Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

PROGRAM ANALYSIS FOR ANDROID SECURITY AND RELIABILITY

by
Sydur Rahaman

The recent, widespread growth and adoption of mobile devices have revolutionized the way users interact with technology. As mobile apps have become increasingly prevalent, concerns regarding their security and reliability have gained significant attention. The ever-expanding mobile app ecosystem presents unique challenges in ensuring the protection of user data and maintaining app robustness. This dissertation expands the field of program analysis with techniques and abstractions tailored explicitly to enhancing Android security and reliability. This research introduces approaches for addressing critical issues related to sensitive information leakage, device and user fingerprinting, mobile medical score calculators, as well as termination-induced data loss. Through a series of comprehensive studies and employing novel approaches that combine static and dynamic analysis, this work provides valuable insights and practical solutions to the aforementioned challenges. In summary, this dissertation makes the following contributions: (1) precise identifier leak tracking via a novel algebraic representation of leak signatures, (2) identifier processing graphs (IPGs), an abstraction for extracting and subverting user-based and device-based fingerprinting schemes, (3) interval-based verification of medical score calculator correctness, and (4) identifying potential data losses caused by app termination.

PROGRAM ANALYSIS FOR ANDROID SECURITY AND RELIABILITY

by
Sydur Rahaman

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

August 2023

APPROVAL PAGE

PROGRAM ANALYSIS FOR ANDROID SECURITY AND RELIABILITY

Sydur Rahaman

Dr. Iulian Neamtiu, Dissertation Advisor — Date
Professor, Department of Computer Science, NJIT

Dr. Ali Mili, Committee Member — Date
Professor, Associate Dean for Academic Affairs, Department of Computer Science, NJIT

Dr. Cristian Borcea, Committee Member — Date
Professor, Associate Dean for Strategic Initiatives, Department of Computer Science, NJIT

Dr. Abdallah Khreishah, Committee Member — Date
Professor, Electrical and Computer Engineering Department, NJIT

Dr. Zhiyun Qian, Committee Member — Date
Associate Professor, Computer Science and Engineering, UC Riverside, CA

BIOGRAPHICAL SKETCH

Author: Sydur Rahaman

Degree: Doctor of Philosophy

Date: August 2023

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2023
- Bachelor of Science in Computer Science and Engineering,
  Bangladesh University Of Engineering and Technology, Dhaka, Bangladesh, 2015

Major: Computer Science

Presentations and Publications:

Sydur Rahaman, Raina Samuel, and Iulian Neamtiu, "Diagnosing Medical Score Calculator Apps". *In Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, IMWUT 2023.*

Sydur Rahaman, Umar Farooq, Iulian Neamtiu, and Zhijia Zhao "Detecting Potential User-Data Save and Export Losses due to Android App Termination". *In Proceedings of the 4th ACM/IEEE International Conference on Automation of Software Test, 2023.*

Sydur Rahaman, Iulian Neamtiu, and Xin Yin "Algebraic-datatype Taint Tracking, With Applications to Understanding Android Identifier Leaks". *In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021*, Association for Computing Machinery, New York, NY, USA, Pages 70–82. https://doi.org/10.1145/3468264.3468550.

Sydur Rahaman, Raina Samuel, and Iulian Neamtiu "Quantifying Nondeterminism and Inconsistency in Self-organizing Map Implementations", *2021 IEEE International Conference on Artificial Intelligence Testing (AITest)*, Pages 85-92, doi: 10.1109/AITEST52744.2021.00026.

Raina Samuel, Iulian Neamtiu, and **Sydur Rahaman** "Could Medical Apps Keep Their Promises?," *E-Health '22: 16th Multi Conference on Computer Science and Information Systems*, Pages 173-180, July 2022.

Raina Samuel, Iulian Neamtiu, **Sydur Rahaman**, and James Geller "Characterizing Medical Android Apps," *E-Health '22: 16th Multi Conference on Computer Science and Information Systems*, Pages 155-162, July 2022.

Priyam Patel, Gokul Srinivasan, **Sydur Rahaman**, and Iulian Neamtiu "On the Effectiveness of Random Testing for Android: or How I Learned To Stop Worrying and Love the Monkey". *In Proceedings of the 13th International Workshop on Automation of Software Test , AST 2018*. Association for Computing Machinery, New York, NY, USA, Pages 34–37. https://doi.org/10.1145/3194733.3194742.

Rayhan Shikder, **Sydur Rahaman**, Farzia Afroze, and ABM Alim Al Islam "Keystroke/Mouse Usage Based Emotion Detection and User Identification". *2017 International Conference on Networking, Systems and Security (NSysS)*. Dhaka, Bangladesh, Pages 96-104, doi: 10.1109/NSysS.2017.7885808.

To my family and my loved ones:
Onti (my beautiful wife), Abba, Amma,
Vaiya, Sani, Ajim vai, Farhan, BSA family
and my inspiration CR7 (the GOAT)

ACKNOWLEDGMENT

I would like to express my deepest gratitude and appreciation to the following individuals who have contributed significantly to the completion of my PhD dissertation:

First and foremost, I extend my sincere gratitude to my dissertation advisor, Professor Iulian Neamtiu. Your guidance, expertise, and unwavering support throughout the research process have been invaluable. Your insightful feedback and encouragement have greatly shaped the outcome of this work, and I am truly grateful for the opportunity to have worked under your mentorship.

My sincere appreciation goes to the honorable members of my dissertation committee, Dr. Ali Mili, Dr. Cristian Borcea, Dr. Zhiyun Qian, and Dr. Abdallah Khreishah. I am honored to have had the privilege of benefiting from your expertise and thoughtful feedback. Your invaluable guidance and constructive criticism have significantly shaped the development and quality of this research.

I would like to express my deepest gratitude to the National Science Foundation (NSF) for their generous financial support throughout my doctoral research. The funding provided by the NSF under grant numbers CCF-2106710 and 2007730 has played a crucial role in enabling the successful completion of this dissertation.

I would also like to thank my colleagues, Raina Samuel, Umar Farooq, Muyeed Ahmed, and Xin Yin, for their collaboration, insightful discussions, and support throughout this journey. Your contributions and friendship have been invaluable, and I am grateful for the shared experiences and mutual support we have provided to one another.

Lastly, I want to express my deepest gratitude to my family, and in particular, to my wife. Your unwavering support, love, and understanding have been the bedrock of my academic journey. Your patience, encouragement, and belief in me have been

instrumental in overcoming the challenges faced throughout this endeavor. I am truly blessed to have you by my side, and I cannot thank you enough for your constant support and sacrifices.

To all those mentioned above, as well as to anyone else who has contributed to my academic and personal growth, I extend my heartfelt appreciation. Your support and encouragement have made this achievement possible, and I am forever grateful for the role each of you has played in my journey towards earning my PhD.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF TABLES
(Continued)

Table **Page**

LIST OF TABLES
(Continued)

Table **Page**

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

Mobile applications ("apps") have become an integral part of our daily lives, offering a wide range of functionalities and services. However, the rapid growth of the Android platform and the increasing complexity of mobile apps have introduced numerous challenges related to security, privacy, and reliability. Addressing these challenges is crucial to ensure the trustworthiness and effectiveness of Android apps in meeting user expectations. This dissertation aims to investigate and provide insights into several key areas of concern, including: identifier use and abuse; app reliability; and app testing. We now provide an overview of the scope of this work, describe the problem landscape, state the objectives, and outline the organization of the dissertation.

1.1 Background

1.1.1 Android devices and apps

As of July 2023, there are 16.8 billion mobile devices in use globally; of these, 70% run Android and 28% run iOS [9]. Google Play, the main Android app store, offers more than 3.5 million apps [1]. The number of apps on Google Play has increased alongside Android's rising popularity until December 2017, when nearly 700,000 apps were taken down [2]. Google has consistently taken measures against apps that impersonate other apps, contain inappropriate content, or pose potential harm. Users have the option to download apps from alternative sources like F-Droid [3] or OEM-based stores such as the Samsung Galaxy Store [7].

Over the years, there has been a significant increase in mobile app usage. On average, smartphone users: spend 3 hours daily on their devices; use 10 apps per day; and have over 80 apps installed on their phones [4]. Initially, apps were primarily used for general productivity and information retrieval purposes, such as email clients,

web browsers, or weather. However, due to growing demand, social media apps have become the most downloaded and popular apps [8]. Mobile apps now serve a broader range of functions, beyond leisure and entertainment. Most importantly, apps are being increasingly employed for critical tasks, including e-commerce [6], finance, or healthcare. For example, 86.5% of Americans use mobile banking apps [5]. Furthermore, medical apps have seen a rise in popularity, offering the ability to handle sensitive information and perform essential tasks. For these reasons, app reliability and security have become crucial concerns.

1.1.2 Android platform

The Android system is based on the Linux kernel; on top of the kernel sits middleware written in C/C++ and Java. Android apps, written in Java and (optionally) C/C++ native code, run on top of the middleware. Java code is compiled into Dalvik bytecode (DEX), which is translated to native binary code (in older Android versions, apps were executing on top of the Dalvik VM). Android apps are distributed as APK files – compressed archives which hold the DEX files, native code if present, and the app resource files. Android apps are high-level, event-driven, and with shallow method stacks. Below the application layer is the Android Framework which provides libraries containing APIs. This allows the use of hardware functionality without incurring any complexities inherent when working at a lower level. Since Android is constantly evolving with the introduction of new features and APIs, fragmentation becomes an issue that app developers must address. Thus, Android developers should test their apps on multiple Android versions and multiple hardware devices, to ensure that other platforms are also supported.

1.2 Dissertation Scope

This dissertation focuses on exploring critical aspects of Android app security, privacy, and reliability. It encompasses a range of topics, including algebraic-datatype taint tracking, understanding and breaking user and device fingerprinting in Android, diagnosing medical score calculator apps, detecting potential user-data save and export losses due to Android app termination, and evaluating the effectiveness of Monkey as a random testing tool.

1.3 Problem Description

The Android platform poses unique challenges in terms of security vulnerabilities, privacy risks, and reliability issues. The diverse and interconnected nature of Android apps makes them susceptible to identifier leaks, deceptive practices, and unauthorized data access. Furthermore, the accuracy and reliability of medical score calculator apps, as well as the potential losses of user data due to app termination, are critical concerns. Additionally, there is a need to evaluate the effectiveness of existing testing tools, such as Monkey, in ensuring the reliability and robustness of Android apps.

1.4 Dissertation Objectives

The primary objectives of this dissertation are as follows:

- Quantify the security risks associated with identifier leaks and deceptive practices in Android apps by precisely representing the identifier leaks.
- Explore techniques to understand user device-based fingerprinting schemes represented in Android and how to subvert them.
- Develop methods for diagnosing and verifying the accuracy of medical score calculator apps.
- Detect and analyze potential user-data save and export losses due to Android app termination.
- Evaluate the effectiveness of Monkey as a random testing tool for Android apps.

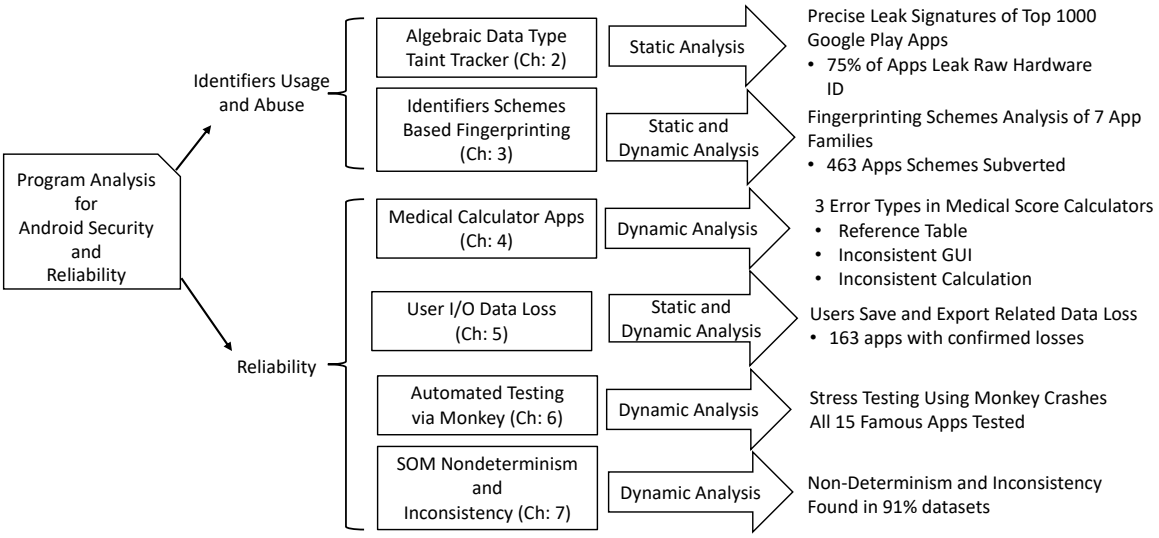


Figure 1.1 Dissertation overview.

- Analyze the nondeterminism and inconsistency present in SOM implementations.

We provide a visualization of this dissertation in Figure 1.1.

1.5 Dissertation Organization

The dissertation is organized along two main thrusts:

Identifiers Usage and Abuse. The investigation on "Identifiers Usage and Abuse" in this dissertation explores the ways in which Android app developers utilize and potentially exploit user and device identifiers for various purposes. By delving into the realm of "Identifiers Usage and Abuse," this research sheds light on the potential privacy risks associated with the collection, storage, and transmission of sensitive user and device information by Android applications.

Algebraic-datatype taint tracking: this research investigates the security of Android apps by analyzing the flow of sensitive data, identifying leaks, and characterizing their behavior using novel algebraic-datatype taint analysis techniques (Chapter 2).

Financial attacks and "limitless" offers: using our dynamic slicing approach, we extract and characterize fingerprinting schemes in Android and show how these schemes are constructed from a wide range of customer-identifying information: from ephemeral sources such as registration information to persistent sources such as device information. We also discuss different methods to break these fingerprinting schemes to achieve unlimited offers/promotions leading to financial losses (Chapter 3).

Reliability. The examination of "Reliability" in this dissertation focuses on investigating and enhancing the dependability and consistency of Android applications, including the effectiveness of testing tools, the diagnosis of potential errors, and the quantification of uncertainties within the app ecosystem. *Diagnosing medical score calculator apps:* this research focuses on verifying the accuracy of medical score calculator apps through automated correctness checking of reference tables and dynamic analysis-based verification techniques (Chapter 4).

Detecting potential user-data save and export losses due to Android app termination: this investigation aims to identify and address issues related to the termination of Android apps, specifically focusing on potential losses of user data and analyzing the reliability of file-write operations (Chapter 5).

Effectiveness of Monkey as a random testing tool: this research evaluates how Monkey's parameters affect code coverage (at class, method, block, and line levels) and answers several research questions centered around improving the effectiveness of Monkey-based random testing in Android and how it compares with manual exploration (Chapter 6).

Quantifying nondeterminism and inconsistency in self-organizing map implementations: this study investigates the reliability of self-organizing map implementations in neural network-based unsupervised learning. It examines the properties of determinism and consistency in four popular self-organizing map implementations using internal

and external metrics. The research identifies violations of these properties across a range of datasets, highlighting the importance of multiple runs and toolkit comparisons for reliable results in critical applications (Chapter 7).

In the following chapters, we will delve into each of these research areas, present our findings, discuss the implications, and provide practical solutions to enhance the security, privacy, and reliability of Android apps.

CHAPTER 2

ALGEBRAIC-DATATYPE TAINT TRACKING

Current taint analysis techniques used for tracking data flow from sources to sinks in Android apps have certain limitations, making them imprecise and ineffective in real-world scenarios. These shortcomings include difficulties in handling mutually exclusive sources and flows that combine multiple sources, which are common in complex Android app environments. To address these issues, we introduce a novel approach to taint analysis using algebraic-datatype representations. This technique generates expressive and concise taint signatures that involve operations such as AND, XOR, and hashing, akin to algebraic types. The proposed analysis is implemented as a static analysis tool for Android apps, specifically focusing on deriving app leak signatures that represent how sensitive hardware and software identifiers are manipulated before being exfiltrated to the network.

We introduce our algebraic representation for taint analysis in Section 2.2. The static analysis design in Section 2.3 includes a dataflow-centric call graph and two refinement phases to generate algebraic leak signatures. The evaluation in Section 2.4 demonstrates its effectiveness with 2.1x more leak discoveries than FlowDroid, achieving 72.6% precision and 100% recall. Section 2.5 presents empirical studies on privacy leaks in top Google Play apps and their embedded libraries. We also compare the apps' versions from 2018 with their 2020 counterparts finding a decrease in the use of raw/hardware identifiers, indicating that apps have become more privacy-friendly.

2.1 Motivation and Design Choices

Taint analyses determine whether data from a privacy-sensitive source (e.g., MAC Address) flows to an insecure sink (e.g., Internet). Current analyses' imprecision affects

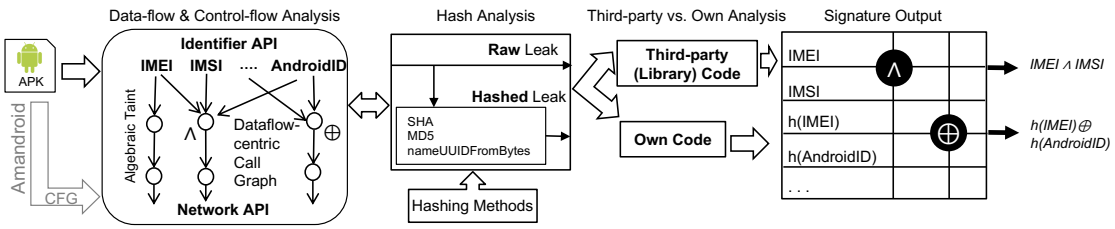


Figure 2.1 Overview.

their usability. For example, applying a standard Android taint analysis produces the same result (IMEI→Internet), in three different scenarios:

- The app exfiltrates the IMEI to a third-party server (e.g., Blink Health RX [34]); this practice is discouraged or forbidden by Google Play guidelines, depending on the nature of the app.
- The app (e.g., CareZone [26]) links with a financial library that uses the IMEI for payment fraud prevention; this is allowed by the guidelines.
- The app (e.g., Spectrum TV [60]) concatenates the IMEI with another identifier, e.g., AndroidID, hashes the result, and uses this hash value for customer identification. Since the actual IMEI cannot be reverse-engineered, the privacy loss is lower compared to the first and second scenarios.

Conflating these three use cases is problematic, as they are very different in terms of guidelines compliance and privacy implications. We combine a precise static analysis with an algebraic representation that can distinguish between these, and other, scenarios.

Android identifiers. Our analysis considers the seven popular Android IDs described in Table 3.1. The first four are "hardware" identifiers, i.e., tied to the specific phone hardware, and cannot be reset/changed in software; the remaining three are resettable identifiers. Google/Android developer guidelines have specific policies designed to protect user privacy [14] by discouraging, or even forbidding, access to hardware identifiers, such as:

- "Avoid using hardware identifiers".

Table 2.1 Identifiers Considered and Their Semantics

	ID	Semantics
Hardware	IMEI/MEID	15-digit; identifies mobile phone
	IMSI	15-digit; identifies SIM/network subscriber
	MAC Address	48-bit; identifies the network card -actual value reported by Android <6 -"02:00:00:00:00:00" reported by Android 6−9 -randomized value reported by Android ≥10
	Serial#	Manufacturer-assigned; identifies device
Resettable	AndroidID	Android ≥8: unique for an app or app group Android <8: unique user&device combination
	GUID	identifies app instance
	AdvertisingID	identifies user for ad tracking purposes

- "Only use an Advertising ID for user profiling or ads".
- "Use an Instance ID or GUID whenever possible for all other use cases, except for payment fraud prevention and telephony".
- "By its nature, fraud prevention requires proprietary signals".

To sum up, the only acceptable use of hardware identifiers is financial/fraud detection; all other scenarios, e.g., advertising or analytics, require the use of resettable identifiers. Many apps violate these guidelines; to counter this abuse, as shown in Table 3.1's "MAC Address" line, the Android platform's recent versions took increasingly stringent measures, first reporting a constant MAC Address, and then a randomized one. Android version 10 (used by 8.2% of Android devices as of February 25th, 2021, per Android Studio) restricts access to hardware identifiers to privileged (e.g., system, vendor) apps; this does not affect the generality of our approach.

2.2 Algebraic-datatype Representation for Signatures

In type theory, a product type is the type of an n-ary tuple, e.g., in OCaml, the tuple:

$$(1,3.14,\text{"foo"})$$

has type

$$\mathsf{int} * \mathsf{float} * \mathsf{string}.$$

A sum type is the type of a union, e.g., the C language union:

$$\mathsf{union\ u\ \{int\ i;\ float\ f;\}}$$

or the OCaml [150] variant

$$\mathsf{type\ number} = \mathsf{Int\ of\ int} \mid \mathsf{Float\ of\ float}$$

have the product type:

$$\mathsf{int} \oplus \mathsf{float}$$

(either an int or a float, but not both, inhabit the variant). Our core insight is an *algebraic-datatype* definition of taint: identifiers are base types and leak signatures are finite (non-recursive) algebraic types over base types.

2.2.1 Definitions

We define these shorthands for the Android identifiers: e for the IMEI, s for the IMSI, a for the AndroidID, r for Serial, m for MAC Address, v for AdvertisingID, g for GUID.

Signatures can be identifiers; hashes; or combinations thereof introduced via AND or XOR. We use S and T as metavariables for signatures. Hence our signature grammar is defined simply as:

$$\begin{array}{lrcl} Identifier & i & ::= & e \mid s \mid a \mid r \mid m \mid v \mid g \\ Signature & S & ::= & i \mid h(S) \mid S \oplus S \mid S \wedge S \end{array}$$

AND, denoted $S \wedge T$, indicates that *both* S and T (which can be identifiers or signatures), are used. This corresponds to a product type in type theory, and Cartesian product in set theory. Note that we deliberately use '$\wedge$' instead of the standard type theory symbol '$\times$' as it is more suggestive in our context.

XOR, denoted $S \oplus T$, indicates that either S or T is used, but not both. This corresponds to a sum type in type theory,[1] and disjoint set union in set theory. We use '$\oplus$' instead of the standard '$+$' from type theory as it is more suggestive in our context.

Hash. The hash, denoted $h(S)$, indicates that an identifier's hash (or the hash of an identifier combination) is leaked, not the actual "raw" identifier(s); e.g., $h(e)$ could be computed via hashing methods `nameUUIDFromBytes(IMEI.getBytes())` or `md.digest(IMEI.getBytes())`.

2.2.2 Properties

Defining formally what it means for a program to *leak less* than another program is key. For this purpose we introduce the *subsumption* relation, '$<:$', induced by subset semantics. Informally, app A leaks less than app B, aka B subsumes A, if the set of all possible values leaked by A is a subset of the set of all possible values leaked by B. We now define subsumption for the algebraic representation.

Subsumption (AND). An app whose signature is S leaks less than an app whose signature is $S \wedge T$; this is denoted $S <: S \wedge T$. Similarly, an app whose signature is T leaks less than an app whose signature is $S \wedge T$; this is denoted $T <: S \wedge T$.

Subsumption (XOR). An app with signature $S \oplus T$ leaks less than an app whose signature is $S \wedge T$.

Subsumption (hash). An app with signature $h(S)$ leaks less[2] than an app with signature S; this is denoted $h(S) <: S$.

First, note how subsumption introduces a partial order (in certain cases, a total order) on apps' leaking properties: its power becomes apparent in Section 2.5 when

[1]Technically, in the Curry-Howard isomorphism [131], sum types correspond to OR in logic, not to XOR. However OR is not our intended semantics, since $a = TRUE$ in $a \vee b = TRUE$ does not force b to be $FALSE$ whereas in our semantics it does (mutual exclusion); a longer explanation is available here [162]. Our semantics is readily apparent in the Church Boolean [104] function XOR, i.e., $\lambda a.\lambda b.a$ (*not* b) b, where if a reduces to TRUE, (*not* b) must reduce to FALSE for the XOR to reduce to TRUE.

[2]"Leaks less" in a privacy/cryptographic sense, rather than strictly $h(S) \subseteq S$.

```
1  String getTelephonyDeviceId(Context context) {
2      String deviceIMEI = ((TelephonyManager) context.getSystemService("phone")).getDeviceId();
3      return deviceIMEI; }
4  String getAndroidId(Context context) {
5      String androidId = Secure.getString(context.getContentResolver(), "android_id");
6      return androidId; }
7  String getWifiMacAddress(Context context){...
8      String mac=wifiManager.getConnectionInfo().getMacAddress();
9      return mac; }
10 String getUniqueDeviceID(Context context) {
11     return generateDeviceId(getTelephonyDeviceId(context), getWifiMacAddress(context),
           getAndroidId(context));
12 }
13 String generateDeviceId(String str, String str2, String str3) {
14     if (! TextUtils.isEmpty(str)) { // str3 → {a}
15         str3=str; // str3 → {e}
16     }
17     else if (! TextUtils.isEmpty(str2) && !TextUtils.isEmpty(str3)) {// str3 → {a}
18         str3 = new UUID((long) str3.hashCode(), (long) str2.hashCode()).toString();// str3 → {h(m) ∧ h(a)}
19     } else if ( TextUtils.isEmpty(str3)) {// str3 → {a}
20         str3=UUID.randomUUID().toString();// str3 → {g}
21     }
22     return str3; // str3 → {e ⊕ a ⊕ h(m) ∧ h(a) ⊕ g }
23 void SendDeviceinfo() {...
24     httpParaMap.put("deviceID", getUniqueDeviceID(context).toString());
25   ...}
```

Figure 2.2 UUID generation in the Audiobooks.com app.

we use it to check whether a signature subsumes another (i.e., an app leaks more than another app, or more than a different version of the same app). Second, the algebraic representation naturally induces *equivalence classes*: apps with the same signature will leak the same identifiers (and semantically, the identifiers are manipulated in the same way, e.g., hashed).

2.3 Approach

The architecture of our system is shown in Figure 4.4. Given an Android app (APK file), we perform a chain of analyses to construct the app's leak signature: control- and data-flow analyses that compute and propagate algebraic taint; a secondary analysis to detect hashing; and a third-party vs. own analysis. The initial control-flow graph is produced by the Amandroid static analyzer [208] (shown in gray; not a contribution of this work).

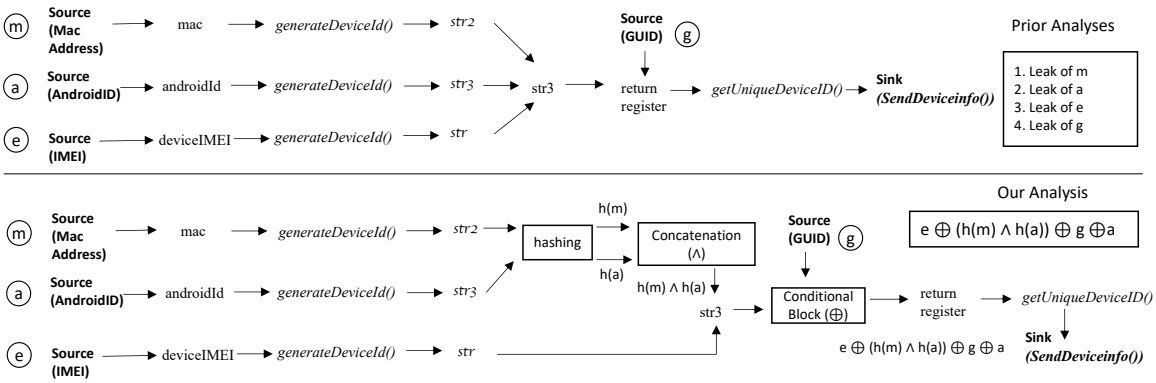


Figure 2.3 Prior taint approaches (top) vs. our approach (bottom).

Our approach operates from generic sources/sinks: any variable or API method can be used as source; similarly for sinks. These are specified in a source input file and sink input file, respectively, as is common for flow trackers.

2.3.1 Motivating example

We illustrate our approach, and contrast it with prior taint analyses, on the Audiobooks.com [32] app. The relevant source code is shown in Figure 2.2. The app attempts to leak an unique device ID, aka UUID onto the network via SendDeviceInfo() (lines 23–25). The UUID is: the IMEI, if available (retrieved on lines 1–3); if not available, the hashes of MAC Address and AndroidID if they are available (lines 4–12); otherwise the GUID, if available (line 20); finally, if none of these conditions are met, AndroidID is the UUID.

Traditional taint analyzers, e.g., FlowDroid or Amandroid, perform taint analysis of each source separately and report the leaks separately. For the aforementioned code snippet, such tools perform four tainted paths calculations for four different sources (IMEI, MAC Address, AndroidID, and GUID), as illustrated in Figure 2.3 (top). Eventually, they produce a report stating that all four identifiers are leaked. This, however, is imprecise for two reasons. First, the identifiers' *hashed* values are leaked, which is less dangerous than *raw* leaks. Second, the tools fail to report the

aggregation: actually MAC address and AndroidID are used *together*, exclusive of IMEI and GUID – i.e., a signature, whereas the tools report separate leaks.

In contrast, our analysis produces the correct signature. The high-level view is shown in Figure 2.3 (bottom); lower-level dataflow analysis will be discussed shortly. Instead of simple taint propagation, we propagate algebraic taint. For the example shown in the figure, instead of four different leaks, we report one precise signature, the leak actually present here, that is: $e \oplus (h(m) \wedge h(a)) \oplus g \oplus a$.

```
1  public class PersistentUUID {
2  JSONObject jsonObject = new JSONObject();
3  private static final String UUID_KEY = "nr_uuid";
4  ...
5      private void generateUniqueID(Context context) {...
6        TelephonyManager tm = (TelephonyManager) context.getSystemService("phone");
7        hardwareDeviceId = tm.getDeviceId();
8        putUUID(hardwareDeviceId);...
9      }
10     protected void putUUID(String uuid) {...
11       jsonObject.put(UUID_KEY, uuid); ...
12     }
13     public String getPersistentUUID() {...
14       uuid = jsonObject.getString(UUID_KEY);...
15       return uuid;
16     }
17 }
18 public class AndroidAgentImpl {
19     public void sendDeviceInformation() {...
20       hashMap.put("Model",Build.MODEL);
21       hashMap.put("deviceID",persistentUUID.getPersistentUUID()) ;...
```

Dataflow-centric Call Graph

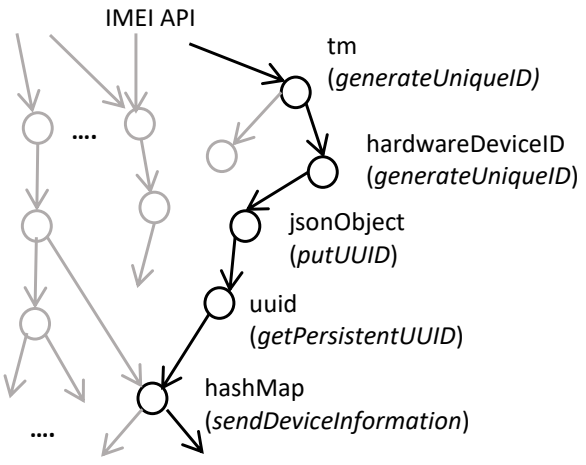


Figure 2.4 Source code and its dataflow-centric call graph.

2.3.2 Dataflow-centric call graph construction and analysis

While other static taint analyzers [88, 208] perform interprocedural control- and data-flow analyses (as we do), their taint facts and propagation are both imprecise and insufficient for our purposes. We address this via a series of analyses, the first of which is *call graph extraction*, as explained next.

Call Graph Extraction. We start our analysis from the Amandroid-generated control-flow graph, and soundly[3] extract the sub-graph where the algebraic taint-relevant data propagates. We illustrate our approach in Figure 5.7: the top shows the source code while the bottom shows our dataflow-centric call graph (the grey edges/vertices depict the parts of the control flow graph that can be soundly abstracted away). In the source code, the IMEI is obtained via the Android API on lines 6 and 7, and stored in hardwareDeviceId. On line 8, the IMEI flows into the putUUID() method as a parameter; the IMEI is saved into a JSONObject on line 11. Our graph captures this dataflow. We can see a dataflow edge between the hardwareDeviceID and jsonObject variables where the IMEI data is saved as key-value JSON data. Next, if we follow the dataflow of the jsonObject currently holding the IMEI data, the data is saved into a new variable uuid on line 14 in the getPersistentUUID() method. This is captured by the edge between jsonObject and uuid. Finally, on line 21, the hashMap creates an entry (key-value pair) with deviceID as key and IMEI as value, resulting in an edge from uuid to hashMap in the graph. Our dataflow analysis (forward/may – a variant of reaching definitions [79]) propagates algebraic taint on top of the graph.

Dataflow Analysis. We illustrate our dataflow analysis on the program in Figure 2.2, method generateDeviceID. A simplified-for-legibility version of the $Out(s)$ sets limited to variable str3 are shown as comments on the right side of the code. At the beginning,

[3]Technically *soundy* [160] – our approach is sound up to native code (because we leverage Amandroid, which is sound up to native code) which is par for the course for Java/Android analyses.

the set contains the value {a}, i.e., AndroidID. At line 15, str3 is assigned a new value which contains the IMEI, {e}. The $Out(s)$ set for the statement at line 18 contains the combined hash value signature, (h(m)∧ h(a)); str3 is then assigned {g}, GUID, at line 20. For each statement, the union of the $Out(s')$ of all the predecessors s' of s gives the $In(s)$ value [139, 79]. In our example, line 22's predecessor set is lines {14,15,18,20} (of course, in the actual analysis, conditional statements are more fine-grained hence we typically performs two-way joins rather than a four-way join). Hence at program joint point (line 22), the *In* set, in this case identical to the *Out* set, which represents the UUID signature, is:

$$e \oplus (h(m) \wedge h(a)) \oplus g \oplus a$$

A key factor that informed our analysis design, and helps keep the analysis precise, was our observation (drawn from manual taint analysis, Section 5.3.5) that apps' code for constructing the signature, such as the code discussed above, tends to lack back edges, which helps contain dataflow sets size.

2.3.3 Hash analysis

As illustrated in Figure 4.4, hashed leaks are leaks that flow through hashing methods, e.g., MD5. We detect such flows by setting up another flow analysis as follows. First, we set the entry of hash methods as *sinks*. Next, we set the return of hash methods as *sources* and network API methods as sinks. As a result, we separate the underlying identifiers leaks into *raw leaks* and *hashed leaks*. Note that our analysis takes a "Hashing Methods" list as input; we constructed this list based on an exhaustive analysis of hashing functions/practices available in Java and practices used by manually-analyzed Android apps. For example, some common hashing Java API methods include MD5, SHA and nameUUIDFromBytes(), or Java classes such as MessageDigest. This list is user-configurable hence easily extended.

```
1  public static String getDeviceId(Context me) {
2      TelephonyManager telephonyManager = (TelephonyManager) me.getSystemService("phone");
3      String IMEI = telephonyManager.getDeviceId();
4      MessageDigest md = MessageDigest.getInstance("SHA");
5      byte[] dat = null;
6      if (IMEI == null) return "";
7      md.update(IMEI.getBytes());
8      return hashByte2String(md.digest());
9     }
```

Figure 2.5 Cryptographic hashing in app CGTN.

2.3.4 Third-party vs. own code analysis

To separate own leaks from third-party leaks we used a predefined list of common third-party libraries as reference,[4] along with flow partitioning. Specifically, if an identifier's entire flow involves only third-party library methods, we tag that leak as *third-party* leak; otherwise we tag it as *own* code leak. We have not found any cross-flow between third-party code and own code in our examined apps.

2.3.5 Example

In Figure 2.5, we show an example that illustrates both hashing and third-party vs. own analysis, from app CGTN. The IMEI is being hashed by the cryptographic hashing method MessageDigest(SHA) (lines 4, 7 and 8). The leak happens inside the app's own package, so we categorize it as 'own'. The leak signature is, therefore : $h(e)$ [own]

2.4 Evaluation

We evaluated our approach, and performed six studies, on 1,000 top apps from Google Play. The 1,000 apps span 19 popular categories from Google Play. The number of apps varied slightly across categories as we favored popular apps. For the evolution study only (Section 2.5.6), we compared apps' year 2018 versions with their year 2020 counterparts, i.e., 2,000 APKs.

[4]The same library can appear under different names in different apps due to obfuscation; we mapped obfuscated libraries' names to a unique name, common across all apps, for that library.

Table 2.2 The Number of Top Google Play Apps Where FlowDroid, and Our Approach Respectively, Found Leaks

	IMEI	IMSI	Serial	MAC	And.ID	AdvID	GUID
FlowDroid	104	23	n/s	101	336	n/s	519
Our Approach							
Total	405	108	316	372	722	455	728
Raw	334	43	235	324	695	455	728
Hashed	145	79	142	83	297	0	0
False Negatives	3	0	n/a	7	0	n/a	11

Table 2.3 The Number of Ground Truth Apps Where FlowDroid, and Our Approach Respectively, Found Leaks

	IMEI	IMSI	Serial	MAC	And.ID	AdvID	GUID
FlowDroid	11	9	n/s	5	24	n/s	53
Our Approach	21	9	18	15	44	32	56

2.4.1 Effectiveness

We first evaluate the effectiveness of our approach by comparing, on the 1,000 apps, with state-of-the-art FlowDroid; next, we compare with ground truth on 64 apps where flows were tracked manually.

Comparison with FlowDroid. We ran the July 2020 version of FlowDroid from its official GitHub page [27] on our 1,000-app dataset. We configured FlowDroid to match our configuration: we enabled implicit flow analysis and context sensitivity. Dataflow analysis, callback collection during call graph construction, and result collection time limits were set to 1000 seconds, 1000 seconds, and 500 seconds respectively. As a point of reference, our analysis' median time per app was 347 seconds (Section 2.4.2), so we believe the aforementioned time limits are reasonable. We directed FlowDroid to use sources and sinks that match ours. As sources, we used the API methods responsible for retrieving the 7 identifiers we track (Table 3.1). For sinks, we used the SuSi list [28], i.e., all possible sinks under NETWORK_INFORMATION category, as

we are only interested in exfiltration to the network. Note that the API methods that read the 'Serial' and 'AdvertisingID' cannot be expressed in FlowDroid's taint source format, so we marked those as 'n/s'.

We show the results in Table 2.2: the number of apps where leaks were found, by FlowDroid and our approach, respectively. We make three observations. First, FlowDroid misses a substantial number of leaks, as it reports 46% of the leaks we report, 1083 vs. 2335 (for those five IDs we could run FlowDroid on); prior work suggests that false negatives' root causes in FlowDroid/SuSi include inter-component communication (ICC) and imprecise sink/source lists [174]. Our approach handles ICC by default (via Amandroid). Even with the generous time limits we set, FlowDroid timed out and could not find all the leaks. Second, FlowDroid cannot distinguish between raw and hashed leaks, as our approach does (third and fourth rows show the raw/hashed split). Third, our approach has some false negatives compared to FlowDroid (i.e., we miss leaks that FlowDroid does not miss), as depicted in the last row. We found that false negatives originate in the CFG provided by Amandroid – when Amandroid missed some control-flow edges, our approach missed those edges as well.

Comparison with Ground Truth. We measured the False Positives (FP) and False Negatives (FN) by comparing the results of our static analysis with ground truth – known flows found in prior work via a manual analysis on 64 apps;[5] these "ground truth flows" are not a contribution of this work. The confusion matrix is:

True Positives: 186	**False Positives**: 70
False Negatives: 0	**True Negatives**: 512

[5]The manual flow analysis was exhaustive, e.g., went so far as capturing and rewriting network packets.

Table 2.4 Efficiency Results

Analysis time (seconds)				**Bytecode size (MB)**			
min	max	median	mean	min	max	median	mean
140	47,651	347	411	0.04	103.4	16.6	15

These figures, a 72% precision and 100% recall, are par for the course for a static analysis, indicating that our approach is effective.

We also show a comparison of our approach with FlowDroid on these 64 ground truth apps in Table 3.8. Our approach found more leaks than FlowDroid on these apps as well.

2.4.2 Efficiency

We conducted the experiments on a MacBook Pro (3.5 GHz dual-core Intel Core i7 with 16GB RAM), running Mac OS X 10.14.6. We show statistics (computed across the entire app dataset) of analysis running time, along with app bytecode size, in Table 5.5. A typical app took about 6 minutes to analyze – median 347 seconds, geometric mean 411 seconds – which is efficient for a static analysis; the longest analysis time was 13 hours, which we believe can be reduced substantially with more engineering. The app bytecode statistics – median 16.6MB, geometric mean 15MB, maximum 103MB – show that our approach is capable of analyzing large apps.

Table 2.5 Identifiers Stats

	IMEI	**IMSI**	**Serial**	**MAC**	**And.ID**	**AdvID**	**GUID**
Apps (%)	51	21	40	47	91	58	92
Raw (%)	83	38	75	88	96	100	100
Hashed (%)	37	74	46	23	41	0	0
Raw & Hashed (%)	20	12	21	11	37	0	0

2.5 Applications

We now present six studies that provide evidence for the expressiveness and effectiveness of algebraic-datatype taint tracking.

2.5.1 What IDs are leaked, and in what form?

We first studied the frequency and nature of identifier leaks. In Table 2.5 we show the percentage of apps that leak that identifier, and the form of the leak. Three critical identifiers, IMEI/Serial/MAC Address, are leaked by 40–51% of the apps, which is the first reason for concern. The second reason for concern is that identifiers are leaked *raw* by 75–88% of the apps that leak them; 23–46% of apps leak these IDs hashed – in lieu of, or in addition to, the raw leak. On a more positive note, the IMSI is leaked to a lesser extent, only 21% of the apps, and mostly hashed (74%).

For the remaining three, resettable identifiers, we found that the AndroidID and GUID are leaked routinely: by 91% and 92% of the apps, respectively. The Advertising ID is seeing a reduced leak rate (58% of the apps). Raw leaks are the norm for these identifiers: 96–100% of the leaks are in raw form.

We observed that certain apps leak *both* the raw and hashed ID (last row of Table 2.5). Note that for IMEI, IMSI, Serial, and AndroidID, this figure is quite high, 12–37% of the apps. We believe this practice to be particularly pernicious, because such apps essentially have the $h(ID) \rightarrow ID$ mapping. If these apps communicate the mapping to other apps that only have $h(ID)$, then the raw ID value, unique to the device, *can be de-anonymized*.

2.5.2 Multiple-identifier leaks

We now study cases where multiple identifiers are leaked by a single app. We present the most frequent signatures in Table 2.6. On a positive note, 3 out of top-10 most common signatures are $h(a) \wedge g$, $a \wedge g$, and $a \wedge v$, that is, resettable identifiers (10%, 7%,

Table 2.6 Most Common Multi-ID Leaks; R=Raw, H=Hashed

IMEI		IMSI		Serial		MAC		And.ID		ADvID	GUID	#Apps
R	H	R	H	R	H	R	H	R	H	R	R	
									✓		✓	100
								✓			✓	70
	✓				✓				✓		✓	55
							✓		✓			41
✓								✓				40
					✓				✓			33
								✓		✓		31
	✓		✓									28
	✓						✓					27
	✓								✓		✓	26
	✓		✓		✓		✓		✓			24
✓						✓		✓				24
	✓				✓				✓			17
	✓		✓						✓			14
					✓				✓		✓	13
✓											✓	13
✓		✓										11
✓				✓								11
									✓	✓		10

and 3.1%, respectively). The flip side is that the other 7 out of top-10 use hardware identifiers: we have $h(e) \wedge h(r) \wedge h(a)$, then $h(m) \wedge h(a)$, then $e \wedge a$, at 4% and above. We have $h(r) \wedge h(a)$, then $h(e) \wedge h(s)$, then $h(e) \wedge h(m)$, then $h(e) \wedge h(a) \wedge g$, at 2.6% and above.

Note how these findings underline the effectiveness of our approach. A standard taint analysis would conflate the 100 apps whose signature is $h(a) \wedge g$ with the 70 apps whose signature is $a \wedge g$; and would conflate the 24 apps using $h(e) \wedge h(s) \wedge h(r) \wedge h(m) \wedge h(a)$ with the 4 apps using $e \wedge s \wedge r \wedge m \wedge a$.

Examples: complex yet common signatures. Our prior work on manual taint analysis (Section 5.3.5) has revealed groups of apps with common signatures – apps use the same mechanism for constructing a unique "DeviceID". Our analysis can group apps into equivalence classes induced by app signatures; this has a variety of

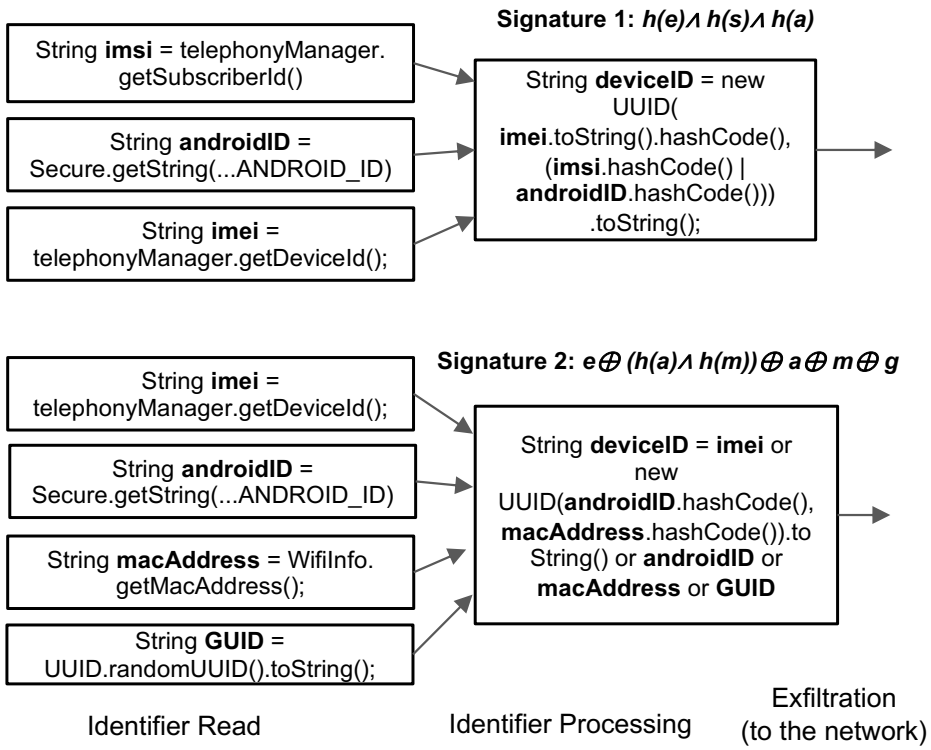


Figure 2.6 DeviceID signatures.

applications, from finding groups of apps with common behavior [120] to groups of apps with common developers, etc. We show two such examples in Figure 3.4. The left side (first stage) of the figure lists all the identifiers involved in signature construction. The second stage shows how those identifiers are combined or processed to generate a hashed unique DeviceID, which is then exfiltrated.

Signature 1 (app: Texas Roadhouse Mobile [62]) creates a DeviceID from the combination of IMEI, IMSI, and AndroidID. Since all identifiers are used, the signature uses ANDs:

$$h(e) \wedge h(s) \wedge h(a)$$

Signature 2 (library io.intercom) is quite complex, as the DeviceID is exactly one of: either the IMEI, or the AndroidID, or the MAC Address, or the GUID, or the AND of hashed Android ID and hashed MAC Address. Our representation captures this effectively:

$$e \oplus (h(a) \wedge h(m)) \oplus a \oplus m \oplus g$$

Table 2.7 Third-Party vs. Own Code Statistics: Number and Percentage of Leaks (T=third-party, O=own code)

	IMEI		IMSI		Serial		MAC		And.ID	
	T	**O**	**T**	**O**	**T**	**O**	**T**	**O**	**T**	**O**
R	183	208	28	16	128	120	223	145	586	418
H	97	63	56	25	120	33	61	23	198	144
R(%)	33	38	22	13	32	30	49	32	44	30
H(%)	18	11	45	20	30	8	14	5	15	11

Table 2.8 Third Party Libraries: the Number of Methods Leaking Each ID, and the Form of the Leak (H=Hashed, R=Raw). Raw Hardware Leaks in Non-financial Libraries Shown in Red

Library	**IMEI**		**IMSI**		**Serial**		**MAC**		**AndroidID**		**AdvID**	**GUID**	**Purpose**
	H	R	H	R	H	R	H	R	H	R	R	R	
com.paypal	35	8	35	4	35	4	35	6	35	4	30	142	**finance**
io.fabric	0	38	0	0	0	30	0	35	0	543	0	285	analytics
net.hockey.app	0	0	0	0	0	0	0	0	114	69	0	266	analytics
com.apps.flyer	0	2	0	0	0	0	0	2	7	91	95	106	ads
com.kochava	0	3	0	0	0	0	0	3	0	36	34	33	ads
com.threat.metrix	2	24	0	6	3	32	0	0	2	12	0	37	analytics
com.google	0	0	0	0	0	0	0	0	3	22	164	16	ads
io.intercom	0	9	0	0	0	0	0	8	0	9	0	106	analytics
io.branch	0	0	0	0	0	0	0	0	0	65	0	66	analytics
com.appsee	15	0	15	0	0	15	0	0	0	0	0	60	analytics
bo.app	0	0	0	0	0	7	0	0	0	0	0	60	analytics
com.tune	0	0	0	0	0	0	0	0	0	12	0	107	**finance**
com.segment	0	24	0	0	0	24	0	0	0	24	3	76	analytics
com.adjust	0	0	0	0	0	0	0	55	0	55	0	57	**finance**
com.leanplum	0	0	0	0	0	0	12	0	1	11	0	11	analytics
com.nielsen	0	0	0	0	0	4	0	1	0	26	6	4	analytics
com.iovation	0	9	0	8	0	0	0	9	0	9	0	7	analytics
com.startapp	0	0	0	0	0	0	0	0	0	32	32	99	ads
com.newrelic	34	0	0	0	34	0	0	0	34	0	0	174	analytics
com.mobvista	0	22	0	0	0	0	0	22	27	22	22	65	analytics

2.5.3 Library leaks vs. app's own leaks

We motivate this analysis via two scenarios. In the first scenario, a developer submits an app for publishing onto Google Play, and the app is rejected for violating guidelines, e.g., a raw hardware leak in a non-financial app. Even though the developer has used no IDs, the app is linked with a "leaky" advertising library that causes the ID leak. The developer should be able to extract the library's signature and the app's signature to determine the leak's cause and course of action. In the second scenario, the Google Play marketplace itself tries to determine whether a raw hardware leak is allowable or not, prior to publishing an app. If the app uses a payments services library, the leak would be allowed in the name of fraud prevention. Hence it is essential to find whether

Table 2.9 "Leakiest" Apps. Non-financial Apps With No Financial Libraries Shown in Red; R=Raw, H=Hashed

App	#Installs (million)	Category	IMEI		IMSI		Serial		MAC		AndrID		AdvID	GUID
			R	H	R	H	R	H	R	H	R	H	R	R
Spectrum TV [60]	10	Entertainment	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓
CGTN [37]	5	News & Magazines	✓		✓		✓	✓	✓		✓		✓	✓
GPS Navigation System [46]	10	Maps & Navigation	✓		✓		✓		✓		✓		✓	✓
WiFi Map [44]	50	Productivity	✓	✓	✓	✓	✓		✓		✓		✓	✓
Bitcoin, Crypto News [33]	1	Finance	✓		✓		✓		✓		✓		✓	✓
CheapOair [38]	1	Travel & Local	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓
Greyhound Lines [47]	1	Travel & Local	✓		✓		✓		✓		✓			✓
JCPenney [49]	5	Shopping	✓		✓		✓		✓		✓			✓
CBS News [36]	1	News & Magazines	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓
Wendy's [64]	5	Food & Drink	✓		✓		✓	✓	✓		✓	✓	✓	✓
Zipcar [66]	1	Maps & Navigation	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓
Lyft Rideshare [51]	10	Maps & Navigation	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Western Union [65]	5	Finance	✓	✓	✓		✓	✓	✓		✓	✓		✓
NJ TRANSIT [55]	1	Maps & Navigation	✓		✓		✓	✓	✓	✓	✓			✓
Curb The Taxi App [40]	1	Maps & Navigation	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
Apartments.com [29]	5	House & Home	✓		✓		✓		✓		✓			✓
CareZone [26]	1	Medical	✓		✓		✓		✓		✓	✓		✓
BURGER KING [35]	10	Food & Drinks	✓		✓		✓		✓		✓		✓	✓
Fox Now [43]	10	Entertainment	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓
One Dollar [56]	0.5	Shopping	✓		✓		✓		✓		✓	✓	✓	✓
Sam's Club [58]	1	Shopping	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓
Aaptiv [30]	1	Health & Fitness	✓		✓		✓		✓		✓		✓	✓
Letgo [50]	100	Shopping	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
Amber Weather [31]	1	Weather	✓		✓		✓	✓	✓		✓	✓	✓	✓
Blink Health Rx [34]	0.1	Medical	✓		✓		✓		✓		✓			✓

a leak is caused by a library or the app itself. Our analysis isolates the source of the leak (Figure 4.4) and attributes it to either *third-party* (library) code or *own* code.

In Table 2.7 we present the results of leak attribution in our examined apps. For each ID, we show the number of third-party (T) vs. own (O) leaks, whether the leak is raw or hashed, as well as the percentage distribution. All the hardware identifiers – IMEI, IMSI, Serial, and MAC Address – as well as the AndroidID, are leaked more by libraries than own code (51%, 67%, 62%, 63%, 59%, respectively). For identifiers

AdvertisingID and GUID (omitted from the table for space), leaks were substantial but balanced: 297 third-party vs. 291 own for AdvertisingID and 639 vs. 631 for GUID.

This finding – hardware ID leaks are attributable more to third-party code than own code – is important, because it shows that apps could *unwittingly* be the source of problematic leaks, e.g., due to "leaky" libraries, and could be unfairly blamed for leaks that apps' own developers did not introduce, or were not even aware of.

```
1  // JCPenney app
2  static String g(Context context) {...
3      hashMap.put("di",(telephonyManager.getDeviceId()).digest());
4       ...
5      jSONObject.put("di", hashMap.get("di"));...
6  }
7
8  // Dunkin app
9  final HttpParameterMap getRiskBodyParameterMap(){...
10     httpParameterMap.add("imei", (TelephonyManager) context.getSystemService("phone")).getDeviceId(),
           true);...
11     return httpParameterMap;
12 }
```

Figure 2.7 Hashed leak (top) vs. raw leak (bottom) in the same, com.threatmetrix library.

2.5.4 Leakiest libraries

As mentioned previously, libraries are a significant source of leaks. Summarizing leaks in libraries is non-trivial, however, because of *context-sensitivity*: a leak would materialize (or not) depending on how an app invokes the library. We illustrate this in Figure 2.7, on library com.threatmetrix. When the library is invoked from the JCPenney app (top), the IMEI is leaked *hashed*: on line 3 the IMEI is read and its hash (digest) added to hashMap. However, when the library is invoked from the Dunkin app (bottom), the IMEI is leaked *raw*: on line 10 the IMEI is read and added, raw, to httpParameterMap.

Therefore, in Table 2.8 we present the results of our library analysis; of the 821 libraries used in our apps, we show the top-20 "leakiest"; for each library and each ID, we show the number of library methods that leak the raw ID and the number

Table 2.10 Identifier-centric Study Results: 2018 → 2020 Changes in Identifier Use

	IMEI		IMSI		Serial		MAC		AndroidID		AdvID		GUID	
	TP	**O**	**TP**	**O**	**TP**	**O**	**TP**	**O**	**TP**	**O**	**TP**	**O**	**TP**	**O**
Raw	+8	+19	+1	+8	+14	+23	+8	+10	+41	+44	+24	+44	+47	+52
	-45	-59	-8	-2	-28	-25	-48	-28	-75	-52	-48	-23	-45	-27
Net	*-37*	*-40*	*-7*	*+6*	*-14*	*-2*	*-40*	*-18*	*-34*	*-8*	*-24*	*+21*	*+2*	*+25*
Hashed	+12	+8	+5	+5	+15	+10	+5	+5	+33	+26	0	0	0	0
	-19	-16	-10	-8	-21	-5	-12	-2	-34	-18	0	0	0	0
Net	*-7*	*-8*	*-5*	*-3*	*-6*	*+5*	*-7*	*+3*	*-1*	*+8*	*0*	*0*	*0*	*0*

of library methods that leak the hashed ID. For example, library com.paypal has 35 methods that leak the hashed IMEI, 8 methods that leak the raw IMEI, 35 methods that leak the hashed IMSI, etc.

For each library, we also present the library's purpose, as indicated on the library's website or GitHub page. Note that only three libraries are financial: com.paypal, com.tune, com.adjust; hardware ID leaks are expected, and allowed, in these libraries. However, the analysis shows that most leaks are in non-financial libraries, the overwhelming majority of which are advertising and analytics.

Table 2.8 paints a grim picture of the Android library landscape when it comes to privacy: advertising and analytics libraries make heavy use of hardware IDs, but this use appears aimed at identifying users and devices rather than preventing fraud. Ironically, financial libraries com.tune and com.adjust are among the most privacy-friendly libraries (least intensive users of hardware IDs).

2.5.5 Leakiest apps

We examined the "leakiest" apps in light of the Google guidelines for acceptable use of hardware IDs. We focus on the top-25 apps that manage to leak *all hardware identifiers, raw*. Moreover, many of these apps also leak hashed versions of hardware identifiers; leaking both raw and hashed versions is a concern for de-anonymization. We show the results in Table 2.9. For each app we show the popularity (the floor of the number of installs, as indicated on Google Play on February 25th, 2021), the app category, and the list of leaks.

We identified those apps that have a legitimate financial reason to use hardware IDs as follows: apps that are in the Finance category, or apps that link with a financial library, and the leaks are due to the library (third-party) code rather than the app code. The apps that did not meet these conditions, shown in red in the table, potentially violate ID usage guidelines. Our approach distinguishes between raw and hashed, and between third-party vs. own leaks, helping spot potential violations. In contrast, an approach that misses these nuances might flag a substantial number of benign, policy-abiding apps as problematic (i.e., a high rate of false positives).

Altogether, our dataset had 190 apps that either use a financial library, or the app itself is in the Finance category. These apps might need hardware identifier information for fraud & abuse checking purposes, so leaks from these apps can be accepted. However, 47 out of these 190 apps leak at least one raw hardware ID via a non-financial third-party library, which is a concern.

2.5.6 Longitudinal study: 2018 vs. 2020

To investigate whether apps are becoming more guidelines-compliant and privacy-friendly, we conducted a longitudinal study, comparing the 2018 versions of 1,000-app dataset with their 2020 counterparts.

Identifier-Centric Study. We first investigate how the prevalence/use of a certain identifier has changed over two years. We tabulate the findings in Table 2.10. For each ID, each code location (third-party (TP) or own (O)), and each leak type (hashed or raw) we show the number of apps that added that ID leak with '+' and the number of apps that removed that ID leak with '-'. For example, for raw IMEI we have: in third-party code, 8 apps have added this leak and 45 apps have removed this leak, yielding a net change of *-37*; whereas in own code, 19 apps have added this leak and 59 apps have removed this leak, yielding a net change of *-40*. The results reveal

Table 2.11 App-centric Study Results: Subsumption Kind, Informal Definition, and # of Apps Exhibiting Subsumption

Kind	Subsumption Definition	#Apps
AND	hardware ID leaks decreased	108
	hardware ID leaks decreased, software ID leaks increased	11
	raw IMEI leak removed	87
	raw MAC Address leak removed	71
	raw Serial leak removed	49
	raw IMSI leak removed	12
	raw AndroidID leak removed	107
Hash	raw hardware ID leak → hashed hw. ID leak	26
	raw IMEI → hashed IMEI	14
	raw IMSI → hashed IMSI	3
	raw MAC Address → hashed MAC Address	6
	raw Serial → hashed Serial	3
	raw AndroidID → hashed AndroidID	20
Reverse	hashed hardware ID leak → raw hw. ID leak	35
	hashed IMEI → raw IMEI	20
	hashed IMSI → raw IMSI	3
	hashed MAC Address → raw MAC Address	7
	hashed Serial → raw Serial	5

several trends. First, the use of raw IDs has decreased across the board: notice the negative net figures for IMEI, Serial, MAC Address, AndroidID. Two groups saw an increase: AdvertisingID and GUID,[6] especially in own code, as well as hashed own code (AndroidID, Serial, MAC). These results, also corroborated by the app-centric study in Section 2.5.6, indicate (1) a move away from hardware identifiers and toward resettable identifiers, and (2) replacing raw with hashed values, which is encouraging.

App-Centric Study. The second part of our study is app-centric. Assuming the signature of an app in 2018 was S_{2018} while in 2020 the signature is S_{2020}, we check whether $S_{2020} <: S_{2018}$. We show how our notion of subsumption allows for flexible definitions, hence we can gauge, along several dimensions, whether the apps have become more privacy-friendly.

[6]We keep the '0' values for AdvertisingID and GUID in the table for uniformity; since these IDs were not used in the hashed form to begin with, there was no change.

We show the results in Table 2.11. We start with *AND subsumption*, e.g., $e \wedge s <: e \wedge s \wedge r$ indicates a reduction in hardware identifiers; we found that 108 apps exhibit this condition, which is encouraging as it means dropping the use of one or more hardware identifiers. When relaxing the subsumption notion to allow for increases in software IDs, we found a further 11 apps that exhibit this condition, which is still positive, as the use of software IDs is preferred to the use of hardware IDs. We also show the number of apps that drop each ID; IMEI and MAC Address are the most-dropped hardware identifiers (87 and 71 apps, respectively), while AndroidID was dropped by 107 apps.

Hash subsumption, e.g., $e \wedge h(ID) <: e \wedge ID$, indicates that the app has switched from leaking the raw ID to leaking the hashed ID. While few apps exhibit this subsumption (26 for hardware IDs, 20 for AndroidID), it is nevertheless a privacy gain.

Reverse subsumption. Finally, 35 apps were in the undesirable "reverse subsumption" situation: at least one hardware ID leak was added in the 2020 version. We show these findings in the last five rows of Table 2.11. Of the 35 apps that went from a hashed to a raw leak, the majority did so for the IMEI (20 apps), while fewer apps did so for the IMSI, MAC Address, and Serial, respectively.

To conclude, the longitudinal analysis reveals an overall move away from usage/leaks of hardware IDs, toward resettable IDs; and to smaller extent, a move toward hashed hardware IDs.

2.6 Summary

We introduce an algebraic taint representation that solves a key problem with existing taint analyses: distinguishing between programs that leak data in ways that are similar on the surface, but very different underneath. We implemented algebraic taint tracking as a static analysis for Android, and demonstrate its effectiveness through six

studies on identifier (ab)use in top Android apps and libraries. We found that being able to capture subtle yet critical differences is key for understanding app behavior w.r.t. user privacy or abiding by developer guidelines. Our longitudinal study shows that over the past two years, apps have become more privacy-friendly.

Now that we have demonstrated the effectiveness of precise representation of leak signatures involving device identifiers in capturing critical differences in app behavior, we will turn our attention to the topic of fingerprinting in the next chapter.

CHAPTER 3

IDENTIFIER SCHEMES BASED FINGERPRINTING

In this chapter, we tackle the challenge of automatically extracting and understanding fingerprinting schemes used for uniquely identifying mobile users or devices in mobile development. To achieve this, we introduce identifier processing graphs (IPGs) that capture fingerprinting mechanisms concisely, allowing for high-precision, high-recall dynamic taint analysis. Through our study, we analyze and break fingerprinting schemes in 436 Google Play apps, and we demonstrate how automated subversion of these schemes enables us to obtain free or discounted offers and credits. Our findings shed light on the vulnerabilities of fingerprinting schemes and their financial implications.

The identifier processing graphs (IPGs) are a novel, graph-based encoding of identifier processing to streamline the extraction process in Section 3.2. We present our systematic and scalable approach for breaking fingerprinting schemes including a re-registration attack using ReOffer and network traffic injection in Section 3.3. Our experiments reveal vulnerabilities. The evaluation of 436 apps shows a 100% success rate in breaking fingerprinting schemes (Section 3.4). Finally, we discuss strategies to hinder attackers and offer improvement suggestions for developers, vendors, app markets, and users based on our findings in Section 3.5.

3.1 Overview

We start with a brief discussion of unique identifiers in Android (Section 3.1.1), followed by our threat model (Section 3.1.2), then our workflow (Section 3.1.3).

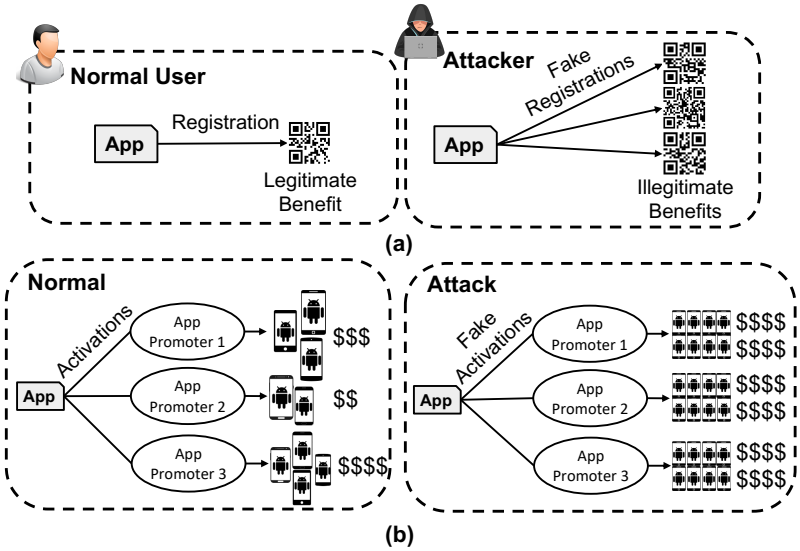


Figure 3.1 (a) Offer re-generation; (b) fake device activation.

3.1.1 Unique identifiers

App developers and businesses with a mobile presence have an interest in fingerprinting users or devices via unique identifiers. We call these interested entities "fingerprinters." A wide range of identifiers (IDs) can be employed; depending on whether the ID is tied to the user, app, software installation, or hardware, it can survive various level of reset. However, the "deeper" the identifier, the more intrusive the fingerprinting, hence Google has provided ID usage guidelines for app developers, e.g., "Avoid using hardware identifiers"[14]. Generally, IDs are tied to either users or devices, so fingerprinting schemes fall into two categories:

Type 1: Registration-based: Fingerprinters use registration information (e.g., email address or credit card number) to identify a unique entity.

Type 2: Device-based: Fingerprinters identify a unique entity by using device IDs, e.g., IMEI, Android ID, serial number; no registration information is required.

Naturally, fingerprinters can also employ a combination of these identifiers, from both types. We investigate weaknesses (unsafe assumptions) in these uniqueness enforcement schemes in two scenarios, as discussed next.

3.1.2 Threat model

We analyze how fingerprinting mechanisms are applied in practice for discounts and promotions, i.e., to enforce "one device/user, one discount/promotion". In our model, *victims* are businesses that rely on a fingerprinting scheme to either 1) limit offers to a unique user/device, or 2) track unique device activations. *Attackers* aim to gain multiple unauthorized benefits or cause financial losses to app developers/vendors by breaking the fingerprinting scheme.

Scenario 1. For this scenario, illustrated in Figure 3.1(a), we consider mobile apps that give out initial offers to new users. The attacker breaks the "***one offer per unique user***" scheme, gaining multiple illegitimate offers. Our analysis has revealed various types of offers in terms of "redeemability". In the first type, the offer appears as a QR code or barcode in the app. Hence, an attacker generates codes repeatedly. In the next type, offers are shown as string codes or in-app GUI elements confirming offer eligibility; unlike the previous case, the offer can usually be redeemed by making an in-app purchase (e.g., delivery). In a few cases, the user needs to show the string code to the cashier, who then enters the code into the register to redeem the offer.

Attackers can profit from multiple offers in several ways, while reducing the risk of getting caught: codes can be sold to, and redeemed by, other users (besides the "original" offer recipient); or the attacker can use different codes in different stores. All these translate to lost revenue for the victim.

Scenario 2. For this scenario (Figure 3.1(b)), we consider the case where app developers/vendors distribute their apps through several channels such as third-party app *promoters* (or different app stores, named *distributors*). Promoters charge developers an amount proportional to the number of unique devices that have installed the app. Hence, developers need a precise mechanism to track the number of unique devices that have installed the app, to ensure that promoters charge fairly. The attacker aims to break the "***one credit per fresh install on unique device***"

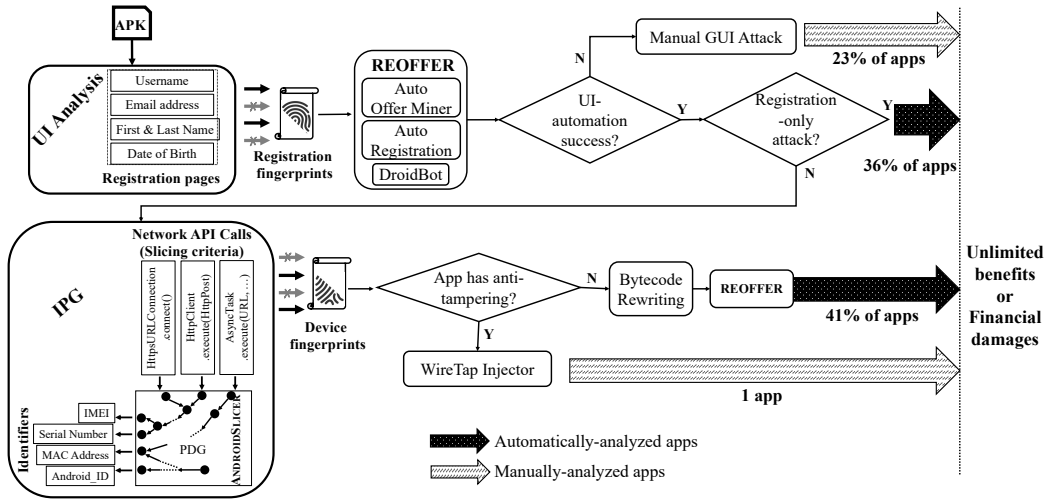


Figure 3.2 Methodology: Extracting the fingerprinting mechanism (left) and conducting the attack (right).

scheme, e.g., by deceiving the back-end servers via multiple fake activations (no real device installs their app); as a result, promoters receive illegal financial gains, while victims (developers/vendors) are charged abusively.

The main difference between these two threat models is that in Scenario 1, a single device can have multiple, unique users, whereas Scenario 2 allows only one user per device.

3.1.3 Workflow

Our attacks consist of two main phases: a) Extracting and categorizing the fingerprinting schemes according to the types introduced in Section 3.1.1; and b) Constructing and conducting the attack, which can give the attacker repeated benefits or cause financial damages.

Figure 4.4 presents our approach. For a given APK (Android app), our automated UI analysis checks whether the app requires a user account and finds registration pages (Section 3.3.1). Registration information is saved and ReOffer "mines" app offers. For 36% of our examined apps, ReOffer automatically regenerates offers via re-registration. For 41% of the apps, a re-registration attack alone is not enough, so

we leverage IPGs to understand how device IDs are combined and processed into a fingerprinting scheme (Section 3.2); then we use a combination of bytecode rewriting (Section 3.3.2) and ReOffer to automatically re-generate offers. For the remaining 23% of apps, ReOffer' UI automation is not powerful enough to bypass customized, sophisticated GUIs, or verification steps; in such cases we conduct the attack manually. There was a single app ('App_X') that employed anti-tampering, where bytecode rewriting was insufficient. For App_X we "injected" IDs into network traffic to subvert the fingerprinting scheme (Section 3.3.3).

3.2 Extracting and Categorizing the Fingerprinting Mechanism

Extracting the fingerprinting scheme is akin to "*finding the needle in the haystack*": we need to precisely identify relevant data- and control-flow in the app execution, from fingerprint construction to fingerprint sending.

Motivating Example. Consider, for example, extracting the fingerprinting mechanism for the rewards-offering, Penn Station Subs app.[1] Intuitively, we know that the app performs fingerprinting (i.e., accesses, processes, and sends identifiers), but we do not know *which identifiers*, *how identifiers are combined/processed*, and *how this identifying information is exfiltrated*. The most popular approach for exposing identifier leaks, taint analysis, can help with the "ends" but not with the crucial middle: taint analysis can at best expose leaks of individual identifiers, but not the intricate way in which identifiers are processed and combined into a scheme. Other disadvantages of taint tracking are imprecision (in Section 3.4.7 we quantify this inadequacy) and the risk of over-tainting aka "taint explosion" [105]. An alternative to taint analysis, dynamic dependence tracking, holds promise, but the resulting number of dependences can be overwhelming, since only a tiny fraction of dependences are

[1] `https://apkpure.com/penn-station-subs/com.ak.app.pennstation/download/18-APK`, Retrieved on DATE: 2023-06-01

germane to fingerprint construction. For example, dynamic slices can be in the range of 500–5,000 instructions for popular apps [89].

Insights. Our strategy is based on three insights:

1. Dynamic program dependence graphs soundly capture data- and control-dependences in a specific execution.

2. Fingerprinting information is constructed from registration and/or device information.

3. The fingerprint is sent to the fingerprinter's servers.

Based on these insights, we can construct the IPG-based fingerprint, as discussed next. (Figure 3.3 shows Penn Station Subs app's fingerprint, as extracted by our approach).

3.2.1 IPG: Definition and extraction

A dynamic program dependence graph $G = (V, E)$ is induced by control- and data-dependences between instruction instances (nodes v_i) as follows. Let v_i^k be an instruction instance, i.e., the k's executed instance of bytecode instruction v_i; without loss of generality we omit k in the subsequent presentation. A data-dependence edge $v_j \leftarrow_d v_i \in E$ is created when computation performed in v_i depends on values produced by v_j. A control-dependence edge $v_j \leftarrow_c v_i \in E$ is created when the decision to execute v_i is made by v_j, that is, v_j contains a predicate whose outcome controls the execution of v_i. Dynamic program dependence graphs are created via dynamic program analyzers that track dependences, e.g., slicers [89].

IPG construction as transitive reduction on G. We show the algorithm for IPG construction as transitive reduction on G in Algorithm 1. The algorithm takes as inputs the program dependence graph (PDG) of G; the slicing criteria (which in our

case is X, the set of network API calls); and the identifier APIs set U. The algorithm produces the IPG as output. We define D as the set of data dependence edges and C as control dependence edges (lines 3–4). Let $X = \{x_1, \ldots, x_j, \ldots, x_m\}$ be the bytecode instructions responsible for *exfiltrating the fingerprint*, e.g., when the fingerprint is sent over the network, these x's are network API calls. Let $U = \{u_1, \ldots, u_i, \ldots, u_n\}$ be the bytecode instructions responsible for *reading unique identifiers*, e.g., the API calls for retrieving the MAC address, or the IMEI. Naturally, X and U are subsets of V; let $V' = X \cup U$ (line 5) and $G' = (V', E')$ (line 10) be the subgraph of G induced by this restricted set of nodes; we calculate the restricted set of edges E' from the union of data dependence and control dependence edges that have program dependences on any of the APIs from the identifier APIs set U (line 8). The *transitive reduction* of G' is $G'_{rdx} = (V', E'_{rdx})$, that is, the subgraph of G' with the minimum set of vertices so that if there exists a path from u_i to x_j in G', that path exists in G'_{rdx}. Therefore we add a new edge (x_j, u_i) to edges set E'_{rdx} of graph G'_{rdx} (line 15). Since dependence paths point "backwards" from x_j's to u_i's, we transpose G'_{rdx} to obtain a natural, ID $\rightarrow$ exfiltration, flow. To conclude, we define the IPG as $(G'_{rdx})^T$, the transpose of G'_{rdx} (line 20), i.e., the graph capturing the crucial paths from ID-retrieving to processing/combining to exfiltration, $u_i \rightarrow p_m \rightarrow x_j$. Consequently, IPGs are concise: typically 3–8 vertices and 2–7 edges per app.

3.2.2 Constructing precise and effective IPGs

Our approach first extracts program dependences via dynamic slicing (the AndroidSlicer tool [89]). Given a slicing *criterion* that defines the property of interest (we explain next how we achieve this), a dynamic slicer captures program traces, containing control- and data-dependences, then reduces the dependence to a small set of executed instructions. This reduction typically leaves just 0.3% of executed instructions [89] for

Algorithm 1 Constructing the IPG

Input: Program dependence graph (PDG), Slicing Criteria=X, Identifier APIs=U
Output: IPG (Transitive reduction graph, G'_{rdx}=(V', E'_{rdx}))

procedure IPGCreation(*PDG, X, U*)
    $V \leftarrow$ X
    $D \leftarrow$ DataDependence(PDG)
    $C \leftarrow$ ControlDependence(PDG)
    $V' \leftarrow X \cup U$
    $E' \leftarrow \emptyset$
    for each node n in V **do**
        $E' \leftarrow E' \cup$ calculateEdgesSET($(D_n \cup C_n)$, U)
    end for
    $G' \leftarrow (V', E')$
    $E'_{rdx} \leftarrow \emptyset$
    for each x_j in X **do**
        for each u_i in U **do**
            if IsConnected(x_j, u_i, G') **then**
                $E'_{rdx} \leftarrow E'_{rdx} \cup$ CreateEdge(x_j, u_i)
            end if
        end for
    end for
    $G'_{rdx} \leftarrow (V', E'_{rdx})$
    $G'_{rdx} \leftarrow$ Transpose(G'_{rdx})
end procedure

further processing; while a 99.7% reduction is substantial, for popular apps this can still leave hundreds or thousands of instructions to be analyzed.

We present an example of an IPG reduction graph for the Penn Station Subs app in Figure 3.3. On the left side of the figure, we present a small portion of the program dependence graph, consisting of nodes representing bytecode-level instructions. The edges between these nodes capture data-flow dependences (red arrows) and control-flow dependences (black arrows). In practice this graph typically consists of thousands of nodes and edges, most of which are not relevant to fingerprinting. The black-background boxes highlight nodes of interest, namely those representing Identifier APIs, intermediate processing of identifiers, and network APIs. We identify these intermediate processing nodes by tracing data flow edges from the identifier

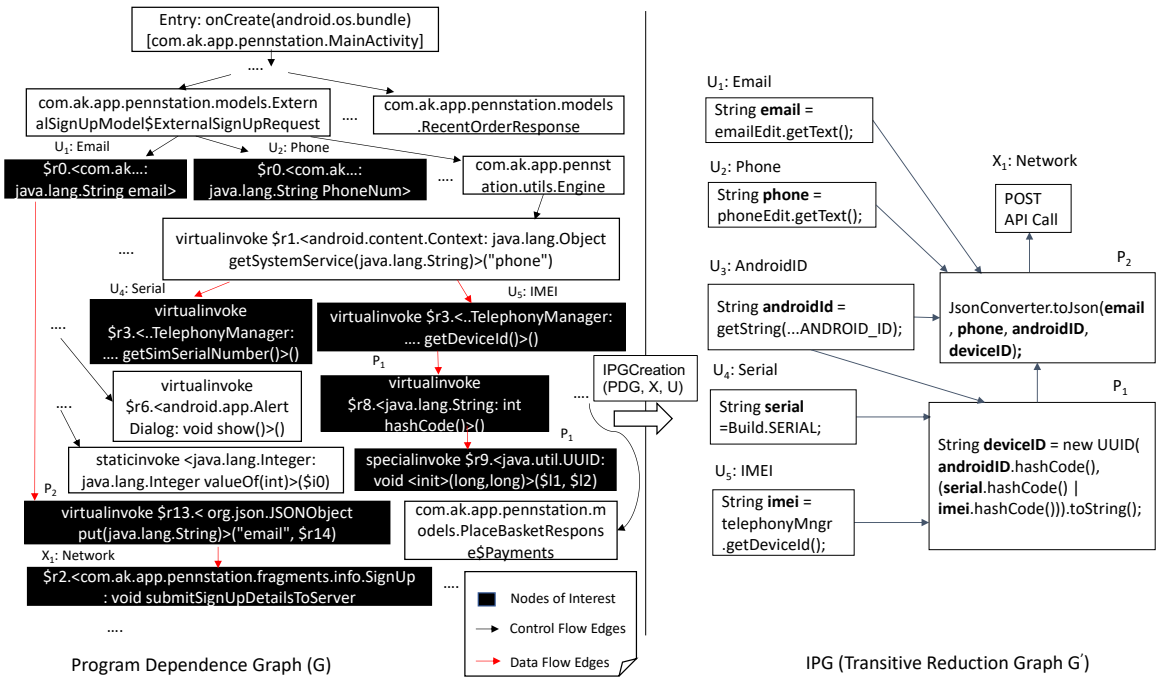


Figure 3.3 PDG reduction to IPG for the ***Penn Station Subs*** app.

APIs to other PDG nodes. The nodes of interest and the edges connecting them are then translated (using Algorithm 1, IPGcreation) into a precise IPG subgraph derived from this PDG, which originally contained numerous instructions.

To specify the slicing criteria that maintain soundness but increase effectiveness, we need to understand where uniqueness checks originate; we observed that all uniqueness checks occur on servers, hence, slicing should start at the point where the app puts the fingerprint onto the wire (the x_j's). This initial app–server communication takes place after app start (e.g., after launching the app, or as part of registration/login); therefore our insight is that slicing criteria should include the start and registration communication. We specify Android Framework's network API as criteria; Figure 4.4's IPG box (bottom left) contains examples of such API calls. Note that a pre-defined API list is not a limitation, as other API calls can simply be added to the list. As a result, after we run the app in AndroidSlicer with the aforementioned slicing criteria, the program slices contain *fingerprint-relevant code and IDs*.

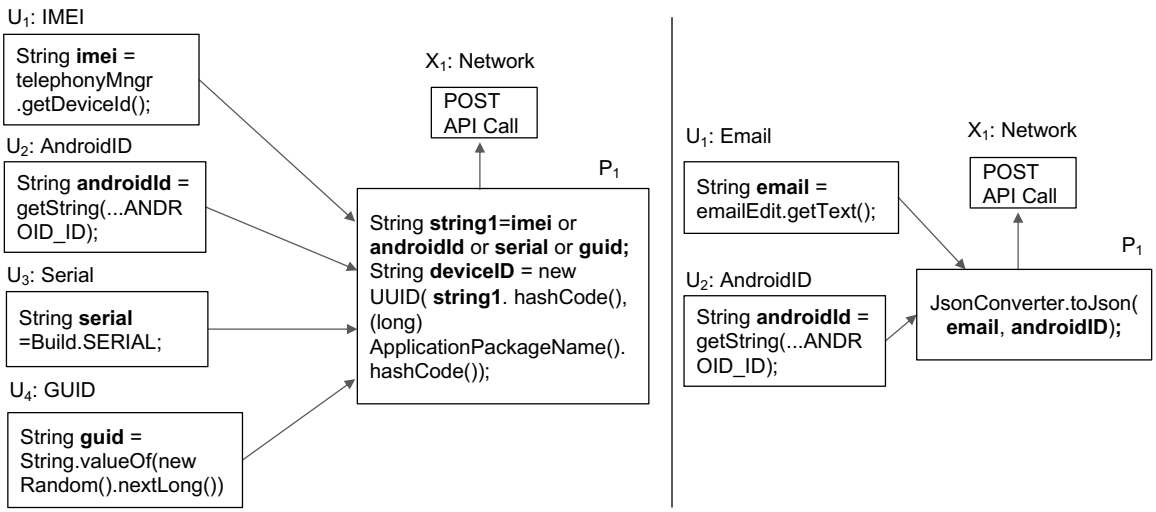


Figure 3.4 Fingerprint scheme in the ***Fazoli's Rewards*** app (left) and ***Pita Pit*** app (right).

In the third step, we compute the IPG via transitive reduction on the slicing-exposed relevant dependences using Algorithm 1. In Figure 3.3, the IPG is the right-side graph, computed from the left-side program dependence graph. Figure 3.3 (right) shows the scheme for the Penn Station Subs app: it consists of vertices connecting ID-retrieving nodes (u_i, e.g., Android ID), with nodes that process these IDs (p_m, e.g., via hashing), and nodes that send the fingerprint over the network (x_1, e.g., POST). Note how this fingerprint is *concise yet effective*; it is also *sound* because registration must involve server communication and transitive reduction preserves reachability.

We show two additional examples of extracted fingerprinting schemes and their corresponding IPGs in Figure 3.4. On the left, the IPG for app Fazoli's Rewards shows which identifiers are used (IMEI, AndroidID, Serial Number, and GUID); and how they are processed, in this case hashed, before being leaked onto the network; the IPG has just 6 vertices and 5 edges. On right, the IPG for the Pita Pit app, shows how the source identifiers (Email and AndroidID) are combined and converted into a key-value pair JSON object, which is then leaked onto the network; the IPG has

Table 3.1 Android Identifiers' Persistence

Persists after	IMEI	IMSI	Serial	MAC	AndroidID	Advert.ID	GUID	Registration Info
App restart	✓	✓	✓	✓	✓	✓	✓	✓
App reinstall	✓	✓	✓	✓	✓	✓	✗	✗
Factory reset	✓	✓	✓	✓	✗	✗	✗	✗

just 4 vertices and 3 edges, illustrating the conciseness and effectiveness of our scheme extraction approach.

Comparison with existing dynamic analyzers. In contrast with our precise IPG representation, existing dynamic analyzers suffer from program dependences explosion, and traditional taint trackers are subject to over-tainting. We compare our toolchain with existing approaches in detail in Section 3.4.7.

3.2.3 Identifiers used in practice

Our study has revealed four categories of unique identifiers used by apps for fingerprinting.

(1) Hardware IDs. The IMEI/MEID uniquely identifies mobile phones, whereas the IMSI identifies the cellular network subscriber. The MAC address uniquely identifies the network interface controller. The Serial Number, only present in certain devices, is manu-facturer-assigned and unique.

(2) Software IDs. Android ID [13], the most popular software ID in our analysis, uniquely identifies an app or app group: it is a 64-bit identifier, constructed from a combination of APK signing key [20], user, and device.[2] Android also provides a random, instance-scoped ID, named GUID (Global Unique Identifier) [14].

(3) Advertising ID. Google Play Developer Policy[3] mandates the use of Advertising IDs for advertising purposes (users can easily reset this ID). The policy

[2]This is the Android ID semantics in Android 8.0 (API level 26) and later, and the semantics used in this dissertation.

[3]`https://play.google.com/about/monetization-ads/ads/`, Retrieved on DATE: 2023-06-01

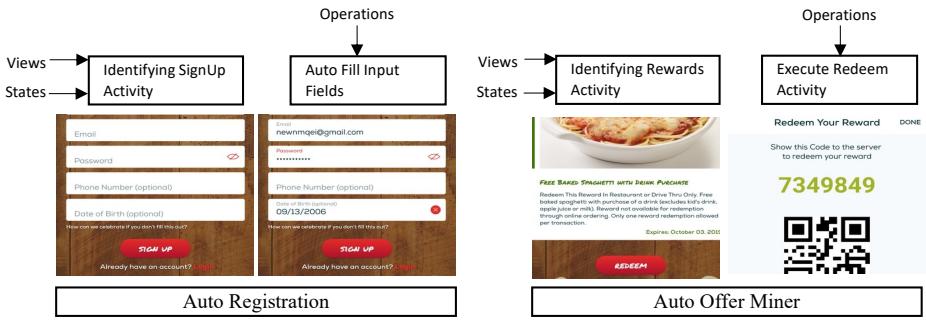


Figure 3.5 ReOffer: workflow (top), tool in action (bottom).

also requires that the Advertising ID not be connected to any personally identifiable information (PII) or persistent (hardware) identifier.

(4) Registration-based IDs. Fingerprinters can also leverage user registration information (e.g., email or phone number) as identifier. This information is not tied to devices or software; rather, it is loosely tied to individual users. Credit cards are "tighter" registration-based identifiers. Section 3.4 shows that registration-based information is implemented insecurely in many apps, enabling offer abuse.

We categorize the IDs observed in our examined apps, based on ID persistence, in Table 3.1. For example, the IMEI, MAC, and Serial survive factory reset, while Android ID survives app reinstalls but changes upon a factory reset. To balance usability (e.g., asking users to provide additional IDs such as emails) and ID persistence, fingerprinters combine these IDs, or create their own custom identifiers. Note that Google provides privacy-focused guidelines for selecting appropriate IDs depending on use [14]: resettable IDs are preferred, hardware IDs should be avoided. Google warns developers that "fraud prevention requires proprietary signals" and hardware IDs can be spoofed. Hence, the guidelines per se do not offer protection against (financially motivated) attacks on fingerprinting.

3.3 Constructing and Conducting the Attack

UI analysis reveals the registration information involved in fingerprinting (*registration-based*), whereas IPGs reveal the unique identifiers and the associated relevant code

(*device-based*). Therefore, we consider three fingerprinting categories: (a) registration-based, (b) registration- and device-based, and (c) device-based. Next we discuss how we conduct the attacks with respect to these categories.

For category (a), we use our automated ReOffer tool (Section 3.3.1) to register multiple fake accounts, hence, gain multiple offers. For categories (b) and (c), that build the fingerprint on device IDs (alone or coupled with registration), we employ *device masquerading*: generating and sending fake values for relevant device IDs. Masquerading can be achieved in many ways – rewriting app bytecode, instrumenting the network layer (wiretap injection), instrumenting the Android Framework, etc. We use two such techniques: *Bytecode Rewriting* and *Wiretap Injection*. For bytecode rewriting, we modify the fingerprinting methods' return values (Section 3.3.2).

When bytecode rewriting is not possible, e.g., an app employs anti-tampering, we use our "Wiretap Injector" module to send fake IDs to servers (Section 3.3.3). We combine masquerading with ReOffer to automate the attacks on schemes that use both registration- and device-based fingerprinting. Section 3.3.4 shows an end-to-end automated attack example on the app *Sixty Vines* [59].

3.3.1 Re-registration

Fingerprinting analysis. In this category, apps use registration information alone as the fingerprint. Users first register, e.g., by providing their email address, phone number, username, password, name, date of birth. Once the user is logged in for the first time, the app makes the offer available.

Attack. Regardless of offer type, e.g., QR code or barcode, since the offer is only tied to the user account, an attacker can register again and again using the same device, by creating different accounts; therefore redeeming offers again and again. We have automated the registration and sign-up processes for such apps using our tool, ReOffer (discussed next). Using ReOffer, an attacker can create as many

accounts as desired on the same device, as a new code (QR, barcode, or in-app code) will be generated every time.

ReOffer Design. This section presents our automated ReOffer (Offer ReGenerator) tool. The ReOffer box in Figure 4.4 shows the major components. As input, ReOffer takes the app (APK file), automatically logs-in using fake credentials and redeems offers repeatedly, at scale. Next, we describe ReOffer's components.

Input generator. ReOffer uses DroidBot [154] as input generator to automatically fill-in registration information and navigate app pages to complete the registration. DroidBot's GUI-based model enables ReOffer to identify various screens, such as sign-up and offers' pages, and different views, such as username and password fields. Three main components in DroidBot are `events`, `states`, and `views`. An `event` triggers the transition between different `states`. A `state` contains `views` and is represented by a foreground Activity (screen). Finally, a `view` is any user interface element such as a TextView or an ImageView. We use UI-based scripts (described next) on top of DroidBot to direct the transitions. The main challenges when dealing with automated UI-based analysis are (1) identifying the relevant screens (e.g., sign-up page) and (2) filling out the relevant views (e.g., email address) in those screens properly as shown in Figure 3.5. We apply the UI-based scripts in two stages: Automatic Registration and Automatic Offer Mining, as discussed next.

Automatic registration. The first step in guiding app execution to the "redeem offer" or "show offer" page is registering/logging into the app. Apps use a wide variety of UI models, which complicates scalable, automatic registration. We extended the DroidBot model by attaching auto registration scripts to its main exploration model. As discussed previously, the first challenge is to identify the relevant screens. We found that view attributes make effective indicators of whether a view is registration-related or not. These attributes include the element's `text`, `resource-id`, `content-desc`,

`checkable`, `checked`, and `password`. The first three attributes indicate registration relevance for a majority of elements such as username, email, phone number, name, ZIP code, etc. The next three attributes are good indicators for widgets, e.g., checkboxes, passwords, and their status. For example, the `checked` attribute's value indicates the status of a checkbox. The `password` attribute is a unique indicator of password fields. While the aforementioned strategy covers the majority of relevant elements, there are some cases in practice where this information is not enough (due to custom UIs). For example, an app might use an ImageView with two states: On (visible) and Off (invisible) as a checkbox. For these cases, we leverage coordinate (`bound`) and `size` attributes.

For the second challenge – filling out the relevant views – we have extended DroidBot's model with a state machine of operations per screen (it originally supported one state, i.e., one operation per screen). We have also automated the sign-up (input filling) portion by appropriate random values for each parameter.

Automatic offer mining. Once ReOffer has logged into the app, it guides the input generator to the "Redeem Offer" page (which contains a QR code, barcode, text, or digit code). The underlying mechanism is the same as for auto registration. Here, ReOffer looks for offer-related UI elements using the aforementioned set of attributes (`text`, `resource-id`, `clickable`, etc.). For example, ReOffer looks for view objects whose `resource-id` or `text` are offer-related, and their `clickable` attribute is true. After navigating to the "Redeem Offer" page, the generated QR code or barcode is saved as a screenshot, so it can be redeemed (Section 3.1.2).

3.3.2 Bytecode rewriting

We now discuss our approach for identifying relevant device IDs and generating fake ID values by rewriting app bytecode.

Fingerprinting analysis. For device-based fingerprinting, the app collects device information, usually for the purpose of limiting offers to new users or new devices.

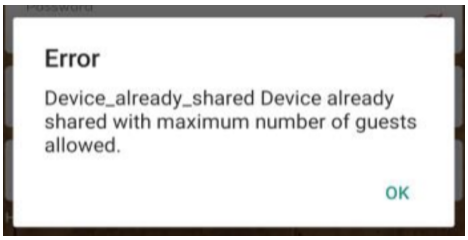


Figure 3.6 App restricting the number of user accounts.

This scheme can be used by itself or combined with registration-based fingerprinting. Figure 3.6 shows an example of a combined case – the Fazoli restaurant chain's Rewards app:[4] the app throws an error when the user attempts to create multiple accounts using the same device but different user registration data. When subverting device-based fingerprinting, our approach (1) detects the set of identifiers that are involved in fingerprinting, and (2) conducts the attack, by replacing original values with fake generated values, then sending these to "fool" the servers into accepting a fake activation. The *Bytecode Rewriting* module (Section 3.1.3) replaces identifiers with forged values.

Attack. We illustrate our attack methodology by showing successful attacks on two restaurant chain apps.

Case Study: Fazoli's Rewards apps. To enforce one-per-user restrictions, the app saves the fingerprint in local storage and sends it to the server for verification. Figure 3.7 shows the app code for unique ID generation. The app first checks whether a local file exists or not. If the file exists, meaning the ID has already been created (line 3), it returns the string in the file as Device_ID (line 6). Otherwise, it creates a new UUID (lines 11–22), stores it in the file (line 23), and returns the value as Device_ID (line 24). Lines 11–22 show device ID information (depending on availability, this can be the IMEI, the Android ID, the serial number, or a random value) being hashed to form a new UUID and saved in a local file to be retrieved later. Using the *Bytecode*

[4] `https://play.google.com/store/apps/details?id=com.punchh.fazolis`, Retrieved on DATE: 2023-06-01

```
1  synchronized(f.class) {
2      File file = new File(context.getFilesDir(),"CONSTANT_INSTALLATION_APP_GENERATED_ID");
3      if (file.exists()) {
4          String a = a(file);
5          g.a(context).a("DEVICE_ID", a);
6          return a;
7      }
8      TelephonyManager tm = (TelephonyManager)context.getSystemService("phone");
9      String aid = Secure.getString(context.getContentResolver(), "android_id");
10     long acontext = (long)context.getApplicationContext().getPackageName().hashCode();
11     if (tm.getDeviceId().toString() == ""){
12         if (aid.equalsIgnoreCase("9774d56d682e549c"){
13             if ((TextUtils.isEmpty(Build.SERIAL))
14                 uuid = new UUID((long) (String.valueOf(new Random().nextLong())).hashCode(),
                        acontext);
15             else
16                 uuid = new UUID((long) (Build.SERIAL).hashCode(), acontext);
17         }
18         else
19             uuid = new UUID((long) (aid).hashCode(), acontext);
20     }
21     else
22         uuid = new UUID((long) (tm.getDeviceId()).hashCode(), acontext);
23     g.a(context).a("DEVICE_ID", uuid);
24     return uuid;
25 }
```

Figure 3.7 Fingerprinting in ***Fazoli's Rewards*** app.

Rewriting module, we replace the identifier with fake values to enable multiple fake registrations on the same device. Specifically, we decompile the app and replace the return values in the Smali (Android) bytecode by a hard-coded UUID of our choice. Figure 3.8 shows the code after replacement; the replacement allows us to forge the actual Device_ID and generate a new fingerprint.

Case Study: Texas Roadhouse Mobile app.[5] Figure 3.9 shows the app's fingerprinting code. The app does not store the fingerprint in a file, instead calculating the device ID by combining the IMEI, IMSI, and Android ID, as shown in getDeviceID()'s return statement (line 6). To bypass this fingerprinting, we modified line 3's bytecode, replacing the on-the-fly constructed value with a hard-coded UUID. For both case study apps, we leveraged ReOffer for automated re-registration. As a result, the apps do not show an error anymore, and we receive the offer again.

[5] `https://play.google.com/store/apps/details?id=com.relevantmobile.texasroadhouse`, Retrieved on DATE: 2023-06-01

```
1   ...
6   return "00000000-6950-bd75-0000-00007a908258";
... ...
12  return "00000000-6950-bd75-0000-00007a908258";
```

Figure 3.8 Bypassing fingerprinting in ***Fazoli's Rewards***.

```
1  public String getDeviceID(Context context) {
2      TelephonyManager tm = (TelephonyManager)context.getSystemService("phone");
3      int androidId = Secure.getString(context.getContentResolver(),"android_id").hashCode();
4      int simserial = tm.getSimSerialNumber().toString().hashCode();
5      int imei = tm.getDeviceId().toString().hashCode();
6      return new UUID((long) androidId, ((long) simserial | (long) imei << 32)).toString());
7  }
```

Figure 3.9 Fingerprinting in ***Texas Roadhouse Mobile***.

As these two apps belong to app "families" (PunchhTech and Relevant Mobile, respectively, see Section 3.4.1), each of these case studies is representative of conducting an attack *on the entire family*. In our experiments for similar device-based fingerprinting apps (within these apps' families, and beyond), replacing return values with hard-coded ones successfully subverted fingerprinting.

3.3.3 Wiretap injector

We now describe our wiretap injection attack, for apps that employ anti-tampering (which hinders bytecode rewriting).

Fingerprinting analysis. Similar to the previous case, the main challenge here is detecting the set of identifiers and devising an automated process to replace original values, by replying with fake generated values, to trick the servers into accepting fake activations.

Case Study: 'App_X'. We studied the app of a popular Chinese company focused on retail and online group buying. For security reasons, we anonymize the company name by the term 'App_X'. We established collaborations with the company which gave us the unique opportunity to peek into the activation results at their backend and get ground truth for our analysis/attack.

As discussed in Section 3.1.2, in this scenario, app vendors (here, App_X) distribute their app through multiple channels, e.g., different app stores or third-party app promoters. Promoters, who charge App_X-like vendors for promotion services, compute the charged amount based on the number of unique devices that have installed the app. Hence, a precise mechanism to track the number of devices that have installed the app is very important for such vendors. We analyzed and subverted the mechanism used by App_X. Our automated fingerprint extraction has revealed that multiple IDs are sent to the server via an elaborate scheme. Specifically, the app sends the IMEI, Android ID, serial number, MAC address, brand, model, etc. over the network during the first communication with the server. For retrieving the IMEI, the app needs a specific permission. If the permission is not granted, App_X has its own algorithm (combining hashes of parameters such as Android ID, serial number, MAC address, brand) to form a 15-character device ID similar to IMEI.

Attack. With the IDs and relevant fingerprinting code revealed by IPG extraction, we fake the parameters using network packet replay: the *Wiretap injector* module (Figure 4.4) forges network packets containing device info to contain real or fake device info, depending on the configuration. The reason for sending real parameters in some configurations and fake parameters in other configurations is that checking takes places on the server, hence, App_X's activation acceptance logic is unknown (and practically unknowable) to us. Therefore, we analyze the vulnerability of the system as a blackbox.

IPG analysis on network API calls has revealed the App_X protocol, shown in Figure 3.10. Three back-to-back network packets containing the above IDs are sent to App_X servers in the first moments of running the app, indicating that the servers compute the number of unique device installations based on the parameters received over the wire. The response packets, from *register* and *report* servers, always contain a UUID value and a successful status code, meaning that the server (regardless of

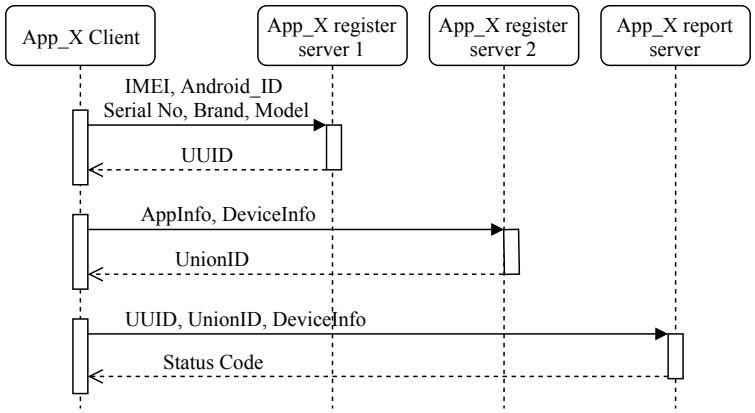


Figure 3.10 App_X network packet sequence diagram.

whether parameters sent from the phone are unique or not) always assigns a new UUID to each installation and the unique device installation acceptance logic is separately computed on the server. Therefore, based on server responses alone, *the client cannot distinguish between a successful and a failed activation.* Because of this, we asked the App_X company to verify which activation attempts were successful.

3.3.4 End-to-end automated attack example

To provide an overview of our automated approach and the rationale for each phase, we present an example end-to-end attack in Figure 3.11. First, we compare other existing solutions (Figure 3.11(a) and (b)) with ours, highlighting their inadequacy in effectively countering fingerprinting. Subsequently, we illustrate how our approach (Figure 3.11(c)) combines these solutions to successfully bypass fingerprinting.

One solution, shown in Figure 3.11(a), would be to use just a customized version of DroidBot to allow re-registration; however, this approach would fail to actually repeat the offer generation due to the *one offer per user* limitation, and would just lead to the error message shown in the figure (or a similar error message). Another solution would be to use slicing, shown in Figure 3.11(b), to attempt identifier extraction. However, slicing alone would be inadequate, since fingerprinting extraction cannot

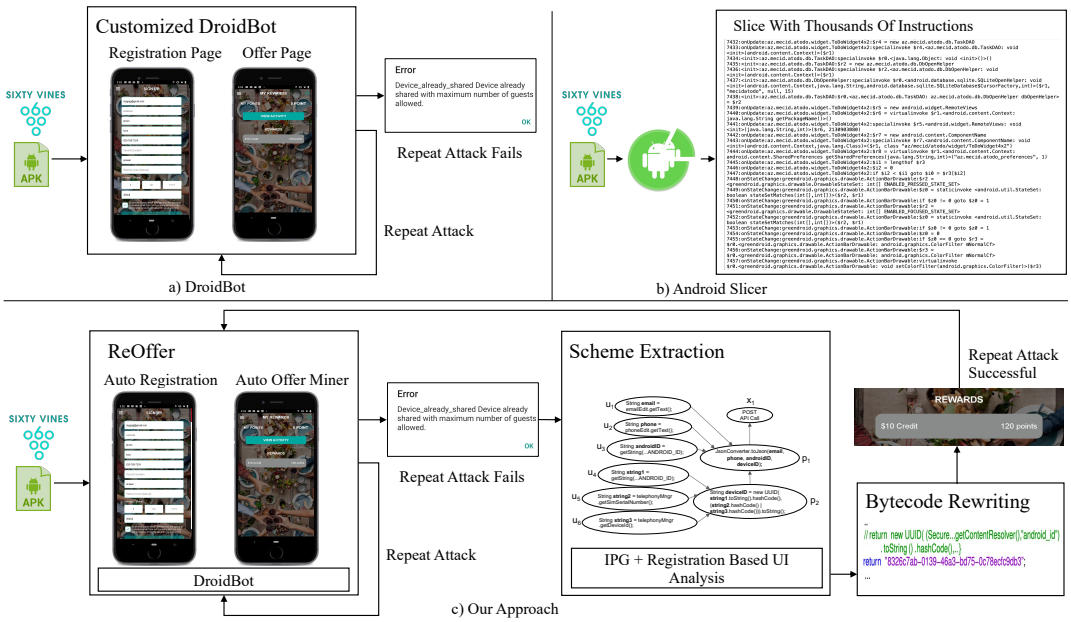


Figure 3.11 a) DroidBot-only attack; b) Ineffective identifier extraction via dynamic slicing; c) Complete automated attack using REOFFER on app *Sixty Vines*.

be achieved effectively: slices, even when specifying precise criteria, might include *thousands of instructions*, as shown in the figure.

In contrast, Figure 3.11(c) shows our approach: the flow of the attack on an example app, *Sixty Vines* [59]. Given the APK, first, we perform re-registration as discussed in Section 3.3.1. After the offer is generated ($10 in this case), the app checks whether the device is already registered on their servers, to enforce the one-offer-per-user limitation. Hence, repeating the attack fails with the error message as shown in the figure. This app falls under the registration- and device-based fingerprinting category. Therefore, we first use IPG-based fingerprinting extraction as discussed in Section 3.2.1. Then, we leverage bytecode rewriting as explained in Section 3.3.2 which leads to successful, repeated generation of the $10 credit.

3.4 Evaluation

We now present the experimental results of extracting and attacking Android fingerprinting schemes on a large dataset of apps collected from Google Play, in

accordance with our threat model (Section 3.1.2). We first introduce our dataset, followed by the categorization of apps based on the type of offer they provide and the type of identifiers used for detecting unique users in those apps. We then demonstrate the effectiveness of our attack in successfully redeeming different offers, the effectiveness of the IPG representation, and the overall effectiveness of our toolchain by comparing it with existing dynamic analyzers and taint trackers.

3.4.1 Dataset

As discussed in Section 3.1.2, businesses that provide offers assuming fingerprint-centered uniqueness are vulnerable to fingerprint scheme subversion. To ensure broad coverage, we chose Google Play apps using several criteria: (1) *popular*, by #installs, apps in the Food and Drink category; (2) apps of restaurant chains with *high sales figures*; (3) specific types (*families*) of apps with initial offers; and (4) *other potential offer-related* apps with relevant keywords in their Google Play description. We now discuss each criterion.

Popular Food and Drink Apps. We analyzed the top-200 (top-grossing and highest #installs) Food and Drink apps; we found that 38 out of 200 apps provide some types of initial offers for new users.

High Annual Sales Apps. We analyzed top-200 apps ranked based on their companies' most recently available (2018) sales [23]: 47 out of 200 apps have initial offers.

App Families (Loyalty Solutions). Our analysis of popular and high-sales apps has revealed that many apps share the same developer. These developers are actually "loyalty solution" businesses providing loyalty programs (e.g., initial offers for new users). We studied seven such families. The set of apps induced by the same loyalty solution is mass-exploitable: the common scheme among those apps can lead to low-effort, massive exploitation (Section 3.4.4). Within a family, different apps

Table 3.2 App Dataset for Our Study

Category	Apps in category	Apps with initial offers	Offer available on website
Top Google Play	200	38	9
Top Yearly Sales	200	47	4
Families			
LevelUp Cons.[15]	100	74	2
Paytronix Sys.[17]	100	53	42
PunchhTech[18]	56	39	18
Relevant Mobile[19]	10	5	0
TapMango Inc[21]	100	70	0
Thanx[22]	44	40	0
Total Loyalty S.[24]	100	97	0
Others	100	15	0
Total		*436*	*67*

have different offer types, from free food to credits/discounts to loyalty points which can be used to get free items. Some families (e.g., Total Loyalty Solutions and Thanx) publish apps in various Google Play categories, e.g., Shopping and Lifestyle. We found that most offers can be redeemed without any up-front expense; for a small fraction of offers, the user has to spend some threshold amount to get the discount, e.g., $10 off user's first purchase. For apps with loyalty points as initial offers, points can be accumulated via subsequent purchases; once the points reach a certain threshold, the user can redeem them. In our evaluation, we only consider apps that offer free products, or discounts as their *initial offers*; we do not consider apps with initial points as they might not be profitable targets.

Other Initial Offer-related Apps. To extend our coverage, we used a Google Play scraper to find additional offer-related apps. Specifically, we selected apps whose descriptions met the following criterion: a word from set A is followed (up to a distance threshold) by a word from set B:

A=[sign up, sign-up, signing up, signing-up, new, first]

B=[free, discount, off, offer, reward]

While we found 100 candidate apps, 85 overlapped with apps from previous criteria; we put the 15 new apps in the "Others" category.

Table 3.3 Most Common IDs

Scheme	**Registration-based**					**Device-based**				
Top-5 IDs	Email	Password	Name	Phone#	C.Card#	And.ID	IMEI	Serial#	IMSI	Mac
Apps using the ID	97%	83%	75%	53%	41%	36%	14%	12%	4%	2%

3.4.2 Categorization

Offer categories. Table 3.2 shows the dataset of our study. We found 436 unique apps that have some types of initial offers (note that there is some overlap between categories). The fourth column shows that most of these initial offers are only available or redeemable via a mobile device and not through their corresponding websites; in the majority of cases, the website counterpart does not exist. Where websites do exist, they usually direct users to the mobile app for registration or offer redemption. We also found cases where the offer was higher in the app than on the website (e.g., *Wildflower* app in the LevelUp Consulting family). In total, only 67 out of 436 apps (17%) have offers available on their website as well.

Identifier Categories. Table 3.3 shows the most common IDs used in fingerprinting. Apps in the registration-based fingerprinting category ask for *email* in 97% of the cases and *password* in 83% of the cases. Other registration information collected by apps are user's *name* (*First Name*, *Last Name* or *Full Name*), *phone number* and *credit card*. On the other hand, for apps in the device-based fingerprinting category, the top-most used unique device identifiers are *Android ID* and *IMEI* (36% and 14% of the cases, respectively). Other top device IDs used for fingerprinting are *Serial Number*, *IMSI*, and *MAC Address*.

3.4.3 Ethical considerations

Before presenting the evaluation results, we discuss the measures we took for ethical experiments. When verifying multiple redemptions of one-time offers (Section 3.4.4), we made sure to conduct the experiments ethically. For each offer, we made only one real purchase (as intended by the promotion). We then verified that the offer can

Table 3.4 Core Results. Regeneration Success Rate was 100%, i.e., All Apps' Fingerprinting Schemes were Subverted

Category	# Apps	Redeemed		Unique identifiers							ReOffer		Manual GUI attack
		In-app	In-store	Email	Phone#	C.Card#	IMEI	And. ID	Serial #	GUID	Alone	+Bytecode rewriting	
Top Google Play	28	10	18	28	15	1	6	10	6	6	5	10	13
Top Yearly Sales	15	15	-	15	8	-	5	5	5	5	7	5	3
Families													
LevelUp Consulting	74	74	74	74	-	-	-	74	-	-	-	74	-
Paytronix Systems	53	-	53	53	53	-	-	-	-	-	35	-	18
PunchhTech	39	-	39	39	39	-	39	39	39	39	-	39	-
Relevant Mobile	5	-	5	5	5	-	5	5	5	-	-	5	-
TapMango Inc	70	-	70	70	70	-	-	-	-	-	-	-	70
Thanx	40	-	40	40	-	40	-	40	40	-	-	40	-
Total Loyalty S.	97	-	97	97	-	-	-	-	-	-	97	0	-
Others	15	5	10	15	1	-	2	2	2	2	3	2	10

be generated again by getting to the point where the price/item was confirmed, and stopped there. We believe that reaching this point is sufficient because once the offer is confirmed on the checkout screen, vendors typically have to honor the price as it is considered a *material claim* by the FTC [45].

For App_X, our analysis was performed in collaboration with the App_X company. The logic for accepting unique device installations in App_X is executed independently on the server. Consequently, if we were to rely solely on server responses, we would be unable to differentiate between a successful and unsuccessful activation. Therefore, we sent the computed unique deviceID from the app to App_X company for verification, determining which activation attempts were successful.

3.4.4 Attack effectiveness

We now evaluate the effectiveness of our automated scheme breaking approach. Table 3.4 shows the results. The first and second columns show app families and the number of apps with initial offers for new users, respectively. We were able to break the fingerprinting scheme for all these apps – in other words, the offer regeneration success rate was 100% – as explained shortly.

The "Redeemed" split column shows how offers can be redeemed. *In-App* redeemable offers are presented either as plain text confirming offer eligibility, or as

a digit code. Users redeem these offers via in-app purchases (e.g., order free food or apply the discount/code to their order). *In-Store* redeemable offers are presented as QR codes, barcodes, or digit codes. Users redeem the offers in-store, by showing the code to the cashier. In most cases, screenshots of QR codes/barcodes can be used by other users on other phones. We found that 406 apps use codes which can be redeemed in-store, while 104 apps have offers redeemable via in-app purchases.

The "Unique identifiers" grouped columns show the IDs used in each scheme (number of apps in a certain family/set using that ID). The columns capture each scheme concisely, e.g., the PunchhTech family makes use of all device-based IDs, whereas the Thanx family uses two device-based IDs and credit card information.

Attack techniques. The last three columns show the attack techniques required: ReOffer alone (36%), ReOffer and bytecode rewriting (41%), or manual GUI attack (23%). Apps which use device IDs, e.g., the LevelUp Consulting, PunchhTech and Thanx families, require ReOffer and bytecode rewriting. In contrast, Paytronix and Total Loyalty Solutions families use only registration information, hence, could be subverted by ReOffer alone. Finally, there were cases requiring a manual GUI attack (last column). The reasons that ReOffer cannot automate the attack in these apps are as follows: some apps use email or phone verification, hence, hindering ReOffer; apps in the TapMango Inc family use a *WebView* [25] which complicates identifying relevant screens/views automatically; and lastly some apps use highly customized UIs which complicate ReOffer's interaction. These failures are not a fundamental limitation in ReOffer, and would be overcome with additional engineering (Section 4.2.3).

Financial Losses. To show the financial implications of attacks on fingerprinting, we present potential financial losses in Table 3.5: the minimum, average, and maximum dollar value offered by apps in each family. In those cases where the dollar value was not explicit, we assigned free drink as a $2, free desert as a $3, and free food as a $5

Table 3.5 Financial Losses Statistics

Category	Dollar Amount			Qualitative Range
	Min	Avg	Max	
Top Google Play	2	3	5	free beverage–free food
Top Yearly Sales	2	2	5	free drink–free sandwich
Families				
LevelUp Consulting	2	3.2	10	free drink–free dessert
Paytronix Systems	2	3.8	10	free drink–50% off wine
PunchhTech	2	4	5	free mini blizzard–free pizza
Relevant Mobile	2	6.5	10	free drink–free meal
TapMango Inc	2	2.3	10	free topping–free ice cream
Thanx	2	10	25	free dessert–free car wash
Total Loyalty S.	2	4	10	free beverage–free movie tickets
Others	2	6.6	50	free drink–free car wash
Overall	2	5	50	

value. We chose these amounts based on the fact that many apps use similar amounts as the upper bound for such offers. Among the total 436 apps, in 131 apps, the offer (e.g., free sandwich or pastry) can be redeemed *without having to purchase anything*; for the rest, the offer is in the form of discounts (e.g., "buy one, get one free" or "X dollars off of first purchase"). The highest offer we observed was $25 for restaurants and $50 for groceries.

To conduct the "free meals and groceries, using just a single device" attack, we redeemed offers resulting from subverting fingerprinting schemes in 4 apps. Note that for ethical reasons, we performed only one real purchase, which has no ethical implications, as the offers are available to the general public (including the authors). For subsequent repeated orders, we got to the checkout point where the price/item was confirmed and stopped there to avoid repeated financial gains. We now present the attacks. The *Seamless* delivery app (>1M installs), offers discounts for first time in-app orders of newly registered users. We were able to register multiple user accounts on the same mobile device. For verification, we ordered food one time and verified that $7 and $8's worth of food for pickup is ready, for free, for next times. In the *Del Taco* app, weak fingerprinting allows receiving the initial offer, two free tacos, multiple times. The offer is limited to one per device, hence, simply re-installing the app issues an "already redeemed offer". Using bytecode rewriting, we enabled the offer repeatedly; we ordered the free tacos (original value: $2) one time and confirmed the

free tacos are ready to pickup for next four attempts. *Jamba Juice* offers $3's worth of free food per unique user; with bytecode rewriting we could re-enable the offer. Each time, despite using a deliberately invalid credit card number (only for profile creation), the app still issued the offer. For the *Delivery.com* app, we got $11's worth of groceries ($17 worth of groceries minus a $6 delivery fee) one time and verified the next attempts by getting to the point that items/offers are confirmed. To conclude, an attacker can get free meals and groceries by leveraging the fingerprinting schemes' weaknesses in these apps.

3.4.5 App_X

As shown in Figure 3.1b, unlike the apps with initial offers, the scenario in App_X is different: the uniqueness test is performed silently on the server and is not disclosed to the app or user (all financial exchanges occur in the backend). We had the opportunity to collaborate with the App_X company, hence, we could check whether each activation in our experiments was successful or not.

We subverted the fingerprinting scheme via wiretap injection (Section 3.3.3) on App_X. Experiments were performed on real Android phones, as follows. We used real device info as a base and unilaterally, then multilaterally, modified hardware IDs (*IMEI*, *Android ID*, etc.) with both legitimate and forged values; we checked for successful activations with the company. Table 3.6 shows the minimum set of identifiers required to successfully activate a fake device – in two cases, forging just the IMEI is sufficient. Hence, despite the vendor being aggressive in catching any potential fake fingerprints, holes still exist. In the end, considering that the global cost per install for an Android app is $0.44 [11], an adversary can generate fake activations at scale, and as long as a reasonable fraction of them go through (e.g., 50%), cause serious financial damage.

Table 3.6 App_X Successful Activations via Wiretap Injection

Phone	Minimal Successful Attack
Google Pixel 3	IMEI
Nexus 5	IMEI/AndrID/Serial
Galaxy 8	IMEI
Xperia XA2	IMEI/AndrID/Serial/MAC

Table 3.7 IPG Results for Different App Families

App Family	Nodes									IPG	
	Source Identifiers							Intermediate Nodes	Sink (Network)	$\|V\|$	$\|E\|$
	Email	Phone#	C.Card#	IMEI	And. ID	Serial #	GUID				
LevelUp Consulting	1				1			1 (serialization)	1	4	3
Paytronix Systems	1	1						1 (serialization)	1	4	3
PunchhTech	1	1		1	1	1	1	1 (hashing)	1	8	7
Relevant Mobile	1	1		1	1	1		2 (concatenation, serialization)	1	8	7
TapMango Inc	1	1						1 (serialization)	1	4	3
Thanx	1		1		1	1			1	5	4
Total Loyalty S.	1							1 (serialization)	1	3	2

3.4.6 IPG effectiveness

We demonstrate the effectiveness of our IPG scheme representation and extraction by showing how IPGs capture the processing of various identifiers for our seven examined app families. Table 3.7 shows the results. The first column lists the app families, while the next column displays the various types of nodes within the IPG for each specific app family. The nodes, or vertices, fall into three categories: source identifiers (e.g., email, IMEI), intermediate nodes that summarize how the identifiers are processed or combined before being leaked (e.g., hashed, concatenated), and sink nodes, which represent the destination of the leaked information. The last two columns contain the total count of IPG vertices and edges.

In the case of the LevelUp Consulting app family, the fingerprinting scheme involves utilizing email and AndroidID as source identifiers, which are serialized in the form of a key-value pair JSON object (intermediate processing node), and subsequently leaked onto the network. Therefore, the IPG associated with this family comprises a total of 4 vertices (email, AndroidID, intermediate JSON object, and network sink) and 3 edges. Similarly, the PunchhTech and Relevant Mobile app families use email,

phone number, IMEI, AndroidID, and serial number as source identifiers. In the case of PunchhTech, these identifiers undergo hashing using the `hashCode()` function, while for Relevant Mobile apps, the identifiers are concatenated and then exposed on the network as a JSON object. The last two IPG columns show the median graph size among apps in that row. Typically, IPGs have 3–8 vertices and 2–7 edges, which confirms the conciseness of our graph-based encoding and the effectiveness of our scheme extraction.

3.4.7 Comparison with alternative approaches

In this section, we compare our approach with possible alternatives: taint analysis, GUI automation, and slicing.

Taint trackers, static or dynamic, are popular tools for studying ID leaks, but their goal is to enumerate the identifiers leaked (as sink→source pairs), whereas in our fingerprint setting, our IPG-based approach summarizes how identifiers, including registration, are *processed and combined* to produce a fingerprint. Moreover, in the context of offer abuse, identifiers deemed as "leaked" by a taint tracker might or might not be involved in fingerprinting.

To quantify the suitability of taint tracking in our setting, we performed a study that compared our approach with the state-of-the-art (and de facto benchmarks) FlowDroid and Amandroid taint trackers, on 64 ground truth apps that cover all categories in Table 3.2. For each app, we performed an exhaustive manual flow analysis, including network packet analysis, to find all possible device identifiers' flows to the network. We show the results in Table 3.8. The first column contains the ground truth, i.e., the number of apps (out of 64), where the identifiers have an actual flow to the network. Then, in each column, we show the number of apps where the tool has found a network leak for the specified identifier. For FlowDroid, we added source methods in the SuSi list (FlowDroid's predefined sink and source list) for AndroidID,

Table 3.8 Number of Identifier Flows for 64 Ground Truth Apps

	Ground Truth	Our Approach	FlowDroid	Amandroid
IMEI	28	28	11	21
IMSI	9	9	9	9
Serial	29	29	n/s	18
MAC	10	10	5	15
Android ID	57	57	24	44
Advert. ID	9	9	n/s	32
GUID	35	35	53	56
FP	n/a	0	18	49
FN	n/a	0	55	24

Serial, AdvertisingID, GUID (not included by default). FlowDroid missed IMEI, MAC, and AndroidID leaks in about half the apps that actually contain a leak; for GUID, FlowDroid reported leaks that do not exist; FlowDroid could not find any flows for Serial and AdvertisingID, though for the same APIs, our approach and Amandroid did find flows. Overall, FlowDroid under-reported leaks in 55 apps and over-reported leaks in 18 apps. Amandroid performed slightly better than FlowDroid because it handles ICC (Inter Component Communication) by default: the tool under-reported 24 leaks and over-reported 49 leaks. While false positives are inherent due to static analysis' over-approximation, false negatives are particularly pernicious in our setting as they lead to incomplete fingerprinting schemes. In contrast, as shown in the second column, our approach captures all the flows precisely, with *zero false-positive and false-negative rates*. Finally, our approach produces concise IPGs, with median $|V| = 5$ and median $|E| = 4$ across the 64 apps.

GUI automation. Even a highly effective GUI automation approach, e.g., a customized version of DroidBot that could reliable navigate app activities and fill-in fields, would fail to repeat the offer generation due to the *one offer per user* limitation.

Slicing. Note that slicing already improves precision compared to standard dynamic bytecode analysis, e.g., program dependence tracking, thanks to slicing criteria that reduce the set of program dependences to only those dependences that end in (or start at) the stated criteria. Still, using slicing alone would be inadequate, since fingerprinting extraction cannot be achieved effectively: slices, even when specifying

precise criteria, might include *thousands of instructions* [89] for substantial real-world apps (our setting).

3.5 Discussion

We now discuss actionable insights and key takeaways: defense schemes for developers to protect against fingerprinting attacks; methods to prevent anti-tampering; recommendations for app markets, third-party developers, and users; the generalizability of our approach, as well as its limitations.

3.5.1 Defense schemes

We propose several measures that developers and vendors can take to ensure more robust defenses against attacks on fingerprinting.

a) *Verifying Phone Number's Authenticity.* Phone numbers are commonly used IDs (Table 3.3), but only a few apps verify their authenticity (e.g., Uber Eats, Dunkin, Postmates). Apps can authenticate phone numbers via: One Time Passwords (though blocking "free SMS" websites [93]); handshaking, where users have to both *send* and *receive* an SMS; or receiving a phone call. Interestingly, we found no email authenticity checks: throwaway addresses worked perfectly fine.

b) *Limiting Losses.* Businesses can limit losses by asking users to first spend \$X, or fill the in-app account with \$Y from a credit card, before receiving an offer. For example, the LevelUp family [15] asks users to add at least \$25 into the in-app account from a credit card to redeem the initial sign-up offer. The maximum available offer is \$10, limiting losses to $10/25 = 40\%$. In contrast, Seamless directly provides a \$15 offer without minimum purchase or credit card requirements, hence, losses can be 100%. Per Table 3.4, in 41 apps, a credit card is required to redeem the offer. Surprisingly, we could easily bypass this requirement in all 41 apps using made-up, confirmed invalid credit card numbers [73], showing that these apps do not verify the

credit card information when the amount due becomes $0 (after the offer is redeemed). Another defense strategy is to limit the total number of offers or the offer budget for the promotional campaign, e.g., $10K. Such limits though may actually lead to a failed marketing campaign, e.g., when the campaign benefits one attacker as opposed to 1,000 legitimate users.

c) *Hardware Cryptographic IDs.* Modern hardware (e.g., TPM, Intel SGX, and ARM Trustzone) stores private keys, allowing devices to be "attested" (device is unique and valid) by a remote party [12]. Such technologies, in principle, enable much more reliable uniqueness tests, as an adversary would need multiple pieces of real hardware to create multiple profiles (which can be too expensive to be practical).

d) *Biometric Authentication.* Authentication using biometrics such as fingerprint scanning can verify real users and ensure one offer per unique user scheme. While biometric authentication is vulnerable to presentation attacks and puppet attacks, FINAUTH [209], Bianchi et al. [95] show defense schemes against such attacks.

e) *App Anti-tampering.* Preventing app bytecode and traffic tampering, discussed shortly, can add another layer of security.

3.5.2 Anti-tampering

We now discuss methods that can be used to protect apps from unauthorized modifications (bytecode-rewriting) and tampering with the API communication between a client app and server.

Preventing Bytecode Rewriting. One effective method to detect tampering is through checksum verification. By incorporating a checksum within the application's binary file and verifying it during runtime, any modifications made to the app will result in a mismatched checksum [190]. Another approach involves dynamic class loading, where crucial segments of the app's code are loaded at runtime instead of being included in the binary file. This presents a challenge for attackers attempting to

alter the code since they would need to modify the loaded code during runtime [156]. Additionally, encryption and decryption techniques can be employed. This method entails encrypting vital components of the app's code or data and decrypting them during runtime. By doing so, attackers face increased difficulty in modifying the app's code or data, as they would have to reverse-engineer the encryption algorithm and acquire the decryption key [144].

Preventing Wiretap Injection. A MAC (Message Authentication Code) is a short piece of information that is used to authenticate a message and detect any changes made to it. By using MACs, both the client and server can verify that messages have not been tampered with during transit [143]. Another technique that can be employed for authentication is API token authentication, i.e., using unique tokens to verify API requests. These tokens are generated on the server side and provided to the client app upon login. By authenticating API requests with these tokens, unauthorized access and tampering with API communication can be mitigated [127].

3.5.3 Other target groups

Our toolchain could benefit other target groups: app markets by enhancing regulatory checks, third-party developers by promoting compliance and privacy-aware development, and users by providing them with knowledge and tools to protect their privacy in the face of fingerprinting practices.

App Markets. App stores/markets can integrate an automated fingerprinting vulnerability detector, similar to our methodology (Section 3.1.3). This mechanism would allow app stores to assess whether app developers are adhering to the guidelines set by the stores. For instance, the Google Play store could determine whether an app adheres to the developer guidelines (Section 3.2.3), e.g., regarding the use of persistent hardware identifiers, and then decide whether to keep or remove the app.

Third-party developers. Vendors, such as restaurant chains, can extract and check the fingerprinting schemes implemented by 3rd party developers (e.g., developers of app families) to ensure compliance with the vendor's fingerprinting requirements. Research has shown that a substantial share of identifier leaks occurs in 3rd party code, i.e., libraries [179]. Developers can extract fingerprinting schemes from apps to understand the extent of, and mechanisms for, fingerprinting performed by 3rd party code, e.g., analytics and advertising libraries.

Users. Our toolset empowers users to better understand the extent of fingerprinting performed by apps on users' devices, and the nature of sensitive information leaked onto the network by apps. As a potential solution to protect their privacy, users can deny apps the permission to access sensitive device information. For example, in Android, the android.permission.READ_PHONE_STATE permission allows an app to read the phone's state, including information such as the device's IMEI, IMSI, network information, and call status. Users can deny such permissions, thereby regaining control over their personal data and minimizing their exposure to fingerprinting. Another solution to avoid being tracked by third-party trackers is to reset the *Advertising ID* periodically.

3.5.4 Generalizability

We now discuss potential generalizations of our approach to other settings.

iOS. Apple uses unique identifiers such as the IDFA [74] (similar to Android Advertising ID), Vendor ID, MAC address, and Bluetooth MAC address to fingerprint iOS devices. While in this dissertation we only studied Android apps, since iOS apps can use similar fingerprinting approaches they could potentially be vulnerable to similar attacks. While it is possible to reverse-engineer iOS apps, the process is generally more difficult than for Android apps due to the closed ecosystem, obfuscation techniques, and stronger security features [109, 206].

Browser fingerprinting. Our general techniques of dynamic dependence analysis and reduction to IPG could be combined with a dynamic JavaScript tracer to understand browser fingerprinting. However, browser fingerprinting techniques are commonly circumvented by simply restarting the browser, blocking cookies, or utilizing a different browser [146].

3.5.5 Limitations

For the 23% of apps that employ WebView or have a scrolling feature in the UI, we had to conduct a manual attack in addition to ReOffer. With additional engineering, this limitation can be resolved, and manual efforts can be automated. For example, to overcome the WebView issue, one option is to inject custom JavaScript code into the WebView, leveraging its built-in JavaScript support to extract specific elements and content. Another approach is to utilize the Android Accessibility APIs, accessing the WebView's accessibility tree to retrieve information about the rendered content. In complex scrolling scenarios where ReOffer alone is insufficient, we can combine it with other testing frameworks. For instance, we could integrate our toolchain with Espresso [72], a UI testing framework that provides more advanced scrolling capabilities.

3.6 Summary

Fingerprint encoding, extraction, and studies, require abstractions and techniques beyond the capabilities of current analysis tools. We address this by introducing IPGs. By automatically constructing IPGs we could encode, extract, characterize, and ultimately subvert fingerprinting schemes. We conducted the first study that explored and attacked fingerprinting schemes in 436 Android apps. We show that registration-based schemes are vulnerable, yet widely used; we constructed a tool, ReOffer, for conducting re-registration attacks at scale. We show that device-based

(and combined registration/device-based) schemes can be subverted via bytecode rewriting and network injection, leaving apps, or entire app families, vulnerable.

We expect our contributions to be usable and generalizable in other contexts. Using our approach, users can understand how they are being fingerprinted, or how to establish new identities. Developers/companies can implement more effective promotion systems and promotional campaigns: more user privacy-friendly yet avoiding financial losses or unfair app promotion charges. Program analysis and security researchers can use our approach to extract the semantics of complex leaks that combine multiple taint sources.

Now that we have examined the aspects of device identifiers usage and abuse in detail, we turn our attention to another crucial aspect of our research: reliability. In the upcoming chapters, we will explore various dimensions of reliability in the context of Android applications. Through these chapters, we will address different aspects of reliability in Android applications, ranging from medical calculators to user interactions, automated testing, and neural network implementations.

CHAPTER 4

DIAGNOSING MEDICAL SCORE CALCULATOR APPS

Mobile medical score calculator apps are widely used in clinical settings to aid in patient treatment and diagnosis decisions. However, errors in score definition, input, or calculations can have severe consequences. In this chapter, we address this issue with a novel, interval-based score-checking approach. We first introduce automated correctness checking of medical reference tables to identify errors that may propagate to apps. Next, we implement an automatic, dynamic analysis-based approach to verify score correctness in Android apps. Our evaluation of 90 Android apps revealed violations and incorrect score calculations, leading to improvements and fixes. We aim to enhance the reliability and accuracy of medical score calculators, particularly crucial in acute care settings, where precision can significantly influence outcomes. Specifically, we aim to answer the following research questions:

RQ1) Can our approach extract and verify medical score specifications?

RQ2) Is our approach effective at analyzing and finding errors in real-world apps?

In Section 4.1, we define the properties we check, highlighting three sources of errors in score calculations: inconsistent reference tables, GUI inconsistency, and incorrect score calculation. In Section 4.2, we describe our approach, utilizing the Z3 theorem prover to automatically check medical reference tables for partition condition violations. We then present a novel dynamic approach in Section 4.2.3 to verify the app for GUI consistency, reference table compliance, and correct score calculation. We examine 12 long-established medical scores, uncovering errors in five reference tables in Section 4.3. In Section 4.4 we evaluate our app-checking approach on 90 Google Play apps, achieving 100% precision and 80% recall. The results reveal GUI inconsistencies and score calculation errors that can have critical implications for patient care.

SOFA score	1	2	3	4
Respiration				
PaO_2/FiO_2, mmHg	<400	<300	<200	<100
			with respiratory support	
Coagulation				
Platelets $\times 10^3/mm^3$	<150	<100	<50	<20
Liver				
Bilirubin, mg/dl (μmol/l)	1.2–1.9 (20–32)	2.0–5.9 (33–101)	6.0–11.9 (102–204)	>12.0 (<204)
Cardiovascular				
Hypotension	MAP < 70 mmHg	Dopamine ≤5 or dobutamine (any dose)[a]	Dopamine >5 or epinephrine ≤0.1 or norepinephrine ≤0.1	Dopamine >15 or epinephrine >0.1 or norepinephrine >0.1
Central nervous system				
Glasgow Coma Score	13–14	10–12	6–9	<6
Renal				
Creatinine, mg/dl (μmol/l) or urine output	1.2–1.9 (110–170)	2.0–3.4 (171–299)	3.5–4.9 (300–440) or <500 ml/day	>5.0 (>440) or <200 ml/day

Missing value 12.0

Should be > 204

Missing value 5.0

[a] Adrenergic agents administered for at least 1 h (doses given are in μg/kg·min)

Figure 4.1 SOFA Score (from Vincent et al. [201]).

4.1 Motivation

To motivate our approach, we first quantify the adoption of mobile apps in clinical care; next, we define the key terms and concepts used throughout the dissertation, and then provide examples of errors in actual reference tables and apps.

4.1.1 Mobile medical apps usage

Medical apps and calculators in acute cure. The adoption of mobile medical apps in a clinical setting in general is already strong, with clinical smartphone use among physicians being reported at 70% and above as early as 2012 [169, 204]. Mobile medical apps and smartphone-based reference material are widely used in emergency/acute care. For example, Hitti et al.'s 2021 study [129] regarding emergency department personnel has revealed that 91.8% of those surveyed used medical apps on their devices during their shifts, amidst heavy workloads and a stressful environment. Flynn et al.'s 2018 study [115] showed that 98% of acute care nurses used a smartphone in acute settings "to access information on medications, procedures, and diseases". Green et al.'s 2019 study [122] revealed that "60% of users indicated that they are somewhat or very likely to use newly published medical calculators".

Higher app usage for inexperience personnel. Another impetus for studying and improving medical app reliability is that less-experienced personnel might rely more on apps. There is evidence that users of medical calculator apps are clinicians and nurses, especially *inexperienced and younger doctors*, according to Hitti et al. [129]. A 2015 study among surgeons found that "Junior doctors were more likely to use medical apps over their senior colleagues ($p = 0.001$) as well as access the Internet on their smartphone for medical information ($p < 0.001$)" [172]. Additionally, per Green et al.'s survey [122], clinicians with less experience are more likely to use medical calculator software; conversely, experienced clinicians had doubts on the credibility of medical calculators.

Medical score calculator accuracy. Pelletier et al.'s 2022 study [175] on both online and mobile bleeding risk calculators, such as HAS-BLED, has found inconsistencies in calculated risk estimates which can result in harmful clinical decisions. The study has shown that such imprecise results found in apps are due to incorrect calculations, using alternative validation studies, and inaccurate translations of risk factors to risk elements. Fajardo et al.'s 2019 study [112] of online type 2 diabetes risk calculators has revealed that while calculator results are generally understandable, such calculators may not be suited for patients who lack general health literacy; the study also found that these calculators have high variability in terms of determining estimated risk.

Therefore, the reliability of scores and score calculator apps is important in general due to heavy reliance on under-regulated mobile apps, and particularly important in acute care settings where *rapid* and *correct* decisions are key for achieving positive patient outcomes. However, there is a lack of a regulatory framework and enforcement regarding medical apps. For example, the US Food and Drug Administration (FDA) has jurisdiction over medical apps. However, the FDA does

not regulate apps that "automate clinical calculations and basic tasks for health professionals" [70]. Additionally, apps that are FDA-approved went through an approval process initiated by the developers themselves.

4.1.2 Definitions

Reference Table We use the term *reference table* for the table in the form it was first introduced, e.g., in a medical research article or a regulatory agency document. For example the Sequential Organ Failure Assessment (SOFA) score, shown in Figure 4.1 and discussed shortly, was introduced by Vincent et al. [201] in the *Intensive Care Medicine* research journal in 1996. The NEWS score, another score we consider, was introduced by the UK's National Health Service (NHS) in 2012 and later updated in 2017 to NEWS2 [54]. Score tables are structured as follows: most commonly, each cell in the table contains intervals for one physiological parameter, while the row or column header contains a numeric value, typically 0–4. For example, in SOFA's reference table (Figure 4.1) the third row shows intervals 1.2–1.9, 2.0–5.9, and so on, for parameter *Bilirubin*. The header row in the table shows numeric values, in SOFA's case 1 through 4, which correspond to individual scores for the intervals in that column. Occasionally, a table entry contains just a threshold value, e.g., $MAP < 70$ *mmHg* in SOFA's fourth row. Finally, a cell can contain intervals or thresholds for more than one parameter, e.g., *Dopamine* > 15 *or norepinephrine* > 0.1 in SOFA's fourth row. Next we describe score computation.

Score A score is computed by adding the individual scores corresponding to each cell. For example, a patient with *Respiration=350* (score=1), *Coagulation=90* (score=2), *Liver=7.0* (score=3), *Central nervous system=13* (score=1), *Renal=1.5* (score=1) would have an overall SOFA score:

$$SOFA\,score = 1 + 2 + 3 + 1 + 1 = 8$$

The overall value determines the course of action. In Table 4.1 (discussed at length later) we show action thresholds, e.g., "aggressive treatment if HEART score ≥ 7"; hence, an accurate value is critical for patients' health outcomes.

Partition. Let $min \in \mathbb{R}$ and $max \in \mathbb{R}$ be the minimum and maximum values for a parameter, respectively. Let $min < p_1 < p_2 < ... < p_n < max$ be ordered values. Based on the p_i's we can define intervals (e.g., closed, $I_i = [p_i, p_{i+1}]$, open, $I'_i = (p_i, p_{i+1})$, or combinations thereof). Let $I_1, I_2, \ldots, I_i, \ldots, I_n$ be nonempty ($I_i \neq \emptyset$) intervals on $[min, max]$. Then $I_1, I_2, \ldots, I_i, \ldots, I_n$ form a *partition* of $[min, max]$ if two conditions are met:

1. Coverage (exhaustion):

   $I_1 \cup I_2 \cup \ldots \cup I_i, \ldots \cup I_n = [min, max]$

2. Non-overlap (disjointness):

   $\forall i, j,\ 1 \leq i < j \leq n \rightarrow I_i \cap I_j = \emptyset$

As we will illustrate shortly, many errors, in reference tables themselves or the apps implementing the tables, stem from violations of the aforementioned partition conditions.

4.1.3 Error source #1: inconsistent reference table

Errors such as inconsistent definitions in reference tables are the most concerning kinds of issues we found, because, unlike apps, tables are hard to update or fix. Moreover, as our evaluation shows, an incorrect reference table is likely to lead to incorrect implementations in apps, because developers tend to implement tables *ad literam*. Finally, an inconsistent table will lead to an inconsistent GUI that confuses app users and invites score calculation errors.

We illustrate several such inconsistencies on the *SOFA Score* reference table. The SOFA score predicts ICU mortality as follows: it evaluates the dysfunction of six

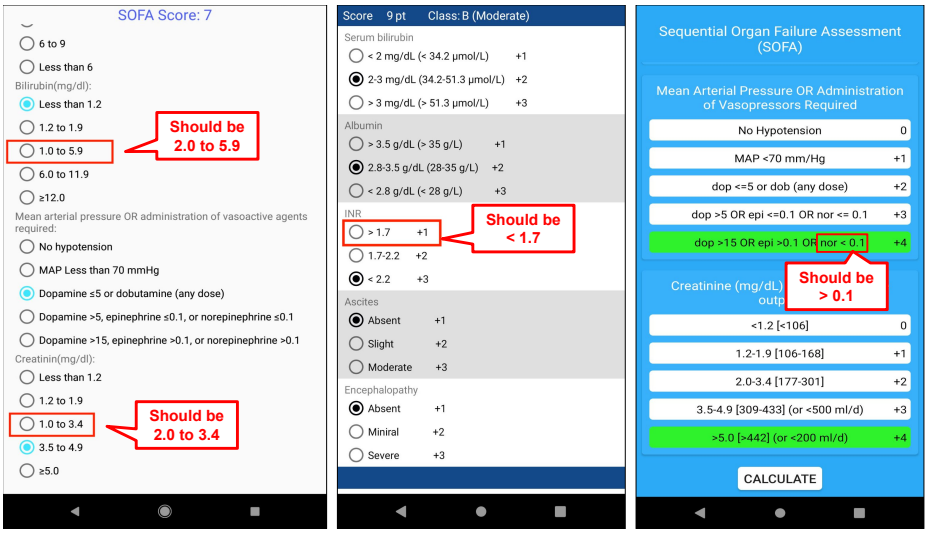


Figure 4.2 Inconsistent GUI errors in three apps: Nursing Calculator (left), Child-Pugh Score (center), SOFA 1.2.0 (right).

systems by scoring each organ from 0, which is considered normal functionality, to 4, the most abnormal [201]. Thus the highest possible score to obtain would be 24, indicating severe morbidity, and the lowest would be 0.

The reference table for the SOFA score is shown in Figure 4.1. Notice how for Liver (Bilirubin), the second-to-last interval is defined as 6.0–11.9. As the parameter is a real number, the actual interval specification is $[6.0, 12.0)$. That is, a value such as 11.95 would still be in the interval because only the first decimal is specified. The last interval for bilirubin is > 12.0. Hence, the interval-based specification for these two entries is: $[6.0, 12.0)$ and $(12.0, max)$. *This squarely violates the coverage property of the partition, because value 12.0 is not covered by any interval.* The same issue is present for parameter Renal (Creatinine), where value 5.0 is not covered. It is unclear how developers are supposed to cope with this incorrect specification, e.g., the SOFA score of a patient with Bilirubin=12 and Creatinine=5 can be *off by as much as 2 points*, depending on how the table is interpreted.

Finally, when bilirubin is specified in μmol/l, the table's last column shows '(< 204)' which is incorrect: the entry should be '(> 204)' (note how values < 204 are already covered in the preceding intervals). If the developer implements the table ad

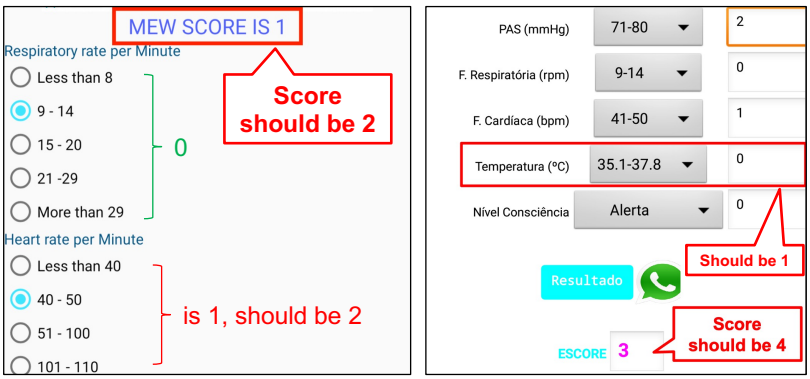


Figure 4.3 Nursing Calculator incorrect score (left); MEWS Brasil incorrect scores for *Temperature* and overall (right).

literam and offers '(< 204)' as a GUI option, the SOFA score of a patient can be *off by as much as 3 points*.

Note that even a off-by-one error can affect patients' condition classification, e.g., between "patient should be monitored" and "urgently inform a clinician". Section 4.3.2 discusses these issues at length.

4.1.4 Error source #2: inconsistent GUI

We now turn to the first kind of implementation errors, where the GUI is inconsistent; our approach detects two kinds of errors. In the first kind, the user can input the same parameter value into *two different GUI boxes*, which impacts the score; in the second kind there is no input box for a certain value. Essentially, these embody violations of the coverage and non-overlap conditions, respectively; in Section 4.2.3 we discuss how we check GUIs for such errors automatically via dynamic analysis and constraint solving.

Example 1: SOFA Score in App Nursing Calculator. The app Nursing Calculator,[1] with over 50,000 installs, provides a variety of medical calculators, including the SOFA score. The app's GUI has two inconsistency errors (first kind), as highlighted in

[1] `https://play.google.com/store/apps/details?id=com.niya.lijo.nursingcalculators`, Retrieved on DATE: 2023-06-01

Figure 4.2 (left), and described next. The option for Bilirubin shows a range of 1.0–5.9 when it is supposed to be 2.0–5.9. Moreover, for Creatinine the range in the app is 1.0–3.4, when it should be 2.0–3.4. Due to these errors, a patient's score can be *off by as much as 2 points*.

Example 2: Child-Pugh Score in App Child-Pugh Score. The Child-Pugh score is generally used to assess the potential for liver diseases, mainly cirrhosis. The app Child-Pugh Score[2] has an inconsistency error (first kind) regarding values for INR, as highlighted in Figure 4.2 (center): the first option should be '< 1.7' instead of '> 1.7'. Due to this error, a patient's score can be *off by as much as 2 points*.

Example 3: SOFA Score in App SOFA 1.2.0. This app exemplifies the second kind of error. The app SOFA 1.2.0,[3] removed from Google Play in the course of our research, exhibited a GUI inconsistency error as highlighted in Figure 4.2 (right). Note that the reference table's last column in the *Cardiovascular* row specifies the score for ... $norepinephrine > 0.1$; the app however incorrectly lists '$norepinephrine < 0.1$'. Basically, the app offers no option where users can indicate the $norepinephrine > 0.1$ condition; this error can alter the score by 1 point.

4.1.5 Error source #3: incorrect score calculation

Even with a consistent table and consistent GUI, apps can still be prone to errors in score calculation, e.g., per the table the score is 4, but the app displays 6. These calculation errors are silent, hence, particularly pernicious (the user does not have any indication that the calculation has gone awry).

[2] `https://play.google.com/store/apps/details?id=br.child_pugh`, Retrieved on DATE: 2023-06-01

[3] `https://apksos.com/app/com.varendrasoft.sofascore`, Retrieved on DATE: 2023-06-01

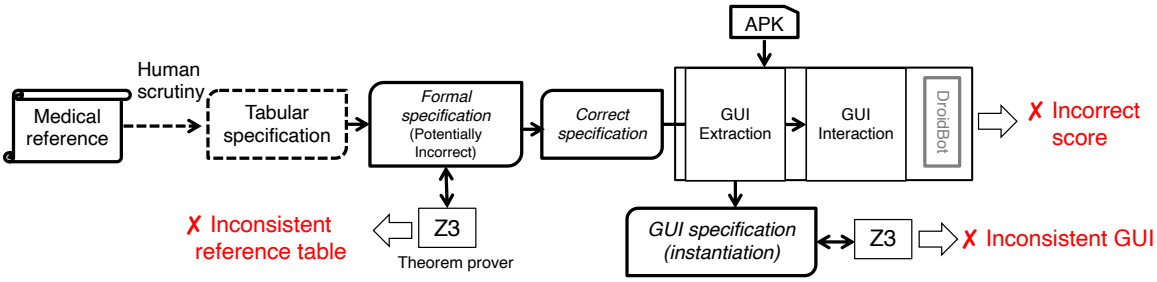


Figure 4.4 Overview of our approach and toolchain.

We now present several examples, based on the Modified Early Warning Score (MEWS), used by professionals to determine whether or not a surgical in-patient requires intensive care [118].

Nursing Calculator. When the app starts, parameter values are in their default settings, i.e., individual scores are 0, hence, the MEWS score is 0. After the user changes the Heart rate to 40, the output score is 1 instead of the expected value of 2 (screenshot in Figure 4.3 left). Note that a higher MEWS value indicates a more severe situation.

Another example is the MEWS Brasil app [69], Figure 4.3 (right). Suppose the user inputs a 41–50 Heart rate and BP between 71–80; cumulatively, the score is 3. The error manifests when the temperature is in the interval 35.1–36; per the table, the individual temperature score value is 1. In the app however, the value is 0, hence, the displayed overall MEWS score, 3, is incorrect (correct value: 4). This error is particularly problematic, because for MEWS, 4 is a threshold value: "[at $\geq$ 4] surgical team should be informed immediately" [118].

These incorrect implementations can lead to differences in assessment, and subsequently outcome, making verification of scoring systems crucial.

4.2 Approach

We now describe our approach to finding errors in reference tables, app GUIs, and app score calculations. An overview is shown in Figure 4.4. In the first stage for each

scoring system, we extract a *specification* for the reference table; our toolchain then checks the specification for consistency, i.e., for coverage and non-overlap violations using Z3. We then fix the inconsistency found in the reference table before using it as a reference. Next, for a given app (APK) implementing that score, our toolchain first performs a dynamic analysis to extract a *GUI specification* (aka the GUI instantiation of specification) using the DroidBot automator [155]. The GUI specification is (a) validated against the correct reference table specification, and (b) verified for consistency, using Z3. Finally, our approach drives app execution (GUI interaction) automatically, according to specific input parameter combinations, to produce the app's output score, and verifies this score against the reference score for that parameter combination. We now discuss each phase, including a brief introduction to the underlying tools.

4.2.1 Partition checking via satisfiability

The Z3 Theorem Prover. Z3 [107] is a widely-used automated theorem prover for solving logical formulas in various domains. Z3 is based on Satisfiability Modulo Theories (SMT) techniques, which allow it to handle a wide range of logical theories, including arithmetic over integers or reals, bit-vectors, or arrays. Given an input formula, Z3 returns "`Sat`" or "`Unsat`". "`Sat`", short for "satisfiable," indicates that there exists at least one assignment of values to the variables in the formula that makes the formula true. For example, assuming A and B are integers, the formula:

$$A > 20 \wedge B > A$$

is satisfiable, with Z3 returning `Sat`, and the model $A = 21, B = 22$.

"`Unsat`", short for "unsatisfiable," indicates that there is no assignment of values to the variables in the formula that makes the formula true. For example, formula:

$$A > 20 \wedge B > A \wedge B < 19$$

is unsatisfiable, hence, Z3 returns `Unsat`.

We use Z3 to check the satisfiability of logical formulas composed of numerical ranges/intervals. Note that most score calculations involve parameters whose values span a set of ranges, or intervals. In order to be defined as a valid set of input ranges, the ranges should meet the coverage and non-overlap conditions, as per Section 4.1.2. Hence, we encode parameter ranges into a Z3 specification (sets of intervals over integer or real numbers, as appropriate) and then use Z3 to check whether the set of intervals meets the partition conditions.

Coverage (exhaustion) Checking. We encode coverage checking into Z3 by requiring that the union of a given set of intervals cover a range, defined by its minimum and maximum values. When Z3 finds a counterexample, it proves that there exists a "gap" in coverage. For example, the intervals [≥10, 6-9, ≤5], encoded into Z3 as:

$$\neg(\vee(X \geq 10, \wedge(X \geq 6, X \leq 9), X \leq 5)))$$

cover the range, hence, Z3 will not find a counterexample. However, the intervals [≥10, 6-9, <5] with encoding:

$$\neg(\vee(X \geq 10, \wedge(X \geq 6, X \leq 9), X < 5)))$$

do not cover the range, and Z3 will successfully find the counterexample *X=5*, i.e., a non-covered value. Note that our implementation automatically generates a Python program that invokes Z3 via its Python API.

Non-overlap (disjointness) Checking. In this case, we use the equation solver feature of Z3. Given a set of variables and corresponding constraints, Z3 generates a solution (if a solution exists) that satisfies the constraints. Hence, to check for overlap between two intervals, we add them as constraints composing two sets, and as an additional constraint we check for overlap between the sets. For example, given an interval with incorrect partitioning [≥65, 45-65], which we encode as:

$$(X \geq 65, \wedge(Y \geq 45, Y \leq 65), X == Y)$$

Z3 finds an overlapping value $X = 65, Y = 65$. If this set of intervals were correctly partitioned ([>65, 45-65]), Z3 would return `Unsat`.

4.2.2 Reference table validation

We check the validity of reference tables in two steps. First, we extract the reference table from the source PDF file into a *tabular specification* in CSV format. Though we employed a PDF→CSV conversion tool, the resulting CSV is still subject to human scrutiny to ensure accurate conversion (this is one of the only two manual steps of our approach; the accuracy of this extraction step is discussed in Section 4.3.3).

The tabular specification is then automatically encoded into a Python program that invokes Z3 to verify that each parameter of a reference table meets the partition conditions. Note that some of the scores we considered contain real numbers, specified to one or two decimal places (e.g., 5.9, 5.32). We convert such decimals to integers, multiplying by 10 or 100, respectively (5.9 becomes 59; 5.32 becomes 532). Section 4.3.4 discusses the reference table errors we found.

4.2.3 App verification and validation

App score verification and validation presents several challenges: a) mapping heterogeneous GUI elements to reference table cells, which we solve via semi-supervised clustering (Section 4.2.3); b) systematically and automatically exercising the GUI, which we address by coupling Depth-first Search with DroidBot-based static and dynamic GUI information extraction (Section 4.2.3, Section 4.2.3); and c) identifying app score-related heterogeneous GUI elements and extracting the corresponding score value (Section 4.2.3). We now present our approach to solving these challenges in detail.

Mapping Heterogeneous GUIs to Specifications. We need to handle heterogeneous GUIs to create the correct specifications set for different parameters of a

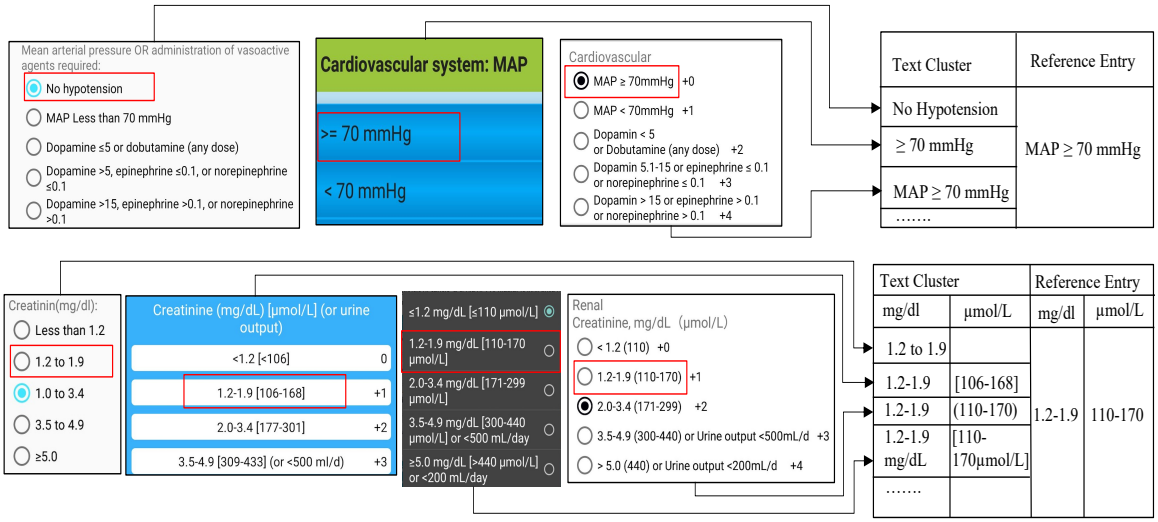


Figure 4.5 Heterogeneous GUI mapping example for *Cardiovascular Mean arterial pressure* attribute and *Renal function Creatinine* attribute in apps Nursing, Nursing Calculator, SOFA score, SOFA score, and SOFA Score.

specific scoring system. In Figure 4.5, we provide two sets of examples of heterogeneous GUI elements that must be translated to SOFA score parameters. The *Cardiovascular Mean arterial pressure* parameter (top) has the score value *0* which appears as a "*No Hypotension*" Radio Button in the Nursing app (top left) whereas the Nursing Calculator app (top center) uses a different GUI element, Spinner, labeled "$\geq$ *70 mmHg*", while the SOFA score app (top center) uses a different label, "*MAP* $\geq$ *70 mmHg*"; all for the same parameter value. To address this challenge, we first extract all the GUI information from the apps automatically using DroidBot (to be introduced later) and then we create a mapping from the heterogeneous, free-text GUI to the reference table entry. We use semi-supervised clustering, i.e., we manually labeled the first points, and when analyzing a new app, the new text is binned into the cluster containing the most similar text. Each of these clusters' title represents the reference table entry – typically an interval or parameter. We defined reference table entries manually (right side of Figure 4.5); text clusters (shown to the left of reference entries) are populated automatically using edit distance as a similarity metric. In Figure 4.5 (top right), we show how different phrasings of *Cardiovascular Mean arterial pressure* parameter are

mapped. The text clusters are formed from the phrasings (*No Hypotension*, $\geq$ *70 mmHg*, *MAP* $\geq$ *70 mmHg*), which are then mapped to the same cluster, whose title is '*MAP* $\geq$ *70 mmHg*'. Similarly, the *Creatinine* parameter (bottom) poses the challenge of heterogeneous GUIs: note how the same score value, *1.2-1.9*, is expressed in four different ways in four different apps. In this case, the text cluster entries are mapped to the reference entry '*1.2-1.9 (110-170)*'. We separate reference table entries by units, e.g., mg/dl were not mixed with μmol/l; this was necessary only for the SOFA score.

Background (DroidBot). DroidBot [155] is a tool that facilitates test automation for Android apps. DroidBot allows users to customize the app testing/exploration strategy by specifying input events for certain app states. DroidBot models the app as a finite state machine and is driven by a specification consisting of States (e.g., a particular screen), Views (specific GUI elements), and Operations (action to perform on a View, e.g., click, swipe, or enter text). A JSON script specifies the input events to generate for each state transition. For example, the script might specify that when the app is in a particular state, DroidBot should generate a specific sequence of taps, swipes, or other input events to simulate a user interacting with the app. DroidBot can also monitor app behavior. We use DroidBot to extract the GUI and automate the exploration, e.g., systematically exploring Radio Button elements to select an interval, clicking the 'Calculate Score' button, and retrieving the resulting score.

Leveraging DroidBot. Our approach uses DroidBot to collect score-relevant GUI information dynamically, i.e., while the app runs. Note that, for practical reasons, dynamic analysis is required for GUI extraction, because static GUI information alone (e.g., from app resources) is incomplete: extracting the GUI at runtime, when all the GUI View objects have been instantiated, is more effective. DroidBot's GUI model helps automatically identify various View objects (such as Radio Button or TextView) related to target score calculation. To trigger the necessary events for completing

the score calculation procedure (*GUI Interaction* in Figure 4.4), we wrote custom DroidBot scripts that automatically activate GUI elements. For example, the scripts automatically trigger different user inputs such as choosing from RadioButtons and selecting from Spinner items. Note that score calculators appear in different Android activities in different apps. We manually directed DroidBot to the activity that corresponds to the score of interest (this is the second of the two manual steps of our approach; the manual effort could be avoided with more engineering, which we leave to future work). In the course of dynamic analysis, i.e., when using DroidBot scripting to populate different score calculations automatically, the states and transitions, e.g., events, are recorded into separate JSON files. These JSON files contain the necessary dynamic information required to construct the GUI specification, as explained next.

Algorithm 2 Constructing the GUI Specification and Finding GUI Specification Errors

Input: ReferenceGUISpecification (Correct specifications from the score reference table)
JSON files (DroidBot-generated GUI information from the target APK file)
Output: GUI specification errors

```
1: procedure FindCalculatorErrorsViaGUISpecification(ReferenceGUISpecification,
   JSON files)
2:     for each JSON file F in JSON files do
3:         ExtractedIntervals ← []
4:         for each view objects V in F do
5:             IsScoreRelevantParameterOptionCheck(V)
6:             RunDFSToGetParameterValueOptions(V)
7:             IntervalText ←ProcessParameterGUIText(V["text"])
8:             MappedInterval ←MapExtractedIntervalToCluster(IntervalText)
9:             ExtractedIntervals.add(MappedInterval)
10:        end for
11:        FindGUISpecificationMismatch(ReferenceGUISpecification,
   ExtractedIntervals)
12:        CheckCoverageViolation(ExtractedIntervals)
13:        CheckOverlappingViolation(ExtractedIntervals)
14:    end for
15: end procedure
```

Constructing GUI Specification and Finding GUI Specification Errors. Algorithm 2 presents our GUI specification construction and verification approach. The FindCalculatorErrorsViaGUISpecification algorithm takes the target score system's correct reference GUI specification, the DroidBot-generated JSON files from the target APK as input, and outputs the GUI specification errors found.

For a given app, our toolchain explores its score-relevant screens (aka activities) using DroidBot, generating multiple JSON files. These JSON files are then fed to our validation method to check for GUI specification errors. We perform a DFS search to find and save all the View objects representing score parameters and their values (lines 5,6). The GUI text, e.g., *"dopamine $\geq$ 5 mg/kg/min"* is extracted (line 7). Then, we map the extracted text to its corresponding reference entry using the semi-supervised clustering approach discussed in Section 4.2.3 (line 8). The mapped values will be used as final extracted intervals (line 9). Finally, GUI specification discrepancies (*Inconsistent GUI errors*) are found by comparing correct reference parameter value options and extracted parameter value options from the app (line 11). The extracted GUI intervals are checked for coverage violations (line 12) and overlapping violations (line 13) via Z3 (Section 4.2.1).

Finding Calculation Errors Via GUI Exploration. We also check the score resulting from GUI interaction against the formal specification; in other words, we check the validity of the app-computed final score, given the selected parameter values, w.r.t. the reference table's score calculation. This step discovers *incorrect score* errors.

Algorithm 3 shows our approach to finding score calculation errors. The FindCalculatorErrorsViaGUIExploration algorithm takes the DroidBot-generated JSON files and the reference GUI specification as input. We first check whether the View object corresponds to a score parameter option or interval (line 5). If the View object is a relevant score parameter option, then all the attributes and values of that

Algorithm 3 Finding Calculation errors via Automatic, DroidBot-driven GUI Exploration

Input: ReferenceGUISpecification (Correct specifications from the score reference table)
JSON files (DroidBot-generated GUI information from the target APK file)
Output: Score calculation errors

```
1: procedure FindCalculatorErrorsViaGUIExploration(ReferenceGUISpecification, JSON files)
2:     for each JSON file F in JSON files do
3:         SelectedParameterValues ← []
4:         for each view objects V in F do
5:             IsScoreRelevantParameterOptionCheck(V)
6:             RunDFSToGetParameterAttributes(V)
7:             ParameterValue ←ProcessParameterGUIText(V["text"])
8:             if V["checked"]==true OR V["selected"]==true then
9:                 SelectedParameterValues.add(ParameterValue)
10:            end if
11:        end for
12:        CorrectScore ←GetReferenceScore(SelectedParameterValues, ReferenceGUISpecification)
13:        AppScore ←GetAppScore(V)
14:        CheckScoreValidity(CorrectScore, AppScore)
15:    end for
16: end procedure
```

parameter are extracted using DFS (lines 6,7). Next, on lines 8 and 9, we find which score parameter option is selected (set to `true` for spinners) or checked (set to `true` for radio buttons). By looping through all the score parameters all their selected values are saved and passed as input to GetReferenceScore along with the reference GUI specification (line 12). This method calculates the expected correct score by mapping each selected option to its correct value and then simply performing an addition to get the total score. We extract the app-generated score via GetAppScore method (line 13) using a similar JSON file View object analysis and text processing approach. We show an example of score value extraction in Figure 4.6. The figure shows three different apps, where the computed score appears in three different forms. Finally, we compare the two scores: the app-calculated score and the expected correct score, to find any existing score-related error (line 14).

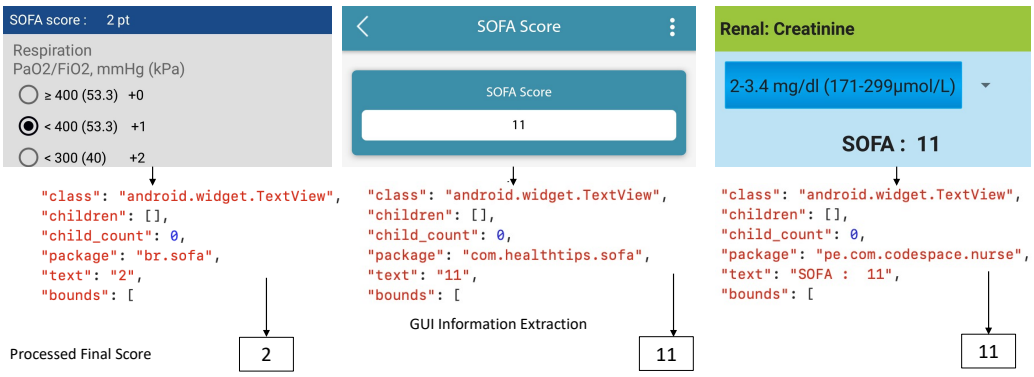


Figure 4.6 Score extraction from heterogeneous GUIs: Blue Rock SOFA (left); SOFA Score (center); Nursing (right).

Comparison With Existing Dynamic Analyzers. Our work makes two key advances that permit rigorous, automatic score verification: extracting a formal interval semantics from GUIs, and systematic GUI exploration. We illustrate these advances by comparing with existing dynamic analyses.

GUI Specification Extraction Although traditional dynamic analyzers [149] can extract GUI information from apps, such as button size and word count, they do not capture semantics. Furthermore, they are not suitable for dealing with heterogeneous GUIs as discussed in section Section 4.2.3. Our approach solves these issues by accurately mapping diverse textual representations of GUI specifications to a reference entry, using interval semantics. These intervals are then used as inputs to Z3 for coverage and overlap error-checking.

Figure 4.7 shows how our approach maps GUI elements to intervals for the *Creatinine* parameter in the br_SOFA app (top-right part of the figure). Additionally, we separate the extracted numeric intervals based on the different units present for an attribute (e.g., mg/dL, μmol/L) for error-checking. These interval-based specifications can be encoded into Python and passed to Z3 for error checking (Section 4.2.2). In contrast, prior GUI extractors do not map GUI elements to an abstract semantics but rather produce lexical or layout metrics, such as the number of words and text spacing, as shown on the bottom-right part of the figure.

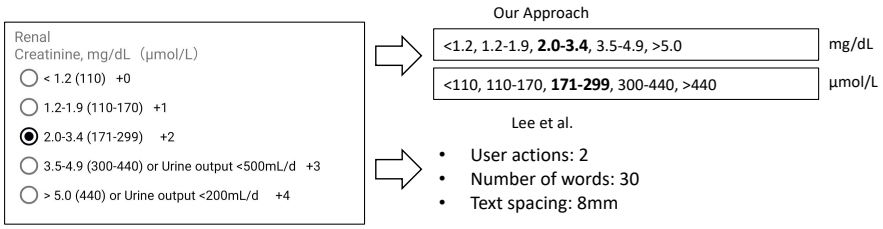
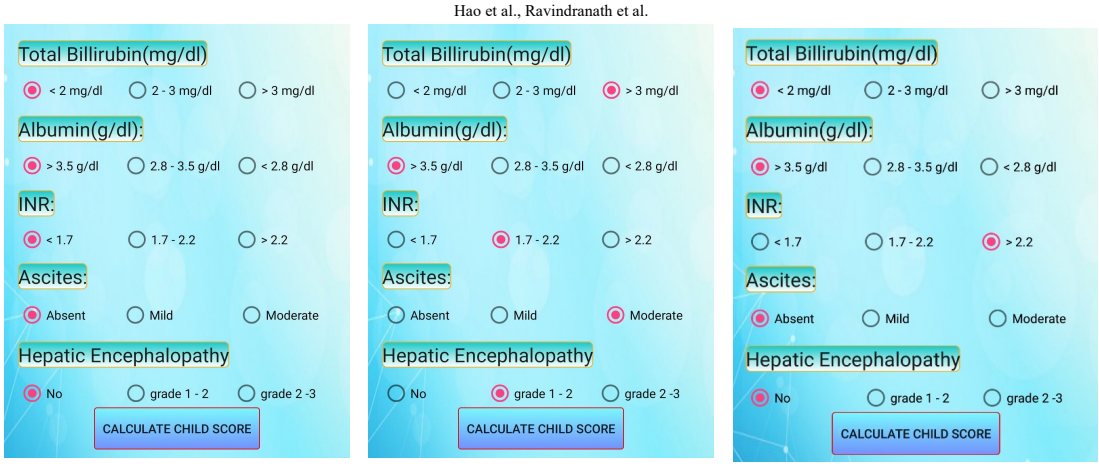


Figure 4.7 GUI of br_SOFA app (left); extracting an interval-based semantics (top-right); prior GUI extraction approaches (bottom-right).

GUI Exploration. Traditional dynamic analyzers typically explore the app based on either fixed static GUI states [149] or random GUI states (explored via Android Monkey [180, 126]). However, the lack of systematic exploration makes these approaches inadequate for identifying calculation errors that only manifest in a specific GUI state, that typically differs from the default GUI state.



(a) Default-based strategy (b) Random strategy (c) Our approach, finding the error

Figure 4.8 GUI exploration of Child-Pugh Score (KSoft Apps).

In Figure 4.8 we illustrate the inadequacy of these approaches and how our approach discovers an actual error in the Child-Pugh Score app. The error only manifests when the GUI state is exactly as shown in Figure 4.8(c), e.g., the attribute *INR* value is "> 2.2".

Defaults-based approaches use the statically-defined default GUI settings, so the GUI setting that induces the error is not explored (Figure 4.8(a)). Random-based approaches simply conduct a single random sampling of GUI settings, as illustrated in

Figure 4.8(b). This has an exponentially low chance of "stumbling" upon the error, as it would require the random configuration to land in the exact GUI state shown in Figure 4.8(c).

In contrast, our approach (Figure 4.8(c)) first reveals a specific attribute for which there exists a discrepancy with the reference score (in this example, the *INR* option parameter "> 2.2" mismatches with the reference specification "> 2.3"). Next, our DroidBot-based exploration (described in Section 4.2.3) reaches that specific GUI state (*INR* value "> 2.2") and generates the overall score. Finally, when our approach compares the app-generated score with the expected reference score, the calculation error is revealed.

Limitations. Our approach has two small limitations which can be addressed with more engineering. First, DroidBot failed to produce proper JSON GUI data in 5 cases when apps used WebView. Second, we had to correct occasional minor errors in PDF-to-CSV conversion, e.g., SOFA mixes numbers with text for Dopamine, while other scores are purely numeric.

4.3 Checking Reference Scores

We evaluated our approach on 12 medical scores. We focused on scores used in critical settings, where errors have serious implications. The scores, ranges, potential errors and action thresholds are shown in Table 4.1. We first discuss scores' nature, argue why score accuracy is critical, and then present the errors we found.

4.3.1 Reference scores

We only provide details and citations numbers (Table 4.1, first column) for those scores that contain errors.

The *SOFA* (Sequential Organ Failure Assessment) score predicts the mortality rate of an ICU patient based on the functionality of six organ systems. The score is

Table 4.1 Medical Scores Analyzed, The Year scores Were Introduced, Errors Found, Score Ranges, and Action Thresholds

	Score Name	Year	Errors		Range	Thresholds
			Interval/Value Not Covered	Overlap		
Error Found	SOFA *(11,162 citations)*	1996	Bilirubin=12.0 Creatinine=5.0	Bilirubin ([102-204], [<204])	0-24	$\geq$ 11: "higher mortality rate" [114]
Error Found	APACHE II *(21,863 citations)*	1985	Age=44		0-71	25: "predicted mortality of 50%" $\geq$ 35 "predicted mortality of 80%" [98]
Error Found	HEART *(582 citations)*	2008		Age([45-65], [$\geq$ 65]) Troponin($\leq$normal limit, 1x normal limit)	0-10	4-6:"cannot discharge; admit for clinical observation, noninvasive investigation" $\geq$ 7: "early aggressive treatment including invasive strategies" [186]
Error Found	Pulmonary Asthma Score *(157 citations)*	2002	Resp. rate (<6 yrs)=30 Resp. rate ($\geq$6 yrs)=20		0-9	9 "severe exacerbation" [189]
Error Found	RAPS "retold" *(357 citations)*	2004	Respiratory rate=5 Heart rate=39 Mean arterial press.=49		0-20	$\geq$ 7 "increased mortality" [168]
No error found	MEWS	2006			0-18	$\geq$ 4:"surgical team should be informed immediately" [118]
No error found	NEWS	2012			0-18	Any value of 3 in a parameter: "urgent ward based response" 5 or 6: "key threshold for urgent response" $\geq$ 7: "urgent or emergency response" [100]
No error found	NEWS2	2017			0-21	5 or 6: "patient should be monitored" $\geq$ 7: "urgently inform a clinician competent in the assessment of acutely ill patients" [167]
No error found	Child Pugh	1973			0-15	8-10: "increased mortality" $\geq$ 11:"hepatic failure" [215]
No error found	HAS-BLED	2010			0-9	$\geq$ 3 "high risk of bleeding" [158]
No error found	CHA2DS2VASc	2010			0-9	$\geq$ 2 "high risk of stroke and thromboembolism" [157]
No error found	Glasgow Coma Scale	1974			3-15	13-15: "mild neuroemergency " 3-5 "mortality is high and long-term neurological outcomes are generally poor" [81]

updated and calculated every 24 hours until the patient is discharged [201]. *APACHE II* (Acute Physiology and Chronic Health Enquiry II) is used to provide a general measure of disease severity while taking into account current measurements, age, and health history [140]. The *HEART* (History, EKG, Age, Risk Factors, Troponin) score is used to predict the risks of a major cardiac event while taking into account risk factors from a patient's history or age and other parameters [186]. The *RAPS* (Rapid Acute Physiology Score) predicts patient mortality in critical care transport [181]. The *Pulmonary Asthma Score* was developed to simplify children's asthma severity calculation [189].

The remaining scores do not contain errors (though apps implementing the scores do); the scores' domains are: identifying the severity of patients' conditions in critical care (*MEWS* [118], *NEWS* [100], *NEWS2* [54]); chronic liver disease severity (*Child-Pugh* [215]); risk of bleeding (*HAS-BLED* [158]); stroke risk (*CHA2DS2VASc* [157]); and severity of a brain injury (*Glasgow Coma Scale* [197]).

4.3.2 Why is score accuracy critical?

We chose these scores because they capture critical conditions, where action is urgently needed. Errors in the app-calculated scores can result in under-estimating the real score, i.e., patient state is more critical than the app indicates, which potentially means that time-critical life-saving actions will not be taken. Conversely, errors that result in the app over-estimating the real score might lead to an overly aggressive, disproportionate intervention, as well as unnecessary use of resources (personnel, ICU beds, etc.).

For each score, Table 4.1's second-to-last and last columns show the range of possible values and threshold values, respectively; the third and fourth columns show errors (if any) and will be discussed in Section 4.3.4. The threshold column is particularly revealing, as it indicates the score value(s) at which a more aggressive intervention is warranted, or values where the prognosis turns dim. For example, for the HEART score, a patient who "scores" ≤ 3 can be discharged; a patient who scores 4–6 would be admitted for noninvasive investigation; whereas a patient who scores ≥ 7 will receive "early aggressive treatment including invasive strategies". Hence, a score calculation error *at* or *around* the threshold value is particularly concerning.

4.3.3 Specification extraction accuracy

We measured the accuracy of specification extraction in terms of true positives (reference table entries that should be checked for coverage and non-overlap), true negatives (entries that should not be checked), false positives (entries that should not be checked, but our approach does check), and false negatives (entries that should be checked but our approach does not check). Specifically, the results obtained via the automated analysis (A) described in Section 4.2.2, were checked and cross-referenced by three human analyzers (H1, H2, H3) as follows.

Human Analysis. To establish ground truth, all the reference tables were verified manually by three human analyzers, H1, H2, and H3. H1/H2/H3 analyzed each table and individually recorded any observed errors, i.e., violations of coverage or non-overlap. Next, each human analyzer compared their findings against the errors reported by the automatic analysis (A) and recorded TP/FP/TN/FN. Finally, a cross-checking was performed to measure agreement. The observed multiple-rater agreement across H1, H2, and H3's findings was 100%.

Results. Across all tables, there were 418 rows, of which 319 should be checked for. The resulting confusion matrix was:

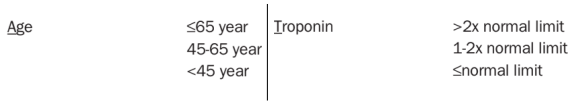
True Positives: 318	**False Positives**: 0
False Negatives: 1	**True Negatives**: 99

i.e., 100% precision and 99.69% recall. The false negative was in the APACHE II score, where the error was hidden in a supplementary, nonstandard, age adjustment footnote (relevant excerpt shown in Figure 4.9b).

4.3.4 Inconsistent reference table

We found errors in the original reference tables for 4 of the 11 scores; we also found errors in one score as defined in follow-up work to the original reference table; these errors are shown in the top part of Table 4.1.

For SOFA, as discussed in Section 4.1.3, the partition condition (1) coverage, is violated for *Bilirubin=12.0* and *Creatinine=5.0*; these values do not appear in the table though values lower or higher do appear in the table (Figure 4.1). The second issue for SOFA was a violation of the partition condition (2) non-overlap, where multiple table entries satisfy *Bilirubin* $<$ *204*. While the latter issue might be alleviated if an app/medical system does not use the μmol/l units, it is unclear how an app developer is supposed to deal with the former issue: should the 12.0 and 5.0 values be included into the left or right cells in the table?



(a) HEART score has two non-overlap violations: *Age=65* and *Troponin=(1x) normal limit*.

Add 0 points for the age <44.2 points. 45–54 years:

(b) APACHE II has a coverage violation for *Age=44*.

Score	Respiratory Rate (breaths/min) <6 Years	≥6 Years
0	<30	<20
1	31–45	21–35
2	46–60	36–50
3	>60	>50

Missing value 30

Missing value 20

(c) Pulmonary Asthma Score has two coverage violations for *Respiratory Rate*.

Physiological variable	+2	3+	+4
Body temperature	32–33.9	30–31.9	<30
Mean arterial pressure	50–69		<49
Heart rate	55–69	40–54	<39
Respiratory rate	6–9		<5

(d) RAPS "retold" has three coverage violations for *Respiratory rate, Heart rate, and Mean arterial pressure*.

Figure 4.9 Reference tables with no straightforward fixes.

The HEART score's reference table (relevant excerpt shown in Figure 4.9a) violates the non-overlap condition at two points: *Age=65* and *Troponin=normal limit*; possible resolutions include changing *Age≤65* to *Age>65* and *Troponin: ≤normal limit* to *Troponin: <normal limit*.

The APACHE II reference table [140], dating back to 1985, violates the coverage condition for *Age=44*. This table would be particularly challenging to verify manually as it has 117 entries (13 rows by 9 columns).

The Pulmonary Asthma Score's reference table (relevant excerpt in Figure 4.9c) violates coverage at two points: *RespiratoryRate=30* and *RespiratoryRate=20*; it is unclear how an app developer is supposed to cope with these, and whether the scores for those values should be 0 or 1.

The "RAPS retold" score was an interesting find. Note that the original RAPS score, introduced by Rhee et al. [181], does not violate the partition conditions. A new score, REMS, was introduced by Olsson et al. [168] to improve upon RAPS; the paper presents both scores, but the "retold" RAPS table (Figure 4.9d) has three coverage violations, as shown in Table 4.1.

4.3.5 Correcting the specification

Although we have found faulty reference tables, they do not introduce false positives, as we first translate such tables into correct ones (partition-wise), then compare apps' GUI specifications against corrected tables. We now discuss approaches for, and challenges associated with, fixing the incorrect tables.

Straightforward case One of the specification errors found in the SOFA reference table (Figure 4.1) for Bilirubin, originally *'<204'*, can be easily fixed by changing it to *'> 204'*.

"Reasonable" fix In addition, the SOFA reference table has no coverage for *Bilirubin=12.0 mg/dl*. However, by observing that $204\mu mol/l = 11.93mg/dl$ we could infer a reasonable fix, that is, to change '> 12.0' to '≥ 12.0'. Similarly, *Creatinine=5.0 mg/dl* is not covered, but since $440\mu mol/l = 4.98mg/dl$, a reasonable fix would be to change '> 5.0' to '≥ 5.0'.

Challenging cases In the remaining cases (Figure 4.9), it is unclear how to "fix" the specification. To fix the HEART reference table (Figure 4.9a), *Age=65* can be assigned a score of either 2 or 1. In fact the paper itself is ambiguous as it says "one point if the patient was between 45 and 65 years and two points if the patient was 65 years or older" [186]. Similarly, in the case of the Pulmonary Asthma score (Figure 4.9c), *RespiratoryRate=30* and *RespiratoryRate=20* can be assigned score values of either 0 or 1. For the APACHE II reference table (Figure 4.9b), *Age=44* can be assigned 0, 1, or 2 points. Finally, for RAPS retold (Figure 4.9d), different score interpretations can be made for *RespiratoryRate=5*, *HeartRate=39*, and *MAP=49*.

How Developers Cope With Faulty Scores. We found 14 instances where developers attempted to correct a faulty score when implementing that score in their apps. For SOFA, we found 4 apps that performed the straightforward *'> 204'* fix, and 4 apps that used the reasonable '$Bilirubin \geq 12.0$' and 'Creatinine ≥ 5.0' fixes.

For APACHE II, 4 apps performed an ad-hoc fix (changing $Age < 44$ to $Age \leq 44$). Finally, for the HEART score, which has overlap errors at *Age=65* and *Troponin* $\leq$ *normal limit*, no app has fixed the overlap errors; moreover, three apps have introduced additional overlap errors at *Age=45*.

We can now summarize our RQ1 findings.

> ***RQ1: Can our approach extract and verify medical score specifications?***
>
> *Answer: Yes. Specification extraction has a 99.5% F1-score and verification has uncovered all 11 violations in the 5 incorrect scores.*

4.4 Finding Errors in Apps

We have evaluated our approach on a dataset of 90 apps; the selection process is explained next, followed by a discussion of the errors we found, effectiveness, and efficiency.

4.4.1 App dataset

We selected our apps from the Medical category on Google Play. We scraped 3,762 apps and their descriptions from Google Play; using ranked retrieval, we identified 556 apps classified as medical calculators. We then focused on apps which computed one or more among the 12 scores we verified, resulting in a total of 90 apps.

4.4.2 App errors: inconsistent GUI

We now present our findings: coverage violations and non-overlap violations.

Coverage violations. Table 4.2 shows the results; we found 23 coverage errors in 11 apps. The first column shows the official app name on Google Play, the second column shows the affected score calculator, while the third column shows values or ranges that

Table 4.2 Inconsistent GUI: Coverage Violations

App Name	Score	Parameter value(s)
Sepsis Clinical Guide	APACHE II	Hct(%) =60
Child Pugh Calculator	Child Pugh	INR =1.7
Child-Pugh Score (Blue Rock)	Child Pugh	INR $<$ 1.7, INR $>$ 2.2
HAS-BLED Score	HAS-BLED	Age=65
Nursing Calculator	MEWS	Systolic BP=70, Resp=8, Temp=35.0
Quick EM	SOFA	PaO2 $\geq$ 400, Dopamine=5
Nursing	SOFA	Bilirubin=12.0, Creatinine (mg/dl)=5.0
SOFA Score (widebitsbd)	SOFA	GCS=15, PaO2 $\geq$ 400, Platelets $\geq$ 150, Creatinine (mg/dl) <1.2
SOFA Score (Blue Rock)	SOFA	Bilirubin=12.0, Creatinine (mg/dl)=5.0, Dopamine=5
Merck Manual Professional	SOFA	Bilirubin=12.0, Creatinine (mg/dl)=5.0
SOFA	SOFA	Bilirubin=12.0, Creatinine (mg/dl)=5.0

are not covered. Interestingly (though somewhat predictably), the "original sin" in the SOFA reference table (no coverage for *Bilirubin=12.0* and *Creatinine=5.0*) leads to non-coverage issues for those parameter values in four apps.

Non-overlap violations. Table 4.3 shows the results; we found 32 non-overlap errors in 12 apps. The parameters with overlapping ranges are shown in the third column. In this case, all the *HEART* score apps' errors appear attributable to the error in the original HEART reference table (Table 4.1).

4.4.3 App errors: incorrect score calculations

Table 4.4 shows errors in score calculation; we found 16 calculation errors in 16 apps. The *Calculation Errors* grouped columns show the parameter values for which the errors manifest, and the app value vs. reference score value. We make several observations. First, app errors lead to both under-estimating the true score (e.g., apps Atrial fibrillation risk calc, MEWS, Nursing Calculator-MEWS) and over-estimating the true score (e.g., apps Sepsis3 or MediCalc). Both error types are problematic due to potential under-intervention and over-intervention respectively, as explained in Section 4.3.2.

Table 4.3 Inconsistent GUI: Non-overlap Violations

App Name	Score	Overlapping Ranges
Child-Pugh Score	Child-Pugh	INR: [>1.7], [1.7-2.2]
HEART Score	HEART	Age: [≤45], [45-65], [≥ 65] Troponin: ≤normal limit, 1-3x normal limit, ≥3x normal limit
HEART Score Calculator	HEART	Age: [45-65], [≥ 65] Troponin: 1-3x normal limit, ≥3x normal limit
Quick EM	HEART	Age:[45-65], [≥ 65], Troponin: ≤normal limit, 1-3x normal limit
HEART Score Gumption	HEART	Age: [45-65], [≥ 65] Troponin: ≤normal limit, 1-3x normal limit, ≥3x normal limit
REBELEM	HEART	Age: [≤45], [45-65], [≥ 65], Troponin: 2-3x normal limit, ≥3x normal limit
Medical Calculators	HEART	Age: [≤45], [45-65], [≥ 65] Troponin: ≤normal limit, 1-3x normal limit, ≥3x normal limit
MEWS	MEWS	Heart Rate: [51-101], [101-111], Respiratory rate: [15-21], [21-30] Systolic BP: [71-81], [81-101], Temperature: [≤35], [35-38.5],[≥38.5]
Nursing Calculator	SOFA	Bilirubin: [1.2-1.9], [1.0-5.9], Creatinine (mg/dl): [1.2-1.9][1.0-3.4] Platelets: [≥ 150], [100-150]
SOFA Score (Blue Rock)	SOFA	Creatinine (μmol/L): [110], [110-170],[300-440],[440]
SOFA	SOFA	Norepinephrine: [≤0.1], [<0.1]
Sepsis Clinical Guide	SOFA	Creatinine (mg/dl): [≤1.2],[1.2-1.9], Platelets: [≥ 150], [≤ 150]

Second, as the last column indicates, certain errors "straddle" the threshold, which, as discussed previously, can put the patient in a different class.

We have reached out to the developers of apps where we found errors. So far, 6 apps (listed in Table 4.5) have been fixed and updated.

4.4.4 Effectiveness

We measured the accuracy of our automated app analysis via a process similar to the one described in Section 4.3.3. Specifically, the results obtained via the automated analysis (A) described in Section 4.2.3 were checked and cross-referenced by three human analyzers (H1, H2, H3) as follows.

Human Analysis. To establish ground truth, all the apps were first explored manually by three human analyzers (H1, H2, H3). The analyzers performed several tasks: (a) ensure that the app implements a score that was among our examined scores, (b) find the app activity implementing the score (which formed the start of the automated analysis), (c) check for inconsistent GUIs, and (d) exercise all GUI

Table 4.4 Calculation Errors in Apps

App Name	Score	Calculation Errors			Meets or Exceeds Threshold
		Parameter Option	App Score	Ref. Table Score	
Atrial fibrillation risk calc	CHA2DS2VASc	Age=75	1	2	N
Child-Pugh Score (KSoft Apps)	Child Pugh	INR=2.3	11	10	Y
Child-Pugh Score (Blue Rock)	Child Pugh	INR=2.3	11	10	Y
Child-Pugh Score (Liver)	Child Pugh	INR=2.3	11	10	Y
Nursing Calculator	MEWS	Heart rate=40	3	4	N
MEWS	MEWS	Temperature=[35.1-36]	3	4	N
Nursing Calculator	SOFA	Platelets=150	12	11	Y
Sepsis 3	SOFA	Dopamine=5	12	11	Y
Nursing	SOFA	PaO2=300	12	11	Y
MediCalc	SOFA	Dopamine=5	12	11	Y
SOFA Score (widebitsbd)	SOFA	Creatinine (μmol/L)=106	12	11	Y
Merck Manual Professional	SOFA	PaO2=300	12	11	Y
SOFA	SOFA	Creatinine(μmol/L)=106	12	11	Y
SOFA - (Sepsis)	SOFA	Dopamine=5	12	11	Y
Sepsis Clinical Guide	SOFA	Bilirubin=1.2	12	11	Y
Sepsis SOFA Calculator	SOFA	Platelets=20	4	3	Y

options and check for score calculation errors. Next, each human analyzer compared their findings against the errors reported by the automatic analysis (A) and recorded TP/FP/TN/FN. A cross-checking was performed to measure multiple-rater agreement among the errors discovered. The observed multiple-rater agreement across H1, H2, and H3's findings was 100%. A fourth human H4 (the research lead) performed a final cross-check between the manual and automatic analysis results. Note that the manual app analysis was a considerable task, due the large space induced by the number of apps, number of scores, and manual GUI interaction; overall the task took H1, H2, and H3 about four person-months.

Table 4.5 Apps Fixed Thanks to Our Reporting

App Name	Package Name	#Installs
Nursing	pe.com.codespace.nurse	100,000
MEWS Brasil	appinventor.ai_blinkeado.InformaticasaudeMEWS	500
Atrial fibrillation	com.gumptionmultimedia.atrialfibrillationriskscore	5,000
Nursing Calculator	com.niya.lijo.nursingcalculators	50,000
Sepsis Clinical Guide	app.escavo.sepsis	100,000
SOFA	gumptionmultimedia.com.sofascore	1,000

Results. The confusion matrix resulting from comparing the automated analysis findings with ground truth was:

True Positives: 20	**False Positives**: 0
False Negatives: 5	**True Negatives**: 65

The false negatives are due to apps using WebView (Section 4.2.3). These figures, a 100% precision and 80% recall, are par for the course for a dynamic analysis.

4.4.5 Efficiency

Table 5.5 provides details on the efficiency of our approach. The median size of app bytecode alone (`.dex`) was 3.8MB; note that app size (`.apk`) would be much larger as that includes app resources. A typical app takes about 3 seconds to analyze. Reference table verification, including running Z3, took less than 1 second for any table; for any app, GUI extraction followed by verification/validation took at most 2 seconds.

Table 4.6 Efficiency Results

Analysis time (seconds)			Bytecode size (MB)		
min	max	median	min	max	median
2	3	3	0.17	125	3.8

We can now summarize our RQ2 findings.

RQ2: Is our approach effective at analyzing and finding errors in real-world apps?

Answer: Yes. The analysis could tackle all 90 real-world apps, achieving 100% precision and 80% recall, and taking 3 seconds per app on average.

4.5 Summary

Mobile health apps are seeing increasing adoption in acute care settings, and mobile app developers, including developers *who are not medically qualified*, are eager to capitalize on this growing demand. Though errors in medical scores and score calculator apps can have severe negative consequences, at this point, such scores and apps are subject to no scrutiny. We tackle this issue via rigorous, automated approaches: (1) extracting reference tables into interval-based specifications and checking them for partition violations, and (2) validating apps against the aforementioned specification, as well as verifying app GUIs. We have uncovered errors in long-standing medical reference articles. We found that incorrect specifications translate to incorrect app implementations, and that even correct specifications can be implemented incorrectly, affecting the resulting scores. Our findings indicate a need for tighter scrutiny of reference scores themselves, as well as apps implementing these scores.

In the next chapter, we will delve into the detection of potential user-data save and export losses resulting from Android app termination.

CHAPTER 5

USER-DATA LOSSES DUE TO ANDROID APP TERMINATION

Saving user data as files onto local storage is a common feature in Android apps. However, the volatile nature of the mobile environment can lead to system-initiated app terminations without notice, resulting in the loss of unsaved data. Testing apps for such potential data losses presents challenges, including identifying user data originating from inputs or actions and reproducing termination scenarios at the right moment. In this chapter, we propose an approach to find potential "lost writes"– data supposed to be written to files but not saved due to system-initiated termination. Our approach employs static analysis to identify potential losses and dynamic verification to confirm errors. Evaluation of a large set of apps revealed numerous cases of data losses that were not detected by existing tools designed to find volatility errors in Android apps. In Section 5.1, we delve into the background and motivation of our research. We discuss the impact of system-initiated termination on file writes and illustrate losses in example apps. Identifying potential losses presents challenges, requiring us to define and identify the data to be saved and understand how this data flows to files. To address these challenges, we introduce an automated approach that combines static and dynamic analysis. We use static analysis to identify user-initiated file writes, producing a list of objects that may be lost due to termination (Section 5.2.1). To verify these losses, we compare system call traces from the original and terminated executions (Section 5.2.2). Evaluating our approach on 2,220 apps revealed confirmed losses in 163 apps, impacting user settings, artwork, edited photos, notes, history, and bookmarks (Section 5.3.2). We also compare our approach with other state-of-the-art methods, highlighting the superiority of our approach in identifying system-initiated

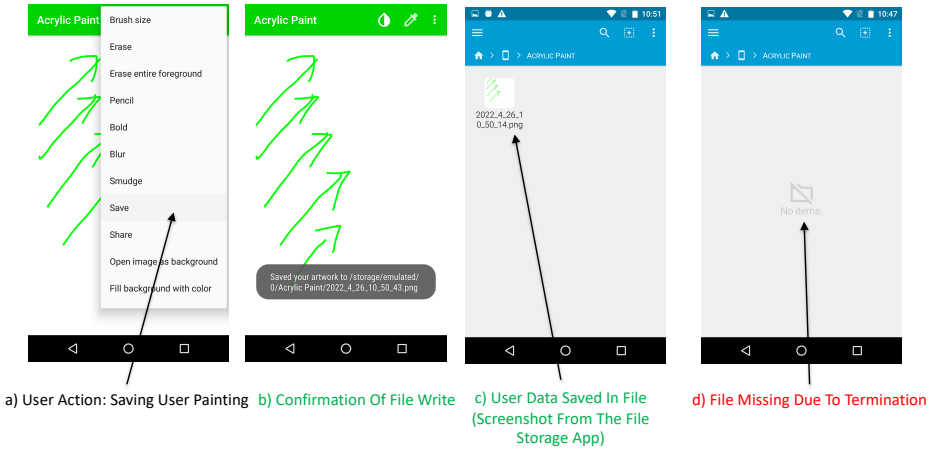


Figure 5.1 Acrylic Paint app: success scenario (a-c) and user file write data loss scenario (d).

termination-induced data losses (Section 5.3.3). Finally, we propose potential solutions to address file write losses (Section 5.3.7).

5.1 Motivation

A fundamental principle in mobile app development – on both Android and iOS – is to "not perform file writes on the synchronous UI thread, to keep the UI responsive" [198, 87]. This forces programmers to run file write operation in a separate thread, e.g., asynchronously via an AsyncTask in Android. However, asynchronous (and potentially time-consuming) file writes are on a collision course with the volatility of mobile platforms. Specifically, mobile apps can be terminated without notice (or on short notice) by the system, due to low memory or runtime changes. We detail this by first presenting a brief overview of Android app construction, file writes in app, and app termination; next, to motivate our approach, we present a suite of examples of file write losses due to termination.

5.1.1 Background: file writes and termination in Android

App Construction and File Writes. Android apps are constructed from four fundamental components: Activities managing the UI, ContentProviders managing access to data, Services that run in the background, and Broadcast Receivers which

```
1  public boolean onOptionsItemSelected(MenuItem item) {
2      switch (item.getItemId()) {
3       ....
4          case R.id.save_menu:
5              takeScreenshot(true);
6              break;
7       ....}}
8  private File takeScreenshot(boolean showToast) {
9       ......
10     Bitmap copyBitmap =
           cachedBitmap.copy(Bitmap.Config.ARGB_8888, true);
11     File file = new File(path,"fileName.png");
12     FileOutputStream output = new FileOutputStream(file);
13     copyBitmap.compress(CompressFormat.PNG, 100, output);
           //Process gets terminated here
14      .....}
```

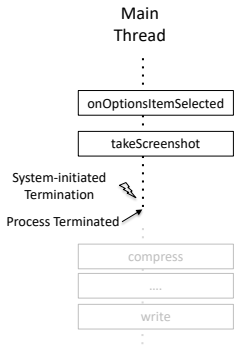


Figure 5.2 Acrylic Paint app code (left) and file write operation termination events (right).

respond to system-wide events. The most common component, Activity, is a "page" in the app; apps with a UI typically consist of one or more Activities. The UI elements in an Activity are owned and managed by a special thread, named the *UI thread* or *main thread*. The UI thread plays a critical role: timely processing of UI events, to keep the UI (and app) responsive. Therefore, one of the first lessons in Android programming is "you should not perform work on the UI thread" [198]: as file write is potentially long-running and blocking, performing a file write on the UI thread could render the app unresponsive. Hence, any long-running or blocking operations should run asynchronously, in a different (background) thread. Occasionally though, apps violate this requirement: among the apps we have analyzed, some perform file writes on the UI thread.

App Termination. There is an inherent tension between long-running operations and the constraints of the mobile platforms. Unlike desktop/server applications, mobile applications cannot expect to "run forever"; rather, mobile applications can be terminated summarily to free resources such as memory [148], conserve energy, and protect user's security (e.g., by preventing background apps from accessing users' location). Therefore, long-running operations can be interrupted or terminated without

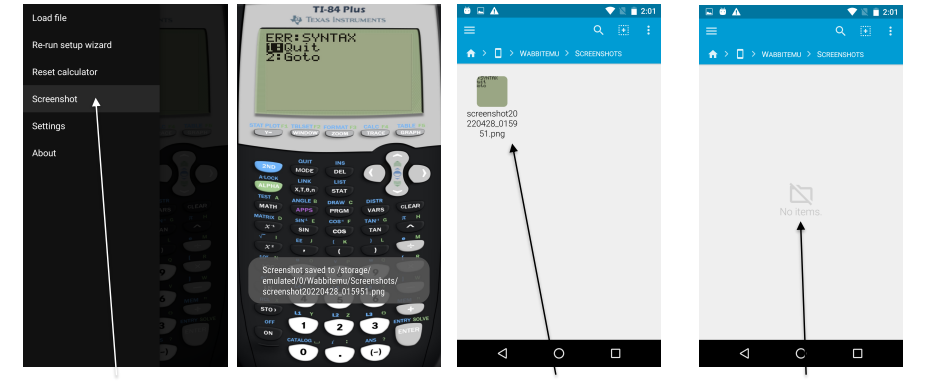


Figure 5.3 Wabbitemu app: success scenario (a-c) and user file write data loss scenario (d).

notice – in fact the entire process enclosing the Android app is terminated – for various external reasons, as listed below.

- **Memory pressure.** Android's Low Memory Killer Daemon (LMKD) [85] handles low-memory situations: when the phone is under memory pressure, the LMKD ranks apps based on their memory usage and acts according to a configuration-described policy, e.g., the app which consumes most memory and is not in the foreground will be killed to release memory.[1]
- **Background process limit.** The Android OS provides a developer option to limit the number of background processes. When the option is set to "No background processes" an app process is killed whenever the app is not in the foreground.
- **Kill via external signal.** Apps can also be terminated by sending them a traditional Unix signal, e.g., SIGKILL.

Besides system-initiated termination, an app process can also be terminated internally, when the app invokes API methods such as System.exit or finishAndRemoveTask; the use of these APIs in our app dataset was practically non-existent, so our approach focuses on system-initiated termination.

[1]In smartphones, I/O consumes substantial energy [173], hence, apps that run background I/O are at higher risk of termination due to resource pressure.

```
1  private class ScreenshotCalcTask extends AsyncTask{
2  protected void onPreExecute() {...}
3  protected Boolean doInBackground() {
4      SaveScreenshotCalc();
5  }
6  void SaveScreenshotCalc() {
7       .....
8      Bitmap Screenshot =
           Bitmap.createScaledBitmap(screenshot,
           screenshot.getWidth() * 2,
           screenshot.getHeight() * 2, true);
9      try {
10         FileOutputStream out = new
               FileOutputStream(new File(outputDir,
               "screenshot" + new
               SimpleDateFormat("yyyyMMdd",
               Locale.getDefault()).format(new Date())
               + ".png"));
11         Screenshot.compress(CompressFormat.PNG,
               100, out); //Process gets terminated here
12          ....
13         }
14       ....
15     }
16 protected void onPostExecute(Boolean success) {...} }
```

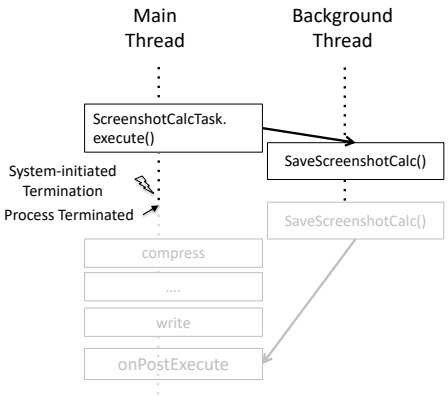


Figure 5.4 Wabbitemu app code (left) and file write operation termination events (right).

5.1.2 Motivational examples

We now present several examples where file write loss can occur due to system-initiated termination.

Example: File write on the main thread. We show a case study on the Acrylic Paint app [77] in Figure 5.1. Specifically, we show two scenarios: a successful scenario where file write operation completes, and an unsuccessful scenario where the file write is eschewed due to system-initiated termination, leading to user's work being lost. Acrylic Paint is a painting app: the user can save painting progress either by clicking the 'Save' button (Figure 5.1(a)) or by exiting the app. The user's progress or changes to the drawing are saved in a local file inside the app directory, as shown in the middle part of the figure: Figure 5.1(b) shows the confirmation message, whereas Figure 5.1(c) shows the saved painting as a file in the app's directory. However, in the case of a system-initiated termination such as low memory, the app process is terminated before

the user's changes, or progress, can be saved. In that case, illustrated in Figure 5.1(d), the valuable file write data is lost and the file is missing from the app's local directory.

In Figure 5.2 we show the app source code (left) and event sequence diagram (right). When the user clicks the 'Save' menu option, the onOptionsItemSelected event is triggered (lines 1–4). The method takeScreenshot is called next (line 5); inside the method, new Bitmap, File, and FileOutputStream instances are created (lines 10–12). Finally, the file writing operation Bitmap.compress (line 13) executes on the main thread. If the app is terminated prematurely, the file write operation is terminated, resulting in data loss. The sequence of events, i.e., code that will not execute due to termination, is shown in gray.

Example: file write on a background thread. Next, we show an example of file write loss due to termination, where the file write operation is performed on a background thread. Wabbittemu [203] is a graphic calculator app. We show relevant app screenshots in Figure 5.3: the user can save calculator screenshots locally in a file by selecting the 'Screenshot' menu item (Figure 5.3(a)); the save confirmation is shown in Figure 5.3(b), whereas Figure 5.3(c) confirms the file's presence in the app's own directory. The file write loss scenario, due to system-initiated termination, is shown in Figure 5.3(d): data is lost, hence, the file is missing from the directory.

Figure 5.4 shows the relevant source code (left) and event sequence diagram (right). When the user selects the screenshot menu item, the ScreenshotCalcTask executes. Note that ScreenshotCalcTask extends the AsyncTask class hence, will execute asynchronously as follows: ScreenshotCalcTask invokes the SaveScreenshotCalc method on a background thread (line 4). Inside the SaveScreenshotCalc method, first, Bitmap and FileOutputStream instances are created (lines 8,10), then the screenshot image is written into a file (line 11). App termination in turn terminates both the main thread and its (child) background thread, hence, the file write data will be lost; specifically, the file write operation and

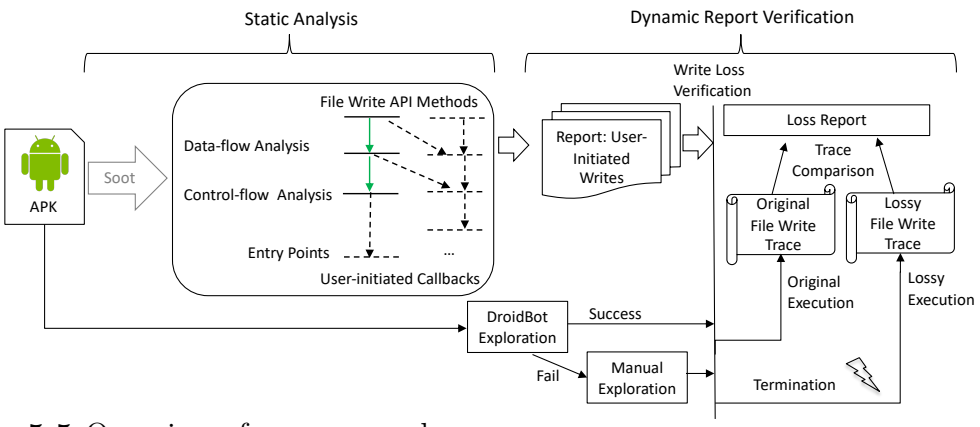


Figure 5.5 Overview of our approach.

the onPostExecute method do not execute. The gray-colored part of the sequence diagram shows the parts that will not execute due to termination.

Hence, our goal is to automatically identify the file-bound data and file write operations that are lost (and do not execute, respectively) due to termination.

5.2 Approach

In Figure 5.5 we provide an overview of our approach. Given an Android app we first perform a *static analysis* which produces reports of *user-initiated file writes*, i.e., file data that originates from UI events, hence, is potentially lost. Then, we verify the potential losses in a *dynamic report verification* phase. We now discuss each phase in detail.

5.2.1 Static analysis

In this phase, we perform a combination of control and data flow analyses to identify data that originates from user interaction (UI) and flows into file write APIs; this data is marked as potential loss.

Our static analysis has two main objectives:

- Finding all the file write operations initiated by (generated from) user interactions with the app, and

- Tracking the data that contains user input and flows into the aforementioned file write operations.

We first define "user-initiated" more precisely, then proceed to describe how we achieved the aforementioned objectives using a combination of control-flow and data-flow analysis.

What is "User-initiated". A key requirement for finding potential data losses is defining exactly what data is "worth saving". Intuitively, user's work or changes are worth saving, whereas logging operations happening in analytics libraries are not. Hence, we define as "worth saving" two kinds of *user-initiated* file write operations.

We consider file write operations where the file content is coming directly from UI input, e.g., the canvas in the Acrylic Paint app shown in Figure 5.1, which has type ImageView. Another example is note contents, whose type is TextView. These file writes are identified via data-flow analysis.

The second kind of worthy content is file-written but does not come from UI input; rather, the file write operation depends on user interaction with the UI such as saving screenshots or exporting log data. One such example is the 'Export to Excel' UI action in the Auto-Away app to export call history log mentioned in detail in Section 5.3.2. Though in that case the exported file does not contain UI data (in contrast to the painting content above), the user nevertheless initiates this file write or export operation. We find these type of file writes via control-flow analysis.

Defining UI Interaction Callbacks. Android apps do not have a main() method; rather, apps have multiple entry points induced by top-level callback events, as follows. Apps can register callbacks for events of interest, such as GPS location updates or UI interaction (menu select, button click, etc.). We create a "dummy" main method (similar to other static analyses such as FlowDroid [88]) which contains all top-level callbacks and will serve as an end point for backward analyses. Among the top-level

```
1  class SketchActivity extends
       AppCompatActivity{
2  public boolean
       onOptionsItemSelected(MenuItem item) {
3        int id = item.getItemId();
4        if (id == R.id.action_reminder) {
5              ....
6        }else if (id == R.id.action_save) {
7              ...
8              saveToExternalStorage();
9              ...
10       }
11       ...
12     }
13 public void onRequestPermissionsResult(int
       requestCode,..) {
14     switch (requestCode) {
15           ....
16           saveToExternalStorage();
17     }
18 }
19 private void saveToExternalStorage(){
20       .....
21     Bitmap bm = overlay(new
           BitmapDrawable(getResources(),
           mFilePath).getBitmap(),
           drawView.getBitmap());
22     ...
23     bm.compress(Bitmap.CompressFormat.JPEG,
           100, new FileOutputStream(file));
24     ....
25 } }
```

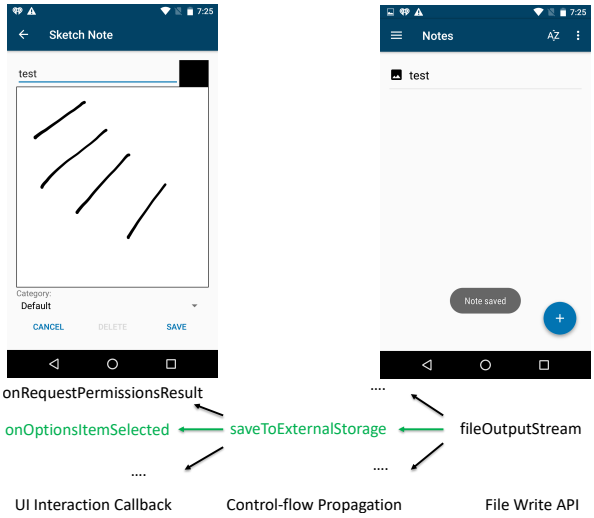


Figure 5.6 Backward control-flow analysis in the Privacyfriendlynotes app.

callbacks, we only retain UI-related ones. There are several UI interaction callback APIs in Android, such as OnClick, onMenuItemClick, etc. Generally these callbacks are part of the Android View (UI) components hence, defined in the android.view class hierarchy. Non-UI callbacks are not a target of our analysis. For example onLocationChanged, defined in android.location.Location and handling GPS location updates, does not correspond to user interaction and is not considered a UI interaction endpoint.

Defining File Write APIs. Our analysis focuses on file write operations. We manually identified 15 java.io classes that support file writes. The first column of Table 5.1 lists these classes and the second column states the frequency of the APIs observed in our evaluated app dataset. Within these classes, we identified API methods that perform file writes, e.g., FileOutputStream.write(), Writer.append(), etc. We observed that

Table 5.1 File Write API Prevalence

File Writing API	% Apps
OutputStream	89
FileOutputStream	82
Writer	81
FilterOutputStream	81
ByteArrayOutputStream	79
StringWriter	72
BufferedOutputStream	66
BufferedWriter	54
ObjectOutputStream	47
DataOutputStream	43
FileWriter	15
PrintStream	10
PrintWriter	9
CharArrayWriter	3
FilterWriter	1

```
1  private File takeScreenshot(boolean showToast) {
2        ......
3      View v = findViewById(R.id.CanvasId);
4      v.setDrawingCacheEnabled(true);
5      Bitmap cachedBitmap = v.getDrawingCache();
6      Bitmap copyBitmap =
           cachedBitmap.copy(Bitmap.Config.ARGB_8888,
           true);
7      File file = new File(path,"fileName.png");
8      FileOutputStream output = new
           FileOutputStream(file);
9      copyBitmap.compress(CompressFormat.PNG, 100,
           output);
10      .....}
```

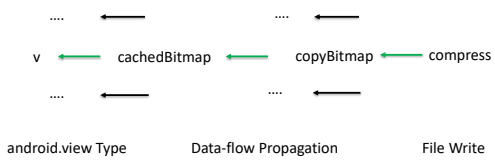


Figure 5.7 Backward data-flow analysis in the Acrylic Paint app.

the most common API classes were the *Stream and *Writer families, e.g., OutputStream, FileOutputStream, Writer. Note that this list is just an input configuration file in our implementation hence, can be easily extended.

Finding User-initiated File Writes. Our static analyzer is built on top of the Soot analysis framework [192]. Using Soot, we first build an inter-procedural call graph, which will form the basis for the control- and data-flow analyses.

To find all the file write calls originating from user interaction, we proceed as follows. We perform a backward control-flow analysis from every file write callsite back

to its app entry point. If the callback belongs to UI interaction callbacks (Section 5.2.1), then the write operation is initiated by the user, and we add this write to our list.

Example. We show an example of how our backward control-flow analysis operates in Figure 5.6. The example is drawn from the Privacyfriendlynotes app (a simple note-saving app). The relevant source code is shown on the left, and the corresponding control-flow diagram along with the app UI screenshots are shown on the right. We start our backward analysis from the file write API. In this case, the file write API is on line 23: the compress method call (taking a FileOutputStream as an argument) is the point where the note or sketch are saved in a file. From compress our analysis lands in saveToExternalStorage; backtracking from the saveToExternalStorage method leads to two different paths, as the method has two callers. One of them is onRequestPermissionsResult (lines 13–18) which does not belong to android.view class, hence, is not a UI interaction callback. The second control-flow path traces back to onOptionsItemSelected which belongs to the android.view.MenuItem class, hence, is a UI interaction callback. When the user selects the 'SAVE' option, the onOptionsItemSelected callback is triggered and saveToExternalStorage is called (lines 6–8). Hence, this particular file write falls under the category of writes initiated by the user. The control-flow path that satisfies our requirement and belongs to the "user-initiated file write" path is marked with green color in the figure. Hence, we add the file write operation on line 23 as a user-initiated file write.

Finding User Input Flowing to File Writes. To find the extent of the data flowing to file write APIs, we perform a data-flow analysis. Similar to the control-flow analysis, we start our data-flow analysis from the write callsite (but now consider the method arguments) and trace whether the data transitively flows from a UI input class. Starting from the file write API callsite, we run a backward data-flow analysis up to the point where the data type belongs to a UI input type (android.view class) or exits via an app entry point. We now illustrate this analysis with an example.

Example. We show the data-flow analysis of the Acrylic Paint app in Figure 5.7. The relevant source code is shown on the left, and the corresponding data-flow edges on the right. The method in consideration here is takeScreenshot, which contains the file write call compress (line 9). Data-flow analysis of copyBitmap leads to line 6, specifically cachedBitmap. Tracing back from cachedBitmap leads to line 5, value v, which belongs to the android.view class as per line 3. Therefore, we end the data-flow analysis here, concluding that the file write content is coming from an UI input; in this example, it belongs to ImageView type UI input. Therefore, we add v as potential loss.

5.2.2 Dynamic report verification

Our dynamic verification phase reduces the false positives resulting from the static analysis phase. Given the list (reports) of user-initiated writes produced by the static analysis, we proceed to verify the potential losses report. Dynamic report verification has multiple components as discussed below:

GUI Exploration. Our goal is to explore the target app to find relevant file write initiating action (i.e., button click to initiate `Save/Export`) and then inject a termination event which should lead to a "lossy" execution and finally compare file write traces in the original and lossy executions. Those writes that are confirmed missing will help us verify whether the file writes containing user data is lost.

DroidBot Exploration. We have used DroidBot[155] to automate the app exploration or trace generation process. DroidBot's GUI-based model helped to automatically identify various view objects (such as Button or TextView) related to target user `Save/Export` actions. We wrote custom DroidBot scripts to identify these target GUI elements and trigger necessary events (e.g., clicking `Save` button).

Manual Exploration. Besides DroidBot, we have explored apps manually (human-driven) for cases where DroidBot failed. DroidBot failed in two types of scenarios. First, the scenario where DroidBot could not reach targeted `Save/Export` options and

the collected traces did not have the desired file write logs. Second, the cases where automated GUI exploration using DroidBot crashed and no trace logs were generated.

Triggering Termination. While termination can be triggered via various system events (Section 5.1.1), we used the background process limit option, based on the observation that background apps are frequently/routinely terminated due to memory pressure [148, 210, 178, 86].[2] In other words, if the user switches away from app A (which in the absence of termination would perform a file write operation, either on the main thread or on a background thread) to app B, the file write operation can be terminated, resulting in data loss. We automated switching from target app A to app B via Monkey [83] (the `adb shell monkey -p package.name` command offered by the Android Debugging Bridge). In case of manual exploration, we manually switched from the target app to another.

Trace Comparison. We confirm the write loss via automated trace differencing: we compare the `strace` Linux-level system call trace [195] across two runs: original execution (normal, uninterrupted) and lossy execution (operation interrupted by triggering termination via app switching). We have automated trace differencing by checking for interrupted I/O, e.g., unfinished openat(), fstat(), or write() system calls. We show an example of successful vs. lossy `strace`s for app PrivacyFriendlyNotes in Figure 5.8. The successful execution trace is shown on top – note the openat() call completing (lines 2–3). The lossy trace is shown on the bottom: the openat() call fails (unfinished) as shown in lines 6–7. Hence, aside from the visual confirmation of file data loss (e.g., Figure 5.1(d), Figure 5.3(d)), which is not scalable for a large set of apps, we automated the dynamic verification process via `strace` differencing; this dynamic verification is key to achieving a low false positive rate (Section 5.3.5).

[2]Typical background process limit (number of concurrent apps): 16 apps for mobile devices with 1GB memory and 26 apps for 2GB memory [178].

```
1 // (a) succesful strace
2 openat(AT_FDCWD,"/data/user/0/org.secuso.privacyfriendlynotes/
3  files /sketches/sketch_1606849053910.PNG", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 71
4
5 // (b) lossy strace
6 openat(AT_FDCWD,"/data/user/0/org.secuso.privacyfriendlynotes/
7  files /sketches/sketch_1607044227961.PNG", O_WRONLY|O_CREAT|O_TRUNC, 0666 <unfinished ...>
```

Figure 5.8 `Strace` differences between a) successful file write and b) lossy execution in app PrivacyFriendlyNotes.

Table 5.2 App Selection and Findings

	#Apps
Save/Export in UI	2,953
Soot Successful	2,220
Contains File Writes	1,476
User-initiated File Writes	298
Confirmed Losses	*163*
Automatically	*107*
Manually	*56*

5.3 Evaluation

We now discuss our evaluation in detail. We introduce our dataset, then quantify the effectiveness of our approach. Next, we present 27 examples of verified losses. We then compare our approach with existing tools, and quantify our analysis' efficiency. Finally, we discuss the limitations of our approach.

Dataset and test platform. To evaluate our approach, we focus on apps that support saving or exporting user data. First, we collected an app dataset of interest from the main Android app store, Google Play, and the open-source app store, F-Droid. To identify apps that offer Save or Export facilities, we used an automated filtering process on GUI data: more precisely, we extracted app resource XMLs from 20,000 popular Android apps split across across various app categories, and retained those apps whose GUI resources (e.g., buttons or menus) match keywords such as `Save`, `Export`, or similar. We found 2,953 apps whose GUI matched our keywords of interest. Next, we excluded 733 apps that could not be analyzed with Soot (on top of which we built our analysis). Soot failing on certain commercial apps is unsurprising, because popular Google Play apps tend to employ anti-analysis techniques such as packing or

Table 5.3 Categories of Confirmed Losses

Category	**# Apps**
App-specific/misc.	61
Notes, documents, PDF files	28
Image, audio, video files	19
Database backups	17
Reports	14
Painting	8
Settings or preferences	6
History	4
Recipes	4
Schedules	2

obfuscation. We ran our dynamic analysis on a Nexus 5 smartphone running Android 6 (API level 23).

5.3.1 Effectiveness

Table 5.2 summarizes our evaluation results, and is discussed in detail next. Among the 2,220 apps where Soot ran successfully, we identified those apps whose bytecode contained file write APIs, yielding 1,476 apps. We found that certain categories of file writes are not of interest: they are performed by third-party libraries, e.g., tracking and analytics packages that write into log files. We configured our analysis to ignore such writes – because they consist of logging and analytics data, they are out of our purview. Instead, our focus is on *user-initiated* writes as mentioned in Section 5.2.1: the static analysis has identified 298 such apps. Among the 298 candidate apps (apps with potential losses) having user-initiated file writes, we found and confirmed losses in 163 apps using our semi-automated dynamic verification approach – a combination of automated and manual analysis discussed next. We show the summary of our evaluation in Table 5.2.

We categorize these confirmed losses in Table 5.3. Most of the losses fall under the app-specific data category (e.g., saving newborn vitals for a baby care app). Other categories of lost data include notes, photos, database backups, artwork, or settings.

DroidBot Exploration Results Using our automated GUI exploration with DroidBot, we produced trace logs for 177 cases out of 298 potential losses. DroidBot crashed and could not generate traces for 121 cases. This automated DroidBot-driven approach verified losses in 107 cases by comparing original and lossy execution traces. For the rest of the 70 cases, there were no differences between original and lossy execution traces due to the lack of file writes in the original execution (DroidBot exploration did not result in the desired file write operations).

Manual Exploration Results. For those 191 apps where DroidBot either crashed or could not reach the target GUI exploration, we performed a manual (human-driven) analysis. Our manual analysis confirmed losses in 56 apps, hence, a total of 163 apps with automatically-confirmed or manually-confirmed losses. For 135 apps, the manual analysis could not run or did not produce traces evidencing losses. These apps fell into several categories: 58 apps could not be explored as they either required a paid membership, their operation was geo-fenced, or could only run when connected to specific hardware devices; 39 apps crashed on our test platform; finally, there were 38 apps where no save- or export-related option was found in the GUI.

5.3.2 Example of confirmed write loss cases

Table 5.4 summarizes data loss examples in 27 apps: 20 apps from Google Play (apps with highest number of installs, shown in the second column) and 7 apps from the open-source F-Droid store. We show a brief summary of the user file write data lost in the third column and the result of running LiveDroid on these apps in the final column (the results are discussed in Section 5.3.3). We now discuss selected apps (more than 5M installations) and the semantics of lost data in detail.

SketchBook. This app allows users to sketch, paint, and draw; due to termination, new sketch data, as well as changes to an existing sketch, can be lost.

Table 5.4 Losses Found and Confirmed by Our Approach; Results of Running LiveDroid

App	#Installs (Million)	Our Approach (Data Lost)	LiveDroid
Google Play			
SketchBook	100	User artwork (sketch, painting)	Failed due to Soot error
WPS Office	100	User-made document changes	No issues found
Drum Pad Machine	100	User created music beats	No issues found
Smart TV Remote	10	Saved TV Channels export	No issues found
Barcode Scanner Pro	10	Barcode scanning history	Failed due to Soot error
Beauty Camera	10	Edited photos	Found 5 app states not saved (user input loss)
AndrOpen Office	5	User-made document changes	No issues found
King James Bible	5	User-settings / preferences	No issues found
K-9 Mail	5	User-settings / preferences	No issues found
Robin	1	App properties/preferences	Failed due to Soot error
Soccer Tactic Board	1	User-created soccer tactic	Failed due to Soot error
Baby Care	1	User-created baby growth data	Failed due to Soot error
Bills Reminder	0.5	Database backup	Failed due to Soot error
SmartTruckRoute	0.5	Truck route exported data	Failed due to Soot error
BCBSM	0.1	Patient's medicare data sharing fails	Failed due to Soot error
Wabbitemu	0.1	Calculator screenshot	Failed due to Soot error
Gallery Slideshow Music	0.1	Edited Video	Failed due to Soot error
TV Show Favs	0.1	User backup data (e.g., favorite TV, watched shows)	Failed due to Soot error
User Dictionary Manager	0.05	Dictionary words	Failed due to Soot error
Bahamas Dining Rewards	0.01	User credentials	Failed due to Soot error
F-Droid			
Sanity	n/a	Settings/preferences (e.g., audio recording, call blocking)	No issues found
Privacyfriendlynotes	n/a	New or updated note	Found 6 app states not saved (user input loss)
MedicLog	n/a	Medic log history	No issues found
Acrylic Paint	n/a	New or updated drawing	No issues found
Auto-Away	n/a	Call or message log export	No issues found
ComfortReader	n/a	New or updated note	No issues found
BeeCount	n/a	database table update	No issues found

Smart TV Remote. This app is used to define and control TV channels via channel logos; due to termination, exported data (TV channels) is lost.

WPS Office. This is an all-in-one office suite app; due to termination, user-made document changes are lost.

Drum Pad Machine. This music mixer app can be used to create beats, mix loops and record new melodies; due to termination, user-created music beats are not saved.

AndrOpen Office. This office suite app allows users to view and edit PDF, Word, Excel, and PowerPoint documents; due to termination, user-made document changes are lost.

King James Bible. This Bible reader app provides options for adding bookmarks and writing notes while reading; due to termination, user settings or preferences (e.g., related to bookmarks, highlights, and notes) can be lost.

K-9 Mail. This is an open-source email client app; due to termination, exported data (user-settings backup) is lost.

Beauty Camera. This app allows editing pictures via filters or stickers; due to termination, edited pictures are not saved.

Barcode Scanner Pro. In this app the user can scan, decode, create, and share QR codes or barcodes; due to termination, the user's barcode scanning history is lost.

5.3.3 Comparison with existing tools

We now compare the results of our approach with the results obtained by running two state-of-the-art tools that aim to find volatility-induced UI losses in Android apps.

Comparison with KREfinder. KREfinder [184] is a static analyzer that looks for incorrectly-handled instance state. Specifically, the analysis looks for app fields that are written to, or modified, and for which there is no subsequent save. KREfinder explicitly looks for state flowing into OutputStream or Writer objects, and generally any Java API methods offering write or save. As the public version of KREfinder is not maintained/updated (latest release: July 2016), we asked the KREfinder's corresponding author to run it on 14 selected apps (7 top Google Play apps, 7 F-Droid apps); KREFinder reported no data losses.

Comparison with LiveDroid. LiveDroid [113] is a tool focused on finding UI fields that might be lost during runtime changes (e.g., phone orientation changes).

Table 5.5 Efficiency Results

Analysis time (seconds)			**Bytecode size (MB)**		
min	max	median	min	max	median
35	25,200	115	0.04	103.4	21.2

LiveDroid identifies variables and GUI input which represent "necessary app state"; this state essentially captures the subset of user input data which must "survive" runtime changes. We ran LiveDroid on the 298 apps with user-initiated file writes. The LiveDroid analysis summary is:

	# Apps
Analyzed	298
Soot Error	157
Issues Found	13

LiveDroid is mainly designed for open-source apps and fails due to Soot error on the 157 Google Play apps with large and/or obfuscated bytecode. For the remaining Google Play apps and all the open-source F-Droid apps, LiveDroid ran to completion; LiveDroid found app state-saving related issues in 13 apps. For example, LiveDroid found app states saving related issues in Beauty Camera and Privacyfriendlynotes app as shown in the third column of Table 5.4, but these issues are unrelated to file write losses; LiveDroid reported no issues in the other 128 apps it successfully run on.

5.3.4 Efficiency

In Table 5.5 we present brief descriptive statistics for static analysis time and app dataset. Analysis time varied between 35 seconds and 7 hours, with a typical time of 115 seconds, which we believe is efficient for a static analysis. App bytecode varied between 40KB and 103MB, with a typical size of 21MB, which shows that our analysis can handle sizable apps.

5.3.5 False positives and false negatives

We measured the False Positives (FP) and False Negatives (FN) by comparing the results of our automated approach with a manual analysis on 60 apps (all containing

file writes) where the file write data losses were confirmed manually. The 60 apps were selected as follows: 30 true positive apps that contain user-initiated file writes and 30 true negative apps that contain file writes, but the writes are not user-initiated. Rather than exploring apps using DroidBot and performing dynamic verification of user data loss via `Strace` difference checking as mentioned in Section 5.2.2, we performed a manual verification of data losses. We manually sent the target app into the background after inducing file write operations and then manually checked whether the expected file writes were missing, i.e., user data is lost. The confusion matrix is:

True Positives: 30	**False Positives**: 0
False Negatives: 5	**True Negatives**: 30

We found 5 False Negatives, i.e., an 85% recall for the automated approach. These are due to automated GUI exploration with DroidBot failing to reach targeted save or export-related GUI options and as a consequence, no file write operation happened, and no file write traces were found. We have no False Positives because static analysis reports are subjected to dynamic verification.

5.3.6 Limitations

Our approach has two limitations. First, the static analysis to find user-initiated file writes is implemented on top of Soot, which failed to produce call graphs for 733 apps. As a result, we were unable to analyze those apps further. Second, our automated dynamic verification used a customized version of the DroidBot input generator [155] to drive app interaction. However, DroidBot, and Android input generators in general, cannot achieve complete coverage [103]. This shortcoming led to manually exploring cases where DroidBot failed to verify user data losses. These limitations can be alleviated with more engineering efforts.

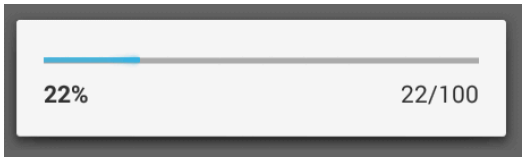


Figure 5.9 Progress bar for ongoing I/O operations in Android.

5.3.7 Potential solutions

File write losses due to termination could be addressed by keeping the target app alive as a foreground process. The app developers could show a progress bar for file writes (e.g., Figure 5.9) so users do not switch to a different app while writes are in progress. Another solution is to keep the app alive as a background process by granting unrestricted battery usage (available for Android 8.0 or above), which ensures the app is running with fewer limits while in background, hence, is less likely to be killed due to memory pressure. Finally, another option for updating data is to write a temporary copy and delete the old data upon successful writing of the new data, or alternatively, use storage with transactional APIs such as SQLite or Firebase.

5.4 Summary

Mobile apps' construction and operation is fundamentally different from "run forever" desktop/server programs which complicates testing whether user data is inadvertently lost due to resource pressure. In this dissertation, we focus on user work/data that should be saved via user-initiated file writes; while expected to be stored in local storage, such work and data can be lost due to system-initiated termination. We constructed a static analysis to find potential losses in users' work due to premature file write termination and verified losses via an automated dynamic approach. We were able to confirm such losses in 107 Google Play and F-Droid apps. Our approach can improve the overall user experience of saving user data and form the basis of further studies and explorations into (a) loss of mobile state due to volatility, (b)

extending the findings from file writes to all possible I/O, and (c) the nature of losses due to unexecuted I/O in programs in general.

Now that we have understood the significance of uncovering hidden bugs, memory leaks, and the consequences of resource inefficiencies, let us now shift our focus to random testing. This approach enables the exploration of diverse app functionalities and features, ensuring their proper functioning across various conditions. By incorporating randomness, the testing process becomes more thorough and resilient, as it uncovers unforeseen issues that traditional testing methods may overlook. In the upcoming chapter, we will delve into the effectiveness of random testing for Android apps, specifically examining the reliability of the Monkey tool.

CHAPTER 6

EFFECTIVENESS OF RANDOM TESTING FOR ANDROID

Random testing of Android apps is attractive due to ease-of-use and scalability, but its effectiveness could be questioned. Prior studies have shown that Monkey – a simple approach and tool for random testing of Android apps – is surprisingly effective, "beating" much more sophisticated tools by achieving higher coverage. We study how Monkey's parameters affect code coverage (at class, method, block, and line levels) and set out to answer several research questions centered around improving the effectiveness of Monkey-based random testing in Android, and how it compares with handcrafted exploration. First, we show that random stress testing via Monkey is extremely effective at crashing apps, including 15 widely-used apps that have millions (or even billions) of installs. Second, we vary Monkey's event distribution to change app behavior and measured the resulting coverage. We found that, except for isolated cases, altering Monkey's default event distribution is unlikely to lead to higher coverage. Third, we manually explore 24 apps and compare the resulting coverages; we found that coverage achieved via Monkey is practically indistinguishable from that achieved via manual exploration. Finally, our analysis shows that course-grained coverage is highly indicative of fine-grained coverage, hence coarse-grained coverage (which imposes low collection overhead) hits a performance vs accuracy sweet spot.

Android apps are updated on average every 60 days. Testing all these apps and their updated versions requires scalable, easy-to-use, and effective tools. One such tool is Monkey [83], a random testing tool that ships with Android Studio (Android's IDE). Monkey simply generates random events from a predefined distribution (according to some parameters, e.g., timing, event distribution, event kind) and sends the events to the app. Many other, sophisticated, automated testing tools for Android have

been proposed – tools that require static analysis, dynamic analysis, or symbolic execution. Surprisingly though, a prior study that compared the coverage attained by such tools has shown that, on average, Monkey manages to achieve the *highest coverage level* across all tools, when comparing across a substantial set of 68 apps [103]. Code coverage is a widely-used measure for quantifying testing effectiveness – the higher the coverage, the more effective the test suite. Code coverage can be defined at various levels, e.g., method coverage indicates what percentage of the app methods are invoked, block coverage indicates what percentage of the basic blocks are entered, while statement coverage indicates what percentage of the app statements are executed. Monkey achieves about 48% statement coverage, on average [103]. In Section 6.1 we provide some background on the Android platform, Monkey, and the Emma code coverage tool we used.

In Section 6.2, we present the approach and results of our study. Given Monkey's unparalleled combination of ease-of-use, negligible overhead, and portability, we study whether Monkey is also *effective*. Specifically, we set up our study around several research questions:

- Can Monkey crash popular apps via stress-testing?
- Can Monkey be made more effective (yielding higher coverage) when appropriately "tuned"?
- Can hand-crafted exploration lead to higher coverage than Monkey's?
- Is collecting fine-grained (e.g., block/line) coverage preferable to coarse-grained (e.g., class/method) coverage?

6.1 Background

6.1.1 Monkey

UI/Application Exerciser Monkey (aka "Monkey") is an open-source random testing tool included in the Android SDK [83]. Monkey can run on either a physical device or

Table 6.1 Monkey's Default Events and Percentages

Event ID	**Description**	**Frequency (%)**
TOUCH	single touch (screen press & release)	15
MOTION	"drag" (press, move, release)	10
TRACKBALL	sequence of small moves, followed by an optional single click	15
NAV	keyboard up/down/left-/right	25
MAJORNAV	menu button, keyboard "center" button	15
SYSOPS	"system" keys, e.g., Home, Back, Call, End Call, Volume Up, Volume Down, Mute	2
APPSWITCH	switch to a different app	2
FLIP	flip open keyboard	1
ANYTHING	any event	13
PINCHZOOM	pinch or zoom gestures	2

an emulator. The tool emulates a user interacting with an app, by generating and injecting pseudo-random gestures, e.g., clicks, drags, or system events into the app's event input stream.

In addition to generating and injecting events into an app, Monkey also watches for exceptional conditions, e.g., the app crashing or throwing an exception.

Monkey sends events from a predefined distribution described in Table 6.1, i.e., 15% of the events it sends are TOUCH events (first row), 10% are drag events, 2% are system key events, 2% are pinch or zoom events, etc. In addition to event probability distribution, Monkey can also vary the "throttle", i.e., the time delays between events; by default there is no delay between events (throttle = 0).

6.1.2 EMMA code coverage

Emma [202] is an open source toolkit used to measure and report Java code coverage. Emma measures coverage at various levels of granularity: from class to method, to basic block, and line; it does so by instrumenting the app so the app "dumps" coverage

information into log files. Importantly, Emma does not require access to source code and its runtime performance overhead is typically less than 20%. Our experiments use Emma to measure coverage.

6.2 Empirical Study

We now present the approach used in, and results of, our study. We phrase each experiment as a research question (RQ); note that at the beginning of each subsection we state the RQ and summarize the finding.

6.2.1 Experimental setup

Android VM. Our experiments were conducted on the same virtual machine setup used by Choudhary et al. [103] for their comparison of automated Android testers. Specifically, we used multiple Android virtual machines (Oracle VirtualBox) running on a Linux server. Each VM was configured with 2 cores and 6 GB of RAM, running Android version 2.3.3 (Gingerbread), API level 10.

App datasets. Our app dataset covered 40 real-world apps, drawn from a variety of categories, and having a variety of sizes. The 40 apps included 15 "famous apps", e.g., Shazam, Spotify, and Facebook for which no source code was available (and many of which use anti-debugging measures hence we did not measure coverage); this set was used for stress-testing only. The remaining 25 apps were popular apps, e.g., K-9 Mail, whose source code is available on GitHub.

Monkey runs. Since Monkey's exploration strategy is random, we use the same seed value for analyses requiring multiple runs to ensure the same input sequences are generated. After Monkey completes its run on each app, the emulator is destroyed and a new one is created to avoid any side effects between runs.

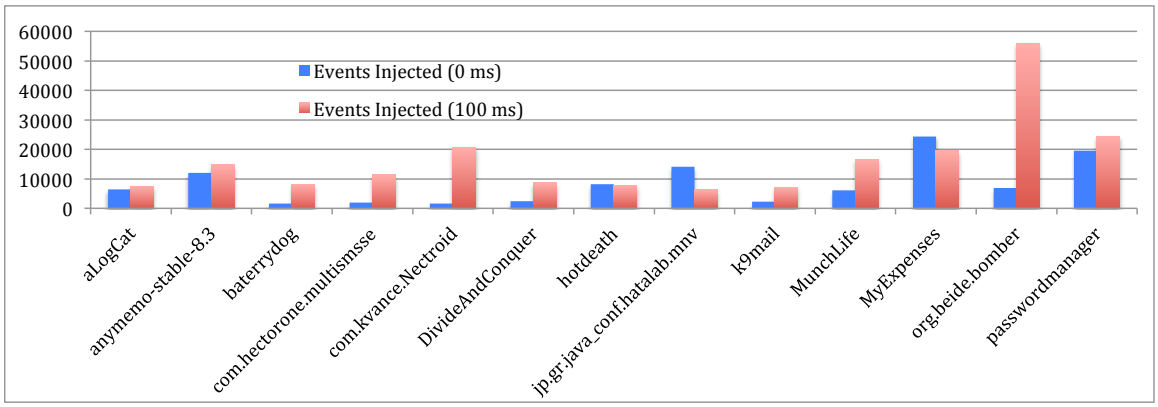


Figure 6.1 Injected events that lead to crash.

6.2.2 Application crashes

RQ1: Can Monkey crash "famous" apps via stress-testing?

Answer: Yes

Stress testing is used to study a program's availability and error handling on large inputs or heavy loads – a robust, well-designed program should offer adequate performance and continuous availability even in extreme conditions. In Android, stress testing reveals the ability of an app to handle events properly in situations where, due to low resources or system load, long streaks of events arrive at a sustained rate. The app should handle such streams gently, perhaps with degraded performance, but certainly without crashing. Therefore, for our first set of experiments, we have subjected the test apps to stress-testing via long event sequences at 0 throttle, i.e., no delay between events – this is the default Monkey behavior.

For the 15 famous apps, we present the results in Table 6.2. The first column shows the app name while the second column shows that app's popularity (number of installations, in millions). Note that four apps have in excess of 1 billion installs, while 12 apps have in excess of 100 million installs. We found that we were able to crash *all 15 apps* by letting Monkey inject 4,739 events on average.

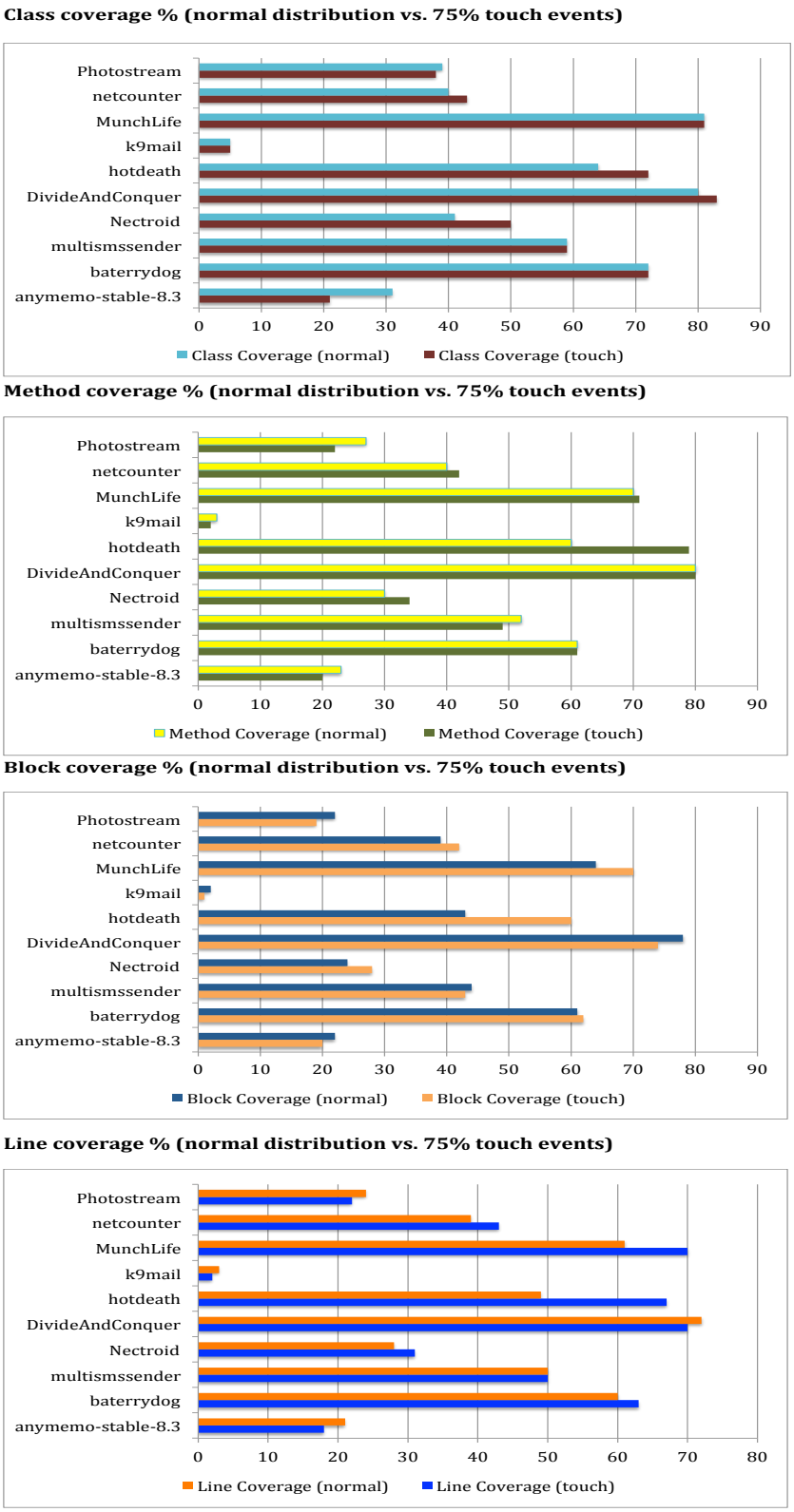


Figure 6.2 Coverage achieved for regular event distribution vs 75% *touch* events.

Table 6.2 Stress Testing Results for Top Apps: The Number of Events at Which the App Crashes

App	Installs (millions)	#Events
Shazam	100–500	2,330
Spotify	100–500	2,041
Facebook	1,000–5,000	637
Whatsapp	1,000–5,000	6,224
Splitwise	1–5	2,123
UC Browser	100–500	17,840
NY Times	10–50	4,949
Instagram	1,000–5,000	4,859
Snapchat	500–1,000	3,773
Walmart	10–50	2,537
MX Player	100–500	9,786
Evernote	100–500	786
Skype	500–1,000	7,342
Waze	100–500	2,484
Google	1,000–5,000	3,380
Mean		*4,739*

Table 6.3 Results for *Touch* Events

Coverage type	Mean	
	Default	75%
Class	52.4	51.2
Method	44.6	46
Block	39.9	41.9
Line	40.7	43.6

Figure 6.1 shows stress testing results for the open-source apps; in this case we ran stress testing at 0 msec and 100 msec, respectively. For brevity, the figure only shows the results for a subset of apps. We found that, on average, apps crash after 8,287 events when throttle is set at 0 msec and 16,110 events when the throttle is set at 100 msec.

To conclude, it appears that *it is not a matter of if, but when* Monkey manages to crash an app. This demonstrates Monkey's effectiveness at rooting out performance stability bugs in such popular apps, and therefore we can answer RQ1 in the affirmative.

6.2.3 Coverage

RQ2: Can we increase coverage by optimizing the event distributions?

Answer: No

We biased the event distribution by increasing the probability for a single event type to 75% while proportionally shrinking the probabilities of the remaining 25% events according to the distribution shown in Table 6.1. Doing so, we boost each event kind's relative importance in achieving coverage. We perform this analysis for all "important" categories in Table 6.1 having a default probability of 10% or higher: touch, motion, trackball, navigation, major navigation. We then measure the difference in coverage when using the biased distribution compared to the baseline Monkey distribution (i.e., default Monkey behavior).

We test this hypothesis via a *one-tail heteroscedastic two-means t-test* set up as follows. Let M_{def} be the mean coverage attained by default Monkey, and M_{75} be the mean coverage attained when using the 75% distribution. We say that the hypothesis is confirmed, i.e., the 75% biased distribution succeeds in increasing coverage, if $M_{def} \leq M_{75}$ at a statistically significant level (p-value < 0.05). Put otherwise, if the hypothesis is rejected, it means that biasing the distribution does not result in increasing coverage and the default Monkey is more effective.

Touch Events. We first increased touch event frequency to 75%. We show the resulting coverage in Figure 6.2 and the means in Table 6.3. We obtained a higher class coverage for isolated cases (Nectroid – a media & video app, DivideAndConquer and hotdeath – games, Netcounter – a tool). Method coverage is similar to class coverage. Block and line coverage results are similar to each other, with two more apps showing higher coverage (Baterry Dog – a tools app, and munchlife – an entertainment app). Note that these apps fall in the categories tools/media/video/entertainment and it is

Table 6.4 Results for *App Switch* Events

Coverage type	Mean	
	Default	**75%**
Class	52.9	52.2
Method	45	45.2
Block	41.6	43.4
Line	42.9	42.7

natural to see them benefiting slightly from the touch events as they are driven mostly by user clicking on certain screen items. Overall, however, the hypothesis is rejected – the 75% distribution does not increase coverage in a statically significant way.

Table 6.5 Results for *Motion* Events

Coverage type	Mean	
	Default	**75%**
Class	45.6	37.8
Method	40.8	31.1
Block	38	26.3
Line	38.8	26.6

App Switch Events. Next, we increased app switch event frequency to 75%. We show the means in Table 6.4. For brevity, we omit per-app charts. We found that class coverage is lower except for app multiSMSsender; method coverage was lower except for apps multiSMSsender and K-9 Mail; block coverage however is slightly (but not significantly) higher; line coverage is lower, except for app multiSMSsender. Note that all these apps have two points in common: (1) background activity and substantial app state. This state has to be saved (and restored) when switching to a different app (and switching back to the app, respectively), which explains local increases in coverage for the 75% app switch event mix. Overall, however, the hypothesis is rejected.

Motion Events. Next, we increased motion event frequency to 75%. We show the resulting coverage in Figure 6.3 and the means in Table 6.5. The 75% motion inputs caused very noticeable *decrease* in coverage across the board: class, method, block, line. The only apps that benefited (slightly) from this distribution were K-9 Mail and

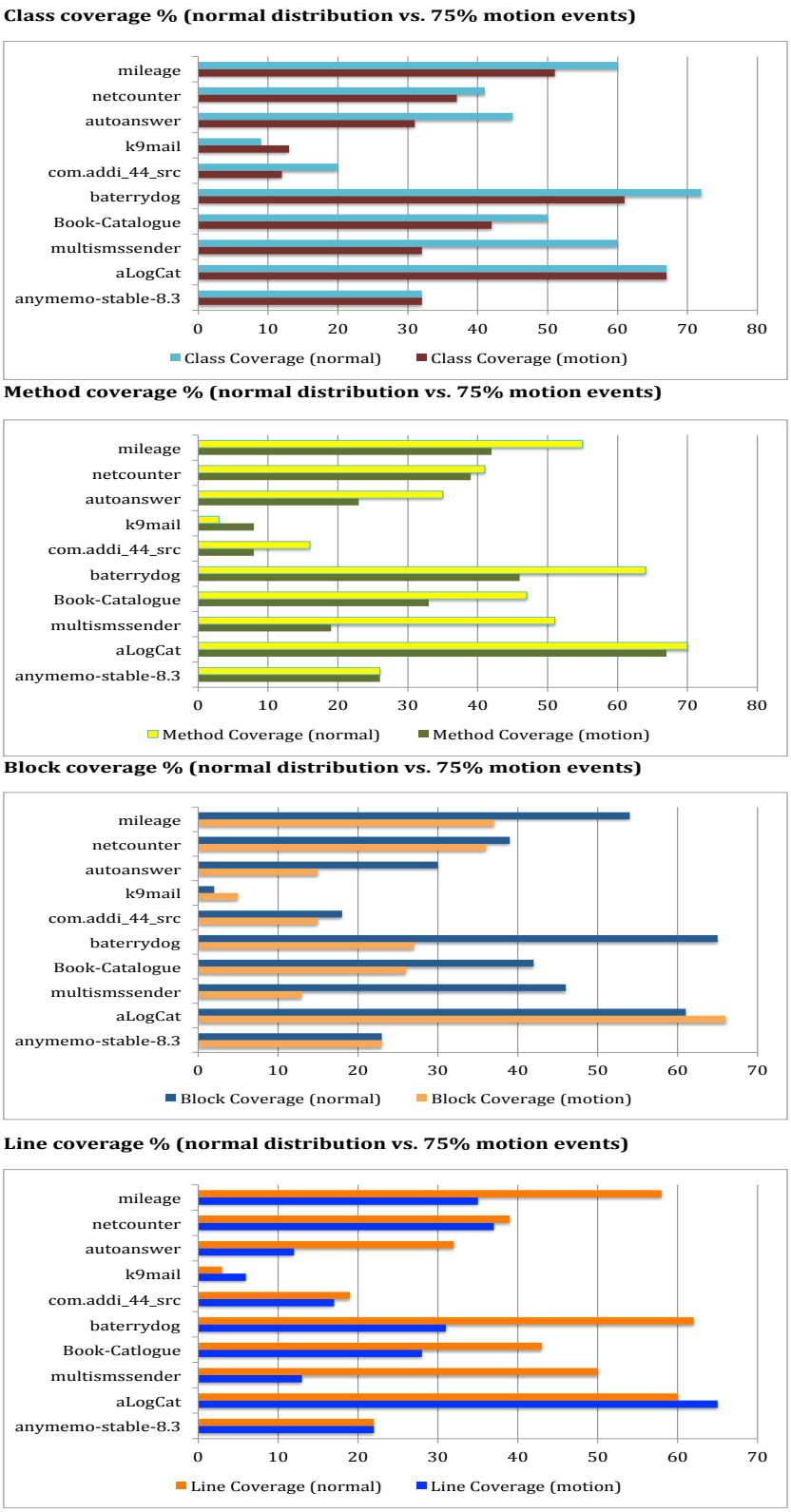


Figure 6.3 Coverage achieved for regular event distribution vs 75% *motion* events.

Table 6.6 Results for *Trackball* Events

Coverage type	Mean	
	Default	**75%**
Class	55.1	52.5
Method	48.1	46.3
Block	43.7	42
Line	44.2	42.4

Table 6.7 Results for *Navigation* Events

Coverage type	Mean	
	Default	**75%**
Class	50.6	39.8
Method	43.2	32.2
Block	37.9	28
Line	38.7	28.5

aLogCat. aLogCat is a `logcat` (system log) viewer; since most of its usage consists of scrolling and dragging, it benefits from increased motion event. However, overall the hypothesis is rejected.

App Trackball Events. Next, we increased app trackball event frequency to 75%. We show the means in Table 6.6. For brevity, we omit per-app charts. We found that class, method, block, and line coverages were lower. The only apps that benefited (slightly) from this distribution were K-9 Mail, Battery Dog and multiSMSsender. Battery Dog runs a background service logging the battery state into a file, so small moves and occasional single clicks are prevalent usage patterns for this app. However, overall the hypothesis is rejected.

Navigation Events. Next, we increased app navigation event frequency to 75%. We show the resulting coverage in Figure 6.4 and the means in Table 6.7. Using 75% navigation inputs caused *substantial decrease*, about 10 percentage points in coverage: class, method, block, line for all apps except DivideAndConquer and Nectroid (a game and a media app, respectively). However, overall the hypothesis is rejected.

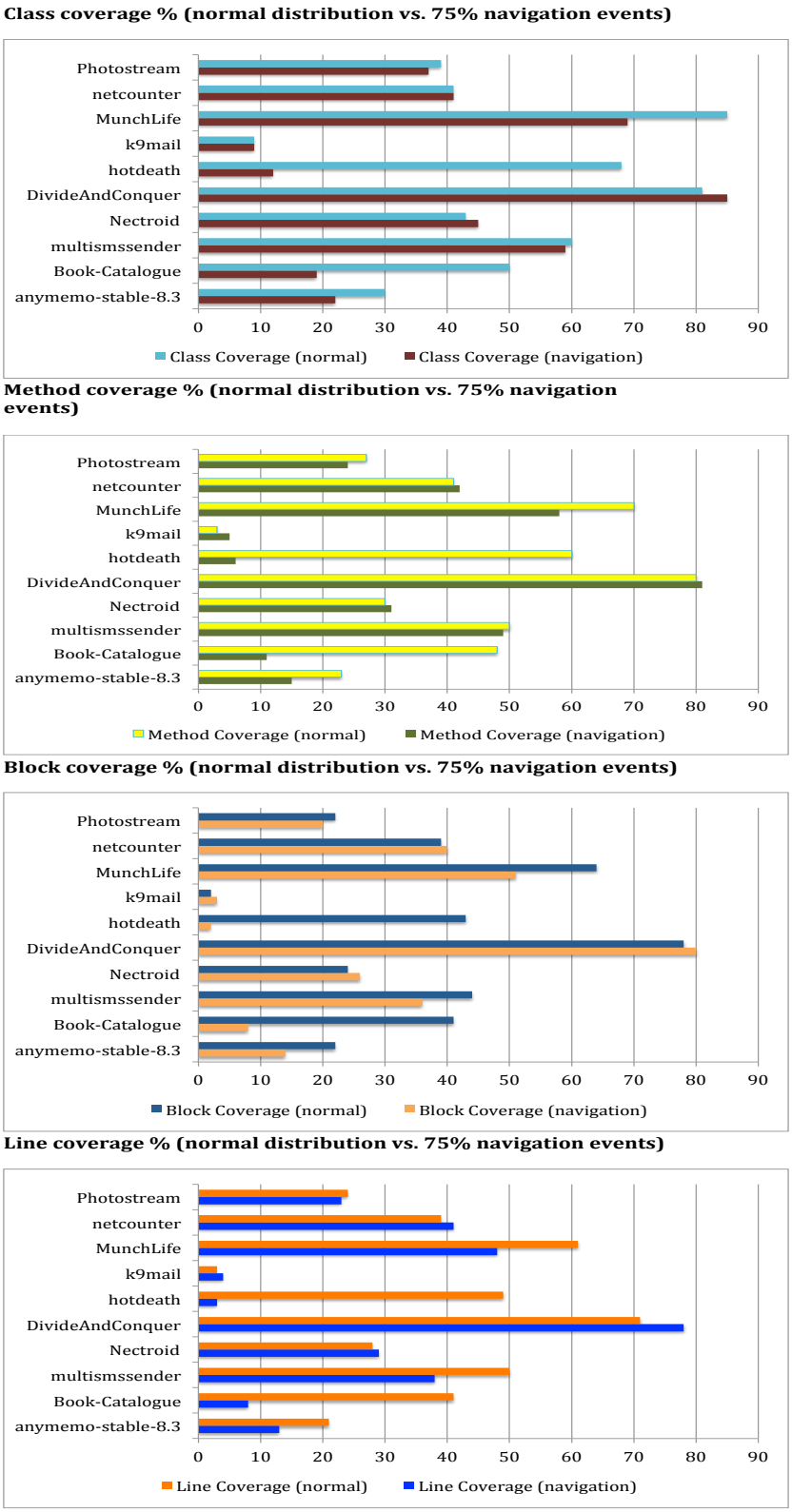


Figure 6.4 Coverage achieved for regular event distribution vs 75% *navigation* events.

Table 6.8 Results for *Major Navigation* Events

Coverage type	Mean	
	Default	**75%**
Class	56.8	62.2
Method	48.4	53.5
Block	45	50.7
Line	45.8	51.1

Major Navigation Events. Finally, we increased major navigation event frequency to 75%. We show the resulting coverage in Figure 6.5 and the means in Table 6.8. We found *increases* in coverage for all types – class, method, block, and line – but the increases were not significant. Therefore, the hypothesis is rejected.

Correlation Between Coverage Values. We now turn to our analysis of correlation among coverage values. For the coverages obtained previously, we compute the pairwise correlation, i.e., for a certain experiment such as "Touch", for each app, we pairwise-correlate the class, method, block, and line coverage values. We show the results in Table 6.9. With two exceptions (Class v. Block and Class v. Line correlation for Motion events, whereas values are 0.78 and 0.82, respectively), the correlation values are *very high, greater than 0.9*. This finding allows us to make two observations about Android methods:

1. Methods are shallow and are generally a good indicator of coverage without having to gather more detailed (but more expensive!) block or line coverage. For example, Android includes a method profiler [82] that could be used in lieu of specialized line coverage tools that require code instrumentation.
2. A method's control flow graph appears to have a low number of paths: covering a method tends to cover all of its constituent blocks.

To conclude, this suggests a pragmatic approach for measuring coverage – using low-overhead but coarse-grained coverage (class or method) as proxy for high-overhead but fine-grained coverage (block or line).

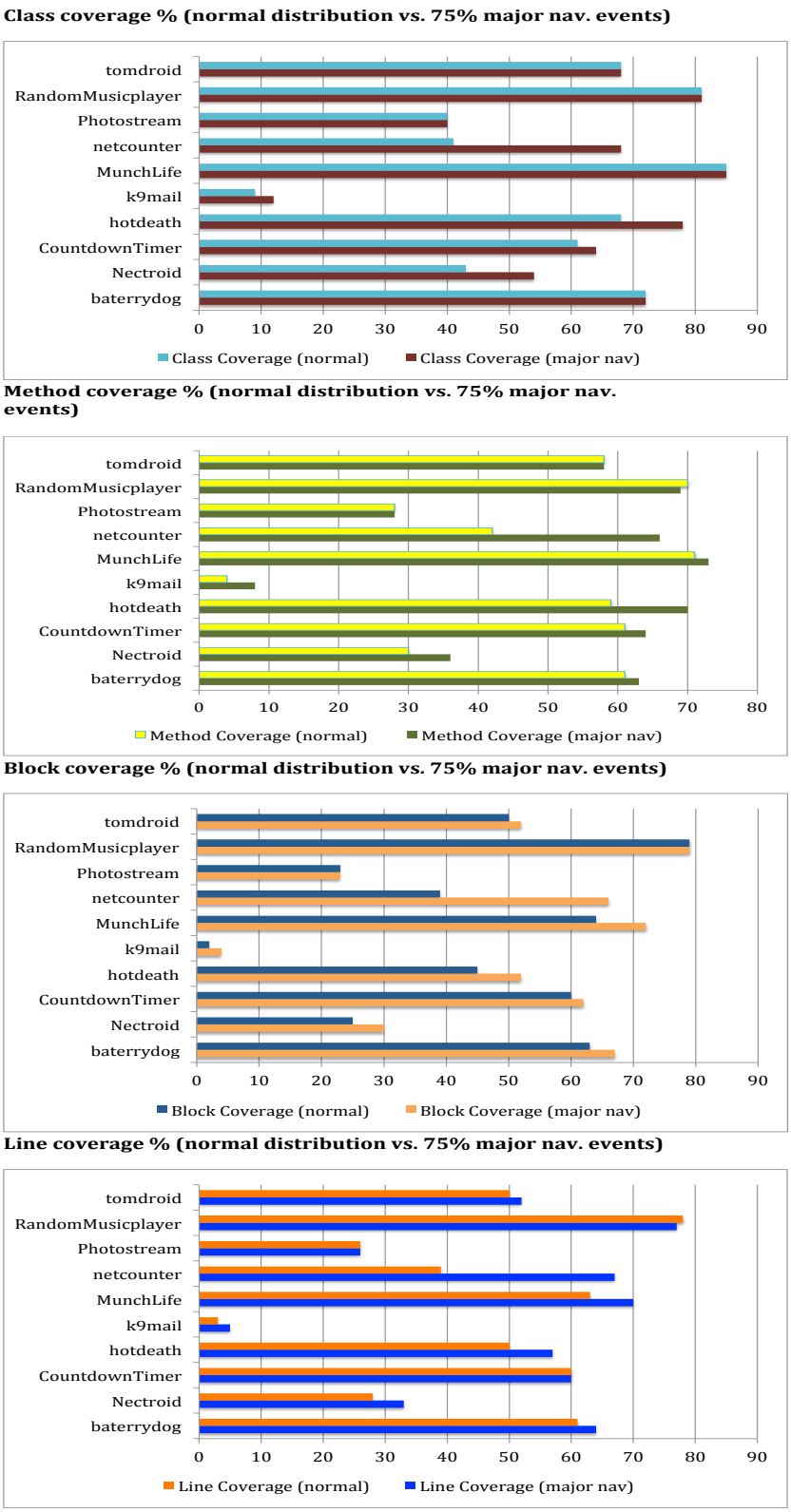


Figure 6.5 Coverage achieved for regular event distribution vs 75% *major navigation* events.

Table 6.9 Correlation Between Coverage Types: Class (C), Method (M), Block (B), and Line (L)

	Touch			App switch			Motion			Trackball			Navigation			Major Navigation		
	M	B	L	M	B	L	M	B	L	M	B	L	M	B	L	M	B	L
C	0.93	0.91	0.92	0.94	0.93	0.94	0.95	0.78	0.82	0.97	0.94	0.96	0.98	0.96	0.96	0.97	0.93	0.95
M	-	0.96	0.97	-	0.97	0.97	-	0.93	0.94	-	0.97	0.98	-	0.99	0.99	-	0.95	0.97
B	-	-	0.99	-	-	0.99	-	-	0.99	-	-	0.99	-	-	0.99	-	-	0.99

Table 6.10 Monkey vs Manual Testing

Coverage type	Mean	
	Monkey	Manual
Class	56.12	56.45
Method	48.45	47.91
Block	47.33	47.45
Line	46.04	45.08

6.2.4 Manual vs Monkey coverage

RQ3: Does manual exploration achieve higher coverage than Monkey?

Answer: No

While manual exploration could be considered a "golden standard" at exploring an app thoroughly, we found that not to be the case. Specifically, we picked 24 apps and manually interacted with each app for 5 minutes trying to explore as many screens and functionalities as possible. We show the results in Figure 6.6 and the means in Table 6.10.

We can see that Monkey obtained higher coverage for between 9 and 13 apps (depending on coverage type), e.g., aLogCat, Milage, Auto AutoAnswer, hectrone, Addi, hotdeath, MunchLife, PhotoStream and WeightChart – these apps fall in the categories of tools, media, and entertainment.

K-9 Mail is an interesting case; the human "beat" the Monkey by a factor of 4.3x (39% vs 9% coverage), 9.6x (29% vs 3%), 13.5x (27% vs 2%), 6.6x (20% vs 3%) for class, method, block, and line coverage, respectively. This is due to a highly customized UI and the inability of Monkey to compose emails. The same holds for app AnyMemo where the Monkey achieves lower coverage because it cannot compose notes.

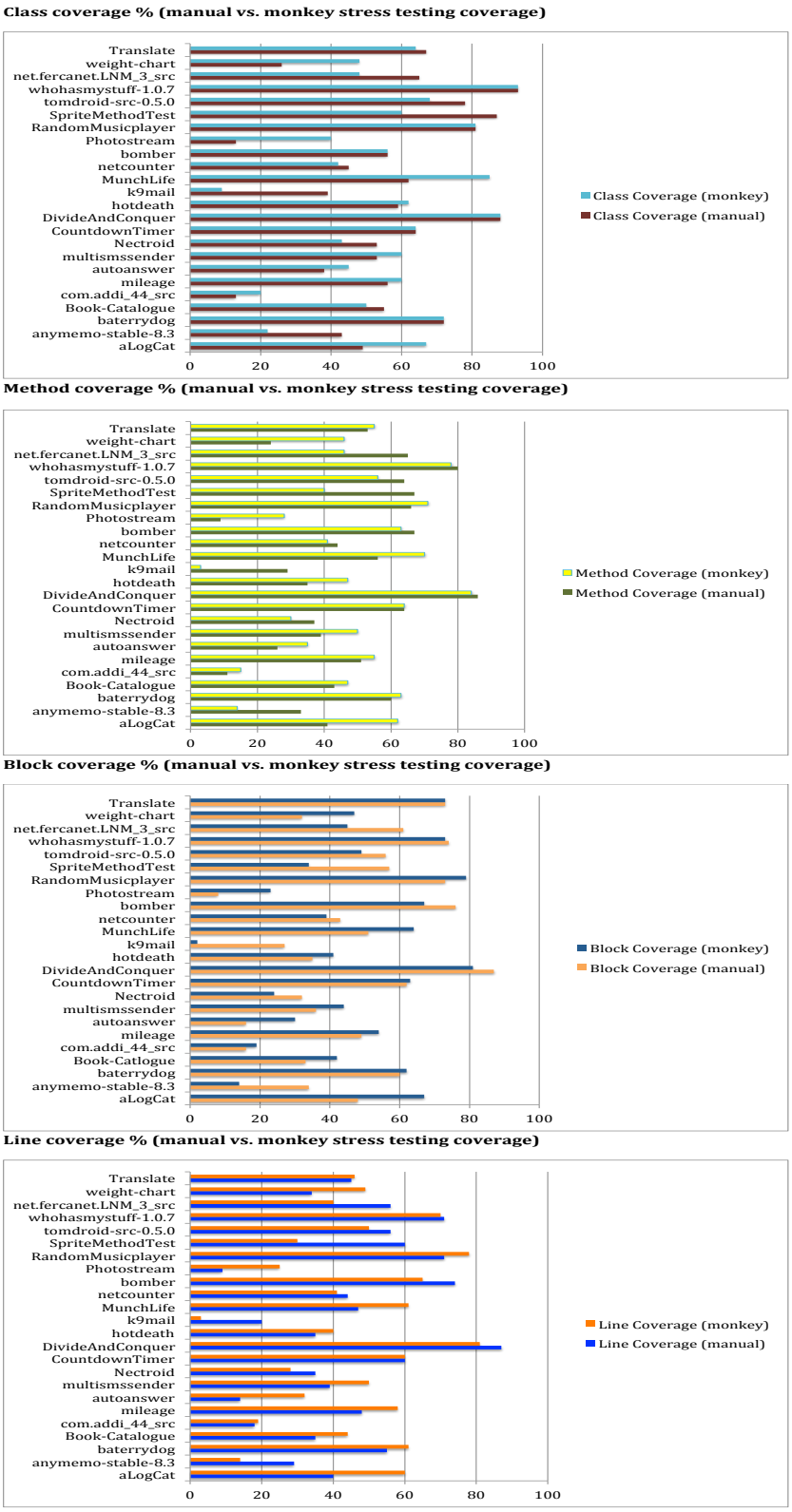


Figure 6.6 Monkey vs. manual testing coverage.

Table 6.11 Line Coverage When Varying Throttle

Throttle (msec)	Mean Coverage (%)
0	40.8
100	44.95
200	42.5
600	42.55

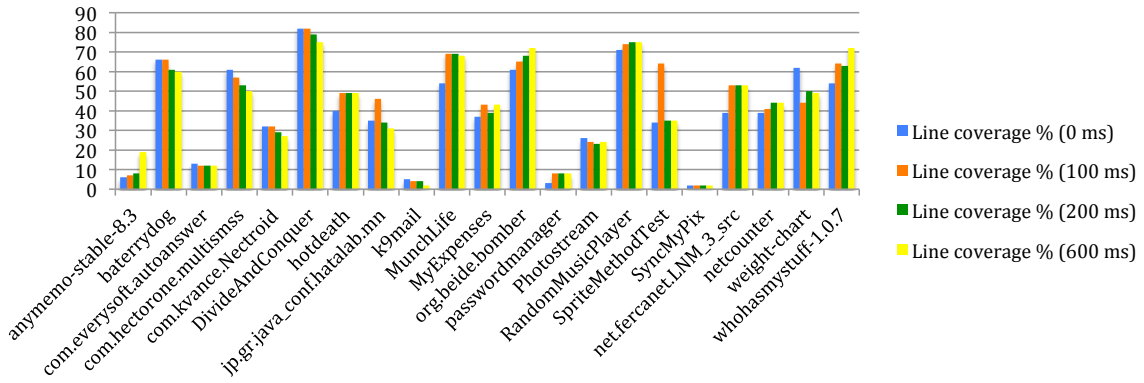


Figure 6.7 Coverage % comparison with changes to throttle.

Conversely, though, Monkey beat the human at Photostream by a factor of 3.1x (40% vs 13%), 3.1x (28% vs 9%), 29x (23% vs 8%), and 2.8x (25% vs 9%) respectively. To conclude, overall Monkey does exceedingly well – its coverage is virtually indistinguishable from a human's.

6.2.5 Throttling

RQ4: Does inter-event time (throttling) affect coverage?

Answer: No

Throttle is the delay between events sent by Monkey. We studied whether varying throttle affects coverage. We experimented with setting the throttle value to 0 (default), 100, 200, and 600 msec, while collecting line coverage only. We present the results in Figure 6.7 and the mean coverages in Table 6.11. We again ran two-means t-tests, pairwise between coverages. We found that, while 100 msec throttle leads to slightly higher coverage, the increase is not statistically significant. Therefore, the

hypothesis that throttle affects coverage is invalidated (at least for our chosen throttle values).

6.3 Summary

We have conducted a study which has revealed that, despite its simplicity, random testing for Android is effective. Specifically, random testing is effective at revealing stress-related crashes, and in terms of coverage is on par with laborious approaches such as manual exploration. Moreover, our study has revealed that, except for isolated apps and event kinds, Monkey's default event type distribution and settings are appropriate for achieving high coverage in a wide range of apps.

We believe that our study reveals two "sweet spots" for Android app developers and researchers. First, Monkey achieves high coverage while being easy to use and efficient. Second, coarse-grained but low-overhead class or method coverage is nevertheless effective (representative of finer-grained, block or line coverage).

Next, we will delve into the reliability of different tools' implementations for SOM neural networks. While SOM implementations may not be directly linked to the reliability of Android applications, the use of neural networks on mobile devices has become increasingly prevalent. Artificial intelligence is widely used in mobile devices and hardware tools. Therefore, understanding the reliability of SOM neural networks is a topic of significant interest. By exploring the reliability aspects of SOM implementation, we can gain valuable insights into the performance and dependability of neural networks on mobile platforms.

CHAPTER 7

SOM NONDETERMINISM AND INCONSISTENCY

Self-organizing maps (SOMs) are a neural network-based approach for mapping relationships between objects in high-dimensional spaces onto a low-dimension space, usually a neuronal grid [142]. Main uses for SOMs include exploratory data mining [200], dimensionality reduction, clustering, or pre-clustering. Figure 7.1 shows an example SOM on dataset Zoo, which is used to cluster 101 animals into 7 groups based on 17 characteristics (features). Each circle indicates a neuron, while the clusters, e.g., "Fish" or "Mammal" are indicated via neurons of the same color. Note how animals that have related attributes in the 17-dimensional space are clustered together in the 2-dimensional output SOM.

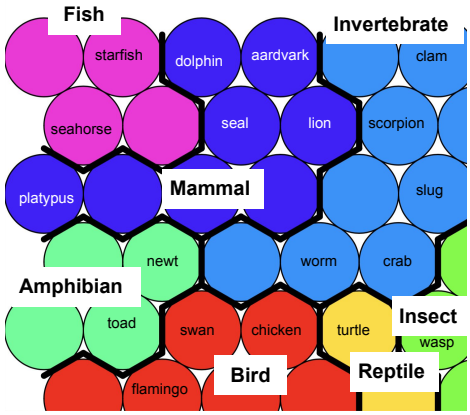
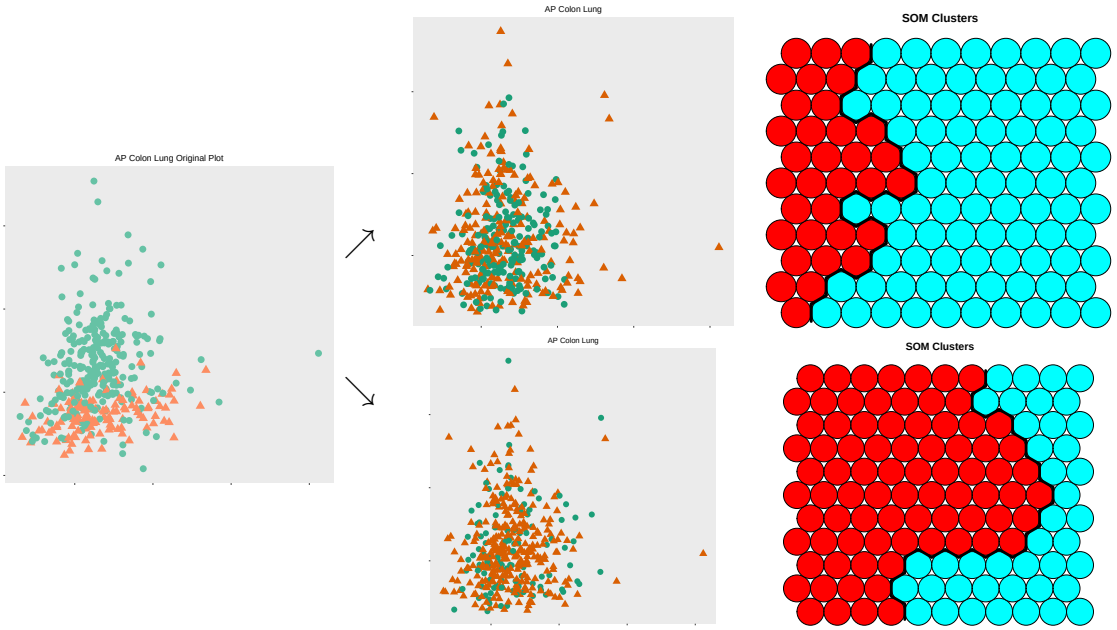


Figure 7.1 SOM for dataset Zoo, toolkit RKoh.

SOM have been used in critical domains, e.g., finance [196, 108], drug discovery [182, 183], or medical sciences [187]. However, SOM reliability has not been questioned. In this dissertation, we do so, by focusing on two key issues. First, *nondeterminism*: when running an SOM implementation repeatedly on the same dataset yields different results. Second, *inconsistency*: when running two different SOM implementations on the same dataset yields different results.



(a) Original dataset (b) Output Clusterings (c) Resulting SOMs

Figure 7.2 Different SOMs obtained via two different runs in RKoh, dataset AP Colon Lung.

We illustrate nondeterminism in Figure 7.2, on the AP Colon Lung dataset, from the Gene Expression for Oncology repository.[1] Specifically, we show that two independent runs of the same, simple procedure – training an SOM on AP Colon Lung – can yield two very different results between the two runs. Figure 7.2(a) shows the original dataset with ground truth (two clusters shown in green circles and orange triangles, respectively). Figure 7.2(b) shows the SOMs constructed by the R/Kohonen (RKoh) toolkit, on this dataset, in two different runs. Finally, Figure 7.2(c) shows the resulting SOMs and clusters, for the two runs in the middle; the red and cyan colors indicate the different neurons clusters on the map (separated by the thick black line). Notice the substantial differences in cluster assignments and SOMs between the top and bottom figures; this is due to nondeterminism.

We now illustrate inconsistency: how SOM clustering outcomes (hence, accuracy) for the same dataset differ not only across runs, but also across toolkits. We conducted 30 independent runs for each toolkit on the aforementioned Zoo dataset. Figure 7.3

[1]GEMLeR by Stiglic and Kokol [194]: using genetic markers to differentiate between different clinical conditions such as various types of cancer.

shows violin plots for clustering accuracy (i.e., distribution of accuracy across the 30 runs). Note that RKoh yields consistently high accuracy (0.78–0.79), whereas for MiniSom, accuracy varies from 0.47 to 0.82 depending on the run. In contrast, TFSom's accuracy (0.23–0.29) is less than the minimum accuracy for the other toolkits. Hence, the choice of toolkit crucially impacts the resulting accuracy.

In the rest of the dissertation we quantify, via statistical tests on internal/external metrics, how SOMs obtained via training *on the same dataset* differ *across runs* and *across toolkits*.

In Section 7.1 we define SOM and discuss the experimental setup: metrics, datasets, toolkits. We investigate four popular toolkits (SOM packages) – MiniSom, R/Kohonen, TensorFlow SOM, MATLAB – described in Section 7.1.3. We ran our analysis on 381 datasets: about 290 of these were medical datasets, and the rest were benchmarking datasets; a qualitative and quantitative description is provided in Section 7.1.4.

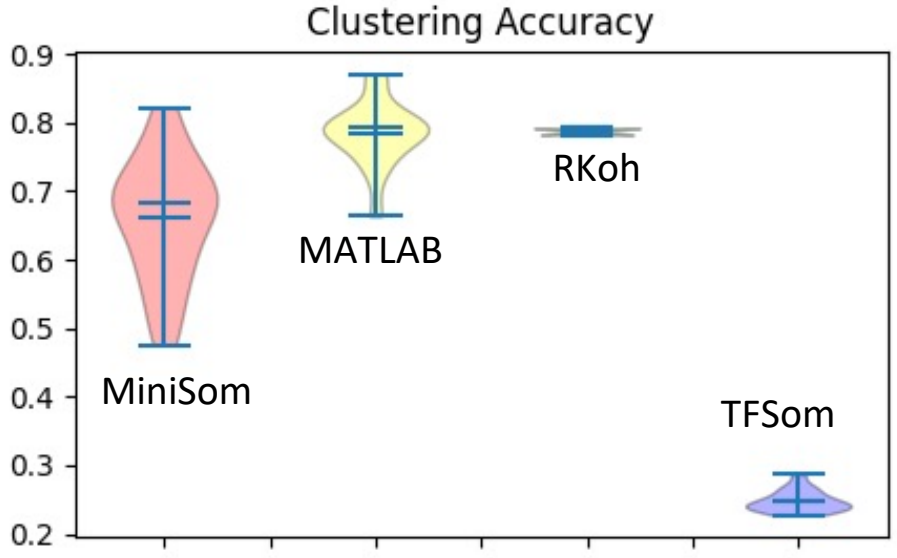


Figure 7.3 Clustering accuracy ranges for dataset Zoo.

In Section 7.2, we define nondeterminism via a rigorous statistical test. In Sections 7.3 and 7.4 we quantify nondeterminism using internal and external metrics. For a given toolkit and dataset, we measure how output SOMs vary across 30 runs. We found that, for our examined 381 datasets, at most 6 lead to deterministic results;

hence, the vast majority of datasets induce nondeterministic SOM outcomes. In Section 7.5, we define inconsistency via a statistical test, and present our findings: for 51–92% of datasets, toolkits yield SOM clusterings with significantly different accuracy distributions.

7.1 Definitions and Experimental Setup

We now define the main concepts and describe the setup for our approach.

7.1.1 SOM definition

SOMs are based on unsupervised competitive learning using neural networks. An SOM clusters (maps) high-dimensional data onto a two-dimensional neuron grid. Typically, the grid topology (how neurons are connected) is hexagonal or rectangular. The network "learns" as the grid neurons adapt to the latent structure of the dataset; in other words, SOMs apply competitive learning to adjust weights to neurons. SOMs are useful for managing and visualizing large datasets or high-dimensional datasets, because the datasets are simplified into clusters in the two-dimensional space. As neurons might shift from run to run, SOMs might yield solutions and results that are potentially inconsistent from run to run.

7.1.2 SOM performance metrics

Prior research [176, 125] has introduced metrics for SOM performance and the quality of the training algorithm. Forest et al.'s SOMperf package [116] measures SOM quality via internal and external metrics. Internal metrics reflect the "native" quality of the SOM construction and its fit to the input data. In contrast, external metrics measure the implementation based on output labels compared to ground truth, e.g., SOM clusters vs. known clusters. We leverage Forest et al.'s metrics and package to collect input data for our analyses.

Table 7.1 Number of Datasets With Statistically Invariant Runs; '-' Indicates that All Datasets' Runs Varied Significantly. "Med" and "TrM" are Short Forms of Median and Trimmed Mean, Respectively

Toolkit	Quantization Error			Topographic Product			Trust-worthiness			Neighborhood Preservation			Distortion			Kruskal-Shepard Error		
	Mean	Med	TrM	Mean	Med	TrM	Mean	Med	TrM	Mean	Med	TrM	Mean	Med	TrM	Mean	Med	TrM
MiniSom	-	1	-	-	-	-	2	2	2	3	5	4	-	1	-	-	-	-
MATLAB	1	4	1	-	4	-	1	4	1	5	6	5	-	-	-	-	-	-
RKoh	-	-	-	-	3	-	3	4	3	5	6	6	-	4	-	-	6	-
TFSom	-	3	-	-	-	-	-	4	-	-	2	-	-	3	-	-	5	-

7.1.3 Toolkits

We investigate four popular[2] SOM packages, as follows. **MiniSom** [53], based on Python/Numpy; **RKoh**[3] – the Kohonen package [39] for R [63]; **MATLAB**'s selforgmap toolbox [52]; and **TFSom** – the TensorFlow Self-Organizing Map package [61] built on top of TensorFlow [10].

7.1.4 Datasets

We used 381 datasets from OpenML [57] for MiniSom, RKoh, and MATLAB. For TFSom we only used 361 of these 381 datasets (on 20 datasets, runtime exceeded our imposed 3-hour limit per run). About 290 of these datasets are drawn from the medical domain or bioinformatics, while the rest are specifically designed to evaluate ML implementations. As these datasets are used to benchmark classification approaches we have cluster labels (ground truth). The following table summarizes the characteristics of our datasets: on average, datasets have 219 instances, 16 dimensions, and 2.38 clusters.

	Min	Max	Geometric Mean
Instances	36	2201	219
Features (attributes)	1	61,359	16
K (# of clusters)	2	50	2.38

[2]As indicated by the number of users [16, 130] or GitHub stars [61, 53].
[3]The analyses run on RKoh implementation of SOM are not a contribution of this dissertation.

7.2 Nondeterminism Definition and Test

We define *nondeterminism* for a toolkit as follows: constructing SOMs repeatedly using that toolkit, on the *same dataset*, with the *same parameters* leads to *statistically significant variation* in the resulting SOM.

Nondeterminism is fundamentally problematic for several reasons. First, it violates users' expectation that repeated runs have the same outcome, or at least outcomes that are statistically indistinguishable. Second, it leaves SOM users at the mercy of the random number generator, i.e., a "lucky" random seed can lead to a better SOM. Finally, nondeterminism undermines users' confidence in SOM reliability in general. We now define nondeterminism in a statistically rigorous way.

Statistical Test for Nondeterminism. We use a sensitive statistical measure of nondeterminism that improves over the tests introduced by Yin et al. in the context of clustering nondeterminism [211]: the metric values are nondeterministic if the 30 runs' outcomes have statistically significant variance. Yin et al. used Levene's [151] test set up as follows: the 30 values constitute one group, while the other group has the same mean, size, and no variance (all 30 elements are equal to the mean of the first group). If Levene's test yields a $p < 0.05$, they concluded that the runs vary significantly. The problem with using Levene's test is that it is a mean-based test hence, it is most appropriate for symmetric, moderate-tailed distributions. We improve the statistical tests as follows: we run a Levene's mean-based test, as well as Brown-Forsythe's median-based test (good for skewed distributions) and Brown-Forsythe's trimmed mean-based test (good for heavy-tailed distributions) [101]. Of these three, we pick the most sensitive, i.e., the one that finds variance across the largest number of datasets. If the underlying test results in a $p < 0.05$ we conclude that the toolkit is nondeterministic.

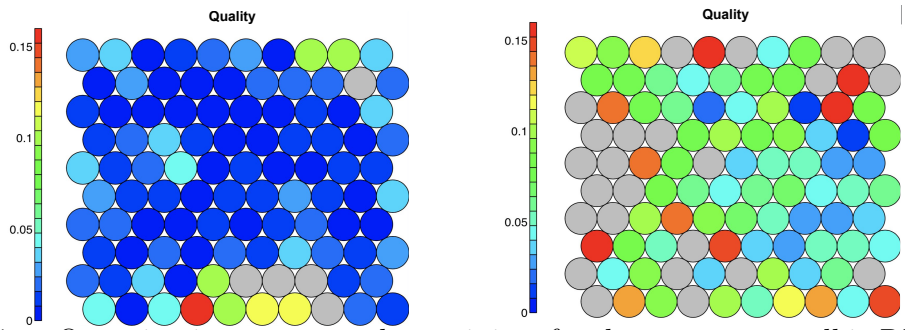


Figure 7.4 Quantization error nondeterminism for dataset ecoli, toolkit RKoh. Low error in dark blue, higher error in light blue/green/red. The run with minimum quantization error (59.75) is shown on the left while the run with maximum error (79.07) is shown on the right.

7.3 Nondeterminism Results: Internal metrics

Internal metrics use the native properties of the SOM model and input dataset in order to evaluate the quality of the SOM implementation on dimensionality reduction. We consider six internal metrics; we discuss their definition, significance, and analysis results shortly.

Parameters. SOM's recommended size is $5 \times \sqrt{N}$ neurons where N is the number of samples in the dataset to analyze [161]. For example, if a dataset has 150 samples, we have $5 \times \sqrt{150} = 5 \times 12.24 = 61.23$. Hence, the recommended map has 64 neurons, arranged in an 8-by-8 grid. To keep the SOM map settings consistent across all the tools, we have used a *hexagonal* topology and *Manhattan distance* as the activation distance. We now discuss each metric in turn.

7.3.1 Quantization error

Definition. Quantization error applies to clustering algorithms in general. The error is computed from the average Euclidean distance of sample vectors to the centroid, or best matching unit, by which they are represented. A lower quantization error value is desirable.

Results. Our nondeterminism hypothesis was confirmed using the three statistical tests. The "Quantization errors" columns of Table 7.1 show the number of datasets for which the tests indicate statistical invariance across runs. These numbers are small and consistent across tests (1–4 datasets depending on the toolkit and test). Put otherwise, for the vast majority, 375–381 datasets quantization error *differs significantly across runs*, confirming our nondeterminism hypothesis.

We illustrate one such difference between runs of the dataset ecoli[4] in Figure 7.4. The figure shows the quality, i.e., mean distance of objects mapped to a neuron to the original data point. For each neuron, quality ranges from dark blue (low error) to green (moderate error) to red (high error). Note the good fit on the left (mostly dark blue) and the worse fit on the right (more green and light blue neurons).

Table 7.2 shows the widest-3 ranges across runs. For example, in MiniSom, for dataset lsvt, quantization error varied between 5.8×10^8 and 9.7×10^8; for the same dataset, but using MATLAB, the range varied between 1.7×10^8 and 4×10^8. Therefore, MiniSom and MATLAB have non-overlapping ranges across our 30-run experiments, which is a source for concern. Finally, note that the minimum vs. the maximum quantization error can vary by 2x–3x, e.g., MATLAB schlvote (min: 1.6×10^5, max: 5.2×10^5) or TFSom sleuth_ex1221 (min: 2.0×10^4, max: 5.8×10^4). A quantization error that differs by a factor of 3 across different runs raises a reason for concern.

7.3.2 Topographic product

Definition. The topographic product (TP) indicates whether the size of the map is an appropriate fit onto the dataset. TP is computed by comparing the ranking orders in the input and output spaces, respectively; essentially, TP measures the quality of the topology preservation. If $TP < 0$, the map size is too small. Conversely, if $TP > 0$, the map size is too large. Surprisingly, we found datasets where the TP can

[4]Protein localization sites in bacteria.

Table 7.2 Widest-3 Differences in Quantization Error Across Runs

Toolkit	Dataset	Min	Max	Range	Stddev
MiniSom	lsvt	5.8E+8	9.7E+8	3.9E+8	9.3E+7
	micro-mass	1.4E+7	1.5E+7	9.1E+5	2.2E+5
	tokyo1	3.9E+5	5.1E+5	1.3E+5	2.7E+4
MATLAB	lsvt	1.7E+8	4.0E+8	2.3E+8	6.2E+7
	micro-mass	9.1E+6	9.5E+6	3.7E+5	7.9E+4
	schlvote	1.6E+5	5.2E+5	3.6E+5	9.0E+4
TFSom	tokyo1	1.2E+6	1.3E+6	9.1E+4	3.9E+4
	analcatdataoly	1.8E+5	2.3E+5	4.6E+4	1.7E+4
	sleuth_ex1221	2.0E+4	5.8E+4	3.8E+4	8.1E+3
RKoh	lsvt	5.5E+8	6.9 E+8	1.4E+8	3.9E+7
	micro-mass	1.2E+7	1.3E+7	4.0E+5	9.3E+4
	schlvote	6.7E+5	9.8E+5	3.1E+5	6.4E+4

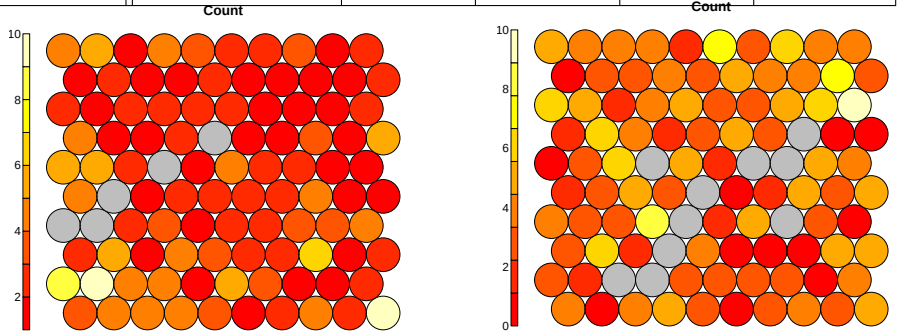


Figure 7.5 Topographic product nondeterminism for dataset colic, toolkit RKoh, exposed by plotting the number of inputs mapped to each neuron. Grey spaces (representing empty nodes) indicate that the map size is too large. The left map (predominantly red or darker orange) shows a more uniform distribution, $TP = 0.0006$. The right map shows more empty nodes and thus a higher topographic product, $TP = 0.0023$; yellow or lighter orange spaces indicate a skewed distribution, where many samples map to a single node.

be *positive* in one run and *negative* in the next run. However, it is important to note that the topographic product presents reliable results only for linear datasets [116].

Results. In Table 7.3, we see how topographic product varies across runs and tools. Certain datasets such as analcatdata_reviewer consistently have a larger topographic product, indicating a larger mapsize. However despite this consistency, certain toolkits have better results. For example, in MiniSom, the min TP for analcatdata_reviewer is 0.70 and the max TP is 1.94, which, while still greater than 0, it is the best performing toolkit as its max value is around the minimum value of the other toolkits. For instance, MATLAB has a max TP of 3.52 and min of 1.86 for the same dataset.

Table 7.3 Widest-3 Differences in Topographic Product

Toolkit	Dataset	Min	Max	Range	Stddev
MiniSom	analcatdata_reviewer	0.70	1.94	1.24	0.29
	arsenic-male-bladder	0.16	0.50	0.34	0.07
	arsenicfemalebladder	0.18	0.52	0.34	0.07
MATLAB	analcatdata_reviewer	1.86	3.52	1.66	0.42
	analcatdata_neavote	2.24	3.84	1.60	0.38
	aids	0.26	1.50	1.24	0.24
TFSom	haberman	0.25	3.77	3.52	1.10
	energy-efficiency	0.37	2.80	2.43	0.75
	rmftsa_ctoarrivals	0.28	2.26	1.98	0.60
RKoh	analcatdata_reviewer	1.66	2.41	0.75	0.14
	Titanic	0.36	0.72	0.36	0.11
	analcatdata_neavote	0.15	0.37	0.22	0.06
RKoh's negative TP values	fri_c0_500_50	-0.17	-0.14	0.03	0.01
	fri_c0_250_50	-0.15	-0.12	0.03	0.01
	fri_c0_500_25	-0.15	-0.13	0.02	0.01
	fri_c1_1000_10	-0.14	-0.12	0.02	0.01
	analcatdatabankruptcy	-0.003	0.01	0.01	0.002
	autoUniv-au6-750	-0.003	0.003	0.006	0.002
	volcanoes-e4	-0.003	0.002	0.005	0.001

In the last seven rows of Table 7.3 we focus on the RKoh toolkit. While TP is positive for the other three toolkits in all cases, RKoh managed to produce negative TP values for some datasets where other toolkits had positive TP values (see the four fri_c* rows). Additionally, RKoh also managed to simultaneously indicate that a dataset's SOM size is too small and too large as shown with the datasets analcatdata_bankruptcy, autoUniv-au6-750, and volcanoes-e4 (last three rows). Figure 7.5 provides a visualization of TP nondeterminism for dataset colic,[5] with a good fit on the left and a poor fit on the right.

7.3.3 Trustworthiness/Neighborhood preservation

Definition. Trustworthiness and Neighborhood Preservation are both topological preservation measures. Trustworthiness displays whether the projected data points that are visualized are actually close to each other in the input space. Whenever one of the neighbors on the map lattice is not one of the closest neighbors in the actual input space, the error is increased. Trustworthiness is calculated from the average of

[5]Horse surgery: surgical lesions and surgery outcome dataset.

Table 7.4 Widest-3 Differences in Trustworthiness

Toolkit	Dataset	Min	Max	Range	Stddev
MiniSom	kc2	0.02	0.85	0.83	0.27
	blood-transfusion	0.24	0.79	0.55	0.1
	cm1_req	0.60	0.99	0.39	0.13
MATLAB	kc2	0.19	0.69	0.50	0.21
	dbworld-subjects	0.41	0.73	0.32	0.07
	dbworld-subjects-stem	0.50	0.78	0.28	0.07
TFSom	kc2	0.35	0.94	0.59	0.14
	Titanic	0.35	0.88	0.53	0.23
	pc1_req	0.45	0.88	0.43	0.14
RKoh	cm1_req	0.66	0.99	0.33	0.05
	blood-transfusion	0.54	0.86	0.32	0.13
	dbworld-subjects-stem	0.67	0.83	0.16	0.03

Table 7.5 Widest-3 Differences in Distortion

Toolkit	Dataset	Min	Max	Range	Stddev
MiniSom	micro-mass	1.5E+15	1.6E+15	1.9E+14	4.6E+13
	oil_spill	1.3E+13	3.3E+13	2E+13	4.9E+12
	tokyo1	1.2E+13	2.2E+13	1E+13	2.2E+12
MATLAB	micro-mass	1.9E+15	2.3E+15	3.8E+14	1E+14
	tokyo1	4.2E+12	6E+12	1.8E+12	4.6E+11
	PieChart3	1E+11	6.2E+11	5.2E+11	1.1E+11
TFSom	oil_spill	3.0E+13	3.5E+13	5.4E+12	2.7E+12
	tokyo1	1.9E+13	2.3E+13	3.5E+12	8.0E+11
	analcatdataoly	1.5E+12	2.2E+12	7.0E+11	2.2E+11
RKoh	lsvt	5.1E+19	7.3E+19	2.2E+19	5.6E+18
	micro-mass	9.2E+14	9.6E+14	4.4E+13	1E+13
	analcatdatabo	1.7E+12	2.9E+12	1.2E+12	3E+11

these errors. By swapping the input and output space rankings in the calculations, we obtain Neighborhood Preservation. This penalizes the data points which are close in the input space but far apart in the output space. Both Trustworthiness and Neighborhood Preservation values are weighted to be kept within 0 to 1 (where 1 means perfect).

Results. Table 7.4 shows trustworthiness results. For RKoh, while there is variation across runs, ranging from 0.33 to 0.16, the overall trustworthiness is ideal, with values being higher than 0.50 and closer to 1. MiniSom, however, has the widest range (min 0.02, max 0.85) for dataset for kc2.

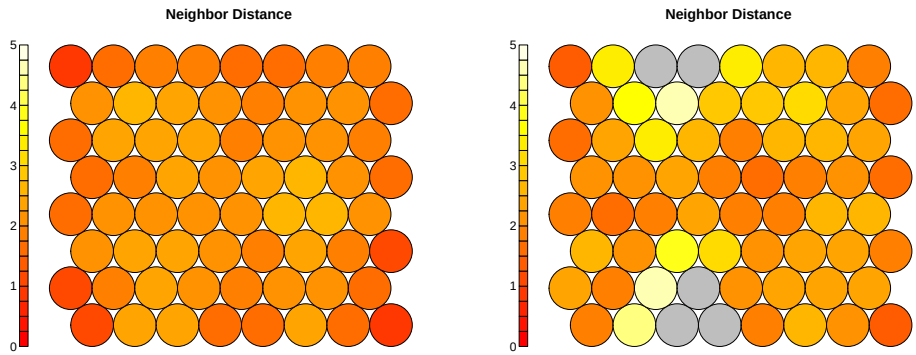


Figure 7.6 Distortion nondeterminism: in the analcatdata_boxing1 dataset, toolkit RKoh, there are variations in distortion between each node and its neighbors. The figure on the left ($distortion = 4.36$) shows significantly less distortion than the right ($distortion = 6.53$): orange indicates more similar nodes. The higher the distance, the more dissimilar the nodes are (depicted in yellow or white). An ideal mapping would have predominantly red nodes.

7.3.4 Distortion

Definition. Distortion is essentially the cost function that the SOM tries to optimize: the sum of squared Euclidean distances between samples and SOM prototypes, weighted by a neighborhood function that depends on the distances to the map's best-matching unit. As distortion measures loss (that the SOM function minimizes), a lower distortion is more desirable.

Results. Table 7.5 shows the differences in distortion. We observed high distortion in RKoh (lsvt where range was 2.2×10^{19}); for MATLAB, the largest range in variation is found in the dataset micro-mass with 3.8×10^{14}. Similarly for MiniSom, we see the same dataset having a range of 1.9×10^{14}. For a more apparent understanding of distortion, Figure 7.6 visualizes how distortion varies between runs on the same dataset, analcatdata_boxing1.[6]

[6]Boxing match results.

Table 7.6 Widest-3 Differences in Kruskal-Shepard Error

Toolkit	Dataset	Min	Max	Range	Stddev
MiniSom	chscase_adopt	0.09	0.24	0.15	0.03
	kc1-binary	0.07	0.21	0.14	0.02
	arsenic-female-lung	0.12	0.26	0.14	0.03
MATLAB	analcatdata_reviewer	0.00	0.22	0.22	0.05
	analcatdata_neavote	0.03	0.18	0.15	0.05
	fri_c4_250_100	0.24	0.39	0.15	0.04
TFSom	eucalyptus	0.05	0.28	0.23	0.07
	fri_c4_250_100	0.21	0.41	0.20	0.04
	fri_c4_500_100	0.16	0.34	0.18	0.04
RKoh	analcatdata_reviewer	0.02	0.14	0.12	0.04
	ar5	0.06	0.15	0.09	0.02
	aids	0.06	0.14	0.08	0.02

7.3.5 Kruskal-Shepard error

Definition. This value measures distance preservation between the input space and the output space. The input space is measured using Euclidean distance; in the output space, Manhattan distance between the best matching units is used.

Results. A low Kruskal-Shepard Error value is desirable as it indicates better preservation between input and output spaces. Table 7.6 shows the differences that occur across toolkits and runs. We see the best error rate (0), but largest range (0.22) for MATLAB with the dataset analcatdata_reviewer. For RKoh, again analcatdata_reviewer shows the greatest variance, having a range of 0.12. Finally, for MiniSom we see the largest error (0.24) and range (0.15) in the dataset chscase_adopt.

7.3.6 Topographic error

Definition. Topographic Error (TE), akin to Trustworthiness, is the ratio of total number of errors and number of data points on a SOM. TE is normalized to a range from 0 to 1, where 0 indicates perfect topological preservation.

Results. For lack of space we omit a Top-3 table, but results are in line with the nondeterministic outcomes we observed for other metrics. For example, analcatdata_neavote's best run with MATLAB has a min TE of 0.04 but its worst run

is a TE of 0.96, showing a range of 0.92, whereas the other toolkits' ranges were 0.73 and 0.64, respectively.

7.4 Nondeterminism Results: External Metrics

We studied internal map qualities; we now change our focus to external qualities, measuring SOM performance on clustering tasks. Specifically, external metrics are computed by comparing SOM-induced output with ground truth's class labels. The number of neurons is set to match the number of distinct output classes to be classified, i.e., map size is C, the number of distinct output classes (recommended size when the number of clusters is known[166, 177]). We used the same hexagonal topology and Manhattan distance as in Section 7.3. We now define external metrics and present the results.

7.4.1 Clustering accuracy

Definition. Clustering Accuracy divides the number of samples correctly classified by the total number of samples.

Results. To emphasize the potential consequences of nondeterminism for medical analysis, Figure 7.7 shows the clustering accuracy of MiniSom, MATLAB, and RKoh on AP Colon Lung. Accuracy of MiniSom varies significantly per run (0.24–0.7): this bimodal distribution can be interpreted as a coin toss for classification, which is undesirable. Table 7.7 further details how the accuracy varies greatly across runs and tools. *Hence, when using SOMs for medical data analysis, a particular run can influence the outcome decisively.*

7.4.2 Purity

Definition. Purity is calculated by assigning each cluster to the class which is most frequent in the cluster, and computing the ratio between how many points are

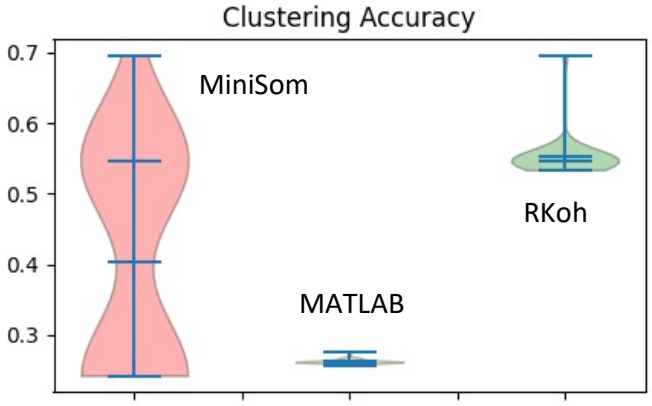


Figure 7.7 Clustering accuracy ranges for dataset AP Colon Lung.

Table 7.7 Widest-3 Differences in Clustering Accuracy

Toolkit	Dataset	Min	Max	Range	Stddev
MiniSom	confidence	0.40	0.86	0.46	0.12
	AP_Prostate_Lung	0.24	0.69	0.45	0.16
	AP_Omentum_Prostate	0.60	0.97	0.37	0.16
MATLAB	solar-flare	0.39	0.69	0.30	0.09
	dbworld-bodies	0.55	0.83	0.28	0.08
	dbworld-bodies-stem	0.58	0.86	0.28	0.07
TFSom	ar3	0.50	0.77	0.27	0.11
	mw1	0.51	0.76	0.25	0.11
	CostaMadre1	0.50	0.75	0.25	0.11
RKoh	AP_Omentum_Prostate	0.57	0.98	0.41	0.11
	AP_Endometrium_Lung	0.50	0.90	0.40	0.12
	water-treatment	0.50	0.83	0.33	0.05

accurately assigned to the total number of points. A higher purity value indicates better SOM clustering performance.

Results. Table 7.8 shows the widest-3 results. For MiniSom and RKoh we observed higher purity values, occasionally at the expense of wide range (e.g., on AP_Omentum_Prostate, purity ranges were as high as 0.41).

7.4.3 Class scatter index (CSI)

Definition. The class scatter index measures how the ground truth labels are scattered in the SOM map. Classes that are not scattered (i.e., distributed into fewer groups of neighboring units) indicate a better map.

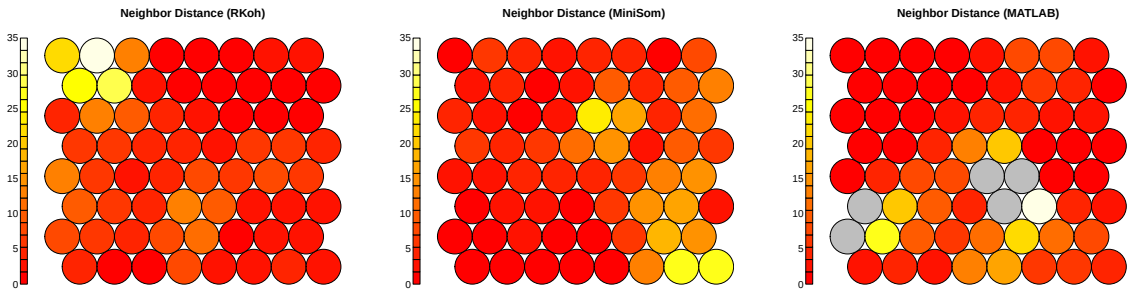


Figure 7.8 Neighborhood Preservation inconsistency in the dataset analcatdata_challenger, toolkit RKoh. Though invariant across runs, NP varies across toolkits. The red indicates an ideal mapping, with fewer samples being mapped to the same node. In contrast, yellow or white indicate many samples mapped to the same node, showing a poor map fit and neighborhood preservation. Grey represents empty nodes, i.e., map might be too large.

Table 7.8 Widest-3 Differences in Purity

Toolkit	Dataset	Min	Max	Range	Stddev
MiniSom	AP_Omentum_Prostate	0.60	0.97	0.37	0.16
	confidence	0.50	0.86	0.36	0.09
	Smartphone	0.48	0.75	0.27	0.07
MATLAB	dbworldbodies	0.54	0.82	0.28	0.08
	dbworldbodies-stem	0.57	0.85	0.28	0.07
	confidence	0.55	0.79	0.24	0.05
TFSom	aids	0.50	0.74	0.24	0.11
	analcatdatahappiness	0.38	0.58	0.20	0.07
	pollution	0.52	0.70	0.18	0.06
RKoh	AP_Omentum_Prostate	0.56	0.97	0.41	0.11
	dbworldbodies-stem	0.54	0.85	0.31	0.05
	dbworldbodies	0.54	0.81	0.27	0.04

Results. Table 7.9 shows the widest-3 CSI results. We see that with RKoh the CSI is varied but better performing with the max found in breast-tissue and user-knowledge of 2.00. Other than leaf which has a large value across toolkits, the remaining datasets have a max CSI of 2.00. Meanwhile, for MiniSom and MATLAB with the dataset amazon-commerce-rev we have a max of 7.90 and 7.16 respectively, indicating a map with larger groups of neighboring units than ideal.

7.5 Inconsistency

We believe that SOM toolkit users should expect toolkits to be interchangeable: when training an SOM on the same dataset via different toolkits one would expect if not the same, at least remotely similar results. However our experiments show that this

Table 7.9 Widest-3 Differences in CSI

Toolkit	Dataset	Min	Max	Range	Stddev
MiniSom	amazon-commerce-rev	4.80	7.90	3.10	0.83
	leaf	3.47	5.53	2.06	0.50
	eucalyptus	1.00	2.60	1.60	0.41
MATLAB	amazon-commerce-rev	5.42	7.16	1.74	0.44
	leaf	3.00	4.67	1.67	0.42
	LED-display	2.00	2.90	0.90	0.17
TFSom	soybean	1.95	3.68	1.73	0.37
	spectrometer	2.54	4.17	1.63	0.53
	leaf	3.27	4.77	1.50	0.36
RKoh	leaf	3.80	5.50	1.70	0.39
	breast-tissue	1.00	2.00	1.00	0.19
	user-knowledge	1.00	2.00	1.00	0.27

expectation is typically not met, e.g., the results of two different toolkits on the same dataset are *inconsistent*. We first illustrate inconsistency, then introduce the statistical test and its results, and finally discuss the toolkits and datasets that display the strongest contrast between toolkits.

7.5.1 Inconsistency Examples

To emphasize the consequences of inconsistency on a medical dataset note that in Figure 7.7, on dataset AP Colon Lung, one toolkit's observed accuracy could be 3 times as high compared to another toolkit. *Hence, when using SOMs for medical data analysis, a particular toolkit can influence the outcome decisively.*

7.5.2 Statistical test and results

To expose statistically significant inconsistency between toolkits, for each dataset and each pair of toolkits, we ran a Mann-Whitney U test where the two populations were the clustering accuracies (30 runs). If $p < 0.05$ we conclude that the toolkits are inconsistent. The number of datasets displaying inconsistency are shown in Table 7.10; these numbers translate to *51–92% of datasets yielding inconsistent results.*

Table 7.10 #Datasets With Statistically Significant Inconsistency

MiniSom vs. MATLAB	MiniSom vs. RKoh	MATLAB vs. RKoh	MATLAB vs. TFSom	MiniSom vs. TFSom	RKoh vs. TFSom
298	254	196	332	333	332

Table 7.11 Worst-3 Inconsistencies (Mutual ARI) Across Tools

MiniSom vs. MATLAB		MiniSom vs. RKoh		MATLAB vs. RKoh		MATLAB vs. TFSom		MiniSom vs. TFSom		RKoh vs. TFSom	
shuttle-landing-c	-0.14	shuttle-landing-cl	-0.14	trains	-0.13	pasture	-0.08	MyIris	-0.11	MyIris	-0.12
trains	-0.07	fabert	-0.08	dbworld-bodies	-0.06	ar4	-0.07	pasture	-0.08	pasture	-0.08
fri_c4_100_50	-0.06	triazines	-0.06	dbworld-sbjs-s	-0.05	wine	-0.07	wine	-0.07	wine	-0.07

Even in those rare cases where toolkits are deterministic on a certain dataset (the few non-zero values in Table 7.1) inconsistencies still arise between toolkits. For example, Figure 7.8 shows how Neighborhood Preservation is inconsistent for the deterministic dataset analcatdata_challenger.[7]

7.5.3 Mutual ARI comparison

We now quantify and discuss those cases where the resulting SOMs disagree strongly between toolkits. We use the Adjusted Rand Index (ARI), a metric introduced by Hubert and Arabie [134] that indicates how dissimilar two clusterings of the same dataset are. An $ARI = -1$ indicates strong dissimilarity between clusterings, $ARI = 0$ suggests that the clusterings are independent, whereas $ARI = 1$ indicates a perfect agreement.

For each dataset, we compute "mutual ARIs" between all toolkits pairs, that is, ARI scores between all six toolkits pairs. In other words, for each toolkits pair, say RKoh vs. TFSom, we compute the 30 runs $\times$ 30 runs ARI scores. We focus on the minimum of these 900 pairs, as it indicates the worst-possible disparity users can experience.

We present the worst disparities in Table 7.11. The strongest observed dissimilarity were between MiniSom and MATLAB, and MiniSom and RKoh, respectively: for dataset shuttle-landing-control we have $ARI = -0.14$. When comparing MATLAB and RKoh, other than trains with $ARI = -0.13$, we see that overall the discrepancy between toolkits is much less than compared with MiniSom. These negative

[7] Space Shuttle Challenger parameters.

ARI values are concerning, because a negative ARI indicates that the clusterings achieved via the two toolkits are worse than unrelated and tend toward disagreement.

7.6 Summary

Given the popularity of SOMs and neural networks in general, we conduct the first study to investigate SOM reliability in terms of determinism and consistency. Running four popular SOM packages on 381 datasets shows that users should expect wide variation across runs and toolkits. Our findings indicate a need to scrutinize SOM results, especially in high-stakes scenarios. Our study could spur further research into the causes of, and remedies for, SOM nondeterminism and inconsistency.

CHAPTER 8

RELATED WORK

Related work regarding this dissertation falls into several categories: program analysis (Section 8.1), fingerprinting (Section 8.2), medical research studies (Section 8.3), GUI extraction (Section 8.4), automated GUI testing (Section 8.5), and Android state volatility testing (Section 8.6).

8.1 Static and Dynamic Analysis

Many static flow analyzers for Android have been developed, including Amandroid [208], DIALDroid [41], DidFail [42], DroidSafe [119], FlowDroid [88] and IccTA [48]. A prior study [174] found that a typical analyzer takes on average 6 minutes per app, on par with our approach. Most of these tools use the predefined sources and sinks list from Susi [28], with the binary goal of deciding whether a source flows to a sink; this renders the results quite imprecise, limiting tools' usability.

Other static flow analyzers for Android (whose goal is still deciding whether a source flows to a sink) improve precision over the aforementioned analyzers, at the expense of running time. For example, P/Taint [121] is a Datalog-based static information flow analyzer. Their evaluation, like ours, include popular Android apps (such as Facebook Messenger or Google Chrome). Thanks to additional features such as taint transfer and sanitization, P/Taint achieves higher precision and recall.

Horndroid [102] focused on improving the precision of existing static analyses by determining whether a sink will be reached by tainted flows, and refining branch conditions to avoid false positives. However, Horndroid does not allow naming sources (as we do with the seven IDs), so their approach is not directly comparable to ours.

DroidInfer [133] uses a context-sensitive information flow type system to improve static analysis precision and scalability, and supports analysis of libraries; their focus is

on sensitive data leaks (to network or logs). Evaluation on top Google Play apps (144) shows high precision (FP=15%). DroidInfer's goal is intuitive source→sink tracking rather than algebraic signatures and ID (ab)use studies.

The Taintdroid [110] dynamic taint tracker has exposed that location and phone information are routinely leaked to advertising and content servers. Taintdroid's focus is on efficiently tracking taint within an app and the Android OS, whereas we perform static tracking, and within the confines of the app only. TaintDroid does not distinguish between raw and hashed leaks.

Myers and Liskov's label model (Jif/DLM) [165] describes "unions" of labels: set union, i.e., AND in our model. Our XORs, not supported in Jif, would be set disjoint union. While we do not support label polymorphism as Jif does, polymorphism would only help if there was cross first-party to third-party flow which we did not find (Section 2.3.4). Stefan et al.'s disjunction category labels [193] are defined as conjunctions and disjunctions on principals; "can-flow" as logical implication governs safe information flow. They implement `dclabel-static`, a prototype information flow control in Haskell, but no evaluation is provided. Montagu et al. [163] introduced label algebras – a set of labels that form a pre-lattice, i.e., with a pre-order (the term "algebraic" in our work, from algebraic data types, refers to the product and sum operations on types). The focus of these approaches was the formalism/flow model. In contrast, for us, the algebraic taint representation is a conduit to implementing a static analysis for Android and conducting six studies on 1,000 top apps.

MAPS [214] distinguishes between first-party and third-party ID leaks by the call site of sensitive API methods but does not perform taint tracking or static analysis – understandable for the scale (1,035,853 apps). This is prone to false positives, e.g., ID-retrieving calls in dead code, or IDs which are read but not used/leaked.

Dynamic taint analysis [110, 105, 145] has different goals compared to us: reduce false positives or track which servers packets go to (which is impossible

with our approach). Our static approach aims to reduce false negatives and allows analysis at scale. Dynamic analysis, in general, needs to overcome two issues (1) low coverage [103, 90, 75, 171], and (2) signing-in successfully – this is problematic in cases such as the Western Union banking app, which requires a Western Union customer account (as do other apps in Table 2.9).

8.2 Fingerprinting

Physical sensor-based device fingerprinting has been used for forensics, fraud prevention, and quality control. Researchers have measured signals from built-in electronic components (e.g., camera, radio frequency front-end) [91], flash memory chips [207], USB firmware [92], audio and accelerometer readings [97] to create unique fingerprints of hardware devices. However, as such measurements are not available through an on-device API (they require measuring the physical components of the device) app developers cannot employ any of these fingerprinting techniques, to identify the device and app users upon app installation.

Network-traffic based fingerprinting techniques analyze the host's online activity to create a unique fingerprint of the host. From Website Fingerprinting (WF) attacks [152, 205] to App Fingerprinting (AF) attacks [153] to TCP/IP Stack Fingerprinting attacks [188], all of them analyze network traffic data to identify remote users and wireless devices [141, 117, 99]. Kuzuno et al. [145] analyzed network traffic for 1,188 free popular apps from Japan's Google Play. They found that hardware IDs (Android ID, IMEI, IMSI, and SIM Serial ID) are leaked over the network by apps' advertising modules and that Android ID is the most frequently leaked sensitive information. Our scope is different: scheme (complex ID manipulation) extraction and subversion.

Studies [147, 191, 199, 111, 136] show that has replaced browser cookies to track users. Browser Fingerprinting constructs a device fingerprint entirely from the

information given by a web browser (JavaScript APIs, HTTP headers). Browser fingerprinting does not cover the concept of, and attacks on, device identification through smartphone apps, as in our case. Kang et al. [138] proposed a zero-permission mobile device identifier similar to browser fingerprinting. The algorithm generates a device fingerprint from device features, such as PixelRatio, ScreenResolution, or UserAgent. However, this approach might not be able to identify the device after changes such as browser upgrades, installing new fonts, or others.

8.3 Medical Research Studies

Bierbrier et al.'s 2014 study [96] tested medical scores (including Child-Pugh and HAS-BLED) and calculations, e.g., BMI. The authors asked 5 physicians to identify relevant scores. Two of their analyzed scores were on our list as well: 5 physicians selected Child-Pugh and 4 physicians selected HAS-BLED as scores of interest. The authors then tested each of the 14 apps (2 Android and 12 iOS) with 10 values: 2 extreme values and 8 middle values. There were errors in two Child-Pugh apps, though they were at the low score ranges hence, would not place the patient over the threshold or in a different class. No issues were found with HAS-BLED implementations. Instead of random testing, our approach verifies reference tables and apps automatically, which, aside from rigor, makes the approach more scalable. As Table 4.2, Table 4.3, and Table 4.4 show, we found issues with both these scores, including issues at the threshold. Haffey et al. [124] performed a study on 23 Android opioid conversion apps. Their study revealed two main issues. First, 11 out of 23 apps failed to identify the sources related to their calculations; 12 apps failed to state whether any medical professionals were involved in the app creation. More importantly, these apps' calculations resulted in highly variable results and significantly different outputs across apps. Huckvale et al. [135] studied 46 insulin dose calculator apps. They found that 31 of the apps pose a risk to users due to incorrect dosage calculation. Further issues included lack of

disclaimers, lack of input validation, no updates in response to changes in input, etc. Hers et al.'s 2021 study [128] has revealed an inaccurate risk assessment in an aortic surgery risk calculator. They found that the calculator underestimated complications consistently when taking into account patient demographics and data. While the study focused on the overall accuracy of the calculator in a clinical setting, it shows general reliability issues with medical calculators which we also manage to uncover in our study. Akbar et al.'s [80] meta-analysis on 74 app studies (none of which have looked at score calculators, however) has revealed numerous safety concerns, including calculation errors, potentially harmful recommendations, etc. Though somewhat complementary to our work, all these efforts underline the importance of verification and tighter scrutiny in the medical app domain.

8.4 GUI Extraction

Extracting semantic information from Android app GUIs is notoriously difficult, especially when it has to be done automatically and at scale (Choudhary et al. [103] discuss challenges and some approaches). Abbas et al. [76] extracted text data from medical text images using Optical Character Recognition (OCR). The extracted unstructured text data is then processed to produce relevant medical terms. Guigle [94] builds a searchable index of GUI elements in Android apps. Liu et al. [159] discuss a method for automatically annotating mobile UIs using a lexical database that contains design semantics. The approach involves identifying different components and concepts in UIs by leveraging the vocabulary provided by the database and a set of labeled examples. The method can automatically identify 25 UI component categories, 197 text button concepts, and 99 classes of icons in Android UIs using code-based properties and a neural network combined with anomaly detection. This approach is applicable to example-based UI search that identifies visually similar UI screens. While these OCR, search-engine, and lexical-database approaches provide detailed

information about the location and functionality of GUI elements, they cannot be applied to find medical calculator apps' GUI errors. Given our need to map GUI information to specific medical score parameters and interval-based semantics, we decided to build our own clustering and DroidBot-based approach; we are not aware of related work that would subsume our approach (Section 4.2.3).

8.5 Automated GUI Testing

The purpose of AMC [149] is to automatically examine user interface designs of vehicular applications and ensure they meet safety and consistency standards. This tool conducts automated dynamic exploration of mobile applications similar to ours, but it verifies apps based on UI properties such as the number of actions per task (should be less than 10 per screen), text word count (not more than 100 words per screen), text contrast (3:1 contrast ratio between foreground and background recommended), and button size (must be greater than $80mm^2$) and spacing (at least 15mm). On the other hand, our focus is on extracting UI elements involving numeric ranges or intervals, and verifying them for partition conditions. VanarSena [180] uses a dynamic exploration approach to test different fault induction modules on the app, including user input, network, and system state faults. They used "many randomized concurrent monkeys" approach that generates input events to test the apps and a hit testing mechanism to ensure that input events are sent to valid UI elements. The hit testing mechanism is implemented as a tree search algorithm that searches for UI elements at a given position, starting from the top-level UI element. They also used fault injection modules to simulate various user scenarios and test the robustness of the apps. The approach was effective in finding numerous crashes and bugs in apps that are already in the marketplace. In contrast, our approach is targeted at finding incorrect definitions of numeric ranges or intervals. PUMA [126] is a dynamic exploration tool similar to DroidBot, designed for automated testing of mobile applications; it uses

Android Monkey-based testing techniques [84] and is programmable to explore the application UI and report any issues or bugs found during the testing process. PUMA requires the application to be instrumented before running the tests. PUMA scripts and apps are input into the tool, and the interpreter instruments the apps to trigger app-specific events. However, random dynamic exploration does not check for, and does not detect, calculation errors as our analysis does. FlowCog[170] uses a combination of static and dynamic analyses to detect information leaks that are caused by improper handling of sensitive data. FlowCog first extracts a context-aware semantics from the Android app's source code and resource files, then uses this information to create a precise model of app behavior, including how it handles sensitive data. Finally, FlowCog performs dynamic analysis to observe the app's actual behavior and compares it against the expected behavior predicted by the model. If the runtime flow deviates from the expected behavior based on the extracted semantics and view dependencies, FlowCog reports it as a potential data leak. In contrast, we extract and verify the GUI specification and report potential errors if the GUI or score result differ from the correct reference specification.

For Android developers creating personal data-handling apps that require accessing sets of interrelated personal data, the Epistenet [106] tool can assist in addressing many of the associated challenges. The use of Epistenet involves storing personal data in a knowledge graph with a semantic structure, exposing the relationships between the data. As a result, developers only have to interact with a single API. Without Epistenet, personal data is stored in silos, and developers must manually interface with each data provider; in addition, the relationships between data are not evident. Epistenet generates a knowledge graph of personal data, categorizing it using ontologies to establish relationships. Each data piece is represented as an object with attributes and meta-attributes linked to ontology classes. This enables developers to retrieve interconnected personal data easily. In comparison, our clustering approach

uses distance metrics to map GUI texts to reference entries, whereas Epistenet relies on manual relationship mapping.

8.6 Android State Volatility Testing

LiveDroid [113] focuses on the issue of finding UI fields that might be lost during runtime changes. LiveDroid's static analysis employs a top-down approach from program variables and UI input instances to the part of saving these inputs into the Android Bundle (the default storage where Android apps can save instance state), whereas we take a bottom-up approach to trace back to UI inputs from file write APIs. However, file writes leading from user interactions are not considered as user data losses in their analysis. LiveDroid has a patching component that injects state-saving code into an app o fix state-saving issues; we do not offer an error repair component. LiveDroid handles, and was run on, F-Droid apps only; in contrast, we successfully analyzed thousands of Google Play apps in addition to F-Droid apps.

iFixDataloss [123] is similar to LiveDroid, detecting and fixing data losses due to Android lifecycle events (e.g., orientation change, back button press). Unlike LiveDroid, their approach is not limited to data losses in a singular instance of an app run, as they also detect and fix data loss issues across multiple runs. Like our approach, iFixDataloss has a reduced false positives rate as they combine static analysis with dynamic testing. However, they do not consider data losses due to system-initiated termination.

KREfinder [184] used program analysis to identify object fields that should be saved during resume-and-restart cycles to avoid user data loss. However, their technique is focused more on finding a path from a field write to an app exit without an intervening save (e.g., in the Android Bundle) rather than finding lost file writes due to system-initiated termination.

The work of Hu et al. [132], Zaeem et al. [213], and Adamsen et al. [78] focused on finding app state issues related to activity restart by generating test cases and performing systematic execution of event sequences. Our goal (lost user file writes) is different; in addition, our approach is based on static analysis whereas their approach is based on testing.

SafeExit [137] is a study on ungraceful exits in desktop applications. They propose cleanup operations on file writes that are interrupted by an ungraceful exit in order to match the program behavior to that of normal execution. However, SafeExit did not categorize file writes based on user interaction, and did not consider user data loss.

8.7 SOM Reliability

We are not aware of any work that addresses SOM reliability. Nondeterminism and inconsistency were studied before, but in the context of discrete clustering algorithms [164, 212, 211] rather than neural networks. The literature discusses how to use SOM effectively, e.g., in image classification [177] and choosing appropriate weights and features correctly [185]. To better understand SOM functionality and performance, a variety of SOM quality metrics have been proposed by Forest et al. [116] Pölzlbauer [176], Lutz [125], yet there were no investigations based on variations of SOM results. Overall, we have found no other investigation into quantifying SOM disparities, either across runs or across toolkits.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

9.1 Conclusions

In this dissertation, we have explored several critical aspects of Android app development, security, privacy, and reliability. Through comprehensive studies and innovative approaches, we have gained valuable insights and provided practical solutions to address the challenges faced in these areas.

First, we focused on the analysis of Android apps using static and dynamic analysis techniques. By leveraging these approaches, we were able to uncover vulnerabilities, identify privacy risks, and understand the behavior of apps at different levels, such as component interactions and data flows. This knowledge contributes to enhancing the overall security posture of Android apps and helps developers in building more robust and trustworthy applications.

Next, we delved into the realm of Android app reliability. By studying the effectiveness of random testing using tools like Monkey, we found that such approaches can be remarkably efficient and effective in identifying app crashes. Additionally, we investigated the impact of Monkey's parameters on code coverage and compared the coverage achieved through manual exploration. Our findings suggest that random stress testing via Monkey is a viable option for reliability testing, with comparable coverage results to manual exploration.

Furthermore, we examined the reliability of Android app components, particularly in terms of identifiers usage and abuse. By introducing algebraic-datatype taint tracking and investigating user and device fingerprinting techniques, we gained insights into potential vulnerabilities and privacy concerns associated with these identifiers.

This research contributes to the understanding of identifier-based attacks and aids in developing countermeasures to mitigate their impact.

Then, we evaluated the reliability of Android apps in diverse scenarios. By diagnosing medical score calculator apps and investigating potential user-data save and export losses due to app termination, we shed light on the reliability challenges faced in critical domains. Lastly, our study quantified nondeterminism and inconsistency in self-organizing map implementations, emphasizing the need for multiple runs and toolkit comparisons in achieving reliable results.

In conclusion, this dissertation has made substantial contributions to the fields of Android app development, security, privacy, and reliability. The insights and solutions presented here can assist developers, researchers, and practitioners in building more secure, privacy-aware, and reliable Android apps. As the Android ecosystem continues to evolve, it is crucial to address these challenges and promote the growth of a trusted and resilient mobile app environment.

9.2 Future Work

In this section, we discuss potential directions for further exploration and improvement of our research results.

9.2.1 Improved leak signatures

While our proposed approach provides a more detailed and accurate representation of leak signatures, there are several avenues for future research and enhancement.

Graph representation of leak signatures. The leak signatures presented in Section 2.2 using AND and XOR cannot differentiate between the order of identifiers when concatenated. The order of concatenation matters because the output of concatenation is different for different orderings. It also does not provide sufficient information on the leak path, i.e., it is unclear whether the leaks occur in the same

data flow path or a different one. To address these issues, we plan to extend the leak signature representation to a graph-based representation.

Policy validation. Our graph-based representation will enable us to impose and validate different privacy policies. For example, we can determine whether hardware and software Identifiers are leaked together or whether an identifier and its corresponding hash are leaked together (thus enabling deanonymization).

9.2.2 Expanding to iOS and WebView-based calculators

While this research has focused solely on Android apps, we acknowledge that the iOS versions of our examined apps may contain errors as well. Therefore, our future research aims to extend the applicability of our toolchain to the iOS platform. Additionally, we intend to streamline the analysis process for apps utilizing WebView (apps where DroidBot encountered difficulties) by reducing the manual effort involved and automating the analysis of such apps.

9.2.3 Addressing partition violations in GUIs

We have observed partition violations or incorrect score implementations in web-based medical score calculators [68, 71], and partition violations non-medical Android apps, e.g., IELTS score conversion [67]. The root cause of such issues is that current software development tools – targeting mobile, desktop, or Web platforms – fail to perform a partition check on GUI elements. These unchecked partition errors lead to user confusion and ultimately incorrect outcome. The GUI partition checks could be made mainstream and performed at app compile time, e.g., for mobile, in Android Studio or Apple Xcode, for Web, in front-end frameworks, or for desktop, in desktop IDEs.

9.2.4 Evolution of medical score errors

As shown in Table 4.1, the papers introducing the reference medical scores have been cited substantially (from 157 to 21,863 times depending on the score) so their impact is wide-reaching and long-lasting. To gain a comprehensive understanding of the propagation of errors, we plan to conduct a study examining how reference scores evolve from the original, flawed reference. This investigation will offer an evolutionary perspective: how certain errors are corrected, which errors are still preserved, and what kinds of new errors are introduced.

By addressing these future research directions, we can further advance the field of leak detection and privacy protection in the context of taint tracking as well as improve the apps' reliability. The continuous evolution and increasing complexity of mobile applications necessitate ongoing research efforts to enhance security, privacy, and data protection in the digital landscape.

REFERENCES

[1] Bankmycell. https://www.bankmycell.com/blog/number-of-google-play-store-apps/. Accessed: 2023-4-4.

[2] Emil Protalinski. Google play removed 700,000 bad apps in 2017, 70in 2016, January 2018. https://venturebeat.com/2018/01/30/google-play-removed-700000-bad-apps-in-2017-70-more-than-in-2016/. Accessed: 2023-4-4.

[3] F-Droid - Free and Open Source Android App Repository. https://f-droid.org/en/. Accessed: 2023-4-4.

[4] Mobile app download statistics usage statistics. https://buildfire.com/app-statistics/. Accessed: 2023-4-4.

[5] Mobile Banking Statistics That Show Wallets Are a Thing of the Past. https://dataprot.net/statistics/mobile-banking-statistics/. Accessed: 2023-4-4.

[6] Rahul Khosla. New data shows 57channels. https://www.heady.io/blog/new-data-shows-57-of-shoppers-prefer-mobile-apps-to-other-channels. Accessed: 2023-4-4.

[7] Samsung galaxy store. https://galaxystore.samsung.com/apps. Accessed: 2023-4-4.

[8] Shanon Roberts. Top 10 most downloaded apps of 2021 so far. https://www.cyberclick.net/numericalblogen/top-10-most-downloaded-apps-of-2020-so-far. Accessed: 2023-4-4.

[9] Statista. https://www.statista.com/statistics/245501/multiple-mobile-device-ownership-worldwide/. Accessed: 2023-4-4.

[10] Tensorflow github. https://github.com/tensorflow/tensorflow. Accessed: 2022-11-11.

[11] Cost Per Install (CPI) Rates (2018), 2018. https://www.businessofapps.com/ads/cpi/research/cost-per-install/. Accessed: 2022-11-11.

[12] A technical report on TEE and ARM TrustZone, 2019. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/a-technical-report-on-tee-and-arm-trustzone. Accessed: 2022-11-11.

[13] ANDROID_ID, November 2019. https://developer.android.com/reference/android/provider/Settings.Secure.html#ANDROID_ID. Accessed: 2022-11-11.

[14] Best practices for unique identifiers. https://developer.android.com/training/articles/user-data-ids#java, 2019. Accessed: 2019-11-11.

[15] LevelUp Consulting Google Play Store Apps, November 2019. https://play.google.com/store/apps/developer?id=LevelUp+Consulting. Accessed: 2022-11-11.

[16] Mathworks fast facts, May 2019. https://www.mathworks.com/company/aboutus.html. Accessed: 2022-11-11.

[17] Paytronix Systems Google Play Store Apps, November 2019. https://play.google.com/store/apps/developer?id=Paytronix+Systems. Accessed: 2022-11-11.

[18] PunchhTech Google Play Store Apps, November 2019. https://play.google.com/store/apps/developer?id=PunchhTech. Accessed: 2022-11-11.

[19] Relevant Mobile Google Play Store Apps, November 2019. http://relevantmobile.com/new/. Accessed: 2022-11-11.

[20] Sign your app, November 2019. https://developer.android.com/studio/publish/app-signing. Accessed: 2022-11-11.

[21] TapMangoInc Google Play Store Apps, November 2019. https://play.google.com/store/apps/developer?id=TapMango+Inc.. Accessed: 2022-11-11.

[22] TapMangoInc Google Play Store Apps, November 2019. https://play.google.com/store/apps/developer?id=Thanx. Accessed: 2022-11-11.

[23] Top 250: The ranking. https://www.restaurantbusinessonline.com/top-500-chains?year=2019&page=0#data-table. Accessed: 2022-11-11, 2019. Accessed: 2019-11-11.

[24] Total Loyalty Solutions Google Play Store Apps, November 2019. https://play.google.com/store/apps/developer?id=Total+Loyalty+Solutions. Accessed: 2022-11-11.

[25] WebView, November 2019. https://developer.android.com/reference/android/webkit/WebView. Accessed: 2022-11-11.

[26] CareZone, July 2020. https://carezone.com/. Accessed: 2022-11-11.

[27] FlowDroid, 2020. https://github.com/secure-software-engineering/FlowDroid. Accessed: 2022-11-11.

[28] SuSi, 2020. https://github.com/secure-software-engineering/SuSi. Accessed: 2022-11-11.

[29] Apartments.com Rental Search, February 2021. https://play.google.com/store/apps/details?id=com.apartments.mobile.android. Accessed: 2022-11-11.

[30] Aaptiv: 1 Audio Fitness App, February 2021. https://play.google.com/store/apps/details?id=com.aaptiv.android. Accessed: 2022-11-11.

[31] Amber WeatherRadar Free, February 2021. https://play.google.com/store/apps/details?id=com.amber.weather. Accessed: 2022-11-11.

[32] Audiobooks.com, February 2021. https://play.google.com/store/apps/details?id=com.audiobooks.androidapp&hl=en_US&gl=US. Accessed: 2022-11-11.

[33] Bitcoin, Ethereum, IOTA Ripple Price,Crypto News, February 2021. https://play.google.com/store/apps/details?id=com.crypto.currency. Accessed: 2022-11-11.

[34] Blink Health Rx - Best Discount Pharmacy Prices, February 2021. https://play.google.com/store/apps/details?id=com.blinkhealth.blinkandroid. Accessed: 2022-11-11.

[35] BURGER KING, February 2021. https://play.google.com/store/apps/details?id=com.emn8.mobilem8.nativeapp.bk. Accessed: 2022-11-11.

[36] CBS News - Live Breaking News, February 2021. https://play.google.com/store/apps/details?id=com.treemolabs.apps.cbsnews. Accessed: 2022-11-11.

[37] CGTN – China Global TV Network, February 2021. https://play.google.com/store/apps/details?id=com.imib.cctv. Accessed: 2022-11-11.

[38] CheapOair: Cheap Flights, Cheap Hotels Booking App, February 2021. https://play.google.com/store/apps/details?id=com.fp.cheapoair. Accessed: 2022-11-11.

[39] CRAN package kohonen: Supervised and Unsupervised Self-Organising Maps, April 2021. https://cran.r-project.org/web/packages/kohonen/index.html. Accessed: 2022-11-11.

[40] Curb - The Taxi App, February 2021. https://play.google.com/store/apps/details?id=com.ridecharge.android.taximagic. Accessed: 2022-11-11.

[41] DIALDroid, 2021. https://github.com/dialdroid-android/DIALDroid. Accessed: 2022-11-11.

[42] DidFail, 2021. https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=508078. Accessed: 2022-11-11.

[43] FOX NOW: Watch Live On Demand TV Stream Sports, February 2021. https://play.google.com/store/apps/details?id=com.fox.now. Accessed: 2022-11-11.

[44] Free WiFi Passwords Internet Hotspot - WiFi Map, February 2021. https://play.google.com/store/apps/details?id=io.wifimap.wifimap. Accessed: 2022-11-11.

[45] FTC: Advertising FAQ'S: A Guide for Small Business, 2021. https://www.ftc.gov/tips-advice/business-center/guidance/advertising-faqs-guide-small-business. Accessed: 2022-11-11.

[46] GPS Navigation System, Traffic Maps by Karta, February 2021. https://play.google.com/store/apps/details?id=com.kartatech.karta.gps. Accessed: 2022-11-11.

[47] Greyhound Lines, February 2021. https://play.google.com/store/apps/details?id=com.greyhound.mobile.consumer. Accessed: 2022-11-11.

[48] IccTA., 2021. https://sites.google.com/site/icctawebpage/source-and-usage. Accessed: 2022-11-11.

[49] JCPenney – Shopping Deals, February 2021. https://play.google.com/store/apps/details?id=com.jcp. Accessed: 2022-11-11.

[50] letgo: Buy Sell Used Stuff, Cars, Furniture, February 2021. https://play.google.com/store/apps/details?id=com.abtnprojects.ambatana. Accessed: 2022-11-11.

[51] Lyft - Rideshare, Bikes, Scooters Transit, February 2021. https://play.google.com/store/apps/details?id=me.lyft.android. Accessed: 2022-11-11.

[52] MATLAB selforgmap, April 2021.
https://www.mathworks.com/help/deeplearning/ref/selforgmap.html.
Accessed: 2022-11-11.

[53] MiniSom Self Organizing Maps, April 2021.
https://github.com/JustGlowing/minisom. Accessed: 2022-11-11.

[54] National early warning score (news) 2, Nov 2021. Accessed: 2022-11-11.

[55] NJ TRANSIT Mobile App, February 2021.
https://play.google.com/store/apps/details?id=com.njtransit.njtapp.
Accessed: 2022-11-11.

[56] One Dollar - Tap To Win, February 2021.
https://apk.support/app/com.giinger.onedollar. Accessed: 2022-11-11.

[57] OpenML, April 2021. https://www.openml.org/. Accessed: 2022-11-11.

[58] Sam's Club Scan Go: Wholesale Shopping Savings, February 2021.
https://apk.support/app/com.samsclub.sng. Accessed: 2022-11-11.

[59] Sixty Vines, 2021. https:
//apkpure.com/sixty-vines/com.relevantmobile.sixtyvines.activity.
Accessed: 2022-11-11.

[60] Spectrum TV, February 2021.
https://play.google.com/store/apps/details?id=com.TWCableTV.
Accessed: 2022-11-11.

[61] TensorFlow Self-Organizing Map, April 2021.
https://github.com/cgorman/tensorflow-som. Accessed: 2022-11-11.

[62] Texas Roadhouse Mobile, February 2021.
https://play.google.com/store/apps/details?id=
com.relevantmobile.texasroadhouse. Accessed: 2022-11-11.

[63] The R Project for Statistical Computing, April 2021. https://www.r-project.org/.
Accessed: 2022-11-11.

[64] Wendy's – Earn Rewards, Order Food Score Offers, February 2021. https:
//play.google.com/store/apps/details?id=com.wendys.nutritiontool.
Accessed: 2022-11-11.

[65] Western Union International: Send Money Transfer, February 2021.
https://play.google.com/store/apps/details?id=
com.westernunion.moneytransferr3app.eu. Accessed: 2022-11-11.

[66] Zipcar, February 2021.
https://play.google.com/store/apps/details?id=com.zc.android.
Accessed: 2022-11-11.

[67] Ielts band score. https://play.google.com/store/apps/details?id=com.samir.ieltsmarkstoband, 2022. Accessed: 2022-09-01.

[68] Mdapp sofa score. https://www.mdapp.co/sequential-organ-failure-assessment-sofa-score-calculator-184/, 2022. Accessed: 2022-09-01.

[69] MEWS Brasil, March 2022. https://play.google.com/store/apps/details?id=appinventor.ai_blinkeado.InformaticasaudeMEWS. Accessed: 2022-11-11.

[70] Policy for device software functions and mobile medical applications guidance for industry and food and drug administration staff, Sep 2022.

[71] Sofa. https://www.rccc.eu/ppc/indicadores/sofa.html, 2022. Accessed: 2022-09-01.

[72] Espresso — Android Developers, May 2023. https://developer.android.com/training/testing/espresso. Accessed: 2022-11-11.

[73] Validate Credit Card Numbers, May 2023. https://www.validcreditcardnumber.com/. Accessed: 2022-11-11.

[74] What is the identifier for advertisers (IDFA)?, May 2023. https://www.adjust.com/glossary/idfa/. Accessed: 2022-11-11.

[75] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In FSE '13.

[76] Asim Abbas, Muhammad Zaki Ansaar, and Sungyoung Lee. Medical concept extraction using smartphone and natural language processing techniques (poster). In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, page 630–631, New York, NY, USA, 2019. Association for Computing Machinery.

[77] Acrylic Paint. Acrylic paint — f-droid - free and open source android app repository, April 2022. https://f-droid.org/en/packages/anupam.acrylic/. Accessed: 2022-11-11.

[78] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 83–93, New York, NY, USA, 2015. Association for Computing Machinery.

[79] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[80] Saba Akbar, Enrico Coiera, and Farah Magrabi. Safety concerns with consumer-facing mobile health applications and their consequences: a scoping review. *J. Am. Med. Inform. Assoc.*, 27(2):330–340, February 2020.

[81] Wayne M. Alves, Brett E. Skolnick, Brett E. Skolnick, and Shamik Chakraborty. *Chapter 5 - Traumatic Brain Injury*. Academic Press, 2018.

[82] Android Developers. Method Tracer, September 2017. `https://developer.android.com/studio/profile/am-methodtrace.html`. Accessed: 2022-11-11.

[83] Android Developers. UI/Application Exerciser Monkey, September 2017. `http://developer.android.com/tools/help/monkey.html`. Accessed: 2022-11-11.

[84] Android Developers. UI/Application Exerciser Monkey, May 2023. `https://developer.android.com/studio/test/other-testing-tools/monkey`. Accessed: 2022-11-11.

[85] Android Open Source Project. Low memory killer daemon, May 2022. `https://source.android.com/devices/tech/perf/lmkd`. Accessed: 2022-11-11.

[86] GCAL Aponso. Effective memory management for mobile operating systems. *American Journal of Engineering Research (AJER)*, 246, 2017.

[87] Apple, Inc. Swift dispatchqueue, May 2022. `https://developer.apple.com/documentation/dispatch/dispatchqueue`. Accessed: 2022-11-11.

[88] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, June 2014.

[89] Tanzirul Azim, Arash Alavi, Iulian Neamtiu, and Rajiv Gupta. Dynamic slicing for android. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1154–1164, 2019.

[90] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages amp; Applications*, OOPSLA '13, page 641–660, New York, NY, USA, 2013. Association for Computing Machinery.

[91] Gianmarco Baldini and Gary Steri. A survey of techniques for the identification of mobile phones using the physical fingerprints of the built-in components. *IEEE Communications Surveys Tutorials*, 19(3):1761–1789, 2017.

[92] Adam Bates, Ryan Leonard, Hannah Pruse, Daniel Lowd, and Kevin Butler. Leveraging usb to establish host identity using commodity devices. 01 2014.

[93] Md. Hajian Berenjestanaki, Mauro Conti, and Ankit Gangwal. On the exploitation of online sms receiving services to forge id verification. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, pages 22:1–22:5, New York, NY, USA, 2019. ACM.

[94] Carlos Bernal-Cárdenas, Kevin Moran, Michele Tufano, Zichang Liu, Linyong Nan, Zhehan Shi, and Denys Poshyvanyk. Guigle: A GUI search engine for android apps. *CoRR*, abs/1901.00891, 2019.

[95] Antonio Bianchi, Yanick Fratantonio, Aravind Machiry, Christopher Kruegel, Giovanni Vigna, Simon Chung, and Wenke Lee. Broken fingers: On the usage of the fingerprint api in android. 01 2018.

[96] Rachel Bierbrier, Vivian Lo, and Robert C Wu. Evaluation of the accuracy of smartphone medical calculation apps. *J. Med. Internet Res.*, 16(2):e32, February 2014.

[97] Hristo Bojinov, Yan Michalevsky, Gabi Nakibly, and Dan Boneh. Mobile device identification via sensor fingerprinting, 2014. Accessed: 2022-11-11.

[98] Michael J. Breslow and Omar Badawi. Severity scoring in the critically ill. *Chest*, 141(1):245–252, 2012.

[99] Vladimir Brik, Suman Banerjee, Marco Gruteser, and Sangho Oh. Wireless device identification with radiometric signatures. pages 116–127, 01 2008.

[100] Amy Brown, Apoorva Ballal, and Mo Al-Haddad. Recognition of the critically ill patient and escalation of therapy. *Anaesthesia and Intensive Care Medicine*, 20, 12 2018.

[101] Morton B. Brown and Alan B. Forsythe. Robust tests for the equality of variances. *Journal of the American Statistical Association*, 69, 1974.

[102] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. Horndroid: Practical and sound static analysis of android applications by smt solving. 07 2017.

[103] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? (e). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 429–440, Washington, DC, USA, 2015. IEEE Computer Society.

[104] ALONZO CHURCH. *The Calculi of Lambda Conversion. (AM-6)*. Princeton University Press, 1941.

[105] Landon P. Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. Spandex: Secure password tracking for android. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 481–494, San Diego, CA, August 2014. USENIX Association.

[106] Sauvik Das, Jason Wiese, and Jason I. Hong. Epistenet: Facilitating programmatic access processing of semantically related mobile personal data. In *Proceedings of the 18th International Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '16, page 244–253, New York, NY, USA, 2016. Association for Computing Machinery.

[107] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[108] Guido Deboeck and Teuvo Kohonen. *Visual explorations in finance: with self-organizing maps*. Springer Science and Business Media, 2013.

[109] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Pios: Detecting privacy leaks in ios applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[110] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association.

[111] Amin FaizKhademi, Mohammad Zulkernine, and Komminist Weldemariam. Fpguard: Detection and prevention of browser fingerprinting. In Pierangela Samarati, editor, *Data and Applications Security and Privacy XXIX*, pages 293–308, Cham, 2015. Springer International Publishing.

[112] Michael Anthony Fajardo, Guy Balthazaar, Alexandra Zalums, Lyndal Trevena, and Carissa Bonner. Favourable understandability, but poor actionability: An evaluation of online type 2 diabetes risk calculators. *Patient Education and Counseling*, 102(3):467–473, 2019.

[113] Umar Farooq, Zhijia Zhao, Manu Sridharan, and Iulian Neamtiu. Livedroid: Identifying and preserving mobile app state in volatile runtime environments. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[114] Flavio Lopes Ferreira. Serial evaluation of the sofa score to predict outcome in critically ill patients. *JAMA*, 286(14):1754, 2001.

[115] Greir Ander Huck Flynn, Barbara Polivka, and Jodi Herron Behr. Smartphone use by nurses in acute care settings. *Comput. Inform. Nurs.*, 36(3):120–126, March 2018.

[116] Florent Forest, Mustapha Lebbah, Hanane Azzag, and Jérôme Lacaille. A survey and implementation of performance metrics for self-organized maps, 2020.

[117] Jason Franklin, Damon Mccoy, Parisa Tabriz, Vicentiu Neagoe, Jamie Randwyk, and Douglas Sicker. Passive data link layer 802.11 wireless device driver fingerprinting. 01 2006.

[118] J Gardner-Thorpe, N Love, J Wrightson, S Walsh, and N Keeling. The value of modified early warning score (mews) in surgical in-patients: A prospective observational study. *The Annals of The Royal College of Surgeons of England*, 88(6):571–575, 2006.

[119] Michael Gordon, Kim deokhwan, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of android applications in droidsafe. 01 2015.

[120] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 1025–1035, New York, NY, USA, 2014. Association for Computing Machinery.

[121] Neville Grech and Yannis Smaragdakis. P/taint: Unified points-to and taint analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[122] Tim A. Green, Stevan Whitt, Jeffery L. Belden, Sanda Erdelez, and Chi-Ren Shyu. Medical calculators: Prevalence, and barriers to use. *Computer Methods and Programs in Biomedicine*, 179:105002, 2019.

[123] Wunan Guo, Zhen Dong, Liwei Shen, Wei Tian, Ting Su, and Xin Peng. Detecting and fixing data loss issues in android apps. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 605–616, New York, NY, USA, 2022. Association for Computing Machinery.

[124] Faye Haffey, Richard R. Brady, and Simon Maxwell. A comparison of the reliability of smartphone apps for opioid conversion. *Drug Safety*, 36(2):111–117, 2013.

[125] Lutz Hamel. Som quality measures: An efficient statistical approach. In *Advances in Self-Organizing Maps and Learning Vector Quantization*, pages 49–59, 2016.

[126] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, page 204–217, New York, NY, USA, 2014. Association for Computing Machinery.

[127] Dick Hardt. The oauth 2.0 authorization framework. *RFC*, 6749:1–76, 2012.

[128] Tessa M. Hers, Jan Van Schaik, Niels Keekstra, Hein Putter, Jaap F. Hamming, and Joost R. Van Der Vorst. Inaccurate risk assessment by the acs nsqip risk calculator in aortic surgery. *Journal of Clinical Medicine*, 10(22):5426, 2021.

[129] Eveline Hitti, Dima Hadid, Jad Melki, Rima Kaddoura, and Mohamad Alameddine. Mobile device use among emergency department healthcare professionals: Prevalence, utilization and attitudes. *Scientific Reports*, 11(1), 2021.

[130] Mark Hornick. Oracle r technologies overview. `https://www.oracle.com/assets/media/oraclertechnologies-2188877.pdf`. Accessed: 2022-11-11.

[131] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[132] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[133] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, page 106–117, New York, NY, USA, 2015. Association for Computing Machinery.

[134] Lawrence Hubert and Phipps Arabie. Comparing partitions. *Journal of Classification*, 2:193–218, 02 1985.

[135] Kit Huckvale, Samanta Adomaviciute, José Tomás Prieto, Melvin Khee-Shing Leow, and Josip Car. Smartphone apps for calculating insulin dose: a systematic assessment. *BMC Med.*, 13(1):106, May 2015.

[136] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. pages 1143–1161, 05 2021.

[137] Zhouyang Jia, Shanshan Li, Tingting Yu, Xiangke Liao, and Ji Wang. Automatically detecting missing cleanup for ungraceful exits. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 751–762, New York, NY, USA, 2019. Association for Computing Machinery.

[138] Nian-hua KANG, Ming-zhi CHEN, Ying-yan FENG, Wei-ning LIN, Chuan-bao LIU, and Guang-yao LI. Zero-permission mobile device identification based on the similarity of browser fingerprints. *DEStech Transactions on Computer Science and Engineering*, (cst), 2017.

[139] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, page 194–206, New York, NY, USA, 1973. Association for Computing Machinery.

[140] W A Knaus, E A Draper, D P Wagner, and J E Zimmerman. APACHE II: a severity of disease classification system. *Crit. Care Med.*, 13(10):818–829, October 1985.

[141] T. Kohno, A. Broido, and K.C. Claffy. Remote physical device fingerprinting. *IEEE Transactions on Dependable and Secure Computing*, 2(2):93–108, 2005.

[142] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[143] H. Krawczyk, M. Bellare, and R. Canetti. Rfc2104: Hmac: Keyed-hashing for message authentication, 1997.

[144] Krishan Kumar. A thorough investigation of code obfuscation techniques for software protection. *International Journal of Computer Sciences and Engineering*, 3:158–164, 05 2015.

[145] H. Kuzuno and S. Tonami. Signature generation for sensitive information leakage in android applications. In *2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW)*, pages 112–119, April 2013.

[146] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. Browser fingerprinting: A survey. *ACM Transactions on the Web*, 14:1–33, 04 2020.

[147] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. Browser fingerprinting: A survey. *ACM Trans. Web*, 14(2), apr 2020.

[148] Niel Lebeck, Arvind Krishnamurthy, Henry M Levy, and Irene Zhang. End the senseless killing: Improving memory management for mobile operating systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 873–887, 2020.

[149] Kyungmin Lee, Jason Flinn, T.J. Giuli, Brian Noble, and Christopher Peplin. Amc: Verifying user interface properties for vehicular applications. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, page 1–12, New York, NY, USA, 2013. Association for Computing Machinery.

[150] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.11 Documentation and user's manual, 2020. Accessed: 2022-11-11.

[151] Howard Levene. Robust tests for equality of variances. In *Contributions to probability and statistics: essays in honor of Harold Hotelling*, pages 278–292. Stanford University Press: Palo Alto, CA, USA, 1960.

[152] Jianfeng Li, Shuohan Wu, Hao Zhou, Xiapu Luo, Ting Wang, Yangyang Liu, and Xiaobo Ma. Packet-level open-world app fingerprinting on wireless traffic. *The 2022 Network and Distributed System Security Symposium (NDSS '22)*.

[153] Jianfeng Li, Shuohan Wu, Hao Zhou, Xiapu Luo, Ting Wang, Yangyang Liu, and Xiaobo Ma. Packet-level open-world app fingerprinting on wireless traffic. *The 2022 Network and Distributed System Security Symposium (NDSS '22)*.

[154] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: A lightweight ui-guided test input generator for android. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 23–26, Piscataway, NJ, USA, 2017. IEEE Press.

[155] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: A lightweight ui-guided test input generator for android. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 23–26, Piscataway, NJ, USA, 2017. IEEE Press.

[156] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. *ACM SIGPLAN Notices*, 33, 05 2000.

[157] Gregory Y. Lip and Deirdre A. Lane. Stroke prevention in atrial fibrillation. *JAMA*, 313(19):1950, 2015.

[158] Gregory Y.H. Lip, Lars Frison, Jonathan L. Halperin, and Deirdre A. Lane. Comparative validation of a novel risk score for predicting bleeding risk in anticoagulated patients with atrial fibrillation: The has-bled (hypertension, abnormal renal/liver function, stroke, bleeding history or predisposition, labile inr, elderly, drugs/alcohol concomitantly) score. *Journal of the American College of Cardiology*, 57(2):173–180, 2011.

[159] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. Learning design semantics for mobile apps. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, page 569–579, New York, NY, USA, 2018. Association for Computing Machinery.

[160] Benjamin Livshits, Dimitrios Vardoulakis, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, José Amaral, Bor-Yuh Chang, Samuel Guyer, Uday Khedker, and Anders Møller. In defense of soundiness: A manifesto. *Communications of the ACM*, 58:44–46, 01 2015.

[161] Maryam Lotfi, Dara Moazzami, Behzad Moshiri, and M. Delavar. Anomaly detection using a self-organizing map and particle swarm optimization. *Scientia Iranica*, 18:1460–1468, 12 2011.

[162] Bartosz Milewski et al. If Either can be either Left or Right but not both, then why does it correspond to OR instead of XOR in Curry-Howard correspondence?, 2020. Accessed: 2022-11-11.

[163] Benoît Montagu, B. Pierce, and R. Pollack. A theory of information-flow labels. *2013 IEEE 26th Computer Security Foundations Symposium*, pages 3–17, 2013.

[164] Vincenzo Musco, Xin Yin, and Iulian Neamtiu. Smokeout: An approach for testing clustering implementations. In *ICST 2019*, April 2019.

[165] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, October 2000.

[166] NeuPy. NeuPy Neural Networks in Python, March 2021. `http://neupy.com/apidocs/neupy.algorithms.competitive.sofm.html`. Accessed: 2022-11-11.

[167] Royal College of Physicians. National early warning score (news) 2: standardising the assessment of acute-illness severity in the nhs. updated report of a working party 2017. 2021.

[168] T. Olsson, A. Terent, and L. Lind. Rapid emergency medicine score: A new prognostic tool for in-hospital mortality in nonsurgical emergency department patients. *Journal of Internal Medicine*, 255(5):579–587, 2004.

[169] Ortholive. 32 Statistics on mHealth, April 2019. `https://www.ortholive.com/blog/32-statistics-on-mhealth/`. Accessed: 2022-11-11.

[170] Xiang Pan, Yinzhi Cao, Xuechao Du, Boyuan He, Gan Fang, and Yan Chen. Flowcog: Context-aware semantics extraction and analysis of information flow leaks in android apps. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 1669–1685, USA, 2018. USENIX Association.

[171] Priyam Patel, Gokul Srinivasan, Sydur Rahaman, and Iulian Neamtiu. On the effectiveness of random testing for android: Or how i learned to stop worrying and love the monkey. In *Proceedings of the 13th International Workshop on Automation of Software Test*, AST '18, page 34–37, New York, NY, USA, 2018. Association for Computing Machinery.

[172] Rikesh K Patel, Adele E Sayers, Nina L Patrick, Kaylie Hughes, Jonathan Armitage, and Iain Andrew Hunter. A UK perspective on smartphone use amongst doctors within the surgical profession. *Ann. Med. Surg. (Lond.)*, 4(2):107–112, June 2015.

[173] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*,

EuroSys '12, page 29–42, New York, NY, USA, 2012. Association for Computing Machinery.

[174] Felix Pauck, Eric Bodden, and Heike Wehrheim. Do android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 331–341, New York, NY, USA, 2018. Association for Computing Machinery.

[175] Ryan Pelletier, Jeff Nagge, and John-Michael Gamble. Variation in bleeding risk estimates among online calculators. *Canadian Family Physician*, 68(4), 2022.

[176] Georg Pölzlbauer. Survey and comparison of quality measures for self-organizing maps. In *Proceedings of the Fifth Workshop on Data Analysis (WDA'04)*, pages 67–82, 2004.

[177] Dian Pratiwi. The use of self organizing map method and feature selection in image database classification system. *International Journal of Computer Science Issues*, 9, 06 2012.

[178] Ihsan Ayyub Qazi, Zafar Ayyub Qazi, Theophilus A. Benson, Ghulam Murtaza, Ehsan Latif, Abdul Manan, and Abrar Tariq. Mobile web browsing under memory pressure. *SIGCOMM Comput. Commun. Rev.*, 50(4):35–48, oct 2020.

[179] Sydur Rahaman, Iulian Neamtiu, and Xin Yin. Algebraic-datatype taint tracking, with applications to understanding android identifier leaks. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 70–82, New York, NY, USA, 2021. Association for Computing Machinery.

[180] Lenin Ravindranath, Suman Nath, Jitendra Padhye, and Hari Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, page 190–203, New York, NY, USA, 2014. Association for Computing Machinery.

[181] K J Rhee, C J Fisher, Jr, and N H Willitis. The rapid acute physiology score. *Am. J. Emerg. Med.*, 5(4):278–282, July 1987.

[182] Gisbert Schneider and Petra Schneider. Macromolecular target prediction by self-organizing feature maps. *Expert opinion on drug discovery*, 2016.

[183] P Schneider, Y Tanrikulu, and G Schneider. Self-organizing maps in drug discovery: Compound library design, scaffold-hopping, repurposing. *Current medicinal chemistry*, 16:258–66, 02 2009.

[184] Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. Finding resume and restart errors in android applications. In *Proceedings of the 2016 ACM SIGPLAN*

International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, page 864–880, New York, NY, USA, 2016. Association for Computing Machinery.

[185] Navdeep Singh. Self-organizing maps for machine learning algorithms, Jun 2018. Accessed: 2022-11-11.

[186] A. J. Six, B. E. Backus, and J. C. Kelder. Chest pain in the emergency room: Value of the heart score. *Netherlands Heart Journal*, 16(6):191–196, 2008.

[187] André Skupin, Joseph R. Biberstine, and Katy Börner. Visualizing the topical structure of the medical sciences: A self-organizing map approach. *PLOS ONE*, 8(3), 03 2013.

[188] Matthew Smart, G. Robert Malan, and Farnam Jahanian. Defeating tcp/ip stack fingerprinting. In *Proceedings of the 9th Conference on USENIX Security Symposium - Volume 9*, SSYM'00, page 17, USA, 2000. USENIX Association.

[189] Sharon R. Smith, Jack D. Baty, and Dee Hodge. Validation of the pulmonary score: An asthma severity score for children. *Academic Emergency Medicine*, 9(2):99–104, 2002.

[190] Carey G. Smoak. Checksum please: A way to ensure data integrity. 2012.

[191] Konstantinos Solomos, Panagiotis Ilia, Nick Nikiforakis, and Jason Polakis. Escaping the confines of time: Continuous browser extension fingerprinting through ephemeral modifications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2675–2688, New York, NY, USA, 2022. Association for Computing Machinery.

[192] Soot.2022. Soot: a java optimization framework., April 2022. `https://www.sable.mcgill.ca/soot/`. Accessed: 2022-11-11.

[193] Deian Stefan, Alejandro Russo, David Mazières, and John C Mitchell. Disjunction category labels. In *Nordic conference on secure IT systems*, pages 223–239. Springer, 2011.

[194] Gregor Stiglic and Peter Kokol. Stability of ranked gene lists in large microarray analysis studies. *Journal of biomedicine & biotechnology*, 2010:616358, 06 2010.

[195] Strace. Using strace — android open source project, April 2022. `https://source.android.com/devices/tech/debug/strace`. Accessed: 2022-11-11.

[196] Eric Séverin. Self organizing maps in corporate finance: Quantitative and qualitative analysis of debt and leasing. *Neurocomputing*, 73(10):2061–2067, 2010.

[197] G Teasdale and B Jennett. Assessment of coma and impaired consciousness. a practical scale. *Lancet*, 2(7872):81–84, July 1974.

[198] Threading in Android. Better performance through threading — android developers, April 2022. https://developer.android.com/topic/performance/threads. Accessed: 2022-11-11.

[199] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-stalker: Tracking browser fingerprint evolutions. pages 728–741, 05 2018.

[200] Juha Vesanto and Esa Alhoniemi. Clustering of the self-organizing map. *IEEE Transactions on neural networks*, 11(3), 2000.

[201] Jean-Louis Vincent, Rui Moreno, Jukka Takala, S Willatts, A Mendonça, H Bruining, C Reinhart, Peter Suter, and L Thijs. The sofa (sepsis-related organ failure assessment) score to describe organ dysfunction/failure. on behalf of the working group on sepsis-related problems of the european society of intensive care medicine. *Intensive care medicine*, 22:707–10, 08 1996.

[202] Vlad Roubtsov. EMMA: a free Java code coverage tool, July 2017. http://emma.sourceforge.net/. Accessed: 2022-11-11.

[203] Wabbitemu. Wabbitemu, April 2022. http://wabbitemu.org/. Accessed: 2022-11-11.

[204] Sean Wallace, Marcia Clark, and Jonathan White. 'it's on my iphone': attitudes to the use of mobile computing devices in medical education, a mixed-methods study. *BMJ Open*, 2(4):e001099, August 2012.

[205] Tao Wang. The one-page setting: A higher standard for evaluating website fingerprinting defenses. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2794–2806, New York, NY, USA, 2021. Association for Computing Machinery.

[206] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. Jekyll on ios: When benign apps become evil. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, page 559–572, USA, 2013. USENIX Association.

[207] Yinglei Wang, Wing-kei Yu, Shuo Wu, Greg Malysa, G. Edward Suh, and Edwin C. Kan. Flash memory for ubiquitous hardware security functions: True random number generation and device fingerprints. In *2012 IEEE Symposium on Security and Privacy*, pages 33–47, 2012.

[208] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 1329–1341, New York, NY, USA, 2014. Association for Computing Machinery.

[209] Cong Wu, Kun He, Jing Chen, Ziming Zhao, and Ruiying Du. Liveness is not enough: Enhancing fingerprint authentication with behavioral biometrics to defeat

puppet attacks. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.

[210] Mingyuan Xia, Wenbo He, Xue Liu, and Jie Liu. Why application errors drain battery easily? a study of memory leaks in smartphone apps. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13, New York, NY, USA, 2013. Association for Computing Machinery.

[211] Xin Yin, Vincenzo Musco, Iulian Neamtiu, and Usman Roshan. Statistically rigorous testing of clustering implementations. In *AITEST 2019*, April 2019.

[212] Xin Yin, Iulian Neamtiu, Saketan Patil, and Sean T. Andrews. Implementation-induced inconsistency and nondeterminism in deterministic clustering algorithms. In *ICST 2020*, October 2020.

[213] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, page 183–192, USA, 2014. IEEE Computer Society.

[214] Sebastian Zimmeck, Peter Story, Daniel Smullen, Abhilasha Ravichander, Ziqi Wang, Joel Reidenberg, N. Russell, and Norman Sadeh. Maps: Scaling privacy compliance analysis to a million apps. *Proceedings on Privacy Enhancing Technologies*, 2019:66–86, 07 2019.

[215] Heinz Zimmermann and Jürg Reichen. Hepatectomy: Preoperative analysis of hepatic function and postoperative liver failure. *Digestive Surgery*, 15(1):1–11, 1998.