TOWARDS ENHANCING ANDROID APPLICATION AND SYSTEM SECURITY

A Dissertation in
Information Sciences and Technology
by
Lannan Luo

© 2017 Lannan Luo

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2017
The dissertation of Lannan Luo was reviewed and approved* by the following:

Peng Liu
Professor of Information Sciences and Thechnology
Dissertation Advisor
Chair of Committee

Anna Squicciarini
Associate Professor of Information Sciences and Thechnology

Xingyu Xing
Assistant Professor of Information Sciences and Thechnology

Minghui Zhu
Assistant Professor of Electrical Engineering

Andrea Tapia
Associate Professor of Information Sciences and Technology
Director of Graduate Programs

*Signatures are on file in the Graduate School.
Abstract

The market for smartphones has been booming in the past few years. Android dominates the market with 87.5% of the global smartphone shipments and over 65 billion Android apps downloaded. With the unprecedented popularity of Android devices, however, a dramatically increasing number of security threats are posing to Android applications and systems. Tremendous efforts from both industry and academia have been made to mitigate the threats to secure Android. This dissertation mainly focuses on two severe threats: Android App repackaging and Android Framework vulnerabilities, and propose techniques and approaches to address them and, hence, enhance Android application and system security.

First, we propose a decentralized Android app repackaging detection scheme to impede the prevalent app repackaging problem in the market. Existing countermeasures mostly detect repackaging based on app similarity measurement and rely on a central party to perform detection. However, the centralized scheme is unscalable and the detection techniques tend to be imprecise when handling obfuscated apps, resulting in many repackaged apps escaping detection and being widely distributed in the market. To solve the problem, we propose a decentralized repackaging detection scheme, which adds the repackaging detection capability into
the code of an app, such that repackaging detection becomes an inherent part of
the app when it is released.

Second, we design and build the first system that enables symbolic execution of
Android Framework to automatically discover vulnerabilities and generate exploits.
Android Framework is an integral and foundational part of the Android system.
Each of the 1.4 billion Android devices relies on the system services of Android
Framework to manage applications and system resources. Given its critical role, a
vulnerability in the framework can be exploited to launch large-scale cyber attacks
and cause severe harms to user security and privacy. Recently, many vulnerabilities
in Android Framework were exposed, showing that it is vulnerable and exploitable.
However, most existing research has been limited to analyzing Android applications,
while there are very few techniques and tools developed for analyzing Android
Framework. To fill the gap, we develop the first system that analyzes the framework
through symbolic execution, and demonstrate how the system can be applied to
discovering new vulnerability instances and generating PoC exploits.

Given that symbolic execution has proven to be a very useful technique, we
plan to apply the system to other purposes in future work, such as automatic API
specification generation, fine-grained malware analysis, and testing.
# Table of Contents

List of Figures ix  
List of Tables x  
Acknowledgments xi  

Chapter 1  
Introduction 1  
1.1 Motivations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1  
1.2 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2  
1.3 Organization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5  

Chapter 2  
Related Work 6  
2.1 Repackaging Detection . . . . . . . . . . . . . . . . . . . . . . . 6  
2.2 Tamper-proofing Techniques . . . . . . . . . . . . . . . . . . . 7  
2.2.1 Code Encryption and Decryption Based Approaches . . . . 7  
2.2.2 Self-checksumming Based Approaches . . . . . . . . . . . . 8  
2.2.3 Oblivious Hashing Based Approaches . . . . . . . . . . . . 8  
2.3 Symbolic Execution . . . . . . . . . . . . . . . . . . . . . . . . . 9  
2.3.1 Mixing Concrete/Symbolic Execution . . . . . . . . . . . . 9  
2.3.2 Switching Concrete Execution to Symbolic Execution . . . 10  
2.3.3 Bug Finding . . . . . . . . . . . . . . . . . . . . . . . . . . . 10  
2.3.4 Exploit Generation . . . . . . . . . . . . . . . . . . . . . . . 11  
2.3.5 Symbolic Execution of Android Apps . . . . . . . . . . . . 11  

Chapter 3  
Repackaging-proofing Android Apps 12
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>Problem Statement and Threat Model</td>
<td>16</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Problem Statement</td>
<td>16</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Threat Model</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>Overview</td>
<td>18</td>
</tr>
<tr>
<td>3.4</td>
<td>Repackaging Detection</td>
<td>19</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Detection Node Construction</td>
<td>19</td>
</tr>
<tr>
<td>3.4.1.1</td>
<td>Construction Process</td>
<td>19</td>
</tr>
<tr>
<td>3.4.1.2</td>
<td>An Example</td>
<td>20</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Detection Node Injection</td>
<td>21</td>
</tr>
<tr>
<td>3.4.2.1</td>
<td>Candidate Methods</td>
<td>21</td>
</tr>
<tr>
<td>3.4.2.2</td>
<td>Injection Algorithm</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>Repackaging Response</td>
<td>24</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Response Node Construction</td>
<td>24</td>
</tr>
<tr>
<td>3.5.1.1</td>
<td>Stealthy-modification Mechanism</td>
<td>24</td>
</tr>
<tr>
<td>3.5.1.2</td>
<td>Construction Process</td>
<td>26</td>
</tr>
<tr>
<td>3.5.1.3</td>
<td>An Example</td>
<td>26</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Response Node Injection</td>
<td>27</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Stochastic Responses</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Communication Channel</td>
<td>28</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Stealth Communication Channel</td>
<td>28</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Communication Medium</td>
<td>30</td>
</tr>
<tr>
<td>3.7</td>
<td>Security Analysis</td>
<td>31</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Static Analysis</td>
<td>31</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Dynamic Analysis</td>
<td>31</td>
</tr>
<tr>
<td>3.7.3</td>
<td>Replay Attacks</td>
<td>32</td>
</tr>
<tr>
<td>3.8</td>
<td>Evaluation</td>
<td>34</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Experimental Settings</td>
<td>34</td>
</tr>
<tr>
<td>3.8.2</td>
<td>Feasibility</td>
<td>34</td>
</tr>
<tr>
<td>3.8.3</td>
<td>Effectiveness</td>
<td>36</td>
</tr>
<tr>
<td>3.8.4</td>
<td>Resiliency</td>
<td>37</td>
</tr>
<tr>
<td>3.8.4.1</td>
<td>Resiliency to Static Analysis</td>
<td>37</td>
</tr>
<tr>
<td>3.8.4.2</td>
<td>Resiliency to Dynamic Analysis</td>
<td>37</td>
</tr>
<tr>
<td>3.8.4.3</td>
<td>Resiliency to Taint Analysis</td>
<td>42</td>
</tr>
<tr>
<td>3.8.5</td>
<td>Side Effects</td>
<td>44</td>
</tr>
<tr>
<td>3.8.5.1</td>
<td>Impact on the Functionality</td>
<td>44</td>
</tr>
<tr>
<td>3.8.5.2</td>
<td>Impact on the Runtime Overhead</td>
<td>44</td>
</tr>
<tr>
<td>3.9</td>
<td>Discussion</td>
<td>46</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Impacts</td>
<td>46</td>
</tr>
<tr>
<td>3.9.2</td>
<td>Limitations</td>
<td>47</td>
</tr>
</tbody>
</table>
Chapter 4

System Service Call-oriented Symbolic Execution of Android Framework with Applications to Vulnerability Discovery and Exploit Generation

4.1 Introduction .......................................................... 49
4.2 Background .......................................................... 55
  4.2.1 Android Framework ............................................. 55
  4.2.2 Symbolic Execution .............................................. 56
  4.2.3 Java PathFinder ................................................ 57
  4.2.4 Symbolic PathFinder ............................................ 58
4.3 System Overview .................................................... 58
4.4 Identifying Symbolic Inputs ......................................... 60
  4.4.1 App-specific Variables ........................................ 60
  4.4.2 Access Patterns ................................................ 61
  4.4.3 Slim Tainting .................................................. 63
4.5 Execution Context Migration ........................................ 65
  4.5.1 Snapshot Parsing and Information Query .................... 65
  4.5.2 Migration Algorithm ............................................ 66
    4.5.2.1 Migrating Objects ....................................... 67
    4.5.2.2 Migrating Classes ....................................... 70
  4.5.3 Bootstrapping ................................................ 70
  4.5.4 Migration Tree ................................................ 71
4.6 Implementation Details ............................................ 74
  4.6.1 Configuration ................................................ 74
  4.6.2 Classpath ................................................... 74
  4.6.3 Slim Tainting ................................................ 75
  4.6.4 Capturing and Parsing Snapshots ............................ 75
  4.6.5 Handling Special calls ....................................... 76
  4.6.6 Dealing with Handler and State Machine Calls ............ 77
  4.6.7 Handling Calls to Native Code ............................... 78
4.7 Evaluation .......................................................... 78
  4.7.1 Comparison with Under-constrained Symbolic Execution (UCSE) ............................................. 80
  4.7.2 Investigating Inconsistent Security Policy Enforcement (ISPE) ............................................. 81
  4.7.3 Investigating Task Hijacking Attacks ....................... 84
  4.7.4 Consistency of Exploits Generated with Different Snapshots ............................................. 87
4.8 Summary .......................................................... 88
List of Figures

3.1 The SSN architecture and deployment scheme. ........................ 18
3.2 A code snippet of a detection node. ................................. 20
3.3 An example of detection node injection. ............................... 23
3.4 A code snippet of a response node. ................................. 26
3.5 Examples of OpenSudoku when it works normally. .................... 38
3.6 Examples of OpenSudoku when malfunctions occur. .................... 39
4.1 The getProviders service interface method. ........................... 52
4.2 Architecture of CENTAUR. ............................................ 59
4.3 Example of retrieving information from a hash-table-based variable mPackages. ........................................... 62
4.4 Example of a test driver. .............................................. 70
4.5 An example of migrating the heap. Grey rectangles and white ones denote classes and objects, respectively. For each class and object, <conRef, symRef> denotes the mapping between the reference value in the concrete execution world and that in the symbolic world. The migration of a class also triggers the migration of all its super classes, which are not shown for simplicity. ................................. 72
4.6 Part of a migration tree with some classes omitted. Different arrows are used to denote different instructions that trigger the migration. Rectangles with diagonal stripes denote objects that are identified as symbolic inputs. ........................................... 73
4.7 A service call in Android Framework. .................................. 76
4.8 startActivityUncheckedLocked().................................... 85
4.9 An example set of concrete input values. ............................... 86
4.10 Exploit example. .................................................... 86
List of Tables

3.1 Feasibility analysis results ........................................... 35
3.2 Surviving protection nodes of each app. ........................... 40
3.3 Runtime overhead. ................................................... 44

4.1 Taint Propagation Logic. Register variables are referenced by \( v_x \).
\( \tau(y) \leftarrow \tau(x) \) means setting the taint tag of \( y \) to the taint tag of \( x \). 63
4.2 Bytecode instructions (and function) used for migrating heap information. .............................. 67
4.3 List of vulnerabilities and analysis statistics. (\textit{LMS, TSI, PIM, WMS, AMS, WSI, NS}, and \textit{ASS} represent LocationManagerService, TelecomServiceImpl, PhoneInterfaceManager, WindowManagerService, ActivityManagerService, WifiServiceImpl, NsdService, and ActivityStackSupervisor, respectively.) 83
4.4 Effectiveness of the generated exploits. ........................... 87
Acknowledgments

I would like to express my sincere appreciation and gratitude to my adviser Dr. Peng Liu, who is a passionate researcher and a role model. Throughout my years as a Ph.D. student in the College of Information Sciences and Technology at The Pennsylvania State University, Dr. Liu has generously given me priceless advice on both research as well as on my career, and mentored me to become an independent researcher. Without his encouragement and support, I would not have been able to achieve my goal.

I also want to thank my committee members, Dr. Anna Squicciarini, Dr. Xinyu Xing, and Dr. Minghui Zhu, for their insightful comments and thoughtful discussion of my research, which helped me to broaden my research from various perspectives. I would like to sincerely thank my labmates, colleagues, and friends, who has accompanied me on this memorable journey. I also would like to give my special thanks to my best friend, Lingzi Xiaoli, for bringing fun to my life in State Collage.

Finally, a special thanks to my family. Words cannot express how grateful I am to my mother Xiujun Luo, father Houhua Liu, and brother Yu Liu, for their love, support, and encourage, and for all of the sacrifices that they have made on my behalf. I also offer my deepest gratitude to my husband Qiang Zeng. Thanks for supporting me for everything. Your abiding love and support have seen me through difficult times.
Chapter 1

Introduction

1.1 Motivations

The global mobile market is booming, amounting to more than $3.1 trillion of economic value [1]. Capturing 87.5% of the global smartphone shipments, Android dominates the market with 1.4 billion active Android devices and over 65 billion Android apps downloaded [2].

Along with the booming of Android markets, app repackaging poses a severe threat to the Android ecosystem. Dishonest developers repackage apps of others under their names to profit from selling repackaged apps or displaying ads. Moreover, attackers insert malicious code into repackaged apps, which steal user information, send premium SMS text messages stealthily, or purchase apps without victim users’ awareness [3, 4, 5, 6]. Previous research showed that 86% of 1260 malware samples were repackaged from legitimate apps [7]. A repackaged mobile browser app, for example, which obtained millions of downloads, sent text messages to premium numbers without user consent [7]. As another example, the malicious adware family, Kemoge, which infected victims from more than 20 countries, disguised itself as popular apps via repackaging [8].

Android apps are not independent; they rely on the Android Application
Framework (Android Framework, for short) to make them useful. For example, all the User Interfaces designs and multi-tasking features would not work without the WMS (Window Manager Service) system service provided by Android Framework; as another example, apps cannot obtain the GPS location without the LMS (Location Manager Service) system service of the framework. Therefore, Android Framework is an integral and foundational part of the Android system; it runs on each Android device for managing applications and providing a generic abstraction for hardware access [9]. Recently, many vulnerabilities in Android Framework have been identified [10, 11, 12, 13]. Vulnerabilities in Android Framework can cause severe security consequences. For example, malicious apps can exploit them to steal user passwords, take pictures in the background, launch UI spoofing attacks, and tamper with user data [14, 15, 16]. On the other hand, due to Android fragmentation among the 1.4 billion active devices [2], tens of millions of Android devices are left unpatched, “turning devices into a toxic hellstew of vulnerabilities” [17]. Given the severe security consequences and the large number of vulnerable Android devices, attackers are certainly motivated to exploit Android Framework vulnerabilities. If no actions are taken, many new vulnerabilities of Android Framework will be found and exploited by attackers, putting enormous users at risk.

To this end, there has been considerable number of efforts to mitigating the following two severe threats: Android App repackaging and Android Framework vulnerabilities, to secure Android and enhance Android application and system security.

### 1.2 Contributions

This dissertation mainly focuses on two severe threats: Android App repackaging and Android Framework vulnerabilities, and propose techniques and approaches to address them and, hence, enhance Android application and system security.
First, in order to effectively detect and prevent app repackaging in the Android markets, we propose a novel repackage-proofing technique, called Stochastic Stealthy Network (SSN), which builds the capability of repackage-proofing into the code of an app, such that repackaging detection becomes an inherent part of the app when it is released. Repackage-proofing can be classified as a type of tamper-proofing (specialized in tackling app repackaging).

Given an app, SSN inserts a large number of detection nodes into the original code without incurring a high overhead. Each detection node is obfuscated and then woven into the surrounding code, such that the inserted detection code and original code blur together without identifiable boundaries. Upon detection of repackaging, rather than triggering a failure instantly, the detection node stealthily transmits the detection result to a response node via a stealthy communication channel, and the latter delays the failure for a random period of time. As a result, the failure point is apart from the response node, which makes it difficult for attackers to trace back to the response node. In addition, many detection nodes and response nodes form a large network with high-degree connections. Each time repackaging is detected, a response node is picked stochastically to take actions. There are a variety of response nodes, which means that, given the same input, two consecutive executions of a repackaged app may end up with different failures injected by different response nodes, resulting in a more difficult debugging scenario for attackers. Through these strategies, our technique constructs a stochastic and stealthy network of repackage-proofing that is resilient to evasion attacks.

We have implemented a prototype of SSN, which can be applied by legitimate developers during compile time to build repackage-proofing into their apps. We evaluated SSN on 600 Android apps. The evaluation results show that the protection provided by SSN is effective to defeat repackaging, resilient to various evasion attacks, and incurs a very small overhead. We summarize the contributions in this work below:
• To the best of our knowledge, SSN is the first work reported in the open literature which builds repackaging detection code into protected apps and does not reply on authorities to perform repackaging detection.

• We identify the main challenges in designing a repackaging-proofing technique, and propose SSN that overcomes those challenges. Unlike conventional tamper-proofing techniques, SSN leverages unique characteristics of Android apps to construct effective and stealthy protection.

• We have implemented a prototype of SSN compatible with the Android platform. The evaluation shows that SSN is effective, efficient, and resilient to many evading attacks.

Second, we design and build the first system Centaur that enables symbolic execution of Android Framework to automatically discover vulnerabilities and generate exploits. Centaur is a path exploration system that can effectively assist automatic and precise vulnerability discovery. We concretely demonstrate how Centaur can be applied to vulnerability discovery by considering several recently uncovered attacks that exploit the framework vulnerabilities. We show that the process minimizes the manual effort and guarantees zero-false positives, in contrast with recent researches on finding Android Framework vulnerabilities that rely on laborious and error-prone manual work [15, 14]. Finally, we make use of Centaur to generate PoC exploits to validate the findings. We make the following contributions.

• To our knowledge, Centaur is the first system that supports symbolic execution of Android Framework. It provides an approach to exploring paths in Android Framework automatically and precisely. The proposed techniques can potentially be applied to symbolic execution of other complex middleware.

• Unlike previous symbolic execution schemes that either start analysis from the main function or analyze a non-main function without the context information,
we employ a scheme that allows service interface methods of middleware to be analyzed separately for much improved scalability and meanwhile provides a complete execution context.

- A novel tainting analysis technique is proposed to precisely identify the framework variables derived from a given app. It is particularly suitable for vulnerability discovery as it considers all possible values under the control of a malicious app.

- An innovative architecture that builds the symbolic executor out of the Android system is proposed. A powerful algorithm that migrates the execution context information from Android to the symbolic executor is designed.

- We have implemented CENTAUR and evaluated it in terms of the effectiveness and precision in finding vulnerabilities and generating PoC exploits.

1.3 Organization

This proposal is organized as follows: we discuss the related work in Chapter 2. In Chapters 3, we present our repackage-proofing approach (SSN) for impeding Android app repackaging. In Chapter 4, we present CENTAUR, a framework for symbolic execution of Android Framework. Finally, we summarize our work and discuss future directions in Chapter 5.
Chapter 2

Related Work

Repackage-proofing can be classified as a type of tamper-proofing. We first review existing work on repackaging detection and tamper-proofing techniques in Section 2.1 and Section 2.2, respectively. CENTAUR is a framework for symbolic execution of Android Framework; we summarize the related work on symbolic execution in Section 2.3.

2.1 Repackaging Detection

The app repackaging problem has drawn efforts from both industry and academia. Different app repackaging detection techniques use different features and methods for comparing the code between a large number of apps to detect repackaging. For instance, Zhou et al. propose DroidMOSS, which uses hashing of app instruction sequence to detect repackaging [18]. Potharaju et al. uses program syntactic fingerprints to detect plagiarized applications under different levels of obfuscations [19]. Chen et al. use the program dependency graph as features to detect repackaging [20]. Crussell et al. propose AnDarwin which is a scalable approach based on clustering to detecting similar Android apps using their semantic information [21]. AppInk [22] and DroidMarking [23] inject watermarking into apps so that a trusted party with
the knowledge of app watermarkings can help detect repackaging. AnDarwin detects similar apps using the semantic information [21]. Repackaging detection techniques based on code similarity comparison can be evaded by various code obfuscations. Most of them rely on a centralized effort to detect repackaging, which may be costly and incomplete. SSN implements decentralized repackaging detection, which adds the detection capability into apps such that the detection becomes an inherent part of apps when released.

2.2 Tamper-proofing Techniques

We divide the existing tamper-proofing techniques into the following three categories: code encryption/decryption based, self-checksumming based, oblivious hashing based.

2.2.1 Code Encryption and Decryption Based Approaches

Aucsmith proposes an approach utilizing cryptographic methods to decrypt and encrypt code blocks before and after each execution round [24]. The decryption and encryption procedures are controlled by an integrity verification kernel, which communicates with other code segments to create an interlocking trust model. However, this method cannot be done in a stealthy way in bytecode and does not scale well because of the time taken by encryption and decryption. Wang et al. propose a dynamic integrity verification mechanism designed to prevent modification of software [25]. The mechanism utilizes multi-blocking encryption technique to encrypt and decrypt code at runtime, which needs no hash value comparison. Cappaert et al. also propose an approach which enciphers code at runtime, relying on other code as key information [26]; this way, any tampering will cause the code to be decrypted with a wrong key and produce incorrect code.
2.2.2 Self-checksumming Based Approaches

Many tamper-proofing techniques are based on computing checksums of code segments. Chang et al. define small pieces of code called *guards*, to compute checksums over code fragments [27]. Disabling this kind of protection requires all the guards be disabled or removed, making it non-trivial for the attacker. However, the code checking operation has to involve a call to a custom class loader, and thus can be easily found and bypassed; moreover, the guards are hard to automatically constructed and the maintenance cost is very high. Horne et al. extend this technique and utilize *testers* and *correctors* that redundantly test for changes in the executable code as it is running and report modifications [28]. The values of the correctors and watermark values are computed at installation time, resulting in a watermarked and self-checking program. Tsang et al. implement a large number of lightweight protection units, called *protectors*, installed among the program code, to protect any critical regions of a program from being modified [29]. This protection scheme supports non-deterministic execution of functions, resulting in different execution paths and nondeterministic tamper responses. Our stochastic response mechanism is inspired by it and has a similar fashion. Jakubowski et al. present software integrity checking expressions, which are program predicates, to dynamically check whether or not a program is in a valid state [30]. These predicates hold for any valid execution of the program, and fail with some probability for any invalid execution. Jakubowski et al. further propose a scheme to transform programs into tamper-tolerant versions that use self-correcting operation as a response against attacks [31]; it chops a program into blocks, which are duplicated, individualized, and rearranged.

2.2.3 Oblivious Hashing Based Approaches

Chen et al. propose oblivious hashing that implicitly computes a hash value based on the actual execution of the code to verify the runtime behaviour of the
software [32]; however, it requires pre-computation of expected hash values under all possible inputs; thus, it can only be applied to relatively simple functions that produce deterministic hash values. Chen et al. propose a tamper-proofing software technology for stack-machine based languages, such as Java, by improving oblivious hashing [33]. This approach inserts hash instructions into basic blocks at the bytecode level, to monitor the top of the stack to check whether the program running has been tampered with or not. Jacob et al. present an approach which overlaps a program’s basic blocks so that they share instruction bytes to improve the tamper-resistance [34].

One work we know of aiming to improve the stealthiness of the tamper-response mechanism is that of Tan et al. [35]. They introduce a delayed and controlled tamper response technique which makes it difficult to detect tamper responses; the delayed failures are achieved by corrupting a global pointer at well-chosen locations. Although their technique is more tamper-resistant than others that directly cause programs to fail, it still reveals information to attackers for finding tamper responses. Unlike their approach, we attempt to cause the delayed logical malfunctions, such that it is very difficult to find the failure points, as well as trace back to the protection code.

2.3 Symbolic Execution

2.3.1 Mixing Concrete/Symbolic Execution

DART is the first concolic testing tool that uses symbolic analysis in concert with concrete execution to improve coverage of random testing [36]. It runs the tested unit code on random input and symbolically gathers constraints at decision points that use input values; then, it negates one of these symbolic constraints to generate the next test case. EGT [37], EXE [38] and Klee [39] execute external code concretely by using one of the possible concrete values of the symbolic operands. S²E introduces
selective symbolic execution, which allows a program’s paths to be explored without having to model its surrounding environment [40]. These techniques usually take advantage of concrete execution to simplify complex symbolic constraints and execute external code, while CENTAUR makes use of concrete execution to set up the execution context for symbolic execution.

2.3.2 Switching Concrete Execution to Symbolic Execution

Symbolic PathFinder (SPF) begins with concrete execution and can switch to symbolic execution at any point in the program, such that the concrete execution is effectively used to set up the environment for the symbolic execution [41]. CENTAUR is similar in spirit, but decouples the architecture and allocates concrete execution and symbolic execution to two disjoint systems, so that the two systems can evolve independently. As SPF aims at generating unit test cases, it assumes a well-defined interface for specifying symbolic inputs. Instead, CENTAUR has to strategically locate symbolic inputs that scatter in memory.

2.3.3 Bug Finding

Fuzzing and symbolic execution have been applied to checking the existence of bugs. For example, Miller et al. proposed a blackbox fuzzing technique that sends unstructured random input to an application program and considers a failure to be a crash or hang [42], while SAGE is a whitebox fuzzing approach [43], which leverages the technique described in DART [36]. Their main purpose is to find inputs that reveal the existence of program bugs.
2.3.4 Exploit Generation

Brumley et al. proposed techniques for automatic patch-based exploit generation (APEG) and show that the techniques can be used to generate exploits for Microsoft programs [44]. Compared to APEG, AEG does not require access to patches. Both APEG and AEG target stand-alone native executables for exploit generation, while Centaur aims at building malicious apps that exploit vulnerabilities in Android Framework.

2.3.5 Symbolic Execution of Android Apps

There has been a lot of work that leverages symbolic execution for testing Android apps. Mirzaei et al. use symbolic execution for deriving the event parameters, but not for the sequencing of events [45]. Anand et al. proposed a system based on concolic testing for generating screen tap events to exercise Android apps, and it does not address the generation of input values [46]. Jensen et al. proposed to use concolic execution to build summaries of the individual event handlers and then generate event sequences backward, in order to find event sequences that reach a given target line in the Android app [47]. SIG-Droid combines program analysis techniques with symbolic execution to generate event sequences as well as input values [48]. All the techniques use symbolic execution to exercise application code rather than Android Framework. To our knowledge, our system is the first one that performs symbolic execution of Android Framework code.
Chapter 3

Repackaging-proofing Android Apps

3.1 Introduction

The explosive growth of the Android markets over the past few years has led to a booming app economy [23]. However, Android app piracy is rampant. Take game apps as an example, a recent report showed a 95% piracy rate for Android games [49]. Among other forms of piracy, app repackaging is especially notorious, because it does not only cause financial loss to honest developers, but also threatens the overall app ecosystem and users. Attackers may repackage an app under their own names to earn the app purchase profit or change the app’s ad library causing the ad profit to go to attackers [3, 27, 4, 50]. Furthermore, repackaging has become one of the main forms of propagating malware. Previous research showed that 86% of more than 1200 Android malware families repackaged legitimate app to include malicious payloads [7]. A popular app may be repackaged by attackers with malicious payload injected to steal user information, send premium SMS text messages stealthily, or purchase apps without the victim user’s awareness [3, 4, 51, 5, 52].

As Android app repackaging is prevalent and dangerous, tremendous efforts have been made to address the problem in recent years. Repackaging detection [3, 53, 54, 51], e.g., based on code similarity comparison, is performed by authorities
like Google Play, which, however, can be easily evaded by various obfuscations; besides, depending on the effect and timeliness of the detection, a repackaged app may have been widely distributed before it is detected and removed from the markets. Code obfuscation by legitimate authors is used to increase the difficulty of reverse engineering [55, 56, 57], so that it may take attackers more effort to inject malicious code. Watermarking can be used to prove the ownership of a repackaged app when disputes occur [23, 22]. However, none of the existing techniques thwarts threats caused by repackaged apps once they are installed on user devices. A defense that prevents repackaged apps from working on user devices, which we call repackage-proofing, is needed.

Repackage-proofing can be classified as a type of tamper-proofing (specialized in tackling app repackaging). Various tamper-proofing techniques exist. For example, Aucsmith [24] proposed a cryptography-based approach, which breaks up a binary program into individually encrypted segments. The protected program is executed by jumping to a chain of temporarily decrypted segments. The technique requires decryption and jumps; while they are possible in Android bytecode, they cannot be done in a stealthy way since such operations have to go through calls to a class loader due the code-loading mechanism in Android. Many tamper-proofing techniques are based on computing checksums of code segments [27]. They have similar issues when applied to protecting Android bytecode, since the code checking operation has to involve a call to a custom class loader, and thus can be easily found and bypassed. Instead of computing code checksums, Chen et al. [32] proposed to calculate a hash value based on the actual execution trace of the code and compare with a stored value. However, the approach requires pre-computation of expected hash values under all possible inputs; thus, it can only be applied to relatively simple functions that produce deterministic hash values. So far few existing tamper-proofing techniques are applicable to dealing with Android app repackaging. To the best of our knowledge, there is no study in the open literature
that investigates repackage-proofing—tamper-proofing that prevents repackaged apps from working on user devices.

Tamper-proofing of type-safe distribution formats such as bytecode in Android apps is more challenging than tamper-proofing native code [55]. Operations as simple as code reading, which otherwise can be blended with the original assembly code seamlessly, involve calls to a class loader in Android. Thus, how to insert tampering detection into bytecode stealthily is challenging. Furthermore, as in all tamper-proofing implementations, it requires a careful design to hide the response code that injects failures upon tamper detection, as an attacker can leverage debugging to locate the response code once a failure is noticed. While the principle of delaying the effect of an injected failure can be shared, how to achieve it in Android apps needs fresh ideas. In addition to hiding the statically inserted code and dynamic failure generation operations, how to deliver an efficient tamper-proofing implementation compatible with the current Android system is a practical consideration as well as a challenge. We identify these as the main challenges of designing an effective repackage-proofing technique resilient to evasion attacks.

We propose a repackage-proofing technique, named Stochastic Stealthy Network (SSN), that overcomes these challenges. It builds the capability of repackage-proofing into apps, such that repackaged apps cannot run successfully on user devices. Our insight is that a unique identification of the app author and an immutable value bound with the app installation is the public key contained in the certificate of an app package. So, instead of calculating code checksums or hashes of execution traces, our detection code detects repackaging by comparing the public key hard-coded and hidden in the code against the one contained in the app certificate.

Given an app, SSN inserts a large number of detection nodes into the original code without incurring a high overhead. To achieve stealthiness, each detection node is obfuscated and then woven into the surrounding code, such that the inserted
detection code and original code blur together without identifiable boundaries. Upon detection of repackaging, rather than triggering a failure instantly, the detection node stealthily transmits the detection result to a response node via a stealthy communication channel, and the latter delays the failure for a random period of time. As a result, the failure point is apart from the response node, which makes it difficult for attackers to trace back to the response node. In addition, many detection nodes and response nodes form a large network with high-degree connections. Each time repackaging is detected, a response node is picked stochastically to take actions. There are a variety of response nodes, which means that, given the same input, two consecutive executions of a repackaged app may end up with different failures injected by different response nodes, resulting in a more difficult debugging scenario for attackers. Through these strategies, our technique constructs a stochastic and stealthy network of repackage-proofing that is resilient to evasion attacks.

We have implemented a prototype of SSN, which can be applied by legitimate developers during compile time to build repackage-proofing into their apps. We evaluated SSN on 600 Android apps. The evaluation results show that the protection provided by SSN is effective to defeat repackaging, resilient to various evasion attacks, and incurs a very small overhead.

The rest of the section is organized as follows. Section 3.2 introduce the problem Statement and threat Model. Section 3.3 presents an overview of SSN. Section 3.4, Section 3.5, and Section 3.6 present three main functional parts of SSN: repackaging detection, repackaging response, and communication channel, respectively. We analyze the security issues of SSN in Section 3.7. The experiment results and the discussion are presented in Section 3.8 and Section 3.9, respectively. The conclusion follows in Section 3.10.
3.2 Problem Statement and Threat Model

3.2.1 Problem Statement

Android app developers produce apps and sell them in the form of Android application packages (APK) files. Both legitimate users and attackers can download the APK files, and we consider that attackers try to repackage apps and re-release them. An attacker may modify the code, such as the in-app billing and ads components, to earn financial profit, insert malicious payload to infringe upon user privacy, or simply repackage the app without making changes for fun and fame. Repackage-proofing, as a type of tamper-proofing, aims to detect repackaging and prevent repackaged apps from working properly on user devices.

On the Android platform, an app, no matter it is legitimate or repackaged, has to be digitally signed with a certificate before it is released. Each certificate contains a unique public and private key pair. As the private key of the certificate is held by the developer which is unknown to the public, an attacker who wants to repackage an app has to choose a new certificate to sign the repackaged app. Thus, the public key contained in a certificate, which is a unique identification of an app developer, can be leveraged to determine whether an app has been repackaged by an attacker. Specifically, our repackage-proofing technique monitors the change of the public key to detect repackaging, and prevents any repackaged app from working properly on user devices. As a consequence, few users are willing to purchase and play the repackaged app, which limits its propagation as well as harms. Other responses upon detection are possible, for example, notifying legitimate developers and authorities of the detection by emails and removing repackaged apps from markets, which is not explored in our current implementation but discussed in Section 3.9.
3.2.2 Threat Model

We assume attackers can get the APK file of an app; however, they cannot obtain the source code and the private key, which is reasonable in practice. Moreover, it is possible that end users are in collusion with attackers; for example, a user may run a custom firmware that always generates the original public key when running a repackaged app. We do not consider such collusion attacks, and aim at protecting legitimate users.

Attackers may conduct evasion attacks to bypass or nullify the repackage-proofing protection built into apps. There are at least three types of evasion attacks. First, attackers can utilize static analysis techniques, such as text searching, pattern matching, static taint analysis, to find the injected protection code and make any code transformations necessary aiming to disable/remove the protection code. Second, attackers can also perform dynamic analysis, such as dynamic monitoring, dynamic taint analysis, debugging, etc., to execute and examine the app (bytecode and native binary), line by line, to identify the injected protection code and disable/remove the code. Third, in order to assist other evasion attacks and facilitate debugging, an attacker may control the execution to provide the original public key to bypass the detection, similar to the idea of replay attacks in networks. We aim to address these evasion attacks.

Attackers may carefully infer the program semantics and re-write parts of an app to bypass specific protection. Moreover, they may be willing to sacrifice certain functionalities of the original app and re-publish it. We handle both selective re-writing and functionality-pruning attacks.

It is generally believed that a program protected by any software-based approach can eventually be cracked as long as a determined attacker is willing to spend time and effort, which is also true with our protection. However, we assume attackers are interested in repackaging an app only if it is cost-effective, for example, when the cost of repackaging is less than that of developing the app from scratch.
Figure 3.1: The SSN architecture and deployment scheme.

3.3 Overview

We propose a repackage-proofing technique called Stochastic Stealthy Network (SSN), which builds reliable and stealthy protection into apps without relying on authorities. Figure 3.1 shows its architecture and deployment scheme.

As shown in the dashed rectangle of Figure 3.1, SSN is comprised of three functional parts: Repackaging Detection, Repackaging Response, and Communication Channel. An app developer passes the app’s source code $A$ and the public key as inputs to SSN. Multiple distinct detection and response nodes are then inserted into the original code for repackaging detection and response. The communication channel is between the detection and response nodes to construct a highly connected network, so that whenever a detection node detects repackaging, one of multiple response nodes takes actions stochastically in order to confuse attackers. Finally, the resultant source code $A'$ is generated as output and packaged with other app resources, which are then signed with the original certificate to produce an APK file for publication. In the following three sections, we will present the design and implementation details of the three functional parts: repackaging detection, repackaging response and the communication channel, respectively.
3.4 Repackaging Detection

This section presents the repackaging detection part, including how to construct and inject detection nodes.

3.4.1 Detection Node Construction

3.4.1.1 Construction Process

Repackaging detection consists of multiple detection nodes; each checks the public key. Specifically, a detection node extracts the public key from an app during runtime and compares it against the original one to detect whether or not the app has been repackaged. In the following, we refer to $K_r$ as the public key extracted during runtime, and $K_o$ as the original public key. Once a detection node detects that $K_r$ is different from $K_o$, it determines that the app has been repackaged, and transmits the detection result to response nodes via a communication channel (presented in Section 3.6).

$K_o$ is provided by the legitimate developer who uses SSN. The next question is how to retrieve $K_r$ at runtime. We leverage PackageManager, which is a class that (1) installs, uninstalls, and upgrades apps, and (2) stores and retrieves various kinds of application information. If an app being installed has passed the signature verification PackageManager parses its APK file to read the application information and then stores the information in three files under the /data/system folder. One of the three files is packages.xml, which contains package names, permissions, public keys, code paths, etc., of the installed apps. We make use of PackageManager to extract $K_r$ from this file.

We construct detection nodes based on multiple predefined distinct “polymorphic” templates which implement the same detection functionality but look different. For example, reflection is used to call functions. The function names (getPublicKey and generateCertificate, etc.) are generated in different ways in
different templates: a variety of substrings are produced and concatenated to form function names, which are passed to reflection methods to invoke the corresponding functions. This way, attackers cannot identify the detection nodes by statically searching function names. Moreover, different code obfuscation techniques, such as instruction reordering, variable renaming, dummy code injection, and opaque predicate insertion, are also combined and applied to the detection nodes to make them difficult to reverse-engineer and improve their stealthiness [58, 59, 60, 61, 62].

3.4.1.2 An Example

Figure 3.2 shows an example of the code snippet in a detection node with reflection calls omitted. The detection node first obtains a CertificateFactory object with the specified type “X509” (Line 1). It then retrieves an X509Certificate object (Line 3). Next, it calls getPublicKey to get the runtime public key pubKey (Line 5). Note that pubKey is extracted from packages.xml. Then the detection node extracts a substring keySub from pubKey (Line 7), and checks whether keySub equals to the hard-coded substring of the original public key with the same index range (Line 9). Sometimes, the two substrings are applied on the same transformation function (e.g., transform each one to a hash code, etc.), and then

```java
CertificateFactory ctFty = CertificateFactory.getInstance("X509");
//Get the X509certificate
X509Certificate cert = (X509Certificate)ctFty.generateCertificate(new ByteArrayInputStream(sig));
//Get the public key
String pubKey = cert.getPublicKey().toString();
//Extract a substring from the public key
String keySub = pubKey.substring(1,4);
//Compare with substring of the original key
if (!keySub.equals("elc")) {
    //Repackaging is detected.
    SenderCommunicationChannel();
}
```
the equivalence of the two resulting values are compared. Here, we denote the substring of the original public key as $K_{o}^{sub}$. Different detection nodes compare different substrings with varied lengths of the public key. Comparing an individual character is insufficient, as different public keys may happen to have the same character at a given index. Neither do we compare the entire public key, since it is conspicuous to attackers ($K_o$ is a long string). Finally, if $keySub$ is not equal with $K_o^{sub}$, the detection node determines the app has been repackaged, and stealthily transmits the detection result via a communication channel by calling $SenderCommunicationChannel$ (Line 11). How to transmit the detection result will be presented in Section 3.6.

### 3.4.2 Detection Node Injection

#### 3.4.2.1 Candidate Methods

Once the detection nodes are constructed, we next automatically inject them into the app. To achieve it, we first need to determine the candidate methods into which the detection nodes should be injected. We require the candidate methods should not be hot methods, for example, those keep running in the background. If a detection node is injected into a hot method, it will be executed over and again, incurring a high overhead. Instead, we consider relatively cold methods which are invoked for a few times during an execution of the app. There are a variety of methods satisfying the condition, for example, those invoked at initialization, exit, or phase transitions.

To assist selection of candidate methods, we profile the apps using Monkey [63] and Traceview [64]. Monkey is a tool that can generate pseudo-random streams of keystrokes, touches, and gestures, and Traceview a profiling tool that can log the execution trace. We first use Monkey to generate the pseudo-random stream of 100,000 user events and feed these events to the app. Simultaneously, we use Traceview to monitor the app at runtime to log the execution trace. The log
Algorithm 1 Detection Node Injection

\[ D: \text{ a random subset of the constructed detection nodes} \]
\[ G: \text{ the CFG of a candidate method} \]

1: \function{DetectionNodesInjection}(D, G)
2: \textbf{while} D is not empty \textbf{do}
3: \hspace{1em} b ← SelectBlock(G) // randomly select a block from G
4: \hspace{1em} S ← FindDominators(b, G) // store dominators into S
5: \hspace{1em} Insert(d, S) // insert d into S
6: \hspace{1em} d ← deq(D)
7: \hspace{1em} n ← GetLineofCode(d) // get # of lines of code in d
8: \hspace{1em} m ← GetNumofBlocks(S) // get # of blocks in S
9: \hspace{1em} k ← RandomNum(1, min(n, m))
10: \hspace{1em} C ← SplitDetectionNode(k, d)
11: \hspace{1em} B ← SelectBlocks(k, S)
12: \hspace{1em} \textbf{for} each \( c_i \) in \( C \) and each \( b_i \) in \( B \) \textbf{do}
13: \hspace{2em} Inject \( c_i \) into \( b_i \)
14: \hspace{1em} \textbf{end for}
15: \hspace{1em} \textbf{end while}
16: \textbf{end function}

contains the information about the invocation number of each method. We consider the methods which are called more than 50,000 times as “hot”, and exclude them. The rest methods are used as candidate methods.

### 3.4.2.2 Injection Algorithm

To inject a detection node, we do not inject it as a whole; instead, we split it into several parts and inject each part separately in order to achieve better stealthiness.

Algorithm 1 shows the pseudo-code for detection node injection. Given the Control Flow Graph (CFG) of a candidate method and a randomly selected subset \( D \) of the constructed detection nodes, the function \texttt{DetectionNodesInjection} inserts each detection node in \( D \) to \( G \). The algorithm weaves code of each detection node into the candidate method, so that detection nodes are difficult to locate and identify by attackers. To inject a detection node, it first randomly selects a basic block \( b \) from \( G \) (Line 3). Then all the dominators of \( d \) are found and are stored
along with $d$ into a set $S$ (Line 4 and 5). In a CFG, a basic block $a$ dominates a basic block $b$ if every path from the entry block to $b$ must go through $a$ [65]. Next $d$ is split into $k$ parts and stored in $C$, and $k$ basic blocks are selected from $S$ and stored in $B$; $k$ is a random integer between 1 and $\min(n, m)$, where $n$ is the number of lines of code in $d$, and $m$ the number of basic blocks in $S$ (Line 7 to 11); note that $d$ is mainly a straight-line code sequence except for a few conditional basic blocks, each of which is considered as a single line. Finally, each part in $C$ is sequentially inserted into $B$ (Line 12 and 13) following the dominance order of these parts in the CFG.

The injection method guarantees that the execution order of the instructions in each detection node remains after injection. In addition, as we use different names for variables in different detection methods, it does not cause problems even when parts of multiple detection nodes are inserted into one basic block of a candidate method.
Figure 3.3 shows, as an example, how to inject a detection node into a candidate method. In the CFG of the candidate method, we first randomly select a basic block; assume it is node 11. Then we adopt the classic Lengauer-Tarjan algorithm [65], an efficient dominance algorithm, to find all the dominators of node 11: node 1, 4, 5, and 6. The five nodes (node 1, 4, 5, 6, and 11) are stored in a set $S$, and the detection node $d$ is split into three parts: $d_1$, $d_2$, and $d_3$. The number of the split parts is a random integer not greater than the number of instructions in $d$ and the number of nodes in $S$. We then randomly select three nodes from $S$; assume they are node 4, 5, and 11. Finally, we inject $d_1$, $d_2$ and $d_3$ into the three nodes, respectively.

## 3.5 Repackaging Response

This section presents how to construct and inject response nodes with design and implementation details.

### 3.5.1 Response Node Construction

#### 3.5.1.1 Stealthy-modification Mechanism

A few response strategies for tamper-proofing have been proposed [35, 55]. The response is usually conspicuous in the form of, for example, program crashes and infinite loops, which are very unusual behaviors. Such that attackers can quickly locate the injected response, and then try to trace back to the response injection code.

Instead of stopping apps from working, we propose to inject responses in the form of delayed logical malfunctions: (1) after a response is injected, it takes effect after some delay; (2) the response is in the form of logical malfunctions, so that when attackers notice it, little trace is left behind. Combined with stochastic responses (Section 3.5.3), delayed logical malfunctions make debugging and evasion
attacks on the attacker side much more difficult.

We give some examples of responses in the form of delayed logical malfunctions. In the OpenSudoku game, assume a response has been injected, so that when a user selects a puzzle, instead of rendering the selected puzzle, some other puzzle is presented to the user. In this example, the response shows itself as some logical bug. Alternatively, the response may turn the app in some disorder, e.g., showing unrecognizable text, rendering a huge button, or supplying a textbox too small to type in. These effects accumulates very negative user experiences, so that few benign users continue using the app. On the other hand, the response is not evident in the beginning and it even does not show itself until some delay after it is injected; thus, when attackers decide to launch debugging, it is difficult to reproduce the problem and locate the response injection code.

To implement delayed logical malfunctions, we propose *stealthy-modification methods*, which stealthily modify particular features of an app so as to cause logical malfunctions. We consider two types of stealthy-modification methods. The first one is to modify integer variables; for example, modifying the Intent value to disturb inter-activity communication, modifying the attributes of Button, TextView, EditText objects, such as the size, visibility, and inputType, modifying the operands in arithmetic operations. The second type is to modify string variables, for example, appending a random string to it.

The modifications may result in program crashes instead of intended delayed logical malfunctions. For instance, modifying the array index may lead to buffer overflow exceptions and hence crashes. We can check against crashes, for example, by avoiding overflows in the previous example, but we choose not to do so in order to simplify the response nodes. Our goal is to achieve the delayed logical malfunctions in most situations rather than all.
3.5.1.2 Construction Process

To automatically construct the response nodes, we search the app’s code to find all of the candidate variables which our stealthy-modification methods can apply to. We consider all class member variables (i.e., member fields) as candidate variables.

For each candidate variable, a stealthy-modification method is applied to constructing the response node. We then apply code obfuscation techniques on the constructed response nodes to make them difficult to reverse-engineer.

3.5.1.3 An Example

Figure 3.4 shows the code snippet of a response node with respect to a candidate variable \( v \).

As shown in this code snippet, the response node first monitors the communication channel to infer the detection result (Line 2; how to infer the detection result is presented in Section 3.6). If repackaging is detected (Line 4), the response node modifies the candidate variable \( v \) by adding a random number to it (Line 5).

One detail is that when applying modifications to member variables of a class and inserting the corresponding response node to another class, the modifications are subject to the scope of the variables and availability of the assistant methods. For example, the attribute declared in the Button class are all member variables; they can only be modified through set* methods when methods of other classes are to modify them.
Algorithm 2 Response Nodes Injection

$G$: a candidate method

1: function RESPONSENODESINJECTION($G$)
2: $\alpha \leftarrow$ ReferencedCandidateVariables($G$)
3: for each $\alpha_i$ in $\alpha$ do
4: \hspace{1em} $\lambda \leftarrow$ FindJavaClass($\alpha_i$)
5: \hspace{1em} site $\leftarrow$ FindInjectionSite($\alpha_i$, $\lambda$)
6: \hspace{1em} Injection($\alpha_i$, site)
7: end for
8: end function

9: function INJECTION($\gamma$, $\Gamma$)
10: \hspace{1em} (name, type) $\leftarrow$ FindNameType($\gamma$)
11: \hspace{1em} $r \leftarrow$ ConstructResponseNode(name, type)
12: \hspace{1em} RandomInject($r$, $\Gamma$)
13: end function

3.5.2 Response Node Injection

To automatically inject the response nodes, we first determine where to inject them. Again, we use relative cold methods as candidate methods (Section 3.4.2.1). Note that each response node is quite small and simple; thus, unlike injection detection nodes, we inject a response node as a whole rather than splitting it into multiple parts.

Algorithm 2 shows the pseudo-code for injecting the response nodes. Given a candidate method $G$, the function first finds candidate variables referenced by the method (line 2). Then for each candidate variable $\alpha_i$, it determines a proper injection site (line 5). In order to cause a delay between the response injection site and the point where a response is evident, we define a method that does not reference the candidate variable as its proper injection site; such that, after a member variable is assigned with some problematic value by a response node, the variable is not used until the method containing this response node returns and another method references it. In this way, it creates extra difficulty for attackers to locate the response node when the malfunction is evident. If there exist multiple proper
injection sites, a random one is selected. Then the response node is constructed and injected to the selected injection site (line 6).

### 3.5.3 Stochastic Responses

To improve the resilience to evasion attacks, we integrate the *stochastic* response mechanism based on nondeterministic programming. Nondeterministic programming is a programming strategy that does not explicitly specify which further action would be executed at a certain point, called the choice point; instead, it allows the program to make a choice among a number of alternatives based on a selective method at runtime [29]. In our case, each response node is a choice point, and the alternatives of a response node include whether or not the candidate variable is to be modified, and, if it is to be modified, the possible values that can be assigned to it.

Several selective methods have been proposed [29]. For example, one can generate a random number, based on which one of the alternatives is selected; or read an uninitialized variable and use its value to determine an alternative; or use the race condition on concurrent threads, or an unordered iteration, or memory address, or time stamps, etc. We adopt the first one as our selective method.

### 3.6 Communication Channel

This section presents the system design of the communication channel. We first present the requirements for the communication channel in our system, and then propose a new type of communication mediums.

#### 3.6.1 Stealth Communication Channel

To support the information transmission between detection and response nodes, we implement several different communication channels. Specifically, we divide all
of the detection and response nodes into several sets, and implement a different communication channel for each set; this way, the detection and response nodes within the same set communicate based on the same communication channel.

We do not implement different communication channels for each pair of detection and response nodes. Because an app is an event-driven application, it is difficult to pre-determine whether both of detection and response nodes of the same pair will execute at an execution path. If not, there will be no response even if repackaging is detected, which will decrease the probability of successfully carrying out responses. We neither attempt to implement the same channel for all pairs of detection and response nodes, in order to improve the resilience.

We aim to implement a reliable stealthy communication channel that is difficult to detect and prevent. Since if the channel involves transmission errors, an app will still incur malfunctions even though it has not been repackaged. There are a few stealthy (covert) channels for Android apps proposed in the literature [66, 67, 68]. The first one is based on vibration settings; when they are changed, the system sends a notification to interested apps. The second one is based on volume settings; it is similar to vibration settings. The third one is based on the mobile phone screen; the changes on the screen will trigger a notification to interested apps. The forth one is based on Intent; the transmitted data is encoded into an additional field of an Intent. We do not adopt them for our system. For the first three channels, they incur some noise when a user himself performs some actions on the settings which cannot be avoided. In addition, they are not stealthy in the context of apps (few app modifies these settings); thus they can be easily identified by attackers. For the last one, the number of detection and response nodes that can be injected is restricted by Intent, which limits the feasibility of our system.
3.6.2 Communication Medium

We propose a new communication medium which can be used to implement a reliable stealthy channel that meets our requirements. The communication medium is a resource (a final variable) in the R class. The R class is a Java class, automatically generated by aapt (the Android Asset Packaging Tool), containing the resource IDs for all the resources in the res directory. For example, R.drawable.icon is a resource; it has a resource ID which is a static integer and can be used to retrieve the icon resource. We randomly pick several resources in the R class as our communication mediums, each of which is used to implement a reliable stealthy channel. We modify their resource IDs if repackaging is detected. Because a final variable cannot be modified once it has been assigned, we use Reflection to achieve it. Since there are usually many places in the original app that access the resources, the injected code are stealthy in the context of the app. Moreover, when DVM (Dalvik Virtual Machine) compiles the source code to generate the Dalvik bytecode, it replaces the references to the resources everywhere in the code with the actual values, namely, the resource IDs; thus the modification on the resource IDs during runtime will not change the semantics of the app.

Because our communication channel relies on Reflection, it should be stealthy in apps that involves enough Reflection operations. In fact, Reflection is widely used in Android apps [69, 70], which is also confirmed in our evaluation. However, there are some apps in which Reflection are scarce. For these apps, one possible attack is to search all Reflection in the app and then disable/remove these Reflection one by one. To counter this attack, we generate decoy Reflection. Specifically, we deliberately select a few callee methods from the original code and transform them to utilize Reflection to call them; such generated Reflection are called decoy Reflection. In this way, attackers cannot distinguish the Reflection used in the communication channels from the decoy Reflection. Moreover, we apply code obfuscation techniques on the communication channels to make them difficult to reverse-engineer and
improve the stealthiness. Furthermore, we adopt opaque predicate to add many instances of Reflection, which confuses attackers and increases their uncertainty as to whether or not they have successfully disabled/removed the communication channels, and thus improve the resistance of our system.

### 3.7 Security Analysis

#### 3.7.1 Static Analysis

One important issue is that if detection and response nodes contain fixed code sequences, they may be easily identified and removed by attackers through statically scanning (e.g., using pattern matching and text searching techniques). For detection nodes, because we construct them based on different predefined “polymorphic” templates and apply code obfuscation techniques, the resulted detection nodes are distinct. Moreover, we split each detection node into several parts and weave these parts into the original app code. In this way, it is quite difficult to identify the detection nodes. With regard to response nodes, because different response nodes target different variables, these response nodes are distinct. In addition, we also apply code obfuscation techniques on them, making them more distinct from each other and improving their stealthiness.

#### 3.7.2 Dynamic Analysis

Attackers can also perform dynamic analysis to identify the detection and response nodes. For the detection nodes, if an attacker executes and observes the protected app in its dynamic environment, he may be able to identify the detection nodes and recognize the hard-coded public key substrings. After that, the attacker may replace the original public key substrings with his own one. However, since we inject multiple detection nodes in terms of the execution paths into the app, the
attacker must iterate this attack for many times to visit all or most of the execution paths. Although automated test input generators can be leveraged to generate the possible inputs (e.g., Randoop [71], Robotium [72]), or directly analyze specific method calls (e.g., Brahmastra [73]), it still requires attackers to inspect the code to pinpoint the detection nodes and make any code modifications necessary so as to disable/remove them. Moreover, because each detection node is obfuscated and then woven into the surrounding code without identifiable boundaries, it greatly increases the investment cost for attackers.

With respect to the response nodes, because our stealthy-modification mechanism causes the delayed logical malfunctions which leave little noticeable traces behind, it is very difficult to find the failure points (where the malfunctions explicit), let alone to trace back to the response nodes. Moreover, to analyze an app, attackers have to feed it with the same input many times and hopefully can observe the same output. However, because of our stochastic response mechanism, they are faced with different delayed malfunctions given the same input, rendering it a more complicated scenario to investigate.

Attackers may adopt the taint analysis to taint the communication mediums once identified, or packages.xml containing the public key. However, because we utilize string operations to manipulate the mediums and file name at runtime and transform them to Reflection for accessing the resources, the communication channel and protection nodes are difficult to be identified by current taint analysis. We examined taint analysis attacks in our evaluation Section 3.8.4.3.

### 3.7.3 Replay Attacks

The last issue is replay attacks. We assume that attackers know we use the public key to detect repackaging. Because the public key is distributed with an app’s apk file (contained in the selfsigned X.509 certificate “META-INF/CERT.RSA”) for signature verification, an attacker can easily obtain it. However, because he does
not know the private key, he has to choose a new certificate to sign the repackaged app. To bypass the repackaging checking of SSN, the attacker’s goal is to provide the original public key to the detection nodes whenever the detection nodes ask for \( K_r \). Because \( K_r \) is extracted from packages.xml, one possible way is to substitute the public key stored in packages.xml with the original one. However, this does not work. Because packages.xml is owned by the system user and system group, it cannot be modified by any app process, which means that once the public key is stored in packages.xml, it is protected by the Android system and cannot be modified by other processes.

Another possible way is similar to the man-in-the-middle attack. It refers to an attacker attempting to hijack an essential function, e.g., getPublicKey, so that whenever it is called, the original public key will be returned. However, when an app is launched, Zygote (an Android system service who launch all apps) creates a clone of itself and initializes a DVM which preloads the required system classes before the app is started; this way, the system classes are already loaded and linked by Zygote. Thus the attacker cannot override the path of the system class so as to load his own class to substitute the system class. In addition, because Android runs on Linux whose kernel implements a strategy called Copy-On-Write (COW), no memory is actually copied and the memory is shared and marked as copy-on-write, meaning that the system classes are read-only and not writable; thus, the attacker either cannot overwrite the system class after it is loaded.

A more advanced way is to hijack vtable. A vtable is a virtual function table, pointed by a virtual table pointer (vfptr). A vtable consists of several virtual function pointers. An attacker can either overwrite a vfptr to point to an attacker-crafted vtable, or overwrite a vtable’s contents to manipulate virtual function pointers, causing further virtual function calls to be hijacked [74]. For instance, an attacker can swap the pointer of getPublicKey with his own defined method, such that whenever getPublicKey is called, the attacker’s method will be executed.
Our work does not deal with such attacks, as it is out of the scope of this work. Many techniques have been proposed to address this problem in the literature and can be adopted [74, 75, 76, 77].

3.8 Evaluation

3.8.1 Experimental Settings

We evaluated our tool SSN based on the following four aspects: feasibility, effectiveness, resilience, and side effects. We randomly collected 600 apps in 10 categories from F-Droid [78], which is an catalogue of open source Android apps. Each category contains 60 apps. We conducted our experiments on a Nexus 4 ARM emulator with Android 4.1 system. SSN is flexible to work on different devices with other Android versions. We choose to use emulator in our experiments as it is flexible to configure and does not affect our results and conclusions compared with using real devices.

3.8.2 Feasibility

We begin by studying the feasibility of SSN. We seek to understand: (1) the percentage of the apps using Reflection, and (2) whether SSN can inject enough detection nodes and response nodes into the apps. We show the results in Table 3.1.

In Table 3.1, there are five columns, including the average lines of code (LOC), the percentage of apps using Reflection, the average numbers of detection nodes and response nodes injected, with respect to the 10 categories of apps. Note that the LOC only includes the Java source code of each app.

For each app, the number of the response nodes is determined by the number of the candidate variables, and the number of the detection nodes is decided by the number of basic blocks in the candidate methods; specifically, for a candidate
Table 3.1: Feasibility analysis results

<table>
<thead>
<tr>
<th>Category</th>
<th>Avg LOC</th>
<th>% of Apps use Reflection</th>
<th>Avg # of Detection nodes</th>
<th>Avg # of Response nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development</td>
<td>5,684</td>
<td>66.7%</td>
<td>368</td>
<td>232</td>
</tr>
<tr>
<td>Office</td>
<td>6,268</td>
<td>40%</td>
<td>406</td>
<td>285</td>
</tr>
<tr>
<td>Multimedia</td>
<td>10,080</td>
<td>73.3%</td>
<td>658</td>
<td>344</td>
</tr>
<tr>
<td>Game</td>
<td>6,900</td>
<td>43.3%</td>
<td>455</td>
<td>276</td>
</tr>
<tr>
<td>Internet</td>
<td>15,170</td>
<td>56.7%</td>
<td>988</td>
<td>546</td>
</tr>
<tr>
<td>Security</td>
<td>10,609</td>
<td>36.7%</td>
<td>685</td>
<td>423</td>
</tr>
<tr>
<td>Reading</td>
<td>14,625</td>
<td>46.7%</td>
<td>946</td>
<td>471</td>
</tr>
<tr>
<td>Navigation</td>
<td>13,938</td>
<td>50%</td>
<td>887</td>
<td>483</td>
</tr>
<tr>
<td>Phone&amp;SMS</td>
<td>13,391</td>
<td>50%</td>
<td>859</td>
<td>503</td>
</tr>
<tr>
<td>Science&amp;Educ.</td>
<td>9,800</td>
<td>43.3%</td>
<td>623</td>
<td>443</td>
</tr>
</tbody>
</table>

method has $n$ basic blocks, we injected $\lceil n/5 \rceil$ detection nodes. Since the detection nodes are mutually independent, the more detection nodes are injected, the better resiliency of the repackage-proofing protection has; however, on the other hand, it results in more runtime overhead to the protected apps. Thus there is a trade-off between the number of injected nodes, the resiliency and the performance overhead. In our experiments, we chose to inject $\lceil n/5 \rceil$ detection nodes to a candidate method with $n$ basic blocks, to achieve good resiliency and small overhead. However, one can easily modify the number for other choices based on the demand. For example, if he wants to protect a real-time app, he may choose to inject fewer nodes to attain smaller overhead. All the detection and response nodes for each app were divided into eight sets; each set corresponded to an independent communication channel. The number of sets can also be configured based on the demand.

Table 3.1 also reveals that almost 51% of the 600 apps in our data set use Reflection. For some apps that do not involve Reflection, we generated decoy Reflection by selectively picking a few callee methods and transforming them to use Reflection to call them. For all the 600 apps, we also adopted opaque predicate to add many instances of spurious Reflection to confuse attackers. Moreover, we
used proGuard [61] to obfuscate the code, to make it difficult to reverse-engineer and improve the resistance.

From the results, we can conclude that (1) Reflection is widely used in Android apps, which is also demonstrated in previous work [69, 70], and (2) SSN can inject many detection nodes and response nodes into apps, and has good feasibility.

3.8.3 Effectiveness

We then evaluated the effectiveness of SSN. We want to understand: (1) whether or not SSN can prevent the repackaged apps from running successfully on user devices, and (2) the percentage of the repackaged apps that are deterred from working properly.

For each app, we first utilized SSN to build a repackage-proofing protection, and signed it using the original certificate to generate the apk file. These apk files are the protected apps. Next, we used Apktool [79] to repackage these protected apps and resigned it using a new certificate. After that, we obtained a set of repackaged apps. We installed these repackaged apps on our Nexus emulator, and tested each of them.

The results show that all of the repackaged apps could not run successfully on the device. Some repackaged apps crashed once they were launched, while some others could be launched but incurred various malfunctions and worked abnormally. Therefore, SSN successfully deterred all of the repackaged apps from working properly. We also evaluated the impact of the injected protection code on the functionality of the protected apps, which is presented in Section 3.8.5.
3.8.4 Resiliency

3.8.4.1 Resiliency to Static Analysis

Attackers may statically analyze the protected app to identify the detection nodes and response nodes. Note that our assumption is that attackers can only get the apk files, but cannot obtain the source code, which is reasonable in practice. Moreover, we also assume that the end users are legitimate ones and will not modify the apk files.

To analyze an app, the attacker has to first utilize a disassembler (e.g., apktool, baksmali) to disassemble the apk file and generate the smali files containing Dalvik bytecode. In the following, we assume that he has obtained the smali files. Then the attacker may statically scan the smali files for some essential functions (e.g., getPublicKey, getCertificate, etc.). However, because we use string manipulation and Reflection to hide the signatures of these functions, it prevents the attacker from finding them. The attacker may also scan the smali files for similar code segments if he has identified one node. However, because the constructed nodes are distinct and are applied on code obfuscation, the resulted nodes are difficult to be identified on the basis of another one. One more sophisticated attack is to adopt taint analysis to taint the communication mediums or packages.xml, so that whenever they are accessed, the location of the nodes could be identified. The related evaluation results are presented in Section 3.8.4.3.

3.8.4.2 Resiliency to Dynamic Analysis

An alternative attack is to dynamically execute and monitor the app to identify the injected nodes. To identify the detection nodes, one option is to combine debugging tools and automated test input generators to execute the app and observe its behaviour, and then pause the app to start debugging once a failure is noticed. A more advanced way is to hook some essential functions (e.g., getPublicKey) to
learn when the hooked functions are activated so as to identify the code regions for
the detection nodes. We conducted three user studies in the following. Attackers
may also adopt dynamic taint analysis to taint the communication mediums or
packages.xml. We present the related evaluation results in Section 3.8.4.3.

We now consider how attackers identify the response nodes. Utilizing taint
analysis is one option. Another option is to find the failure points, and then trace
back to the response nodes. However, because our stealthy-modification mechanism
causes the delayed logical malfunctions which leave little noticeable trace behind, it
is very difficult to identify the failure points, let along to trace back to the response
nodes.

**Examples of stochastic logical malfunctions.** Figure 3.5 and Figure 3.6
show the screenshots of OpenSudoku when it works normally and abnormally,
respectively. From Figure 3.6, we can see that when OpenSudoku is repackaged,
it only incurs logical malfunctions; specifically, Figure 3.6(a), (b) and (c) show
the screenshots when OpenSudoku is subjected to stochastic responses; different
malfunctions are incurred at different times. Besides these, there are some other
stochastic malfunctions. For instance, when an attacker selects a puzzle, a different
one may be presented at a different time; or when an attacker selects a cell in the
puzzle interface and types a number into the cell, a different number may be typed
Figure 3.6: Examples of OpenSudoku when malfunctions occur.

into an arbitrary cell at a different time. These stochastic malfunctions greatly frustrate attackers and maximize the workload. Other apps suffered from similar or different kinds of stochastic malfunctions. Due to limited space, we do not present them here.

Case studies. We now seek to understand how quickly an attacker can identify a node, and how many nodes can be identified in a given time, based on simple dynamic analysis. In principle, it depends on many factors (e.g., the attacker’s skill, the complexity of an app, the automated tools available, etc.) and cannot be easily quantified.

We conducted three case studies. For each app, we generated its protected apk by SSN. To begin, we asked three human players to play these apks to confirm
Table 3.2: Surviving protection nodes of each app.

<table>
<thead>
<tr>
<th>App</th>
<th>LOC</th>
<th>Total # of Det. nodes</th>
<th>Total # of Res. nodes</th>
<th>% of Surviving Det. nodes</th>
<th>% of Surviving Res. nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CatLog</td>
<td>5667</td>
<td>378</td>
<td>231</td>
<td>82.3%</td>
<td>89.6%</td>
</tr>
<tr>
<td>PhotoGallery</td>
<td>2011</td>
<td>137</td>
<td>90</td>
<td>78.9%</td>
<td>76.3%</td>
</tr>
<tr>
<td>OpenSudoku</td>
<td>6079</td>
<td>404</td>
<td>253</td>
<td>81.4%</td>
<td>80.1%</td>
</tr>
<tr>
<td>CEToolbox</td>
<td>2422</td>
<td>168</td>
<td>194</td>
<td>75.0%</td>
<td>82.9%</td>
</tr>
</tbody>
</table>

that the functionalities of these apps were still held. Then we provided these apps to another three human testers who did not know our certificate but knew SSN’s techniques, to analyze the apps separately for twenty-four hours and try to break the repackage-proofing protection built into the apps. The three human testers have three, two, and one years of Android development experiences, respectively. They are skilled in the debugging tools such as DDMS [80] and test input generators such as Monkey, Randoop [71], Robotium [72] and Brahmastra [73]. After twenty-four hours, they repackaged these apps using their own certificate (a new one) to generate the repackaged apps. Finally, we asked the previous three human player once again to play these repackaged apps for one hour; if the app works properly for one hour, we consider it has been successfully repackaged. Note that we do not require every nodes have to be disabled/removed to achieve successful repackaging. Instead, we claim that an app is successfully repackaged as long as it can work properly for one hour.

Table 3.2 shows the results of four sample apps, CatLog, PhotoGallery, OpenSudoku, and CEToolbox, from the category of Development, Multimedia, Game, and Science&Education, respectively. Other apps have shown similar results and are not included here. In Table 3.2, the percentage of surviving nodes is the average percentage among the three human testers. We can see that even after twenty-four hours, almost four-fifths of the detection and response nodes are still not being identified. Moreover, none of the apps was successfully repackaged reported by the
three players.

We then asked the three testers for their experience. They indicated that at first they tried to statically scan the apps but few useful information was obtained. They then had to dynamically execute the apps to observe their behaviour with the help of the debugging tools and test input generators; but unfortunately, they encountered different malfunctions each time. Moreover, the malfunctions caused the apps to work abnormally and they did not know where the malfunctions started. Thus they had to inspect the bytecode carefully using the debugging tools and try to understand the work flow of the apps which cost them a lot of time. When they had identified a detection node or response node, they tried to figure out the final variable that was used as the communication medium. They then adopted the taint analysis tools (e.g., TaintDroid [81]) to taint it and hoped to find the detection and response nodes easily; however, this did not work. Next, they set a watch window in the debugging tool to watch this variable at runtime to detect its modification; however, because the final variable was declared in the R class which was out of the scope chain for debugging, the watch window did not work. Therefore, they had to set multiple conditional exceptions in the code which would suspend the execution once the final variable was modified, and then gradually shorten the searching range. The entire process was very laborious, error-prone and time-consuming.

During their analysis, we also recorded the number of the identified nodes for every two hours. We found that the numbers were decreased during the twenty-four hours, indicating that the difficulty in detecting nodes was increasing. It is intuitive as the more nodes existed, the easier for triggering one, causing more nodes being executed in a certain time, and vise verse. Thus, we infer that less nodes would be detected after the twenty-four hours analysis.

**Advanced dynamic analysis attacks.** Attackers may perform more advanced dynamic analysis attacks. For instance, they may utilize `ptrace` to control
the execution of a process, but to carry out it requires sufficient knowledge of low-level programming. Moreover, the rationale is that only one process can attach to a target process at the same time. To prevent attackers using ptrace, we can let the app attaches to itself at runtime; this way, ptrace cannot attach to it [82, 83]. Attackers may leverage existing binary instrumentation tools such as Adbi [84] and DECAF [85] to monitor some essential functions (e.g., `getPublicKey`), and use automated test input generators such as Monkey and Brahmastra to automatically execute the app, so that when these functions are activated, the locations of the corresponding nodes can be logged. However, some patches on these tools are needed to achieve this goal; in addition, most automated test input generators can only cover partial execution paths of an app [86, 23, 5]. Thus, the attacker still needs to analyze many traces and inspect the code manually to uncover the other nodes.

### 3.8.4.3 Resiliency to Taint Analysis

Attacker can also adopt taint analysis to taint the communication mediums or `packages.xml`, to discover the injected nodes.

**Communication mediums.** To apply taint analysis, the attacker must identify the final variables used as the communication mediums. We assume he finds one through some ways. For static taint analysis, since we adopt string manipulation on the final variable’s name at runtime and transform that to Reflection for accessing the variable, it is difficult to detect the data flow between the sources and sinks (the places where the source is accessed), given the string is unknown statically. Moreover, static taint propagation usually makes the program end up with many false tainted values, and leads to high computation overheads and large memory consumptions. Thus, static taint analysis is impractical to counterattack our technique.

With respect to dynamic taint analysis, we utilized TaintDroid [81] and DE-
CAF [85]. TaintDroid is a dynamic taint tracking and analysis system capable of tracking sensitive data. DECAF is a dynamic binary analysis platform based on QEMU. Assume the communication medium is a final variable str. For TaintDroid, we first added Taint.addTaintString(str, Taint.TAINT_MIC) at the beginning of the create method in the main activity so that once the app started, str would be tainted. Then we inserted the log statements after each detection and response node to check whether or not TaintDroid could detect str was accessed. Next we employed Monkey to execute the app for one hour, and checked the log; however, nothing was logged. The reason is that the communication medium is a resource in the R class and is accessed by Reflection based on strings of its name after manipulated. Similar results were obtained from DECAF.

The packages.xml file. Attackers may attempt to taint packages.xml which contains the public key to reveal the detection nodes. As we use PackageManager to extract the public key, the file name is not explicitly stated in the detection nodes (see the code snippet in Section 3.4.1.2). Thus, static taint analysis is ineffectual under such circumstances.

For dynamic analysis by TaintDroid, we first added Taint.addTaintFile(int-Fd, Taint.TAINT_MIC) at the beginning of the create method in the main activity, where intFd is a file descriptor of packages.xml, and then checked whether TaintDroid could identify intFd was accessed. However, TaintDroid failed to do so. The reason is that instead of directly reading the public key from packages.xml, we adopt PackageManager. The public key is extracted by getPublicKey from sun.security.x509.X509CertImpl in JDK that TaintDroid does not taint track.

In this evaluation, we used two state-of-the-art taint analysis tools. However, a better taint analysis may successfully taint both the communication mediums and packages.xml, which we recognize is possible theoretically. However, it still requires attackers to inspect the code manually to pinpoint the protection nodes and disable/remove them. As the nodes are obfuscated and woven into the surrounding
Table 3.3: Runtime overhead.

<table>
<thead>
<tr>
<th>App</th>
<th>$T_o$ (sec)</th>
<th>$T_p$ (sec)</th>
<th>Runtime Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>CatLog</td>
<td>185</td>
<td>208</td>
<td>12.4%</td>
</tr>
<tr>
<td>PhotoGallery</td>
<td>108</td>
<td>115</td>
<td>6.4%</td>
</tr>
<tr>
<td>OpenSudoku</td>
<td>209</td>
<td>227</td>
<td>8.6%</td>
</tr>
<tr>
<td>CEToolbox</td>
<td>141</td>
<td>152</td>
<td>7.8%</td>
</tr>
</tbody>
</table>

code without identifiable boundaries, it increases the investment cost for attackers. Furthermore, because multiple nodes are injected in terms of the execution paths, attackers have to iterate this attack as many times to visit sufficient execution paths for insuring the safety of republishing the app. The cost may be high and no longer be attractive for attackers.

3.8.5 Side Effects

We further evaluated the side effects of SSN on the apps, from the following two aspects: the impact on the functionality, and the impact on the runtime overhead.

3.8.5.1 Impact on the Functionality

The normal operations of an app should remain intact after protected. To evaluate it, we utilized SSN to build repackage-proofing protection into each app, to generate the protected app. We then asked three human players to play these protected apps for two hours to check whether they could run successfully. All they reported were positive results. Therefore, the protection code injected by SSN has no impact on the functionalities of the apps.

3.8.5.2 Impact on the Runtime Overhead

The application runtime overhead introduced by SSN comes from two major types of computations: (1) the reading public key certificate procedure by PackageManager, and (2) the Reflection operation modifying the value of a final variable. To evaluate
the runtime overhead, we utilized Monkey to generate a sequence of 10,000 user events, and fed the same user events to both the apps with and without the protection code embedded three times to measure the average running time. We denote the average running time of the original app and the protected app as $T_o$ and $T_p$, respectively; the runtime overhead is calculated by $(T_p - T_o)/T_o$. Table 3.3 shows the runtime overhead with respect to the four apps. Results with other apps are similar and are not included here due to the space limitation.

From Table 3.3, we can see that the runtime overhead has a positive correlation with the number of the detection and response nodes injected (however, CatLog is an exception). For example, the amount of the detection and response nodes injected into PhotoGallery is smaller than those injected into CEToolbox; and the runtime overhead of PhotoGallery is lower than that of CEToolbox.

To understand the reason why the runtime overhead of CatLog is particular higher than others’, we examined its source code. CatLog is an app showing a scrolling view of the Android “Logcat” system log, which has multiple reading log procedures that read logs line by line to examine the context of logs (e.g., similar to text searching). The reading log procedures are usually coded as highly repetitive loop statements (e.g., \texttt{while((line=read.readLine())!=null})). If one or more nodes are placed within the highly repetitive loop, they will be executed as many times as the loop iterates, which will greatly sacrifice the performance. Although during the injection, we utilize Traceview to exclude the “hot” methods, we do not exclude the highly repetitive loops. Indeed, in many cases, the nodes do not need to execute over and over again if all they do is to repeat the same self-checking or response action.

To improve the performance, one can opt for a smarter injection strategy. One strategy is to avoid injecting nodes within the performance-sensitive code segments. For instance, one can generate the CFGs of each candidate method, and then exclude the highly repetitive loops so that none of the nodes will be injected into
these loops. In this way, one can quantitatively evaluate the trade-off between the runtime overhead and the resilience based on the demand. Another strategy is to declare a boolean variable for each node to make sure that it will execute only once. We adopted the second strategy and generated another set of the protected apps containing exactly the same number of the detection and response nodes except that in this case each node would execute only once. We redid the experiment and calculated the new runtime overhead; the new result for CatLog is reduced to 8.2%.

From the results, we can see that SSN incurs relatively small runtime overhead to the protected apps, and thus is efficient to defend apps against repackaging.

### 3.9 Discussion

#### 3.9.1 Impacts

SSN is the first work reported in the open literature that prevents repackaged apps from working on user devices without relying on authorities. It builds a complex stochastic stealthy network of defences into apps, such that repackaged apps cannot run successfully on user devices. Unlike repackaging detection techniques based on code similarity comparison that can be easily evaded by various obfuscations, SSN can resist this kind of evasion attacks. No matter whether an app is obfuscated by attackers in order to bypass the code similarity checking, as long as it has been built with the repackage-proofing protection before releasing, the repackaged app cannot run successfully on user devices, which limits its propagation as well as its harms to the financial security of app developers and user privacy. Thus, repackage-proofing that stops repackaged apps from working is a promising direction in defeating the prevalent repackaging attacks.
3.9.2 Limitations

There are several limitations of SSN. First, although SSN has good resiliency to many evasion attacks (as showed in our evaluation), we do not guarantee that determined attackers cannot disable our protection from an app. For example, the attacker may spend much effort and time monitoring the app’s execution to identify the code region for the detection nodes. Or he may leverage a better taint analysis techniques (if available) to taint both the communication mediums and packages.xml, to obtain the tainted data flow revealing the injected nodes.

Attackers can inspect the app code manually to pinpoint the nodes and make any code modifications necessary to disable/remove them. Note that attackers do not need to remove every last node to successfully repackage an app. As long as they ensure that the repackaged app can work properly long enough (e.g., one hour, or longer), they tend to be satisfied. Each time after removing a node, attackers need to confirm the functionality of the app is not broken and the node is successfully bypassed.

However, because of the stochastic response mechanism, they cannot be sure. Moreover, we can also selectively inject many spurious nodes to confuse attackers and increase their uncertainty. Furthermore, as multiple nodes are injected in terms of the execution paths, attackers have to iterate this attack as many times to visit sufficient execution paths for guaranteeing the safety of republishing the app. To increase their risks, we can opt for injecting nodes that notify users the app has been repackaged and may be dangerous, into infrequently executed paths or methods. This way, even only those nodes are remained, and activated only once or twice a day or month, there is still chance that users are informed of the danger and report it to Google Play, which then removes the repackaged app from all user devices. We leave these as our future work.

Another limitation is that attackers can conduct hijacking vtable attacks, which either overwrite a virtual table pointer or manipulate a virtual function pointer, to
bypass the repackaging checking. We refer the reader to Section 3.7.3 for details. There exist many techniques addressing this problem in the literature that can be adopted [74, 75, 76, 77].

It is widely recognized that any software-based protection can be bypassed as long as a determined attacker is willing to spend time and effort, which is also true with our protection. We assume attackers are interested in repackaging an app only if it is cost-effective, for example, when the cost of repackaging is less than that of developing the app themselves.

3.10 Summary

To the best of our knowledge, there is no study that investigates repackage-proofing to prevent repackaged apps from working on user devices. In this paper, we conduct a preliminary first-step study on the problem and introduce a repackage-proofing technique called SSN, Stochastic Stealthy Network, which provides a reliable and stealthy protection for Android apps. We have developed a prototype of SSN. Our experimental results show that SSN is effective and efficient.
System Service Call-oriented
Symbolic Execution of Android
Framework with Applications to
Vulnerability Discovery and Exploit
Generation

4.1 Introduction

The global smartphone market is booming and Android dominates the market with a share of 87.6\% [87]. As of 2016, there were 1.4 billion active Android devices [2] and over 65 billion Android apps that had been downloaded and installed [88]. An Android app relies on the Android Application Framework (\textit{Android Framework}, for short) to make it useful. For example, all the user interface designs and multi-tasking features would not work without WMS (Window Manager Service) and AMS (Activity Manager Service) provided by Android Framework; as another example, apps cannot obtain the GPS location without LMS (Location Manager Service)
of the framework. Therefore, Android Framework is an integral and foundational part of the Android system; it runs on each Android device for managing all applications and providing a generic abstraction for hardware access [9]. Recently, many vulnerabilities in Android Framework were identified [10, 11, 12, 13]. A vulnerability in the framework can lead to large-scale cyber attacks and cause serious harms to user security and privacy. For example, malicious apps can exploit such vulnerabilities to steal user passwords, take pictures in the background, launch UI spoofing attacks, and tamper with user data [14, 15, 16].

Despite the critical role of Android Framework and the security concerns due to the vulnerabilities hidden in its several million lines of code, most of the existing work has been focused on analyzing Android applications [81, 89, 90, 91, 92, 93, 94, 95, 96, 97, 54, 62, 52, 7, 98]. Very few systems and tools are available for analyzing Android Framework [99, 69, 100]; as a result, the insecurity analysis of the framework has been imprecise and/or requires significant manual effort [15, 14]. For example, Shao et al. [15] uncovered a very interesting type of Android Framework vulnerabilities that are due to inconsistent permission checking (detailed in Section 4.7.2); however, due to the overwhelming amount of manual effort needed to validate their findings, the process of vulnerability discovery was tedious and error-prone; moreover, the reported vulnerabilities were hard to verify since no PoC exploits were provided. Thus, there is an urgent need for techniques and tools for precise and automated insecurity analysis of Android Framework. In particular, to our knowledge, there is no tool that is able to analyze the framework through symbolic execution, a precise and automated analysis approach that has proven to be very powerful for automatic vulnerability discovery and exploit generation [101, 38, 43].

This work is to fill the critical gap, aiming to (1) design and build a system that enables symbolic execution of the Android Framework code, and (2) given the description of a new type of attacks exploiting Android Framework, apply the system to precisely and automatically finding zero-day vulnerability instances and
generating PoC exploits to validate the findings; such exploits can also be fed into defense systems for automated malware signature generation [101, 102].

While many symbolic execution systems have been proposed for analyzing Windows programs [36, 43, 44], Unix programs [101, 38, 39], and Android apps [45, 46, 47, 48, 103], none has explored how to effectively analyze such complex middleware as Android Framework. Due to unique characteristics of Android Framework, many new challenges arise when building its symbolic execution system.

First, most of the existing systems target stand-alone executables and start analysis of the code from the main function, while Android Framework is a large piece of middleware with a very complex initialization phase, which parses system and app settings and then prepares all the system services. Symbolic execution that starts from the main entry of Android Framework, SystemServer.main, would quickly cause state explosion and hence cannot reach deep code paths. On the other hand, Android Framework exports system services to apps in the form of a large number of service interface methods (also called entrypoint methods); e.g., in Android Framework 5.0, there are 3,079 entrypoint methods exported. Our insight is that, instead of analyzing Android Framework as a whole, the capability of analyzing each entrypoint method separately is the key to the scalability of the analysis.

However, if the analyzer skips the initialization phase and directly analyzes a service interface method, the context information, such as the type and value of variables, is missing and thus many problems may be caused, as is the case in Under-Constrained Symbolic Execution (UCSE) [104, 105, 106].

For example, Figure 4.1 shows the code for the service interface method getProviders, which returns the names of the GPS providers that the calling app is allowed to access. Line 16 contains a virtual function call to checkPermission through mContext, a reference variable of the Context type; Context is an ab-

1The code snippet has been modified slightly from the original to ease the understanding.
Figure 4.1: The `getProviders` service interface method.

```java
// Defined in the LocationManagerService class
List<String> getProviders(Criteria criteria, boolean enabledOnly) {
    int uid = Binder.getCallingUid();
    int level = getAllowedResolutionLevel(uid);
    ArrayList<String> out = new ArrayList<String>();
    for (LocationProviderInterface p : mProviders) {
        if (level >= p.requiredLevel) {
            out.add(p.name);
        }
    }
    return out;
}

int getAllowedResolutionLevel(int uid) {
    // Inside checkPermission(), the function getUserIdLPr() is invoked
    HashSet ps = mContext.checkPermission(uid);
    if (ps.contains(ACCESS_FINE_LOCATION))
        return 2;
    else if (ps.contains(ACCESS_COARSE_LOCATION))
        return 1;
    else
        return 0;
}

// Defined in the Settings class
static final int FIRST_APPLICATION_UID = 10000;
static final int PER_USER_RANGE = 100000;
ArrayList<Object> mUserIds;
Object getUserIdLPr(int uid) {
    if (uid >= FIRST_APPLICATION_UID) {
        uid %= PER_USER_RANGE;
        int index = uid - FIRST_APPLICATION_UID;
        return mUserIds.get(index);
    }
}
```

abstract class extended by four classes, including `ContextWrapper`, `ContextImpl`, `BridgeContext`, and `MockContext`; each implements the function `checkPermission`. Without the concrete `type` information of the object pointed to by `mContext`, it is hard to precisely determine the dispatch target of the call. Such virtual function calls prevail in the framework code.

Similarly, without the value information of variables, the state explosion problem
can be exacerbated. For example, consider mProviders in Line 6 as an example, which is an ArrayList that stores the currently installed GPS providers; if the elements in the list are unknown, it is difficult to carry out a loop that iterates through the list. One workaround is to regard the list as a symbolic input and then handle it using lazy initialization [107]; this way, however, the loop becomes unbounded and elements of the list become symbolic, which unnecessarily exacerbates the state explosion problem.

Therefore, while it increases the scalability of analysis by symbolically executing each service interface method separately, how to deal with the situation of the missing context information is a challenging problem \((C1)\). To resolve it, we employ an analysis scheme that combines concrete execution and symbolic execution, allowing analysis to start from an arbitrary middleware API method (Section 4.3).

Second, Android Framework contains a large number of data structures maintained for both system services and apps. From the perspective of finding vulnerabilities exploitable by a malicious app, variables that are derived from the malicious app are under control of attackers, and thus should be specified as symbolic inputs, such that branches depending on them are all explored during the analysis. However, given a malicious app, the variables derived from it scatter in the form of object fields and array elements in numerous data structures.

In Figure 4.1, for example, mUserIds at Line 28 is an ArrayList that stores the information of all the installed apps with one element for each app. The object pointed to by ps at Line 16 is an element of mUserIds (Line 33); the object is derived from the calling app and thus should be handled as a symbolic input, such that all the branches in the function getAllowedResolutionLevel will be explored by considering different values of the object. While it is not difficult to identify variables derived from the calling app once one understands the program logic, how to automate the process, given the large number of complex data structures in the framework, is an intriguing new problem as well as a challenge \((C2)\).
Our hypothesis is that, as the framework stores information for multiple clients (i.e., apps), there must exist fixed patterns used to access the client-specific information when the framework services a client call, and this hypothesis is validated through our investigation. Based on the patterns how client-specific variables are accessed upon a system service call, a customized taint analysis approach, called slim tainting, is designed to precisely and automatically pinpoint client-specific variables (Section 4.4).

Third, a straightforward design is to place the symbolic execution engine inside the Android system, such that the analyzer can make use of the host execution environment including the native libraries and other supporting processes (e.g., the Service Manager process used to register and query system services). But this way the symbolic executor is tightly coupled with Android, and the implementation has to handle compatibility with the Android Runtime (ART) in terms of thread management, instruction execution, binary representation of objects, and garbage collection. This significantly complicates the implementation of the symbolic executor and makes the system brittle and hard to debug due to various incompatibility issues. Moreover, since it is unlikely to modularize the code for symbolic execution based on this design, whenever a new version of the Android system is released the symbolic execution code has to be re-inserted. To avoid the complicated and brittle implementation and the endless maintenance, a decoupled architecture is desired. However, how to design an architecture that can make use of the Android execution environment without leading to a complicated and coupled implementation is a challenge (C3).

We propose a decoupled design which builds the symbolic executor outside the Android system but is still able to make use of the Android execution environment. An innovative and critical component of the system is to migrate information generated in Android, such as the classes and objects, to the symbolic execution environment (Section 4.5).
We have overcome the challenges above and implemented the symbolic execution system named CENTAUR for symbolic execution of the Java code in Android Framework. The source code for CENTAUR is publicly available. CENTAUR is a path exploration system that can effectively assist automatic and precise vulnerability discovery. We concretely demonstrate how CENTAUR can be applied to vulnerability discovery by considering several recently uncovered attacks that exploit the framework vulnerabilities. We show that the process minimizes the manual effort and guarantees zero-false positives, in contrast with recent researches on finding Android Framework vulnerabilities that rely on laborious and error-prone manual work [15, 14]. Finally, we make use of CENTAUR to generate PoC exploits to validate the findings.

The rest of the section is organized as follows. Section 4.2 covers some background about Android Framework and symbolic execution. Section 4.3 gives an overview of the system. Section 4.4 describes how to identify variables as symbolic inputs. Section 4.5 presents how to migrate the execution context. Section 4.6 describes the implementation details, and Section 4.7 describes the evaluation. The conclusion follows in Section 4.8.

4.2 Background

4.2.1 Android Framework

Android Framework provides a collection of system services, which implements many fundamental functionalities, such as managing the life cycle of all apps, organizing activities into tasks, and managing app packages. Most of the system services, except for the media services, run as threads in the System Server process [9]. Thus, the System Server process plays a central role in Android Framework. This work uses the services in this process as examples to illustrate the ideas and techniques,
which should be applicable to other services.

A system service exposes its service interface methods invokable by apps, and a system service call is handled in the form of a remote procedural call through the IPC mechanism Binder, which dispatches the call to one of the threads of the target system service. Android Framework is mainly implemented in Java. E.g., in Android Framework 5.0, there are 2.4 million lines of Java code and 880 thousand lines of C/C++ code. Currently, CENTAUR can only perform symbolic execution of Java code.

### 4.2.2 Symbolic Execution

Symbolic execution provides a means of efficiently exploring execution paths [108]. For example, consider the function `getAllowedResolutionLevel` in Figure 4.1, by assigning a symbolic value (as opposed to assigning concrete values as in fuzzing) to `ps`, symbolic execution analysis can iterate every of the three paths and precisely provide the condition that the symbolic value should satisfy for executing a given path; e.g., the symbolic execution analysis can produce the path condition for reaching Line 18: `ps.contains(ACCESS_FINE_LOCATION)`.

Symbolic execution is particularly suitable for vulnerability discovery for several reasons. First, it performs an efficient and automatic path exploration and ideally explores all possible paths. So that it is able to discover as many vulnerability instances as possible. Second, for each path explored, it records a path condition, which is a symbolic expression describing the condition that should be satisfied by the input values in order that the path is taken. Consequently, by resolving the path condition, one can obtain the concrete input values that force the execution to follow the corresponding path; the concrete input values can be used to construct exploits, and can be fed into real program execution for verifying the suspected vulnerability. This way, it guarantees zero false-positives in vulnerability discovery.

One of the main challenges in applying symbolic execution to large-sized pro-
grams is to cope with the path explosion problem, as the number of distinct execution paths is exponential in the number of branches that depend on symbolic values. We mitigate the problem using multiple ways, such as analyzing service interface methods separately and precisely identifying variables as symbolic inputs.

4.2.3 Java PathFinder

Java PathFinder (JPF) [109] is an explicit-state model checker for Java programs. It is built on top of a customized Java Virtual Machine (JVM), which runs on the underlying host JVM, and executes the program in all possible ways. JPF stores all the explored states, and it backtracks when it visits a previously explored state. The user can also customize the search (using heuristics) and it can specify what part of the state to be stored and used for matching. Additionally, JPF’s design encourages developers to create extensions to it. Currently there exist many extensions including a symbolic execution extension (see below), data race detector, and an abstract window toolkit (AWT) extension, etc.

During exploring a target program, JPF stores each state of a particular scheduling or value point, consisting of the heap, static area and stacks of all threads. When traversing the state space, JPF checks whether the current state has been already visited. If so, it backtracks to the nearest scheduling or value point, for which there exist an unexplored branch and continues along that. This backtracking is based on keeping a stack representing the currently explored path in the state space (an item in the stack determines the list of not yet visited branches).

By default, JPF searches the state space of the checked program for “low-level” properties, e.g., deadlocks, unhandled exceptions and failed assertions; however as JPF is extensible via the publisher/listener pattern, it allows to observe the course of the state space traversal. This way, listeners can check for specific and more complex properties in each visited state.
4.2.4 Symbolic PathFinder

Symbolic PathFinder (SPF) [41] is a symbolic execution framework on top of JPF. SPF can be understood as a non-standard Java bytecode interpreter, which enforces path exploration when interpreting the code; e.g., when interpreting an `if` statement, it creates two program states so that both branches will be explored. It provides a set of path selection policies that can be chosen from and various constraint solvers for resolving path conditions.

SPF can be extended by overriding methods that are used to interpret bytecode instructions. It also supports the interception of arbitrary function calls for customized handling during the analysis. Specifically, JPF provides a mechanism called Model Java Interface (MJI) that intercepts method invocations for custom handling. Centaur makes use of MJI to intercept certain method calls (e.g., `getCallingUid`, `getPackageName`, and the `get` functions of various collection data structures), and redirects them to our custom implementation of these functions. Finally, attributes can be added to associate with each of the class/object fields on the heap and variables on the stack to record and track states of interest, such as taints and symbolic expressions.

4.3 System Overview

Our observation of Android Framework is that its execution consists of the initialization phase and the ready-for-use phase, and the initialization phase is fairly stable when the system restarts, since the system boots mainly according to the system configuration, which itself is stable. Thus, to resolve the problem of the missing context information (C1), we propose a phased concrete-to-symbolic execution (PC2SE) for analyzing middleware software like Android Framework; it runs the initialization phase as whole-system concrete execution and then performs symbolic execution starting at one of the entrypoint methods under the execution...
context provided by the concrete execution. It avoids the state space explosion due to the complex initialization phase and meanwhile provides the context for symbolic execution, such that the type and value information of the input variables (i.e., non-locally defined variables read during symbolic execution) is available.

When starting the symbolic execution from an entrypoint method, if only the parameters of the entrypoint method are set as symbolic inputs [110], the path exploration will be severely limited, leading to over-constrained symbolic execution. In the framework, variables derived from the malicious app (mainly its manifest file) are also under control of attackers and can affect the execution of system service calls; thus, those variables should also be set as symbolic inputs. To resolve C2 (i.e., identifying variables derived from the malicious app as symbolic inputs), instead of tracking how information is flowed from an app to the framework, we investigate how the app-specific variables in the framework are accessed and propose slim tainting to identify those variables as symbolic inputs on the fly during path exploration by capturing the characteristic access patterns (Section 4.4). This way, the path exploration considers all possible values of these variables.

To address C3 (i.e., to avoid complicated implementation and endless maintenance due to the coupled design), we propose a novel architecture that is suitable for PC2SE, as shown in Figure 4.2, where the symbolic execution engine is built.
outside of Android. As the symbolic execution engine does not need to take care of the comparability issues but is specialized for path exploration, its design and implementation are largely simplified. Plus, since the code for the symbolic executor is separated from Android code, it does not need to be maintained when the Android system is updated.

The whole-system concrete execution is performed in the Android system. Between the Android system and the symbolic execution engine is the execution context query server, which migrates the context information from Android to the symbolic execution engine. How to correctly interpret the semantics of the bytes and bits in the heap captured at Android and to mitigate the information properly will be discussed in Section 4.5. Finally, an RPC server placed inside the Android system, and JNI calls during path exploration are delegated to the RPC server, which will be discussed in Section 4.6.

4.4 Identifying Symbolic Inputs

4.4.1 App-specific Variables

To deal with over-constrained symbolic execution, it is vital to set Android Framework variables that are derived from the malicious app as symbolic inputs, such that execution paths due to all possible values of these variables are explored by symbolic execution.

A closer look at Android Framework reveals that there are two distinct types of variables. The first type, called non-app-specific variables, are allocated regardless of apps in the system. For example, the aforementioned ArrayList LocationManagerService.mProviders (Line 6 in Figure 4.1) exists no matter what apps are running. This type of variables should not be specified as symbolic inputs, since they are not derived from the malicious app.

The second type, called app-specific variables, stores app-specific information.
Some variables store information for all installed apps; for instance, the variable `Settings.mUserIds` is an `ArrayList` that stores the installation data of each installed app (the code path, signature, granted permissions, etc.). Others store the information of running apps, such as task affinities, intents, and back stacks. Unlike the Linux kernel, which stores most information of a process in a centralized structure `task_struct`, the app-specific information in Android is stored in the many data structures for the system services: Android Framework code is structured as a set of system service classes, each of which points to some objects and arrays for storing app-specific information. Therefore, given an app, the framework variables derived from it scatter and exist as objects fields and array elements among the many data structures.

Note that the task of selecting variables as symbolic inputs is not only to find the app-specific variables but also to locate fields or elements within the variables that are derived from a given malicious app. For instance, in addition to determining `Settings.mUserIds` is an app-specific variable, we need to locate which element in the array is derived from the malicious app. Thus, the task is like looking for a needle in a large pile of hay considering the large number of complex data structures.

### 4.4.2 Access Patterns

In order to determine which variables are derived from a given app, a natural method is to track how the information flows from the app to Android Framework via tainting. However, such information flow is very complex involving multiple intricate steps, including app installation, system boot, and starting the app. Given the complexity of these steps and the huge amount of code involved, it is very difficult, if not impossible, to precisely track the information flow using taint analysis. Note that existing taint analysis, such as TaintDroid [81], Chex [91], and TaintART [111], is able to track whether the information of the return values (e.g.,
Figure 4.3: Example of retrieving information from a hash-table-based variable mPackages.

```java
// Defined in the PackageManagerService class
HashMap<String, PackageParser.Package> mPackages;
int checkPermission(String perm, String pkgNm){
    PackageParser.Package p = mPackages.get(pkgNm);
    ...
}
```

GPS locations) of specific system service calls flow to specific sinks (e.g., sending to network), but none is able to track how the whole app-level information propagates to Android Framework variables. Moreover, conventional tainting techniques suffer from the well-know overtainting and undertainting issues, which lead to imprecise tainting results.

Instead of proposing an even more complex taint analysis technique to track the information flow, we resolve the challenge from a novel angle by looking at how the app-specific variables are retrieved and marking them as symbolic inputs in the process of path exploration. Our hypothesis is that, as the framework stores information for multiple apps, there must exist specific ways to retrieve the information for a given app (i.e., the calling app), rather than any other app, when servicing a system service call. Our further investigation has validated the hypothesis and revealed that app-specific variables are stored in two categories of data structures, array-based ones (built-in arrays, ArrayList, SparseArray, etc.) and hash-table-based ones (HashMap, HashSet, etc.), and the two categories are accessed in two characteristic ways, respectively.

First, given an array-based variable, the framework retrieves an app’s information in the array using an index that is a function of the app’s unique UID (an app’s UID is assigned upon installation and not changed). Our investigation further shows that there are two such formulas used to calculate the index. One is $(uid \% 100,000 – 10,000)$, converting the user app’s UID into an index to retrieve
Table 4.1: Taint Propagation Logic. Register variables are referenced by $v_X$. $\tau(y) \leftarrow \tau(x)$ means setting the taint tag of $y$ to the taint tag of $x$.

<table>
<thead>
<tr>
<th>Inst. / Function</th>
<th>Operation Semantics</th>
<th>Taint Propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td>isub</td>
<td>$v_B \leftarrow v_A - C$</td>
<td>if $C == 10,000$, $\tau(v_B) \leftarrow \tau(v_A)$</td>
</tr>
<tr>
<td>irem</td>
<td>$v_B \leftarrow v_A % C$</td>
<td>if $C == 100,000$, $\tau(v_B) \leftarrow \tau(v_A)$</td>
</tr>
<tr>
<td>concat</td>
<td>$v_C \leftarrow v_A.concat(v_B)$</td>
<td>$\tau(v_C) \leftarrow \tau(v_A)$</td>
</tr>
</tbody>
</table>

the element for the app from a built-in array or ArrayList; the other one is $(uid\%100,000)$, which is used to calculate the index into a SparseArray. Two magic numbers appear in the formulas and are worth interpretation. 10,000 means FIRST_APPLICATION_UID (Line 26 in Figure 4.1), indicating the smallest UID an user app can have, while 100,000 means PER_USER_RANGE (Line 27 in Figure 4.1), indicating the largest UID plus one. For example, as shown in Figure 4.1, the function getUserLpr (Line 29) utilizes the first formula to calculate the index into the ArrayList Settings.mUserIds (Lines 31 and 32).

Second, for hash-table-based variables, no matter it is hash table or a set, the package name (or the package name concatenated with a component name) is used as the key to access elements. Figure 4.3 shows an example of retrieving information from a hash-table-based variable.

While there are a large variety of data structures in the framework, our investigation shows that they commonly follow the two fixed access patterns to retrieve the app-specific information when servicing a system service call.

4.4.3 Slim Tainting

We thus propose slim taint analysis that tracks and recognizes the characteristic access patterns above on the fly during path exploration and sets variables as symbolic inputs when app-specific information is accessed.

Next, we elaborate slim tainting. Similar to other tainting techniques, it consists of taint sources, taint propagation logic, and taint sinks. (1) The return values
of getCallingUID and getPackageName are set as taint sources; they are unique identifications of an app. (2) The taint propagation logic as shown in Table 4.1 is very simple, involving only two instructions and one string concatenation function. They are all derived from the access patterns described in Section 4.4.2. (3) Finally, the taint sinks include the get functions of the collection data structures as well as bytecode instructions for loading elements from built-in arrays, such as iaload (loads from an array of integers) and aaload (loads from an array of references); they check whether the index or key is tainted, and if so, the target element is flagged as a symbolic input.

Example: Let us take the code in Figure 4.1 as an example to illustrate how the slim taint analysis works. First, due to the call to getCallingUID (Line 3), its return value uid obtains the taint. Second, the taint propagates along Lines 31 and 32 according to the taint propagation logic. Finally, at Line 33, the get function works as a sink to set the element (and only this element) accessed through the tainted index as a symbolic input.

Slim tainting comprises very specific taint sources and a simple but precise taint propagation logic; it thus avoids the overtainting and undertainting issues. Section 4.6 includes its implementation details. It is worth mentioning that the implementation intercepts some specific function calls and changes the interpretation of several bytecode instructions; it does not need to change a single line of the source code of Android Framework and does not need any code annotation. Therefore, it is precise and automatic. When the Android system evolves, new access patterns may be used. In that case, we need to update some details of slim tainting in terms of, e.g., taint sources and propagation logic. But the idea of capturing access patterns should still be applicable.
4.5 Execution Context Migration

The decoupled architectural design requires that the execution context due to the concrete execution be mitigated from the Android system to the analyzer. Note that, in the execution context, the program counter, the register file, and the stack all obtain their fresh content when symbolic execution starts at the analyzer; only the heap in the execution context, which is a collection of classes and objects, needs to be migrated. The heap memory image in the execution context is called a snapshot for short. Three problems have to be resolved for migrating the heap information captured in a snapshot: (1) how to obtain the semantics of the bits and bytes in a snapshot? (2) how to conduct the migration during symbolic execution? and (3) how to bootstrap the migration?

4.5.1 Snapshot Parsing and Information Query

A heap snapshot is nothing but an array of bits. However, it would not work if we simply copy the array of bits to the JVM instance for symbolic execution, because the ART process in Android and the JVM instance for symbolic execution differ significantly in terms of the low-level representation of classes and objects. E.g., in our implementation, each object in our symbolic executor needs extra space for recording the taint and the symbolic expression; plus, its heap memory management is different from the one used in Android. On the other hand, given an object, both ART and our JVM should agree on the number of the contained fields, according to the class definition file, and their values. Therefore, given an object, our migration is not to copy its bits but to copy the values of all its fields.

Thus, the parser analyzes the snapshot to obtain all the active objects (and classes) and, for each object (and class), records the values of its fields. The information is organized in a two-tier data structure: the first tier maps an object (or class) address to a second-tier data structure instance, which maps field names
of an object (or class) or element indexes of an array to their values.

After the snapshot is parsed and its information is stored, the execution context query server (in Figure 4.2) is used to service requests from the symbolic executor by returning the information about objects, classes and arrays. Multiple query interfaces are provided: given a reference value, the type of the corresponding object can be queried; given a reference value of an object and the name of one of its fields, the field value can be queried. Snapshot parsing and information query provide the foundation for heap information migration.

4.5.2 Migration Algorithm

Given an object in the concrete execution world, for fields of primitive types we can simply copy the field values after allocating the space from the symbolic executor for the object. But what about fields of the reference type? A deep copy is too inefficient while a simple shallow copy of the reference value will not work as the reference value only indicates the object location in the concrete execution world (where the snapshot has been captured). We choose to enforce a variant of the simple shallow copy: when an object is migrated, we simply copy all the field values, but for each reference-typed field, we mark that it indicates a reference value in the concrete execution world (a boolean attribute snapshotRef is associated with each reference-typed field to indicate whether the filed value is a location in the concrete or symbolic execution world); later, when one of such reference-typed fields is used to access its target object, the target object is either migrated or, if it has been migrated, the field value is updated with the reference value in the symbolic execution world.

Therefore, a hash table, conc2Sym, is maintained to map reference values in the concrete execution world to ones in the symbolic execution world. Every time an object \( o \) is migrated, a new pair \( \langle r_c, r_s \rangle \) is added to the hash table, where \( r_c \) is the reference value of \( o \) in the concrete execution world and \( r_s \) symbolic. The
Table 4.2: Bytecode instructions (and function) used for migrating heap information.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getfield</td>
<td>objRef → value</td>
<td>get a field value of an object</td>
</tr>
<tr>
<td>getstatic</td>
<td>→value</td>
<td>get a static field value of a class</td>
</tr>
<tr>
<td>aaload</td>
<td>arrayRef, index → value</td>
<td>load onto the stack a reference from an array</td>
</tr>
<tr>
<td>initClass</td>
<td>N/A</td>
<td>invoked for class initialization</td>
</tr>
</tbody>
</table>

hash table is maintained for two purposes. First, it prevents duplicate migration of an object; that is, an object pointed to by \( r_c \) is migrated only if \( r_c \) is not found in the hash table. Second, the hash table is used to translate reference values in the concrete execution world, if they exist in the hash table, to ones in the symbolic execution world.

The hash table \texttt{conc2Sym} is handled as part of the process state, and gets stored and restored as the path exploration advances and backtracks, respectively; this way, the migration status keeps consistent during path exploration.

The heap execution context migration algorithm, which is built into the symbolic execution engine, is implemented by overriding a couple of bytecode instructions, shown in Table 4.2, which includes, for each instruction, the effect that the instruction has on the operand stack and the instruction description. Algorithm 3 shows the main migration procedures.

4.5.2.1 Migrating Objects

The instruction \texttt{getfield} is used to access non-static fields in an object. Given a reference to an object (this object must have been migrated; Section 4.5.3 covers the reason) on the stack (Line 2), the instruction \texttt{getfield} pushes a field value of the object onto the stack. Below we describe how an object pointed to by a field when this field is accessed through \texttt{getfield}.

If the field’s \texttt{snapshotRef} attribute is \texttt{false} (Line 5), which means that either
Algorithm 3 Migration of heap information.

1: function `getfield(index)`
2:    objRef = peekStackTop()
3:    fdInfo = getFdInfo(index)  \> Class-specific info.
4:    fd = getFd(objRef, fdInfo)  \> objRef-specific info.
5:    if !fd.getSnapshotRefAttribute() then
6:        return super.getfield(index)
7:    end if
8:    concRef = fd.getValue()
9:    symRef = conc2Sym.get(concRef)
10:   if symRef == NULL then
11:       fdType = fdInfo.getFdType()
12:       if fdType == strRef then
13:          str = snapshot.getStr(concRef)
14:          symRef = searchConstantPool(str);
15:          if symRef == NULL then
16:              symRef = newString(str);
17:          end if
18:       else if fdType == arrayRef then
19:          entryType = fdType.getEntryType()
20:          len = snapshot.getArrayLen(concRef)
21:          symRef = newArray(entryType, len)
22:          snapshot.copyEntries(symRef, concRef)
23:       else
24:          symRef = newObj(fdType)
25:          snapshot.copyFields(symRef, concRef)
26:       end if
27:       conc2Sym.addPair(concRef, symRef)
28:    end if
29:    fd.setValue(symRef)
30:    fd.setSnapshotRefAttribute(false)
31:    return super.getfield(index)
32: end function

33: function `initClass(classInfo)`
34:    if snapshot.isInitialized(classInfo) then
35:        snapshot.copyStaticFields(classInfo)
36:    else
37:        super.initClass(classInfo)
38:        handleBootstrapField(classInfo)
39:    end if
40: end function
it is a primitive-typed field or it has a reference value in the symbolic execution world, the instruction’s interpretation is not changed (Line 6); i.e., the field value is simply pushed onto the stack. If \texttt{snapshotRef} is \texttt{true} and the field value \texttt{concRef} is not found in \texttt{conc2Sym} (Line 10), the object should be migrated (Lines 11–26); after migration, the pair \langle \texttt{concRef}, \texttt{symRef} \rangle is added to \texttt{conc2Sym} (Line 27).

How to migrate an object is determined by its type (Line 11). (Recall that, given the reference value, which is the value of the field being accessed, the execution context query server can locate and return the target object information, i.e., its type and contained field values, from the concrete execution world.) (1) If the object is a string, the algorithm first searches for a string that has the same value within the runtime constant pool in the VM for symbolic execution. If not found, a new string with the same value is created in the symbolic world (Lines 12–17). (2) If the object is an array, an array is allocated and all the elements are copied to the new array (Lines 18–22). This algorithm performs a shallow copy. Thus, for a multi-dimensional array, e.g., \(A[5][10]\), only the five elements in the top-level array are copied at this moment. Later, when any of the five elements is accessed, the instruction \texttt{aaload} has to be invoked, which is the reason the interpretation of \texttt{aaload} (not shown in Algorithm 3) is also overridden, i.e., to migrate second-level arrays. Due to the shallow copy, an array object is not copied until a reference to the object is accessed. (3) A reference to an ordinary object is handled by allocating a new object and copying all its fields (Lines 23–25).

While non-static fields are accessed through \texttt{getfield}, access to static fields is through \texttt{getstatic}. Thus, to migrate objects pointed to by static fields, the interpretation of \texttt{getstatic} has to be overridden, and the interpretation is similar to that of \texttt{getfield} and is thus omitted.
Figure 4.4: Example of a test driver.

```java
public TestDriver() {
    @fromSnapshot
    private static com.android.server.LocationManagerService mService;
    public static void main() {
        // The parameters are configured as symbolic inputs, so their values do not matter
        mService.getProviders(null, false);
    }
}
```

4.5.2.2 Migrating Classes

When an operation (e.g., an object of a class is created or a class’s static fields are accessed for the first time) triggers initialization of a class during symbolic execution, `initClass` is invoked by the underlying VM for symbolic execution automatically. For classes that have been initialized during concrete execution, the symbolic executor has to make sure that they are migrated instead of being initialized, considering that the static fields have obtained their values during concrete execution. Thus, when `initClass` is invoked, the symbolic executor first checks whether the class has been initialized in the concrete execution world; if so, the enclosed static fields in the class are copied from the snapshot to the symbolic execution world (Line 36). In particular, when an object of some class is created in the symbolic world for the first time due to migration (Line 24), it triggers the invocation of `initClass` first, which migrates the class.

4.5.3 Bootstrapping

An important invariant kept during migration is that, when a field of an object `o` (resp. an element of an array `A`) is accessed, `o` (resp. `A`) must have been migrated in the symbolic execution world. Assume `f` is the field whose access triggers the migration of the first object; a natural question is “where does `f` reside?” The answer is that `f`, called the `bootstrap` field, resides in the test driver class, and
it is a reference to the system service class that contains the entrypoint method. Listing 4.4 shows an example of a test driver. A custom annotation fromSnapshot is used to specify the bootstrap field, which is recognized and handled by the migration algorithm; specifically, when the TestDriver class is initialized, it sets the bootstrap field value to the reference value of the system service object in the concrete execution world (note that all the system service classes adopt the singleton design pattern, so there is no ambiguity when specifying the reference value).

Example. In Listing 4.4, when the TestDriver class is initialized, the migration algorithm sets the value of the bootstrap field to the reference to the Location Manager service object in the snapshot; as a result, when the bootstrap field is accessed, the service object is migrated correctly. The migration of classes and objects form a migration tree, which grows as classes and objects get migrated, rooted at the class and object corresponding to the bootstrap field type. We use the code in Listing 4.4 as an example to partially illustrate how a migration tree grows, as shown in Figure 4.5, where the root node is the class and object for LocationManagerService. Note that the step (5), getDefault is invoked due to the call to getProviders.

4.5.4 Migration Tree

The migration of classes and objects forms a migration tree, which grows as new classes and objects are migrated, rooted at the class and object corresponding to the bootstrap field type. We use the test driver in Figure 4.4 as an example to illustrate how the migration tree is built, as shown in Figure 4.5, where the root node is the class and object for LocationManagerService. The migration of a class also triggers the migration of all its super classes, which is not shown in Figure 4.5 for simplicity.

Part of the resulted migration tree is showed in Figure 4.6. It also shows the
Access to the bootstrap field TestDriver.mService first triggers the migration of the LocationManagerService class, which is performed in initClass().

It also triggers the migration of the LocationManagerService object pointed, which is performed in getstatic().

Access to the mService.mContext field triggers the migration of the ContextImpl class, which is performed in initClass().

Next, the object pointed to by mContext gets migrated; the migration is performed in getfield().

Invocation of the static method getDefault() of the ActivityManagerNative class triggers the migration of this class; the migration is performed in initClass().

**Figure 4.5:** An example of migrating the heap. Grey rectangles and white ones denote classes and objects, respectively. For each class and object, \(<\text{conRef}, \text{symRef}>\) denotes the mapping between the reference value in the concrete execution world and that in the symbolic world. The migration of a class also triggers the migration of all its super classes, which are not shown for simplicity.
Figure 4.6: Part of a migration tree with some classes omitted. Different arrows are used to denote different instructions that trigger the migration. Rectangles with diagonal stripes denote objects that are identified as symbolic inputs.
identified symbolic inputs and how the symbolic input attribute propagates. After the element with index 54 in the \texttt{mUserIds} is identified as a symbolic input through tainting, the symbolic input attribute is propagated to other variables pointed to by the element.

### 4.6 Implementation Details

We built the symbolic executor on Symbolic PathFinder (SPF) \cite{41}, a symbolic execution framework on top of Java PathFinder (JPF) \cite{109}. It runs outside the Android system, and does not rely on the Android internals, achieving the goal of a decoupled architecture. We added 6,285 lines of code for implementing CENTAUR through extending SPF. Significant effort has been saved by building upon SPF, which is made possible thanks to the decoupled design.

#### 4.6.1 Configuration

In addition to specifying the entrypoint system interface and the test driver, we need to provide the Android Framework code and the heap memory snapshot for symbolic execution.

#### 4.6.2 Classpath

The Java source code in Android is compiled into \texttt{.jar} files, which comprise standard \texttt{.class} files, and the symbolic executor is built to analyze Java bytecode in such \texttt{.class} files. The classpath below shows the classes analyzed by the symbolic executor.

```
classpath=test_driver_dir;\ 
  services_intermediates/classes-full-debug.jar;\ 
  framework_intermediates/classes-full-debug.jar;\ 
  core-libart_intermediates/classes-full-debug.jar
```
The first line specifies the directory containing the test driver, the next two lines specify the Android Framework code, and the last line the core libraries of ART, such as utility, io, and math libraries. Several classes (e.g., `java.lang.Class`, `Thread`, `StackTraceElement`) are modeled by the symbolic executor, but the folder `core-libart` contains the Android version of these classes; hence, these specific classes have to be excluded from `core-libart` to avoid failures due to duplication.

### 4.6.3 Slim Tainting

Slim tainting is built into the symbolic executor by modifying the interpretation of instructions, such as `isub` (subtraction), `irem` (modular), and `aload`, and intercepting functions, such as `getPackageName`, `getCallingUID`, `String.concat` (string concatenation), and various `get` functions of collection data structures. Centaur adds one attribute indicating the taint and another indicating the symbolic input property for each field, array element, and call-stack variable; Thanks to JPF’s supports for interception of function calls and adding field attributes without modifying the framework source code, there is no maintenance effort needed when there are new versions of Android released, as long as the app-specific variable access patterns (Section 4.4.2) are not changed.

### 4.6.4 Capturing and Parsing Snapshots

After a heap memory snapshot of an Android Framework process is captured (using the `dumpheap` utility), it is first converted to a standard `.hprof` file using the `hprof-conv` utility in the Android SDK. The standard `.hprof` file format opens up the possibility of parsing the snapshot using many existing tools. In our case, the file is then parsed to extract the list of classes and objects stored in the `.hprof` file using a hprof file parser [112]. The extracted information is then organized into the memory space of the execution context query server.
4.6.5 Handling Special calls

Service calls are frequently used among services. While inter-process service calls are made through the intricate Binder IPC mechanism, intra-process calls are actually ordinary method calls. Figure 4.7 shows an example, where the Location Manager service invokes `getProfiles` exposed by the User Manager service; both services belong to the System Server process. The call at Line 4 leads to a service call at Line 11, which is a virtual function call, whose dispatch relies on the runtime type of the object pointed to by `UserManager.mService`. Previous research relies on expert knowledge and specifies the dispatch targets manually to facilitate further analysis [15, 97, 99], while CENTAUR makes use of the runtime type information provided by the execution context, and thus the call is handled as an ordinary virtual function call without requiring expert knowledge or manual effort. This is a concrete example illustrating the advantage of combining concrete and symbolic executions.
4.6.6 Dealing with Handler and State Machine Calls

Two other important IPC mechanisms that are widely used by system services are Message Handler and State Machine calls. A handler sends and processes messages associated with a thread’s message queue [113]. When a new handler is created, it is bound to the message queue of the thread that creates it. From that point on, it will deliver messages to that message queue and execute them as they come out of the message queue. To deal with Message Handler calls, when sendMessage(message) is invoked, the invocation is replaced by that of the corresponding Handler’s handleMessage(message). The symbolic executor interposes the invokevirtual instruction and enforces the replacement on the fly.

A State Machine can also send and process messages, which has states arranged hierarchically. A state is an instance of the State class, which implements processMessage for handling messages. A State Machine sends a message by invoking sendMessage. When a State Machine receives a message, the current state’s processMessage is invoked. Therefore, a key step is to identify the current state. To do it, the field mSmHandler in the State Machine object, which is a reference to the state machine handler, is retrieved (note that when the State Machine object is migrated, all its fields are copied), and then used to migrate the state machine handler object. Next, two fields in the handler object, mStateStack and mStateStackTopIndex, are used to identify the current state (= mStateStack[mStateStackTopIndex].state). To handle messages sent by a State Machine in the symbolic executor, the invocation of mSmHandler.sendMessage(message) is replaced by that of the current state’s processMessage(message). This way, we connect the senders and receivers for messages sent through State Machine.
4.6.7 Handling Calls to Native Code

Part of Android Framework is implemented in native code, which is invoked through the Java Native Interface (JNI) mechanism. Different ways are adopted to handle JNI calls during symbolic execution. First, methods that return the calling UID (\texttt{getCallingUid()}) and the package name of the client app (\texttt{getPackageName()}) are modeled to return the corresponding information for the skeleton app constantly, and the return values are set to be taint sources as aforementioned. Second, the return values of other native methods that return app-specific information of the skeleton app are specified as symbolic inputs. For example, many native methods declared in the package \texttt{android.content.res} access application resources. Third, for native methods that do not have return values, they are ignored; ignoring calls to external code has been used in many symbolic execution techniques [38, 41].

Finally, other calls to native methods are delegated back to Android as remote procedure calls. The RPC client in the symbolic executor is built by extending jpf-nhandler [114]. While jpf-nhandler delegates native calls to a host JVM, this client delegates them to an app running as the RPC server in a remote Android system (Figure 4.2), which issues delegated native calls using reflection on demand. The \textit{GSON} library [115] is used for marshalling (and unmarshalling) method parameters and return values, which are transmitted between the RPC server and the client via socket. Note that though an Android system is used to execute native calls, the symbolic executor is decoupled from it using the RPC mechanism.

4.7 Evaluation

We compare \textsc{Centaur} against under-constrained symbolic execution (UCSE) in Section 4.7.1. Both can start symbolic execution from system interface methods to reach the code deep in the program, but \textsc{Centaur} makes use of the execution context provided by concrete execution. Ideally, we should also compare the
PC2SE scheme used in Centaur against symbolic execution that starts from the main entry of Android Framework, that is, `SystemServer.main`, but note that our symbolic executor runs outside Android, while at the initialization phase the System Server heavily relies on the Android environment, such as the file systems and other supporting processes, to finish its initialization, meaning that it is unlikely to initialize the System Server process outside the Android environment. We thus compare PC2SE with UCSE only.

Centaur provides strong support for vulnerability discovery and exploit generation. To demonstrate how Centaur can be applied to assisting vulnerability discovery, we investigate two distinct types of recently uncovered attacks that exploit Android Framework vulnerabilities. The investigation in Sections 4.7.2 and 4.7.3 shows how zero-day vulnerability instances can be discovered through the application of Centaur.

Finally, the reliability of the approach is investigated. We present exploit generation experiments based on snapshots captured at different times, and analyze the consistency of the results in Section 4.7.4.

The experiments were performed on a machine with an Intel Core i7 4.0Ghz Quad Core processor and 32GB RAM running Linux kernel 3.13. Exploits were generated on Android Framework 5.0 and verified using different versions of Android systems.

We use a skeleton app to act as the malicious app; it contains all the aspects of a regular app, including the manifest file, activities, and services, but does not implement any essential functionality; in particular, the skeleton app used in our experiments borrows the manifest file from the Android developer website, which has “every element that it can contain” [116]. In practice, the analyst can choose any app as the malicious app.
4.7.1 Comparison with Under-constrained Symbolic Execution (UCSE)

The first issue of applying UCSE to symbolic execution of Android Framework is that virtual function calls are frequently used in the framework code, but the runtime types of the receiver objects are unknown. UCSE constructs the receiver objects based on the type hierarchy or relying on manual specifications, which either explores spurious paths or requires much manual effort.

The second issues is that input variables which are treated as concrete inputs in Centaur are treated as symbolic inputs in UCSE. UCSE handles such symbolic inputs using lazy initialization, which causes the following problems: (1) loops that iterate through collection data structures are unbounded, and (2) the generated concrete values may be unrealistic.

We tried to perform UCSE of Android Framework using Java PathFinder, which kept crashing when it was applied directly. We spent a lot of time and tedious effort modifying the framework code (e.g., adding the type information about objects pointed to by references to assist dynamic dispatching) to make the symbolic execution possible. We thus only modified the code with respect to the getProviders API and the startActivity. UCSE spent 138m when analyzing getProviders and ran out of memory in the case of startActivity, while Centaur finished them within 26s and 42m 37s, respectively.

Therefore, path exploration without precise information of the execution context causes many problems, such as requiring tedious manual annotation effort and exploring spurious paths. CENTAUR resolves the problems by migrating the execution context from the concrete execution world to symbolic execution.
4.7.2 Investigating Inconsistent Security Policy Enforcement (ISPE)

**Background.** Android Framework utilizes a permission-based security model, which provides controlled access to various system resources. However, a sensitive operation may be reached from different paths, which may enforce security checks inconsistently. As a result, an attacker with insufficient privilege may perform sensitive operations by taking paths that lack security checks. Recently, static analysis combined with manual code inspection has been applied to finding such inconsistent security enforcement cases in Android Framework [15]. The system, called *Kratos*, first builds a call graph based on the Android Framework code. With the call graph, it finds all the execution paths that can reach sensitive operations. Kratos then compares the paths pairwise to identify paths that reach the same sensitive operation with inconsistent security checks enforced, and reports them as suspected ISPE vulnerabilities, in that they violate the security property that *all paths should have consistent permissions for reaching a given sensitive operation.*

**Combined approach for bug finding.** Static analysis based on the reachability analysis for finding ISPE bugs may report false positives, as some paths are infeasible in real executions. Currently, manual effort is used to scrutinize the code along each reported path, which is laborious and tedious; moreover, it is difficult to verify the correctness of the manual inspection results.

We propose to combine static analysis and symbolic execution to find ISPE bugs. For each suspected vulnerability reported by static analysis, *Centaur* (1) finds all feasible paths that reach the sensitive operation, (2) gives permissions needed for each feasible path (the needed permissions are included in each path condition), (3) verifies permission consistency among the feasible paths, and (4) generates inputs that exercise the feasible paths to verify suspected vulnerabilities. It thus demonstrates the uses of *Centaur* comprehensively. All the steps have been performed automatically, in contrast with previous work that relies on tedious
and error-prone manual inspection. Plus, zero false positives are guaranteed as all suspected vulnerabilities are validated by the runtime log.

**Result summary.** Table 4.3 summarizes the experiment results (the vulnerability shown in the last row is discussed in Section 4.7.3). For each vulnerability, the table lists the vulnerability description, entrypoint(s), the min/max number of migrated classes among different paths, the min/max number of migrated objects among different paths, the number of sets of concrete values generated (“—” means it can be exploited unconditionally; note that we generate one set of concrete values for each unique path explored), the number of sets that can be used to generate exploits, the symbolic execution time, and the code coverage.

Given an entrypoint method, there may be multiple paths that reach the sensitive operation, and the classes and objects involved in the paths may vary, as illustrated by the min/max number of migrated classes and objects. Note that when migrating a class, all its super classes are also migrated, which is the reason the number of migrated classes is greater than that of objects. In the majority of the cases, the symbolic execution of an entrypoint method is finished within less one minute. Note that in some cases we have a relatively low code coverage, e.g., in `WSI.addOrUpdateNetwork`; it is mainly because branches that rely on non-app-specific variables are not iterated, as we consider those variables as concrete inputs. We are only interested in branches that can be affected by the variables derived from the malicious app.

**New findings.** It is notable that some of our results are inconsistent with those of Kratos. First, for the fifth vulnerability in Table 4.3, Kratos reports that it does not exist in Android Framework 5.0, while CENTAUR shows that it still exist (i.e., different permissions are required by the two system interface methods for reaching the sensitive resource) and the result is verified by the log. Second, for the sixth vulnerability in Table 4.3, Kratos reports only one permission `CONNECTIVITY_INTERNAL` for invoking `NsdService.setEnabled`, while CENTAUR reports two permissions,
Table 4.3: List of vulnerabilities and analysis statistics. (*LMS, TSI, PIM, WMS, AMS, WSI, NS, and ASS represent LocationManagerService, TelecomServiceImpl, PhoneInterfaceManager, WindowManagerService, ActivityManagerService, WifiServiceImpl, NsdService, and ActivityStackSupervisor, respectively.*)

<table>
<thead>
<tr>
<th>No.</th>
<th>Vulnerability description</th>
<th>Entrypoint(s)</th>
<th># of migrated classes</th>
<th># of migrated objects</th>
<th># of all sets</th>
<th># of legal sets</th>
<th>Analysis time</th>
<th>Code coverage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>min</td>
<td>max</td>
<td>min</td>
<td>max</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Access</td>
<td>LMS.getAllProviders()</td>
<td>55</td>
<td>55</td>
<td>4</td>
<td>4</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LMS.getProviders(Criteria,boolean)</td>
<td>77</td>
<td>93</td>
<td>14</td>
<td>42</td>
<td>66</td>
<td>66</td>
</tr>
<tr>
<td>2</td>
<td>Read</td>
<td>TSI.getCallState()</td>
<td>48</td>
<td>48</td>
<td>3</td>
<td>3</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TSI.isInCall()</td>
<td>62</td>
<td>69</td>
<td>17</td>
<td>20</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TSI.isRinging()</td>
<td>60</td>
<td>65</td>
<td>16</td>
<td>18</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>End</td>
<td>TSI.endCall()</td>
<td>81</td>
<td>83</td>
<td>21</td>
<td>24</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PIM.endCall()</td>
<td>80</td>
<td>85</td>
<td>23</td>
<td>26</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Close</td>
<td>WMS.closeSystemDialogs(String)</td>
<td>57</td>
<td>57</td>
<td>6</td>
<td>6</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>system dialogs</td>
<td>AMS.closeSystemDialogs(String)</td>
<td>63</td>
<td>67</td>
<td>11</td>
<td>15</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>Set up HTTP proxy working in PAC mode</td>
<td>WSI.addOrUpdateNetwork()</td>
<td>67</td>
<td>122</td>
<td>23</td>
<td>52</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WSI.getWifiServiceMessenger()</td>
<td>65</td>
<td>84</td>
<td>21</td>
<td>24</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Enable/Disable mDNS daemon with insuf. privilege</td>
<td>NS.setEnabled(boolean)</td>
<td>75</td>
<td>114</td>
<td>28</td>
<td>53</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NS.getMessenger()</td>
<td>80</td>
<td>81</td>
<td>11</td>
<td>14</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Task hijacking</td>
<td>ASS.startActivityUncheckedLocked()</td>
<td>324</td>
<td>387</td>
<td>136</td>
<td>182</td>
<td>2,020</td>
<td>810</td>
</tr>
</tbody>
</table>
CONNECTIVITY_INTERNAL and WRITE_SETTINGS. The more thorough and accurate results demonstrate the advantages of the hybrid approach.

**A detailed example.** As an example, we describe in detail how the combined approach was applied to finding the first vulnerability in Table 4.3. (1) First, the static analysis based on path reachability and pairwise path comparison finds that both `getProviders(Criteria, boolean)` and `getAllProviders()` (in the `LocationManagerService` class) have paths reaching the same sensitive operation that returns the names of the installed GPS providers, and the two paths can be executed with inconsistent permissions; thus, it is a suspected vulnerability. (2) Next, **Centaur** is applied to check automatically whether there exist paths that can reach the sensitive operation from the two service interface methods. Specifically, after the Android system is initialized and the skeleton app is launched, a heap memory snapshot of the System Server process is captured, and provides execution context for symbolic execution, and symbolic execution starts from the two service interface methods respectively.

Compared to previous work that relies on enormous and error-prone manual inspection, the combined approach of call graph reachability analysis and symbolic execution eliminates the need for manual work and guarantees zero false positives. It is potential to apply this approach to finding other types of vulnerabilities in Android Framework.

### 4.7.3 Investigating Task Hijacking Attacks

**Background.** The Activity Manager Service allows activities of different apps to reside in the same *task*, which is a collection of activities that users interact with when performing a certain job. The activities in a given task are arranged in a *back stack*, pushed in the order they were opened; users can navigate back using the “Back” button. This feature can be exploited by a malicious app if its activities are manipulated to reside side by side with the victim apps in the same task and hijack
the user sessions of the victim apps. This is a design flaw rather than a program bug, but can be exploited to implement UI spoofing, denial-of-service, and user monitoring attacks [14]. For example, a malicious app may start a malicious activity that impersonates the victim activity, and the UI spoofing attack succeeds if the fake activity resides in the same back stack as the victim activity, and the user may mistake the fake malicious activity for the victim one. This case illustrates unique characteristics of exploits that take advantage of Android Framework vulnerabilities: the malicious “input” is not some single input (a command parameter, a network packet, etc.) but a separate app.

Vulnerability discovery. We use the \texttt{EditEventActivity} activity of the \texttt{calendar} app as an example victim activity. In the skeleton app, the main activity of the skeleton app starts the malicious activity, denoted by \texttt{M}. The goal of the attack is that \texttt{M}, when it is started, will reside in the same task as the victim activity. A bug is identified if such attacks against the victim activity is feasible. We capture the heap memory snapshot when the victim app and the skeleton app are started and the main activity of the skeleton app is to start the malicious activity.

While the method for starting an activity is \texttt{startActivity}, the task selection is done in \texttt{startActivityUncheckedLocked}, which is invoked by \texttt{startActivity}. We thus performed the symbolic execution of \texttt{startActivityUncheckedLocked} to simplify the path exploration; it has eight parameters as shown in Figure 4.8. The first parameter \texttt{r} is an \texttt{ActivityRecord} instance storing the information of \texttt{M}, while the second storing that of the caller activity. The description of other parameters is omitted. They are set to symbolic inputs. The constraint indicating

```java
final int startActivityUncheckedLocked(ActivityRecord r, ActivityRecord sourceRecord, IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor, int startFlags, boolean doResume, Bundle options, TaskRecord inTask) {...}
```
that the task selected for $M$ is exactly the one hosting the victim activity is added to each of the path conditions when it is to be resolved; that is, \( m.task.taskId == v.task.taskId \), where $m$ and $v$ represent the activityRecords of the malicious activity and the victim activity, respectively. A feasible path is found if the path condition is resolvable.

Exploit generation. The symbolic execution generated 2,020 sets of concrete input values (each set corresponds to a unique path), among which some contain illegal concrete values, e.g., due to requiring the malicious activity’s package and activity names to be equal to those of the victim activity. Simple scripts were written to filter out illegal concrete values (1,210 sets totally). Figure 4.9 shows an example of the rest 810 sets of legal concrete values. In this example, $r.intent.mFlags$ and $options$ (whose type is Bundle) guide how to set the two
Table 4.4: Effectiveness of the generated exploits.

<table>
<thead>
<tr>
<th>Android version</th>
<th>4.0</th>
<th>4.1</th>
<th>4.2</th>
<th>4.3</th>
<th>4.4</th>
<th>5.0</th>
</tr>
</thead>
<tbody>
<tr>
<td># of effective exploits</td>
<td>434</td>
<td>674</td>
<td>674</td>
<td>674</td>
<td>702</td>
<td>810</td>
</tr>
</tbody>
</table>

parameters of `startActivity(Intent, Bundle)`, respectively, and others instruct how to configure the malicious activity; for example, `r.launchMode` is mapped to the `android:launchMode` in the manifest file. Figure 4.10 shows the exploit generated according to the set of concrete values, and it has verified that the exploit can be used to launch task hijacking successfully.

We then examined whether the exploits generated on Android 5.0 were effective on other versions of Android systems. Table 4.4 lists the results, which show that the effectiveness of the exploits are affected by the versions of Android systems. Further investigation has revealed that the difference is mainly caused by code changes. For example, the new exploiting condition `FLAG_ACTIVITY_NEW_DOCUMENT` is not introduced until Android 5.0 (discussed below); the API `startActivity(Intent, Bundle)` is not included in version 4.0, and thus only exploits with `options == null` can be used for invoking `startActivity(Intent)`.

Newly discovered exploiting condition. The path conditions generated from symbolic execution reveal an extra exploiting condition (requiring a specific bit in the bitflags `r.intent.mFlags` be 0) that was not reported in previous work [14]. Compared to previous work that relies on ad hoc manual effort for discovering the exploiting conditions, CENTAUR finds them in a systematic and automatic way.

4.7.4 Consistency of Exploits Generated with Different Snapshots

We then investigated whether snapshots captured at different times affected exploit generation. After the system was initialized, 20 snapshots were captured at intervals of 5 minutes on Android 5.0 with random user interactions during the intervals. For
each vulnerability listed in Table 4.3, symbolic execution was performed with each of the 20 snapshots providing the execution context. The results show that, for each vulnerability, the same sets of path conditions were generated with different snapshots, which means that the resulting exploits with the different snapshots are consistent.

There are several reasons that explain the consistency of exploits. First, if a malicious app does not rely on other apps to exploit a vulnerability (e.g., inconsistent security policy enforcement), access control is enforced in Android Framework to make sure the information of other apps is not accessed. Thus, the configurations and statuses of other apps do not affect the path exploration. On the other hand, for exploits that rely on the statuses of other apps (e.g., the victim app in task hijacking attacks), the path exploration may depend on the statuses of one or more apps; hence, during symbolic execution, reasonable setting up is established consistently; for example, the victim activity should already be started in the task hijacking case prior to capturing snapshots. The results show that an attack succeeds as long as the same statuses recur.

Finally, the values of non-app-specific variables do not affect path exploration, at least, in our cases. For example, in the case of inconsistent security policy enforcement vulnerabilities due to accessing the names of installed providers, the path exploration does not depend on the concrete values of the related non-app-specific variable (i.e., LocationManagerService.mProviders), although different provider names may be returned by the service calls when different providers are installed.

4.8 Summary

We have introduced the first system, called CENTAUR, for symbolic execution of Android Framework. To avoid state space explosion due to the complex initializa-
tion, the Phased Concrete-to-Symbolic Execution is proposed that runs concrete execution for the initialization phase, providing execution context to symbolic execution. Among the large number of variables in the execution context, slim tainting tracks characteristic access patterns to identify variables derived from the malicious apps as symbolic inputs. In order to decouple the implementation of CENTAUR from Android, the execution context provided by concrete execution is migrated from an Android ART process to a Java VM. We have implemented the system and evaluated it. The evaluation shows that CENTAUR is very effective in both vulnerability discovery and exploit generation. Given that symbolic execution has proven to be a very useful technique, we plan to apply CENTAUR to other purposes in future work, such as automatic API specification generation, fine-grained malware analysis, and testing.
Conclusion

Recent years have witnessed the rapid growth of mobile devices. The increasing popularity has drawn attention from malware writers, and the number of harmful apps is increasing at an alarming rate. Besides, a handful of serious system vulnerabilities have been disclosed every year, which cause serious harms to user security and privacy. Tremendous efforts from both industry and academia have been made to mitigate the threats to secure Android. This dissertation presents techniques and approaches to mitigate the following two severe threats: Android App repackaging and Android Framework vulnerabilities, to enhance Android application and system security.

First, we propose a repackaging-proofing technique, called Stochastic Stealthy Network (SSN), which prevents repackaged apps from working on user devices. To the best of our knowledge, there is no study that investigates repackaging-proofing to prevent repackaged apps from working on user devices. We conduct a preliminary first-step study on the problem and introduce SSN that provides a reliable and stealthy protection for Android apps. We have developed a prototype. Our experimental results show that SSN is effective and efficient.

Second, we have introduced the first system, called CENTAUR, for symbolic execution of Android Framework. To avoid state space explosion due to the complex
initialization, the Phased Concrete-to-Symbolic Execution is proposed that runs concrete execution for the initialization phase, providing execution context to symbolic execution. Among the large number of variables in execution context, slim tainting tracks characteristic access patterns to identify variables derived from the malicious apps as symbolic inputs. In order to decouple the implementation of Centaur from Android, the execution context provided by concrete execution is migrated from an Android ART process to a Java VM. We have implemented the system and evaluated it. The evaluation shows that CENTAUR is very effective in both vulnerability discovery and exploit generation.

Given that symbolic execution has proven to be a very useful technique, we plan to apply CENTAUR to other purposes in future work, such as automatic API specification generation, fine-grained malware analysis, and testing.
Bibliography


[67] DESHOTELS, L. (2014) “Inaudible Sound as a Covert Channel in Mobile Devices,” in USENIX WOOT.


Vita

Lannan Luo

Education

2012–present  The Pennsylvania State University, University Park, PA.
Ph.D. in Information Sciences and Technology
Advisor: Prof. Peng Liu

2009–2012 University of Electronic Science and Technology of China, Chengdu, China.
M.Sc. in Communications and Information Systems
Advisor: Prof. Lemin Li

2005–2009 Xidian University, Xi’an, China.
B.S. in Telecommunications Engineering

Research Experience

My research has been focused on various aspects of the software piracy problems and
enhancing middleware security of mobile systems. My research approaches are mainly
empirical in tandem with formal methods, combining symbolic execution, theorem proving,
taint analysis, control flow analysis, data flow analysis, reverse engineering, data mining,
and machine learning.
- Obfuscation resilient software and algorithm plagiarism detection
- Android application security
- Android system security
- Vulnerability analysis

2015 Research Intern, Software Analytic Group, Microsoft Research Asia, Beijing, China.
- Research on analyzing characteristics of multiple programs from online judge systems.

2009–2012 Research Assistant, Key Lab of Fiber-Optical Communication Network, University of
Electronic Science and Technology of China.
- Research on network traffic intrusion detection.
- Project on designing and implementing a network route control model.

2008–2009 Research Assistant, State Key Lab of Integrated Services Network, Xidian University.
- Project on three-dimensional optimum path planning and scheduling for Unmanned
  Aerial Vehicles (UAV).

Awards and Honors

Best Paper Award Nomination in ACM SIGSOFT/FSE 2014
Top-class Scholarship with Honor, Univ. of Electronic Science & Tech. of China, 2010–2012
Top-class Scholarship with Honor, Xidian University, 2006–2009