoofed by a malicious task which camouflages as the victim app in the recent task list. To impede such attack, WindowGuard hooks the system UI and shows the name/icon of the app of the root activity of a task, instead of what is described in `taskAffinity` attribute. By this means, it faithfully reflects the real identities of tasks in the system.

### 6.4.3 Preserving User Experience

WindowGuard implements AWI model, which is designed to adapt to the Android use and navigation pattern, such that the user experience is not affected at all in normal use until a security violation is detected. The security violation may indicate an potential GUI attack, or the legitimate use of GUI features that do not strictly follow the norm of Android app model. WindowGuard takes a light-weight response by briefing the user and asking for the user’s final decision. WindowGuard has all the window handing privileges. The alert messages, depending on the emergency and severity of attacks, are delivered via a confirmation dialog, a system notification or a toast message right after the violation occurs. WindowGuard maintains a handful of white lists; one for each security feature. Those on the a particular white list are not confined by the corresponding security feature. WindowGuard always respects the user’s decision and the diversity of user needs; the whitelists are constantly updated based on user input. Moreover, all GUI security protection features can be lifted and re-enforced in a central control panel owned by WindowGuard, which is always under protection regardless of the white
lists.

6.5 Evaluation

We now proceed to the empirical evaluation of the efficacy of Droidmarking in the following facets: effectiveness, usability and performance impact.

6.5.1 Effectiveness

To evaluate the effectiveness of our solution, we install the WindowGuard prototype on a Google Nexus 5 phone and experiment with 15 attack samples from all 12 attack vectors listed in Table 6.1. The attack samples either come from real-world malware/adware, or are proof-of-concept apps we developed based on previous research [53,65–69]. The evaluation shows that WindowGuard is able to effectively detect all attacks. We now show a few case studies to demonstrate how the attack behavior violates AWI and how WindowGuard delivers the potential attack alert to the user.

Back button hijacking: Back button hijacking [69] is one type of task hijacking...
attack. The attack misleads the user to a phishing activity after the user clicks the back button, instead of the original activity the user just visited. Figure 6.5 shows the task states in AMS. In Figure 6.5(a), victim activity A intends to start an legitimate utility activity U to serve the user’s request (e.g. playing a video). However, when activity U is created, it is tricked to be emplaced on top of a phishing malware’s task, whose enclosing activities M1 and M2 are camouflaged as the appearance of A1 and A2. This happens due to use of \texttt{NEW\_TASK} intent flag when starting activity U and the abuse of \texttt{taskAffinity} by the malware. When the user later clicks the back button, the phishing activity M2 is resumed by the system, while the user still believes he/she has went back to the previous activity A2. User’s sensitive information (such as bank login information) is then stolen by the malware.

WindowGuard can readily detect the task hijacking behavior when activity U is launched on the malware task. As shown in Figure 6.5(b), the activity session keeps track of user observation to activity windows from the launcher activity all the way to activity U. When activity U is started, the legitimacy of back stacks is checked by comparing related back stacks (i.e. task A and M) against the focused activity session. The disparity of the comparison is obvious due to the relocation of activity U. In this case, a notification is created in the status bar to notify the user of the possible security threat. Likewise, other task hijacking attacks violate the legitimacy of back stack and can be defeated by WindowGuard. On the other hand, this kind of task manipulation behavior can also used for legitimate purpose. WindowGuard provides detailed information about the security risk to the user, and it is up to the user to make the final judgment based on the context. For example, an useful task management app may frequently use various task-related features provided by the platform in its normal operation and inadvertently trigger the alarm. In this case, the user can easily cease the surveillance of this particular app in the WindowGuard control panel. In some other context, the security alert is particularly alarming for any user when the user is making a money transfer in a banking app.

\textbf{Tapjacking:} Clickjacking attack is well known in web security. The similar attack idea is borrowed to the Android environment, called tapjacking. Tapjacking overlays windows on top of screen and spoofs the user to perform undesirable operations. Bankbot [116] is a family of banking malware discovered in South
Korea Android market in October 2014. It is designed to steal authentication information from the clients of various financial institutions. To avoid itself from being uninstalled, Bankbot disguises itself as Google play store and attempts to acquire admin privileges of the device. Granting admin privilege requires user confirmation, as shown in Figure 6.6(a). Bankbot overlays a free window (system alert window) on top of the admin privilege confirmation activity, claiming that the software needs update to the latest version. Although the free window is opaque, it is intentionally configured to not receive user tap input, such that button tapping on the free window is in turn received by the active window underneath it, i.e. the system confirmation window. As a result, the user naturally agrees to the software update without realizing that he/she has granted the admin privilege to the malware. WindowGuard immediately detects the attempt of free window launch and pauses it before asking for the user’s decision from a security alert dialog, as shown in Figure 6.6(c). The attack is detected because the current display owner is the system settings, who owns the focused confirmation activity. Any window operations (including free and activity window) that affect the focused activity window are reported to the user and ask for user permission to proceed.
Ransomware: Screen-lock ransomware blackmails victim users by locking the screen for money in exchange for the accessibility to the system again. Ransomware has migrated to Android and has been growing at an alarming rate in the past few years. A ransomware usually renders a high-z-order free window to overlay the full screen and hence blocks all user inputs to the system, leaving the system effectively “locked up”. The ransomware can even use a combined GUI attack vectors, e.g. getting admin privilege via tapjacking, to become more powerful and hard to remove, like a recent ransomware called Lockerpin [115]. In addition, [69] demonstrates that a ransomware can also launch activity windows to prohibit user access to targeted victim apps, e.g. an anti-virus app. In either cases, WindowGuard can block the lock screen window as long as window’s initiator is not the current display owner, regardless of whether it is a free window or an activity. Even if the user is spoofed and accidentally get trapped by a lock screen, the user can always escape by clicking the home or recent button, which starts the launcher or system UI activities. Changing the display owner to launcher or system UI make the upfront malicious lock screen invalid and the lock screen is immediately removed. A notification message is then created to inform the user about the security enforcement just occurred.
Recent years have witnessed the rapid growth of mobile devices. On the other hand, the increasing popularity has drewed attention from malware writers, and the number of harmful apps is increasing at an alarming rate. Mobile devices also share many of the vulnerabilities of the PCs world. There are a handful of serious system vulnerabilities disclosed every year, and the path of these vulnerabilities usually takes time and painful. To this end, we propose two directions for future research:

### 7.1 Security Overhaul of Android System Services.

So far, we have studies the fundamental design flaws of the Android multitasking mechanism by systematically analyzing the "snapshot" of task states during the tasks dynamic. We also researched the mechanism of how Android GUI system works in an attempt to devise a defense solution. To thoroughly mitigate the security vulnerabilities in these system services, we would like to further understand the behind feature provider, Activity Manager Service and Window Manager Service and completely disclose the potential security vulnerabilities in its implementation. We plan to achieve this by doing model checking on the Android source code. Given AMS source code, we tend to exhaustively and automatically check whether the implementation of AMS meets given specification. Typically, the specification contains security requirements such as the hijacked task states that can put the system into task hijacking risk. Model checking is used here to automatically verify the correctness properties of AMS implementation.
7.2 Dynamic Analysis of Harmful Applications

Dynamic analysis has been used to test Android apps in an emulated environment where the app runs on provided input. This allow an malware analyzer to monitor the app’s behavior to detect potentially harmful behaviors that may not be apparent in static analysis.

One of the challenges of dynamic analysis is that malicious apps could attempt to detect the simulated environment and hence evade detection. One possible solution is to use a large number of different devices in different situations and monitor the response of the apps to these differences. If they respond very differently, it may indicate the app is trying to evade detection.

Another challenge in dynamic analysis is the scope of functionality that dynamic analysis can reach. For example, if an application starts by asking for an account login, then the automated system that lacks account credentials may not be able to run very much of the application functionality. One possible solution is to use static analysis to identify events and conditions that can provide for increased code coverage in the dynamic analysis. For example, static analysis may be useful to identify the entry points of user input, network activities or data update behaviors.

Sometimes, automated systems are not sufficient to conduct a complete review by themselves. Human intelligence has the unique advantage of understanding the logic of app functions and can be used to provide data directly to the automated system. Letting humans interact with the app increases coverage for the dynamic analysis and provide another opportunity for the system to detect potentially harmful behaviors.

On the other hand, a malware can delay its attack behavior until interesting event happens, in an attempt to avoid being detected by a analysis system. To solve this problem, a honeypot is created to provide a set of data that appears to be legitimate to the app. In fact, the fake data is isolated and closely monitored. For example, fake account data can be created to feed in app under analysis, the behavior of the app is then monitored in a long term.
7.3 Android System Vulnerability Patching

Due to the well-known fragmentation problem of the Android ecosystem, deployment of any OS-level patch or security solution is often complicated and painful: whenever Google comes up with a patch, individual device manufacturers need to customize it for all their devices before passing its variations to the carrier, who will ultimately decide whether to release them and what to do that. Even in the case that the manufacturers are willing to build the protection into their new products, given the slow pace with which Android devices are upgraded, it is almost certain that the new protection will take a long time before it can reach any significant portion of the one billion Android devices worldwide.

Instead of being confined by the slow pace of system update from the device manufacturers, an alternative approach is feasible, hot patching. Hot patching is patching process that does not require re-flashing the device ROM or even shutting down the system. Similar to an ordinary patch, it alleviates bug and vulnerabilities of the system or important system apps, but without going through the cumbersome system update process as described above, and hence provide immediate protection from being exploited via the n-day vulnerabilities. However, there are several challenges: (1) Although it is relatively easy to patch the system services implemented in Java, hot patching the native system code, including libraries, kernel and device drivers is non-trivial. (2) It is critical important to securely conduct the patching, e.g. authenticate the origin of patches, diminishing the vulnerability of the patching process itself. (3) Since different devices and Android versions are heavily customized, hot patching may incur compatibility issues with the original system. In addition, hot patching requires root privilege, which is only applicable to rooted phones.
Bibliography


[40] Peter Hornyack and S. Han and J. Jung (2011) “These Arenâ€™t the Droids Youâ€™re Looking For Retrofitting Android to Protect Data from Imperious Applications,” in Proceedings of ACM Conference on Computer and Communications Security (CCS).


Attacks,” in *Proceedings of International Conference on Trust and Trustworthy Computing (TRUST).*


Vita
Chuangang Ren

EDUCATION

Ph.D. Candidate in Computer Science and Engineering, The Pennsylvania State University
Advisor: Dr. Peng Liu and Dr. Sencun Zhu
Jan 2011 - Now

M.S. in Electrical and Computer Engineering, Boston University, Boston, MA
Concentration: Signal and image processing
May 2009

B.S. in Electrical Engineering, Dalian University of Technology, Dalian, China
Jul 2007

PROFESSIONAL EXPERIENCE

Graduate Research Assistant, The Pennsylvania State University
Jan 2011 – Now
- Android system security, application plagiarism prevention/detection, malware analysis
- Resource, power management and performance evaluation of data centers

Research Intern, Samsung Research America. Mountain View, CA
May 2015 – Aug 2015
- Work in Samsung Knox team, focusing on Android window sub-system security

Android Security Analyst Intern, Fireeye, Inc. Milpitas, CA
May 2014 – Aug 2014
Mentor: Dr. Tao Wei, Dr. Dawn Song
- Android system vulnerability research and analysis
- Developed proof-of-concept exploits, reverse-engineered "towelroot" root exploit

Software Engineer, Lab of Computational Cell Biology, Harvard Medical School, Boston
Jan 10 – Dec 2010
- Independent software development and algorithm co-design of computer vision technologies on cell biology research (using C and Matlab)
- Released software packages including Biosensor, plusTipTracker and u-Track, with hundreds of downloads by cell biology research community (http://lccb.hms.harvard.edu/software.html)

Software Engineer Intern, Novus Laurus LLC, Boston
Oct 2009 – Dec 2009
- Developed JAVA/J2EE-based web applications (using JAVA)

Graduate Research Assistant, Boston University, Boston
Jan 2008 – May 2009

HONORS AND AWARDS

- Best Paper Award (2/21 accepted papers), IEEE IISWC, September 2013
- Best Paper Award (1/48 accepted papers), IEEE MASCOTS, August 2012
- Nomination for Best Paper Award (4/36 accepted papers), ACM SIGMETRICS/Performance, June 2012