UNIVERSITY OF
CAMBRIDGE

Hardware and software fingerprinting of mobile devices

Jiexin Zhang

Robinson College

This dissertation is submitted on November, 2020 for the degree of Doctor of Philosophy

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. It is not substantially the same as any that I have submitted, or am concurrently submitting, for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or is being concurrently submitted, for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. This dissertation does not exceed the prescribed limit of 60 000 words.

Jiexin Zhang
November, 2020

Abstract

This dissertation presents novel and practical algorithms to identify the software and hardware components on mobile devices. In particular, we make significant contributions in two challenging areas: *library fingerprinting*, to identify third-party software libraries, and *device fingerprinting*, to identify individual hardware components. Our work has significant implications for the privacy and security of mobile platforms.

Software-based library fingerprinting can be used to detect vulnerable libraries and uncover large-scale data collection activities. We develop a novel Android library fingerprinting tool, LibID, to reliably identify specific versions of in-app third-party libraries. LibID is more effective against code obfuscation than prior art. When comparing LibID with other tools in identifying the correct library version using obfuscated F-Droid apps, LibID achieves an F_1 score of more than 0.5 in all cases while prior work is below 0.25. We also demonstrate the utility of LibID by detecting the use of a vulnerable version of the OkHttp library in nearly 10% of the 3 958 popular apps on the Google Play Store.

Hardware-based device fingerprinting allows apps and websites to invade user privacy by tracking user activity online as the user moves between apps or websites. In particular, we present a new type of device fingerprinting attack, the *factory calibration fingerprinting attack*, that recovers embedded per-device factory calibration data from motion sensors in a smartphone. We investigate the calibration behaviour of each sensor and show that the calibration fingerprint is fast to generate, does not change over time or after a factory reset, and can be obtained without any special user permissions.

We estimate the entropy of the calibration fingerprint and find the fingerprint is very likely to be globally unique for iOS devices (~67 bits of entropy for iPhone 6S) and recent Google Pixel devices (~57 bits of entropy for Pixel 4/4 XL). By comparison, the fingerprint generated by previous work has at most 13 bits of entropy. Following our disclosures, Apple deployed a fix in iOS 12.2 and Google in Android 11.

Both code obfuscation and factory calibration help to hide software and hardware idiosyncrasies from third-parties, but this dissertation demonstrates that reliable software and hardware fingerprints can still be generated given sufficient knowledge and a suitable approach. Our work has significant practical implications and can be used to improve platform security and protect user privacy.

Acknowledgements

First and foremost I would like to thank my supervisor, Alastair Beresford, for his excellent advice, prompt discussion, and unreserved support. He has not only helped me choose my research topic but also has unhesitatingly given me all the equipment needed for my research. He has provided me with various collaboration opportunities that I would not have otherwise. He is not only an exceptional supervisor but also a great friend in life. This work would not have been possible without his continuous encouragement.

Furthermore, I would like to thank my collaborators that made this dissertation possible. The device fingerprinting work of this dissertation has grown from my close collaboration with Ian Sheret. It is his expertise and experience on motion sensors that bootstrapped the device fingerprinting work. I would also like to thank Stephan Kollmann for his contributions to the library fingerprinting work of this dissertation. Special thanks to Andrew Rice and Amanda Prorok for their valuable feedback on my annual report.

In addition, I would like to thank my collaborators and friends during my internship at Brave, especially Iñigo Querejeta Azurmendi, Antonio Nappa, Panagiotis Papadopoulos, Alexander Sjösten, Gonçalo Pestana, Antonio Pastor, and Mohammad Malekzadeh, for their emotional, intellectual, and technical support during and after the internship.

I also want to express my enormous gratitude to my close friends who have enriched my life and provided unconditional support during the course of my PhD, especially Matthew Hall, Stephan Kollmann, Ricardo Mendes, Evelyn Zhou, Alfredo Mazzinghi, Michael Dodson, Diana Vasile, Martin Kleppmann, and Mickey Whitzer. They kept me going during tough times and made my life in Cambridge so enjoyable. I am blessed to have their company throughout this journey. I am also thankful to my parents for their continuous support and to all members of the Digital Technology Group for interesting conversations and fun times. Last but not least, I am very grateful to the China Scholarship Council, Computer Laboratory, and Robinson College for generously funding my PhD.

Contents

CHAPTER 1

Introduction

Mobile devices, especially smartphones and tablets, have gained considerable popularity due to their portability, user experience, and rich functionality; around 92% of Internet users globally were mobile users and 53.3% of web traffic originated from mobile phones in December 2019 [1]. The ever-increasing popularity of mobile devices has seen significant investment in mobile advertising: more than 190 billion US dollars were invested in mobile advertising in 2019, and spend is expected to surpass 280 billion US dollars by 2022 [2]. Payments from advertisers have encouraged more developers to provide apps without up-front fees to their users and instead receive compensation through ad revenues; around 96.4% of apps on Google Play and 91.9% of apps on Apple App Store were free as of June 2020 [3]. A survey of 400 mobile app developers in the US and the UK shows that approximately 47% used mobile advertising to monetise their apps as of June 2019 [4].

To harvest behavioural user data, advertising companies typically provide a Software Development Kit (SDK) for publishers, such as app and website developers, to use their service. App developers then integrate this SDK into their apps as a third-party library so that the app obtains fresh advertisements and presents them to users. As a result, advertisers get to run their (the third-party) library code on user devices and in return app developers are paid for ad impressions and clicks. Research has found that behavioural targeting can greatly improve sales for advertisers. A survey of 3.3 million responses revealed that by complying with the EU privacy law, which restricted advertisers' ability to use personal data for targeted advertising, banner ads had seen a decline in purchase intent of over 65% [5]. Because behavioural ads offer increased utility for the advertiser and consequently higher revenues to developers, the advertising ecosystem has moved towards behavioural ads at the expense of user privacy.

In addition to advertising libraries, app developers also use third-party libraries to: promote their apps with social media support, facilitate app development, and extend app functionality. A previous study has shown that, on average, over 60% of sub-packages in

Android apps are from common libraries [6]. Nevertheless, the prevalence of third-party libraries has also brought new challenges that may affect the security and privacy of the mobile platform (§1.1). These challenges motivate the development of reliable *library fingerprinting* techniques to identify third-party libraries. *Library fingerprinting* is a particular type of *software fingerprinting* that aims to identify the software components on mobile devices. It has a variety of use cases, such as studying the popularity of a library of interest, excluding third-party libraries from program analysis, identifying the use of vulnerable libraries in apps, etc.

In addition to software fingerprinting, *hardware fingerprinting*, especially *device fingerprinting*, also has significant implications for the privacy of the mobile platform. In order to grow their business and support behavioural ads, mobile advertisers are incentivised to identify potential customers. For this purpose, they often seek a reliable way to generate a distinctive device signature, or *device fingerprint*, to identify user devices. *Device fingerprinting* is the process of generating such a fingerprint and is based on one or more unique features of a device. With a reliable fingerprint, advertisers can track users online and offline, study their behaviour, and deliver tailored content to the users. However, it also raises privacy concerns since advertisers may learn sensitive information about the customers and share it with third parties without users knowing. Therefore, both Android and iOS have included measures to prevent such tracking (§1.2). Device fingerprints can also support other use cases. For example, it can help banking apps and websites verify the identity of visiting devices to protect against identity theft and credit card fraud.

Both library and device fingerprinting are challenging topics because there have been extensive efforts to either block such fingerprinting or hide the idiosyncrasies of these software and hardware modules. However, in this dissertation we show that it is difficult to completely remove unique software and hardware fingerprints despite current best practice. Given sufficient knowledge innovation, both software and hardware components can still be effectively fingerprinted.

In particular, we study the characteristics of common code obfuscation techniques that are often used to thwart reverse engineering. We find these techniques reduce distinctive software features that are important for reliable library fingerprinting. To address this issue, we design a novel library fingerprinting tool that is resilient against these obfuscation techniques (§1.1). In addition, we look into the factory calibration in embedded motion sensors that is intended to reduce the difference between actual and ideal sensor output given the same input. We then present a novel device fingerprinting attack that bypasses existing tracking protection enforced in iOS and Android by exploiting factory calibration patterns (§1.2). Note that, this dissertation focuses on novel techniques to identify software and hardware components (i.e., *third-party Android libraries* and *embeded motion sensors*) instead of on device fingerprinting exclusively. Although library

fingerprinting techniques can also help to identify a smartphone, it is out of the scope of this dissertation. The following sections describe the challenges of both topics and summarise our research questions, threat models, and solutions. Overall, our work has significant practical implications and makes important contributions to protecting user privacy and improving platform security.

1.1 Android library fingerprinting

There are several challenges in identifying a third-party Android library within an app. Firstly, many apps use code obfuscators, such as ProGuard[1], to obfuscate the name of classes, fields, and methods; approaches based on simple identifier matching are not applicable for such apps [7, 8]. Secondly, many detection schemes tried to extract library candidates at scale through cluster-based approaches [6, 9, 10]. Although these schemes require no prior knowledge about the libraries before clustering, they require significant effort to label each cluster later. In addition, due to the lack of detailed information in libraries, clustering-based schemes cannot determine the version of a library inside an app. Thus, they cannot be used to study whether apps use a vulnerable version of a library. Lastly, library code can be dynamically changed during the build process by code obfuscators. For instance, ProGuard, an open-source tool that has been integrated into the Android build system, will, by default, optimise library code and remove unused code during the build process. This process is called *code shrinking* or *dead-code elimination*. In addition, ProGuard can apply package modification operations, which includes *package flattening* and *class repackaging*, to obfuscate packages and change class hierarchies. A recent study has shown that 49% of apps (88% of obfuscated apps) used ProGuard and 21% of them applied package modification operations [11]. Most of the existing studies (e.g., LibRadar [9] and LibD [10]) failed to consider code shrinking and package modification operations, while others that did consider them (e.g., LibScout [12] and Orlis [13]) showed poor accuracy in our experiments.

Our work on library fingerprinting aims to answer the following research questions: *How to reliably identify third-party Android libraries when a large portion of library code is deleted and when package modifications are applied? How does our library fingerprinting solution compare with existing studies? How common do popular Android apps use known vulnerable libraries?*

We use the same threat model as in other library fingerprinting studies [9, 10, 12]. We assume the adversary is able to collect the app binary they want to analyse. The adversary can either download app binaries from an Android app distribution platform such as Google Play or extract the binary of installed apps from a smartphone. The latter can be

[1] `https://www.guardsquare.com/en/products/ProGuard`

achieved if the adversary has physical access to the smartphone or if an app installed on the smartphone is under the control of the adversary and the app can communicate with a remote server. We also assume that the adversary has a collection of library binaries of interest. Furthermore, we assume the adversary has a reasonable amount of computing resources such as a desktop server to run the library fingerprinting algorithm. The goal of the adversary is to accurately identify third-party libraries, including their name and version, used in an Android app, even in the presence of code obfuscation.

This dissertation presents a novel third-party Android library detection tool, LibID, that generates library fingerprints that are resilient to common code obfuscation techniques, including identifier renaming, code shrinking, control-flow randomisation, and package modification. By matching these fingerprints, LibID can reliably identify the library version used in Android apps given the library and app binaries. Our evaluation of LibID is supported by a novel method that semi-automatically generates apps containing third-party libraries, as well as an analysis of hundreds of F-Droid apps. This provides valuable ground truth data to support accurate evaluation and comparison of our approach to previous work. Our experiments show that LibID can detect a much higher percentage of libraries than prior work, especially when code shrinking is enabled and the package hierarchy is modified.

LibID can be of great assistance to Android researchers. For example, program analysers can rely on LibID to remove third-party libraries and consequently reduce the overhead of static analysis. Library developers can apply LibID to gain information about the usage of each library version in real-world apps, and notify app developers when certain versions of libraries are found to be vulnerable. LibID can also be used to check if there are license violation issues in the use of third-party libraries. In addition, security researchers can build their application based on LibID to investigate privacy and security threats of third-party libraries. We hope this work can raise awareness about the importance of third-party libraries and library-centred studies.

1.2 Device fingerprinting

Realising the potential risk to user privacy, both iOS and Android have included measures to prevent device fingerprinting. On iOS, developers do not have access to the UDID (Unique Device IDentifier), IMEI (International Mobile Equipment Identity), and MAC address of hardware modules after iOS 7. Similar restrictions are also deployed on Android O; developers cannot access non-resettable unique hardware identifiers even if users grant the app dangerous permissions such as the `READ_PHONE_STATE` permission. Most recently, Apple announced that apps running in iOS 14 will be required to ask users for permission to track them across apps and websites owned by other companies [14]. While it is still

possible to track users by the advertising identifier on both iOS and Android, this method comes with several drawbacks. First, both platforms allow users to reset this identifier at any time; iOS also provides an option to limit access to this identifier. Moreover, apps that request this identifier but do not serve any in-app advertisements will be rejected by the App Store. Last but not least, the advertising identifier is not accessible from mobile browsers. Although Android still allows apps to access the `ANDROID_ID`, it cannot be obtained from a website and its value is scoped by the app signing key and user since Android 8. Different apps on a device will have different values of `ANDROID_ID` and a factory reset or an APK signing key change may also change its value [15]. Thus, both the advertising identifier and the `ANDROID_ID` cannot be used to track users across apps and websites.

Despite the privacy policies enforced by Apple and Google, it is still possible to exploit the uniqueness of embedded sensors to fingerprint a device. Most mobile devices nowadays are shipped with a variety of embedded sensors (e.g., accelerometer, gyroscope and magnetometer). Mobile apps rely on these sensors to provide rich functionality such as workout tracking and better gaming performance. However, natural variation during the manufacture of embedded sensors means that the output of each sensor is unique and therefore they may be exploited to create a device fingerprint.

Previous studies applied machine learning techniques directly to sensor data in an attempt to create device fingerprints for smartphones [16, 17]. This approach has several drawbacks. First, these models are susceptible to environmental conditions, such as temperature and noise, so accuracy decreases over time. Second, they either require the smartphone to be placed in a stationary position (e.g., on a desk) or have relatively low accuracy when devices are moved. When devices were held in a hand, state-of-the-art work by Das *et al.* achieved an accuracy of around 60% in an open-world setting with 86 iPhone 6 devices, which is equivalent to 13 bits of entropy (§2.3.3) [17].

In our work, we take a different approach. Instead of feeding sensor outputs into machine learning algorithms, we infer the per-device factory calibration data from the output of sensors such as the gyroscope, accelerometer, and magnetometer. This calibration data can then be used to construct a unique device fingerprint. We call this new type of attack a *factory calibration fingerprinting attack*.

We aim to answer the following research questions: *How to practically recover the factory calibration parameters of sensors from their output? What smartphone models are vulnerable to the factory calibration fingerprinting attack? How unique are the calibration parameters and how does it compare with existing motion sensor-based device fingerprinting studies?*

Our threat model is as follows. We assume the adversary is able to record motion sensor samples from a smartphone. The attacker can do this if the user installs an app,

or visits a website (accelerometer and gyroscope only), under the control of the attacker. Furthermore, we assume that the software embedded in the app or web page is able to communicate with a remote server under the control of the attacker; this is typically the case for both apps and web pages. The goal of the adversary is to obtain a reliable device fingerprint of a smartphone.

This dissertation describes this new type of attack and demonstrates its effectiveness on motion sensors (accelerometer, gyroscope, and magnetometer) as available in iOS and Android. We choose these sensors because, when we started our work, access to these sensors did not require any special permission, and accelerometer and gyroscope data can be accessed via both a native app installed on a device and via JavaScript when the user visits a website. Overall, our attack has the following advantages:

Practical The attack can be launched by any website or any app on a vulnerable device without requiring explicit confirmation or interaction by the user.

Efficient The attack takes less than one second to generate a fingerprint.

Unique The attack generates a globally unique fingerprint for vulnerable devices.

Robust The calibration fingerprint never changes, even after a factory reset.

Effective The attack provides an effective means to track users as they browse across the web and move between apps on their device.

With our approach, we achieve 67 bits of entropy for the iPhone 6S and 57 bits of entropy for the Pixel 4/4 XL. We followed a coordinated disclosure procedure and reported the vulnerability to Apple on 3rd August 2018 and Google on 10th December 2018. In iOS 12.2, Apple adopted our suggestion and added random noise to sensor outputs (CVE-2019-8541) while Google decided to round sensor outputs to a multiple of nominal gain in Android 11. In addition, Apple removed access to motion sensors from Mobile Safari by default and in later versions also removed motion sensor access from WebKit. However, we show that Apple's fix is imperfect since a calibration fingerprint can be extracted if more sensor data is available.

1.3 Dissertation outline

Chapter 2 This chapter introduces the necessary background for later chapters. It starts by looking broadly on fingerprinting topics on mobile devices. It then summarises existing studies on third-party library fingerprinting and related topics such as app clone detection. It also reviews the evolution of device fingerprinting techniques and outlines related literature in this area. In addition, it gives a brief introduction to several techniques we used in this dissertation.

Chapter 3 This chapter details the design of our third-party Android library fingerprinting system (LibID), including a scalability-focused design (LibID-S) and an accuracy-focused design (LibID-A). It presents a systematic approach to generating a synthetic app dataset that we use to tune several parameters used in LibID. Then, it applies LibID and several other state-of-the-art library detectors to open-source apps on F-Droid and benchmarks their performance. Finally, it conducts a large-scale study on popular Google Play apps and identifies a high proportion of popular apps using vulnerable libraries.

Chapter 4 This chapter describes the theory of our factory calibration fingerprinting attack. It introduces the background of calibrating motion sensors. It also explains the steps to identify whether a sensor is factory calibrated and to calculate the nominal gain of a sensor if the datasheet is unavailable. Using the gyroscope in iOS devices as an example, it analyses the data representation of sensor outputs and discusses the difference in the sensor outputs obtained from a mobile app and a website. It then elaborates on the factory calibration fingerprinting attack in each case.

Chapter 5 This chapter presents the large-scale analysis results on the feasibility of factory calibration fingerprinting attack on iOS and Android devices. It evaluates the performance of the factory calibration fingerprinting attack and demonstrates the attack is practical, efficient, unique, and robust. It further quantifies the entropy of the generated device fingerprint and reports our vulnerability disclosure practice. Last but not least, it includes our analysis of Apple's fix and introduces a novel attack against the fix.

Chapter 6 This chapter concludes the dissertation by summarising the key results and discusses some directions for future work.

1.4 Publications

Below is a list of publications I co-authored during the course of my PhD.

1. Jiexin Zhang, Alastair R Beresford, and Stephan A Kollmann. "LibID: Reliable Identification of Obfuscated Third-Party Android Libraries." In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA),* pp. 55–65. ACM, 2019.

   This publication forms the basis of Chapter 3 of this dissertation. The original idea emerged from discussions with Alastair. The experiments were designed by myself in collaboration with Alastair. I wrote all the code of LibID independently except

for the part which relates to binary integer programming, which was written in collaboration with Stephan. I executed the experiments, collected and analysed the data, and evaluated the results. I thank Alastair and Stephan for help with writing and Diana A. Vasile, Ricardo Mendes, Martin Kleppmann, and Peidong Zhu for helpful discussion and insight. I also thank anonymous reviewers for their feedback on the paper.

2. Jiexin Zhang, Alastair R Beresford, and Ian Sheret. "SensorID: Sensor Calibration Fingerprinting for Smartphones." In *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP),* pp. 638–655. IEEE, 2019.

   This publication forms the basis of Chapter 4 of this dissertation. The original idea emerged from discussions with Ian and Alastair. The experiments were designed by myself in collaboration with Alastair. I implemented the necessary code for the instrumentation of data collection and analysis. I executed the experiments, collected and analysed the data, and evaluated the results. I thank Alastair and Ian for their help with writing and Stephan A. Kollmann, Diana A. Vasile, Ricardo Mendes, Andrew Rice, and Amanda Prorok for helpful discussion and insight. We also thank our shepherd Adam Bates and anonymous reviewers for their feedback on the paper.

3. Jiexin Zhang, Alastair R Beresford, and Ian Sheret. "Factory Calibration Fingerprinting of Sensors." *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 16, pp. 1626–1639, 2021.

   This publication forms the basis of Chapter 5 of this dissertation. This paper extends our previous work [18]. In particular, we presented our latest findings on accelerometer calibration on iOS devices (§5.2); we conducted a large-scale factory calibration behaviour analysis on popular Android device models (§5.3); we compared the calibration fingerprint with a popular browser fingerprint for Google Pixel phones (§5.4.2); we analysed the calibration fingerprint for vulnerable Pixel devices and estimated entropy for each model (§5.5.2). Last but not least, we analysed Apple's fix and showed that it is imperfect since a calibration fingerprint can be extracted if more sensor data is available (§5.6). The idea came from discussions with Alastair and Ian. I designed and executed the experiments, and collected and analysed the data for the evaluation. I thank Alastair and Ian for their help with writing and Matthew Hall for proofreading.

CHAPTER 2

Background and related work

This chapter gives the necessary background to help better understand the topics of this dissertation, their current research progress, and methodologies we apply to solve the challenges. Specifically, we first look broadly on fingerprinting topics on mobile devices (§2.1). Then, we focus on two types of fingerprinting that are most pertinent to this dissertation: *library fingerprinting* and *device fingerprinting*. We summarise existing work in these areas and present it in §2.2 and §2.3. Furthermore, we give a brief introduction to several techniques used in this dissertation, including *locality-sensitive hashing* (§2.4), *integer programming* (§2.5), and *simulated annealing* (§2.6). Last but not least, we summarise this chapter in §2.7.

2.1 Fingerprinting on mobile devices

Ever since the arrival of smartphones, mobile devices have drastically reduced the need to have a dedicated task-specific device such as a camera, microphone, or speaker. Due to their convenience, people have been using smartphones to record their life, process bank transactions, and handle private communications. Because smartphones have effectively become a central hub of sensitive user data, a great deal of effort in both industry and academia has been devoted to secure the platform and protect user privacy.

In particular, biometric-based authentication, or *biometric fingerprinting*, has drawn an extensive amount of interest because it offers a quicker and more convenient alternative to traditional password-based authentication. Biometric fingerprinting schemes verify the identity of users by their unique physiological characteristics or behavioural traits. By far, the most used physiological features include:

Finger It is generally believed that different fingers have distinct papillary ridge patterns, or *fingerprints*, and therefore they can be used as a person's identity [19]. Fingerprints were first used in crime scenes and forensic science as evidence [20]. Since Apple

introduced Touch ID in 2013, smartphones are increasingly shipped with a fingerprint sensor to allow fast user authentication. Nevertheless, researchers have found that fingerprint sensors can be fooled with a high success rate using the victim's fingerprint mould [21]. It has also been reported that such a mould can be made from just a few photos of the victim's hand taken with a standard camera [22, 23]. In addition, researchers have proposed to use a smartphone camera to capture the changes in the light reflection of the finger skin to identify users [24]. Since the light reflection is associated with the blood flows through the finger, the captured signal reveals unique features of an individual's cardiovascular systems.

Palm Similar to fingerprints, palm prints also have distinctive ridge patterns and thus can be used to authenticate users. Some palm scanners make use of an infrared camera to observe the palm veins to further reduce false positives. In particular, LG G8 ThinQ includes a palm recognition system which they call "Hand ID" that authenticates users by the thickness, shape, and other characteristics of their palm veins as well as unique features of their palm prints [25]. According to LG, two people have less than one in a billion chance of sharing identical palm vein patterns [26].

Face Facial recognition is another type of biometric authentication technique that has been widely deployed in modern smartphones. Although the implementation details may vary, mobile facial recognition systems generally use front-facing cameras in a smartphone to build a geometric profile of a user's face, which may include the placement of eyes, the width of the nose, the size of the face, etc. In particular, Apple's Face ID uses a TrueDepth camera system that projects invisible infrared dots onto a user's face and then takes an infrared image of it [27]. This allows Face ID to build a depth map of a user's face even in the dark. Apple has claimed that Face ID has significantly fewer false positives than Touch ID (1 in 1 000 000 vs 1 in 50 000) [28]. Nevertheless, researchers have found that it can be tricked to authenticate sleeping users by placing a pair of specially made glasses on them [29].

Iris The complex patterns of a person's iris are believed to be unique and do not change over time. Similar to Apple's Face ID, iris recognition systems deployed in a smartphone typically use a special front-facing camera to take infrared images of the iris. Infrared light is used because it is invisible and helps to unveil patterns of dark-coloured eyes. Samsung included an iris scanner in Galaxy S8, S8+, S9, S9+ and Note9. However, it turns out Samsung's iris scanner can be easily bypassed. Attackers only need to print a photo of the victim's iris and put a pair of contact lens on top of the photo to fool the scanner [30]. Samsung stopped using the iris scanner since the release of Galaxy S10.

Ear canal With in-ear wearables gaining popularity, fingerprinting users by the shape of their ear canal has drawn research interest. The general approach of ear canal fingerprinting is to play a probe audio clip, which is often inaudible to the user, via an inward-facing speaker in the earphone and later analyse the response signal captured by a built-in microphone. As the sound wave is reflected inside the ear canal, the response signal reveals the static geometry of the ear canal [31]. A few studies have found the dimensions of the ear canal to be unique for each individual [32, 33]. However, the ear canal geometry estimated from the reflected signal is not perfect. In particular, EarEcho uses this design to continuously authenticate users and it reported 97.55% recall and 97.57% precision in their study with 20 participants [34]. Moreover, other studies have pointed out that head and mouth movements could lead to dynamic changes in the ear canal geometry and thus it is not a stable fingerprint [35].

Apart from aforementioned traits, other types of physiological biometrics such as thermogram and body odour can also be used to verify users' identity. However, they are rarely used in mobile devices and thus we do not elaborate on them here. Behavioural biometrics such as voice [36], key stroke [37], gait [38], eye movement [39, 40], and many others have also been explored for the purpose of fingerprinting. Unlike physiological biometrics, behavioural biometrics check for characteristic behavioural patterns that are likely to change over time. In particular, voice-based speaker recognition that exploits the differences in time and frequency domain acoustic patterns to identify users have attracted a great deal of attention due to the gaining popularity of voice assistants. The difference in voices is a result of both physiological variations (people have differently shaped voice tracts) and behavioural factors (people pronounce words in different ways). However, studies have shown that voice-based authentication schemes are susceptible to replay attacks [41, 42], text-to-speech attacks [43, 44], and phoneme morphing attacks [45].

Both physiological and behavioural biometrics have raised privacy concerns. Biometrics collected for a specific purpose such as authentication may later be used for other purposes or in another context (e.g., forensic investigation). In addition, they may leak unintended information about the user because they represent a rich feature set. For example, biometrics such as iris and palm vein patterns can reveal information about the gender, ethnicity, and health of the user [46, 47]. It is also difficult to change or revoke biometrics. Since users may use the same biometric in different applications, biometric data theft in one application could lead to unauthorised access in others.

Biometric fingerprinting is likely the best-known type of fingerprinting on mobile devices, but other types of mobile fingerprinting topics are also of great importance. Instead of fingerprinting the user, *software fingerprinting* aims to identify the software running on a device, while *hardware fingerprinting* aims to recognise unique hardware modules in a

device. This dissertation explores both software and hardware fingerprinting techniques. In particular, we study a type of software fingerprinting called *library fingerprinting* that seeks to identify third-party libraries including their versions used in an app. In terms of hardware fingerprinting, we present a novel type of *device fingerprinting* that can reliably and uniquely identify individual devices. We describe the background and related work of library fingerprinting and device fingerprinting in detail in §2.2 and §2.3, respectively.

2.2 Library fingerprinting

Library fingerprinting aims to generate a distinctive set of library signatures, or *library fingerprints*, to identify software libraries. Because software libraries are often compiled into binaries, fingerprinting and identifying them requires program analysis techniques. Since our library fingerprinting work targets the Android platform, we give a brief introduction to Android program analysis in §2.2.1.

The extensive use of third-party libraries without appropriate consideration of security issues can put mobile devices and networks at risk of cyberattacks and privacy leakages. If one common library contains severe security concerns, a great number of apps can be compromised. For example, the Facebook Android SDK version 3.15 [48] and certain versions of the `OkHttp` library [49], both of which were very popular in Android apps, contained authentication vulnerabilities. In addition, several other popular libraries spy on SMS messages [50] or even establish backdoors [51]. Moreover, attackers may also inject malicious code into popular libraries and redistribute them through unofficial platforms to remotely control smartphones or steal private information. In fact, according to recent research, more than half of all Android apps with vulnerabilities were at risk due to libraries [52]. There are also academic publications focusing on this problem; we describe them in §2.2.2. A well-related research area to this problem is *app fingerprinting*. App fingerprinting techniques typically aim to identity apps running on a remote device by studying their traffic. It can be combined with library fingerprinting techniques for advanced targeting. For example, attackers can first apply library fingerprinting to identify apps with vulnerable libraries and then launch app fingerprinting to find devices running these vulnerable apps. We briefly summarise studies on app fingerprinting in §2.2.3.

Third-party libraries have also introduced a few novel challenges to mobile platforms. In particular, the presence of libraries constitutes a barrier for many areas of mobile app analysis. App clone detection schemes should not take library code into account during analysis. Since a large proportion of program code is contributed by libraries, which may well be used in many other apps without modification, the accuracy of detection schemes will be significantly skewed if library code is not properly excluded. Static analysis is the basis for many mobile security studies, but it often consumes significant computing power

and resources required [53, 54]. Since libraries are often irrelevant to the main functionality of an app, it is better to separate them from program analysis to improve efficiency [55]. In addition, many developers use third-party libraries without complying with the license terms. In particular, Duan *et al.* found 40K Android apps on the Google Play store potentially violated GPL (General Public License) licensing terms [56]. Similarly, Tang *et al.* reported 18 out of 50 top applications listed in Tencent software centre violated GPL terms [57]. The misuse of licensed components should be identified to protect library developers' copyright. Library fingerprinting is the basis of license violation detection and library code exclusion. A few library fingerprinting techniques have been proposed to help achieve these goals; we summarise these studies in Chapter 3 to better compare them with our library fingerprinting approach.

2.2.1 Android program analysis

Program analysis can be either *static*, through the inspection of the source code or binary, or *dynamic*, through the execution of the program. The former often provides more code coverage, while the latter can capture dynamic behaviour that is hard to detect via static analysis. *Hybrid analysis* exploits results from both static and dynamic analysis. In the context of Android apps and libraries, we briefly introduce related research in each type.

Static analysis Static analysis approaches usually apply reverse engineering and code audit techniques to reconstruct application behaviours and to find hidden malicious code. Egele *et al.* designed a tool named PiOS, which can detect data flows in binaries compiled from Objective-C code by static analysis, to identify private data leaks from iOS devices [58]. For Android applications, Arzt *et al.* developed a static taint analysis system called FlowDroid to find potential privacy leakages, related to Android-specific callback and UI events [53]. However, these static analysis schemes cannot detect self-updating malware where malicious code is injected into the application at runtime.

Dynamic analysis In contrast, dynamic analysis schemes identify possible malicious acts of an application by recording and analysing its behaviour in a controlled run-time environment. Enck *et al.* proposed an extended Android system, named TaintDroid, to detect privacy leakages via dynamic taint tracking [59]. VetDroid [60] further improved TaintDroid by conducting permission use analysis on sensitive operations. Nevertheless, both TaintDroid and VetDroid cannot reveal under what conditions these internal sensitive actions are triggered.

Hybrid analysis Hybrid analysis is gaining more popularity due to the limitations of static and dynamic analysis alone. To detect malware that uses dynamic code update

techniques to defuse static analysers, Bodden *et al.* designed a hybrid analyser, TamiFlex, to reveal this hidden behaviour [61]. TamiFlex logs class loading and reflection call events at runtime and feeds the results to its static analyser to achieve better performance. However, TamiFlex relies on debug information that could be removed when code obfuscation is applied. To address this issue, Zhauniarovich *et al.* proposed a tool called StaDynA that modifies the DVM (Dalvik Virtual Machine) and `libcore` components to obtain required information [62].

2.2.2 Security and privacy of libraries

Many studies have been dedicated to investigating the security and privacy of third-party libraries. For example, Paturi *et al.* and Son *et al.* studied the privacy disclosure threats of advertising libraries [63, 64]. Wang *et al.* and Sounthiraraj *et al.* analysed several authentication vulnerabilities in third-party libraries [65, 66]. In addition, the framework FlexDroid was proposed to isolate the permissions of third-party libraries [67]. These studies also showed that a large number of libraries contained dynamic code execution methods. Chen *et al.* investigated the use of potentially harmful libraries (PhaLibs) in Android apps [68]. They found 117 PhaLibs with 1 008 variations were used in 6.84% of 1.3 million Google Play apps. They then studied the security of iOS libraries by comparing the similarity between Android libraries and their iOS counterparts. As a result, they discovered 23 PhaLibs with 706 variations on iOS. These PhaLibs can record video and audio without users' awareness, access the keychain of their host app, and prompt users to send a message or make a phone call. Recently, Chitkara designed a privacy management system to control what data can be accessed by third-party libraries [69]. We believe our work in library fingerprinting can be of assistance to these library-related studies due to its robustness and high precision in pinpointing library versions.

2.2.3 Mobile app fingerprinting

The knowledge of what apps are installed on users' smartphone can not only leak sensitive user information such as sexual orientation and religion but also expose weak targets that are vulnerable to known vulnerabilities. However, studies have shown that it is possible to identify mobile apps from their traffic. In particular, Dai *et al.* proposed a technique to automatically generate a network profile of Android apps that can be used for app fingerprinting [70]. The basic idea of their work is to use UI fuzzing to traverse multiple execution paths of an Android app in a simulator and extract an app fingerprint from the captured network behaviour. However, their technique only works for unencrypted HTTP traffic and the evaluation is flawed due to the lack of ground truth. Wang *et al.* showed that an attacker can determine users' app usage over an encrypted wireless local area

network (IEEE 802.11 WLANs) without knowing the encryption key [71]. They found that the side-channel information in packets has unique patterns that can reveal the app being used. Nevertheless, they only evaluated their approach on a small sample size of 13 apps and its performance on a larger dataset is unverified. Taylor *et al.* developed a real-time mobile app fingerprinting framework called AppScanner [72]. AppScanner uses supervised learning classifiers including Random Forest (RF) and Support Vector Machine (SVM) to identify Android apps from their HTTPS/TLS traffic. AppScanner is also highly scalable because it relies only on side-channel features such as the timing and size of packets, the destination IP addresses, and connection ports. AppScanner achieved over 99% accuracy when tested on 110 popular Android apps. Taylor *et al.* later extended AppScanner by improving its handling of common app traffic and its evaluation of the robustness of the generated app fingerprint [73]. They found that AppScanner was able to identify these 110 apps after six months with up to 96% accuracy and its performance is largely invariant across devices and app versions. As an improvement, Aceto *et al.* proposed a multi-classifier system to improve the performance of existing mobile app fingerprinting work [74]. They showed that the careful combination of multiple classifiers can provide over 9% boost in recall on the best base classifier. Recently, they presented a systematic framework to compare several deep learning-based mobile app fingerprinting approaches [75]. They concluded that these approaches have not yet reached a maturity level but deep learning-based solutions look promising.

2.3 Device fingerprinting

Device fingerprinting aims to generate a distinctive device signature, or *device fingerprint*, to identify a device. In general, a distinctive device fingerprint can be generated based on either its software or hardware components. These two categories are not exclusive. That is to say, a device fingerprinting technique can be both software- and hardware-based.

In this section we review existing studies on device fingerprinting. In particular, we first summarise traditional device identifiers in §2.3.1. Then, we discuss software- and hardware-based fingerprinting in §2.3.2 and §2.3.3, respectively.

2.3.1 Traditional device identifiers

Device fingerprinting is an important means for app developers and advertisers to track their users. Traditional fingerprint techniques usually adopt the following identifiers:

IP address IP address is one of the earliest identifiers used to fingerprint networked devices. However, the adoption of dynamic IP allocations and Network Address

Translation (NAT), particularly for home PCs and mobile devices, has greatly reduced the effectiveness of this approach.

MAC address App developers have also used the Media Access Control (MAC) address of embedded hardware modules, such as the Wi-Fi and Bluetooth network interfaces, to fingerprint users. Compared with the IP address, a MAC address is globally unique, does not change between networks and is difficult to change by users without replacing the hardware module, enabling straightforward identification of any device. Realising the privacy implications, devices have employed MAC address randomisation, a technique that rotates through random addresses to defeat device fingerprinting [76]. In addition, both Android and iOS would return a constant value of `02:00:00:00:00:00` when developers access a MAC address in recent versions.

Cookie Cookies are one of the most popular types of device identifiers to track users across websites. However, cookies are stored locally and can be changed by users at any time. In fact, many privacy-focused browsers, such as Safari and Brave, block all third-party cookies by default. Evercookie is a technique to improve the persistence of traditional cookies by exploiting multiple types of storage mechanisms to store and respawn cookies [77]. Nevertheless, a recent study shows that evercookie is no longer reliable in modern browsers and is not persistent in private browsing mode [78]. In addition, privacy laws such as the General Data Protection Regulation (GDPR) require websites to obtain explicit user permission before using non-essential cookies, which also decreased the usability of this approach.

Advertising ID An advertising ID is provided by both Android and iOS for ad-serving purposes. Even though the advertising ID is unique, it is user-resettable and its access can be limited. In addition, linking this identifier with any personally-identifiable information would violate the Google Play Developer Content Policy [79]. With iOS 14, iOS developers need to obtain explicit permission from users to allow access to this identifier [80].

Other ID There are a variety of other IDs in a smartphone that could be used to fingerprint devices. A study in 2011 showed that the IMEI (International Mobile Equipment Identity), IMSI (International Mobile Subscriber Identity) and ICC-ID (Integrated Circuit Card IDentifier) number were widely collected in mobile apps [81]. However, as mentioned in the Introduction, both Apple and Google have adopted stringent privacy policies to prevent developers from obtaining these unique IDs. Even though developers can still access the `ANDROID_ID` from an Android app, it cannot be used to track users across apps and there is no JavaScript API in the browser, preventing access by websites. In addition, many information flow tracking

systems, such as TaintDroid [59] and Panorama [82], can capture these malicious behaviours.

2.3.2 Software-based device fingerprinting

In general, device fingerprinting techniques can be either *active* or *passive*; we review existing research on device fingerprinting by this classification.

2.3.2.1 Passive fingerprinting

Passive device fingerprinting is the action of characterising a target device through its network traffic. It does not send any traffic to the device. Instead, it captures data silently and the data is later analysed to reveal patterns, such as the device's software, operating system, or hardware components, to generate a unique fingerprint. Since passive fingerprinting only relies on network traffic, it is compatible with more devices, difficult to discover, and may be able to track users across different browsers.

Many existing passive fingerprinting studies focus on discovering the flaws of communication protocols. For example, Martin *et al.* found a severe flaw in Apple's BLE (Bluetooth Low Energy) Continuity protocol that can be exploited to defeat BLE MAC address randomisation [83]. In addition, they demonstrated that the model and OS version of a device can be accurately inferred by observing Continuity messages. Similarly, Becker *et al.* developed an address-carryover algorithm that can track a device continuously despite MAC address randomisation [84]. In particular, the algorithm made use of a vulnerability in Windows 10, iOS, and macOS devices that exposes identifying tokens in the payload of BLE advertising messages. Sy *et al.* analysed the QUIC protocol for the purpose of user tracking [85]. They discovered several design flaws that allow an adversary who can observe QUIC traffic to track users across multiple sessions; popular browsers, such as Google Chrome, did not protect against such tracking.

A large number of passive fingerprinting techniques rely on machine learning to differentiate and track devices. For instance, Uluagac *et al.* applied ANNs (Artificial Neural Networks) to classify devices based on time-variant behaviour in traffic [86]. Neumann *et al.* evaluated several features extracted from network traffic and found that frame inter-arrival time is the most effective feature for device fingerprinting and it is correlated with hardware status and installed applications [87]. Stöber *et al.* proposed to exploit the characteristic background traffic of all installed apps (e.g., periodic update requests or server synchronisation) to identify a smartphone using SVM [88]. These approaches usually require a lot of computing resources and a large amount of data for training.

2.3.2.2 Active fingerprinting

Active fingerprinting techniques deploy embedded code to actively gather information about a device and use these characteristics to make a distinction between different devices. For example, operating system configurations such as the system version, user groups, and network and flash configurations, may vary between individual devices. A browser may also provide some distinctive information in terms of its user-agent, version, plugins, fonts, and other configurations. In 2010, Panopticlick conducted a large-scale user study to evaluate the efficacy of browser fingerprinting algorithms [89]. Using a rich set of device characteristics that are accessible from websites, they found that 83.6% of 470,161 browsers had unique browser fingerprints.

Canvas fingerprinting is a popular browser fingerprinting technique to identify visitors of a webpage by exploiting the HTML5 canvas element. It was first presented by Mowery and Shacham in 2012 [90]. They found that different systems are likely to produce different pixels when rendering the same text and WebGL scenes to an HTML5 canvas element. The difference is due to variations in software (e.g. browser and driver) and hardware (e.g., GPU) configurations. Because text and scenes can be invisible, adversaries can apply canvas fingerprinting to track users without their knowledge. Acar *et al.* conducted a large-scale analysis on the usage of canvas fingerprinting in 2014 and found that more than 5.5% of the top Top Alexa 100K sites included canvas fingerprinting scripts [91]. However, canvas fingerprinting cannot distinguish devices of the same make and model running the same software (e.g., iPhone 6 with iOS 11.3).

In addition to academic work, there are also industrial efforts dedicated to browser fingerprinting. For example, FingerprintJS[1] is a popular browser fingerprinting library that combines information about a browser, including the user-agent, version, plugins, font, and canvas, to generate a fingerprint. These characteristics are unlikely to be globally unique, but it can be combined with other features from the browser and embedded hardware to increase precision. However, the generated fingerprint is often *transient* because users may adjust the configuration of their system and browsers at any time, and sometimes system and browser updates may also change the fingerprint. In addition, many browser providers have realised the risk of browser fingerprinting and deployed countermeasures to mitigate it. For example, Safari on macOS only exposes generic configuration information and default fonts since macOS Mojave [92]. Brave, by default, blocks many identifying parts of the Canvas, Web Audio, and WebGL APIs to protect against browser fingerprinting [93].

[1] `https://fingerprintjs.com`

2.3.3 Hardware-based device fingerprinting

Hardware fingerprints are generally more persistent than their software counterparts because it is typically difficult to replace embedded hardware. Some embedded hardware, such as motion sensors, can be accessed by both JavaScript running in a web browser and by mobile apps installed on a smartphone and does not require any permission from users. In general, hardware imperfections are inevitable during manufacture, which implies the existence of fingerprints. A variety of hardware modules have been studied for the purpose of fingerprinting, including:

RF module A variety of RF (Radio Frequency) modules have been investigated over the years for the possibility of fingerprinting. Most of the existing work aims to uncover the identity of the source transmitter by analysing received signals [94]. For instance, studies have demonstrated the possibility of fingerprinting several components of a wireless transmitter, including the DAC (Digital-to-Analogue Converter) [95], power amplifier [96], and RF oscillator [97], because the imperfections in the manufacturing process brought small variations in some key parameters of these components. In general, RF signals can be analysed in either the waveform domain or the modulation domain to extract distinctive features [98]. The former is more complex while the later requires knowledge about the modulation scheme. In particular, Brik *et al.* proposed PARADIS, a technique that compares observed signals with the ideal outputs in the modulation domain, to identify transmitter-specific radiometric signatures of signals [98]. They applied PARADIS to make a distinction between over 130 identical NICs (Network Interface Cards) from captured IEEE 802.11 frames and achieved more than 99% accuracy. Most transmitter fingerprinting techniques only work if the configuration of the transmitter remains unchanged. Therefore, they are impractical for fingerprinting cognitive radio devices that can change bandwidth and other transmission configurations dynamically to make better use of available spectrum. To address this challenge, Andrews *et al.* devised a distance metric that is robust to configuration changes and proposed a transfer learning method to fingerprint cognitive radio devices using this metric [99]. Merchant *et al.* demonstrated that applying deep learning, particularly a deep CNN (Convolutional Neural Network), on frequency-corrected signal is an effective approach to fingerprint RF devices in cognitive radio networks [100]. Recently, Gopalakrishnan *et al.* proposed using CNNs with complex weights so that the same model can work across a diverse range of wireless protocols [101]. Nevertheless, fingerprinting RF modules requires the attacker to be in close proximity to the target device and often needs additional equipment. Thus, it may not be the ideal type of fingerprinting technique for identifying smartphones over networks.

Clock Clock skew, which is defined as the ratio between actual and nominal clock frequencies, is a popular feature used to identify remote devices. Kohno *et al.* observed that there is a discernible difference between the clock skew of different devices, even if they have the same model [102]. Then, they proposed a method to measure the clock skew using TCP and ICMP timestamps and demonstrated clock skew is independent of the network environment and is relatively consistent over time. A follow-up study by Murdoch *et al.* showed that temperature change has measurable, predictable impacts on the clock skew [103]. Since temperature is associated with CPU load, a malicious player can exploit clock skew to de-anonymise a Tor hidden service. Sharma *et al.* evaluated the stability of clock skew in a heterogeneous device network [104]. They concluded that the skew estimate is susceptible to the power state, capture duration, and NTP updates. Instead of using TCP or ICMP timestamps, Huang *et al.* proposed fingerprinting the clock by the temporal feature of Bluetooth frequency hopping [105]. Their evaluation suggests that the clock skew is a reliable fingerprint for Bluetooth devices and it cannot be easily forged without a customised baseband. Nakibly *et al.* suggested that the difference between the CPU and GPU clocks can also be used as a device fingerprint [106]. Although the GPU clock is not directly measureable via JavaScript, they showed that it can be estimated by rendering a complex 3D scene in an HTML5 canvas element. More recently, Sanchez-Rola *et al.* reported that a reliable clock-based fingerprint can be generated by timing the code execution on the target machine [107]. In particular, they made use of the pseudorandom generator APIs to construct the target function and implemented a prototype, CryptoFP, in both native language (C/C++) and in JavaScript. Their results suggest that CryptoFP can be used to differentiate between computers with identical hardware and software, albeit it is less effective when implemented in JavaScript due to limits in timing resolution.

Acoustic component Microphones and speakers are pervasive in modern smartphones. Clarkson showed that different loudspeakers typically have distinctive distortion, even when they are of the same make and model [108]. Based on this observation, he presented a method to verify the identity of individual loudspeakers by measuring sound distortion. However, his work failed to uncover idiosyncrasies in microphones. Following this work, Das *et al.* proposed exploiting the combined distortion from both speakers and microphones embedded in smartphones to generate a device fingerprint [109]. In particular, they used the embedded microphone to record audio clips played by the embedded speaker in a smartphone. Then, they analysed commonly used acoustic features extracted from recorded samples and found that Mel-Frequency Cepstral Coefficients (MFCCs) are most effective in fingerprinting smartphones. Their approach was able to achieve a perfect F_1 score when finger-

printing 15 handsets with the same make and model using either k-NN (k-Nearest Neighbours) or GMM (Gaussian Mixture Model) classifiers. Other classifiers such as maximum likelihood and random forest have also been explored in follow-up studies [110, 111]. Nevertheless, their approach is susceptible to background noises. To mitigate this issue, Zhou *et al.* proposed using a carefully crafted inaudible signal with special frequency patterns as the speaker input [112]. Instead of using machine learning to differentiate devices, they estimated the frequency response of speakers and analysed the cross-correlation between these responses as a fingerprint. Although their approach appears to be effective in the lab setting, the generated fingerprint changes over time with the wear of acoustic components. In addition, inaudible sound has the advantage of being stealthy. Arp *et al.* found ultrasonic beacons have been used in several commercial tracking technologies and deployed in stores in two European cities [113]. Similarly, Chen *et al.* exploited the frequency response of the embedded microphone and speaker pair to verify the identity of wireless devices [114]. Recently, Baldini *et al.* applied CNN classifiers to fingerprint the in-device microphone based on recordings of a single tone and showed it is more effective than traditional machine learning algorithms [115]. Nevertheless, accessing the microphone requires explicit user permission and thus it is less practical on smartphone platforms.

Camera A variety of camera artefacts have also been exploited to identify digital cameras. For example, Colour Filter Array (CFA) interpolation brings correlations across adjacent bit planes of images and can be exploited to identify the source camera brand of an image [116, 117]. Lens radial distortion is a common type of camera defect that makes straight lines appear to be curved on the camera sensor. Choi *et al.* proposed an algorithm to estimate the lens radial distortion from images to identify their source camera and achieved 91% accuracy in differentiating three cameras of different models [118]. Vignetting is another type of camera defect that manifests as a brightness reduction at the corners of an image. Bernacki *et al.* attempted to characterise the vignetting defect to detect a difference between camera brands and obtained 72% accuracy in recognising twelve smartphone brands [119].

The most prominent camera fingerprint in digital forensics is the Photo Response Non-Uniformity (PRNU) [120]. The PRNU is a result of manufacturing imperfections and inhomogeneity of silicon wafers. The original algorithm to estimate the PRNU was presented by Lukás *et al.* [121]. It has since been improved by a few further studies [122–124]. Although the PRNU itself is stable to environmental conditions and is likely to be globally unique, the exact value of the PRNU cannot be extracted and the quality of the estimated PRNU is highly dependent on the imaging processing used in the camera. Recently, Ba *et al.* proposed a protocol named ABC to authenticate

smartphones using the PRNU of their built-in camera [125]. According to their study, the PRNU estimated from one photo alone can identify the smartphone camera used to take the photo with high accuracy. However, their study only focuses on two device models with a single camera. It is unclear whether the image fusion process in modern multi-camera smartphones would degrade the performance. To prevent fingerprint forgery attacks, they later proposed a Camera in Motion (CIM) system that challenges users to move their device along a given route while taking burst photos of QR codes [126]. This design is based on the observation that there is a strong correlation between their estimated camera fingerprint and device motion. Deep learning has also been applied to identify the source camera from its images. Although this approach does not require manually finding and extracting distinctive camera features, it requires a large labelled dataset and is computationally expensive to train. In particular, Freire-Obregón *et al.* designed a CNN architecture to infer the camera sensor noise to identify the source camera [127]. Their approach successfully identified the device manufacturer and the exact camera with around 98% and 91% accuracy, respectively, in a dataset consisting of 3 732 images taken from three smartphones of different brands. Nevertheless, accessing the camera or photos would require explicit permission from the user and thus it is less practical.

Motion sensor We discuss the related work on motion sensor fingerprinting in §5.1 to better compare with our work.

There are also fingerprinting techniques targeting other hardware modules. For example, canvas fingerprinting relies on the small variations between GPUs to make a distinction (§2.3.2.2). Olejnik *et al.* found the HTML5 Battery Status API can be exploited to uncover the battery capacity that can be used as a short-term device fingerprint [128]. The mouse wheel event has also been shown to leak information about the type of device used to scroll a webpage and certain characteristics of the device if it is a trackpad [129]. Our device fingerprinting work is orthogonal to these studies and the generated fingerprints can be combined to provide additional entropy. Nevertheless, to the best of our knowledge, our work is the first to fingerprint devices by their factory calibration parameters. This dissertation shows that the factory calibration matrix of motion sensors can be utilised as a reliable fingerprint for iOS devices and vulnerable Android devices. The concept of factory calibration fingerprinting is applicable to a wide range of sensors.

2.4 Locality-sensitive hashing

Locality-Sensitive Hashing (LSH) is an algorithm for quickly approximating the results of the nearest neighbour problem [130]. In other words, LSH aims to find the item in a

given set that is most similar to a given item. The most commonly used index to measure the similarity between items is *Jaccard similarity*. Assume that two items have a set of attributes (or elements) X and Y, respectively. The Jaccard similarity of X and Y is defined below.

Definition 2.1 (Jaccard Similarity)**.** The Jaccard similarity of sets X and Y is defined as:

$$J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

In other words, the Jaccard similarity of two sets is the ratio of the size of their intersection to the size of their union. Although it is possible to find the exact nearest neighbour by calculating the similarity of every combination of items, this simple brute-force strategy is not scalable. In applications where finding the exact nearest neighbour is not always necessary, LSH provides a good approximation and is faster and more scalable. LSH has been widely used in the area of plagiarism detection [131], image search [132], and video fingerprinting [133].

In general, LSH hashes the input items multiple times in such a way that similar items will have a high probability of being hashed to the same value, or *bucket*, while the collision probability is low for inputs that are far apart. Therefore, to find an item in a given set that is most similar to a query item, instead of calculating the similarity of all the input item pairs, we only need to consider those item *candidates* that land in the same bucket as the query item. This drastically reduces the time to search in a large, high-dimensional dataset. However, LSH introduces both *false positives* and *false negatives*; false positives are those dissimilar items that become candidates while false negatives are those similar items that are excluded as candidates. Ideally we would want to reduce the number of both false positives and false negatives but that would increase time cost. Thus, LSH algorithms often allow users to tune the parameters used in LSH hash functions to favour one of these indexes based on the needs of their application.

By way of an example, Figure 2.1 presents the probability of an item becoming a candidate under different parameter settings when using the LSH algorithm described by Rajaraman and Ullman [131]. In practice, many LSH algorithms allow users to choose a threshold for the similarity index in such a way that the chance of items become candidates is high for those that meet the threshold and low for those that do not [134]. This is normally implemented by adjusting the parameters (e.g., b and r in Figure 2.1).

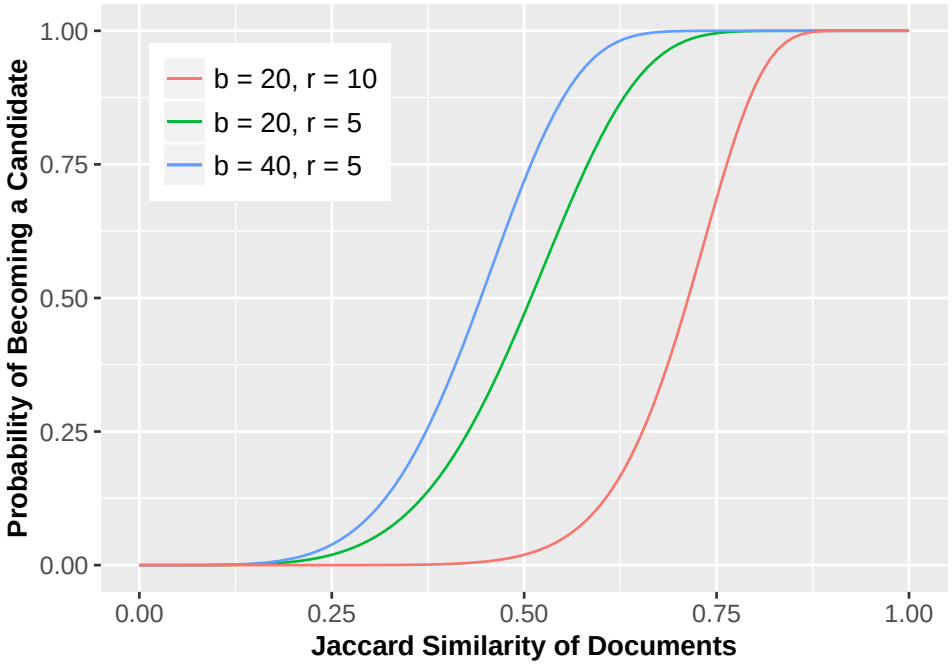


Figure 2.1: Probability of an item becoming a candidate under different parameters

2.4.1 LSH for containment

LSH can also be used for similarity indexes other than Jaccard similarity. In this dissertation we are interested in a particular type of similarity index called the *containment similarity*. The definition of containment similarity is given below.

Definition 2.2 (Containment Similarity)**.** The containment similarity of X in Y is defined as:

$$C(X, Y) = \frac{|X \cap Y|}{|X|}$$

Example 2.1. Suppose we have two sets X and Y as shown in Figure 2.2. The black dots in Figure 2.2 represent set elements. In particular, there are five elements in X and seven elements in Y. Four elements exist in both sets and a total of eight elements that appear in the union of X and Y. Therefore, the Jaccard similarity of X and Y is $J(X, Y) = 1/2$. In addition, we can calculate the containment similarity of X in Y as $C(X, Y) = 4/5$ and the containment similarity of Y in X is $C(Y, X) = 4/7$.

The containment similarity index is a good fit for library fingerprinting. Suppose an app class A is produced from a library class L and we have obtained a feature set X for A and a feature set Y for L. Ostensibly, sets X and Y should be the same since A is generated from L. However, Android projects often use code optimisers, such as ProGuard,

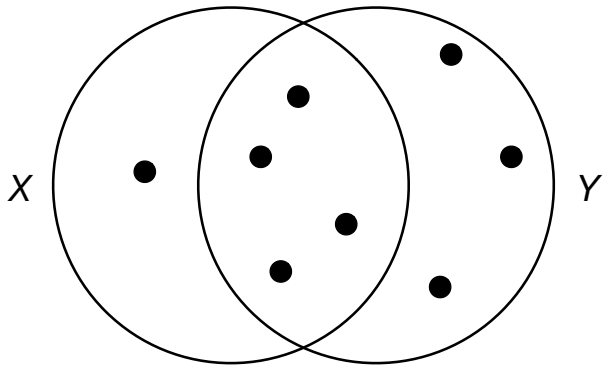


Figure 2.2: Two sets with different number of elements

to eliminate unused code and apply code obfuscation. If features in both X and Y are obfuscation-resilient, then X should be a subset of Y, in which case the containment similarity of X in Y is $C(X, Y) = 1$. By comparison, the Jaccard similarity of X and Y will depend on the intensity of code shrinking. Therefore, we can utilise LSH with the containment similarity index to filter out candidates in an app class set for a library class. In particular, Zhu *et al.* have developed an efficient LSH algorithm for the containment similarity called *LSH Ensemble* [134]. In this dissertation we use *LSH Ensemble* to support our library fingerprinting work; more details are described in Chapter 5.

2.5 Integer programming

An *integer program*, or *linear integer program*, aims to minimise or maximise a linear objective function in which some or all of the variables are restricted to be integers [135]. It often includes equality, inequality, bound, and integer constraints. The canonical form of integer programming can be expressed as:

$$\begin{aligned}
\textbf{maximise} \quad & \sum_{j=1}^{n} c_j x_j \\
\textbf{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad (i = 1, 2, \ldots, m), \\
& x_j \geq 0 \quad (j = 1, 2, \ldots, n), \\
& x_j \text{ is an integer} \quad (\text{for some or all } j = 1, 2, \ldots, n)
\end{aligned}$$

Based on this canonical form, equality and bound constraints can be constructed. For example, we can use the inequality constraints $x_j \geq d$ and $x_j \leq d$ to construct the equality constraint $x_j = d$. Similarly, we can construct the bound constraint $l \leq x_j \leq u$ using two inequality constraints.

Depending on the values the decision variables $\mathbf{x}$ can take, some subcategories of the integer programming problem are defined:

Pure integer program A pure integer program is an integer program in which all decision variables are integers.

Mixed integer program A mixed integer program is an integer program in which some, but not all, decision variables are integers.

Binary integer program A binary integer program, or 0–1 integer program, is an integer program in which all decision variables are binary variables.

In this dissertation we are interested in Binary Integer Programming (BIP) models to express the binary decision about whether a library is used in an app; more details can be found in Chapter 3. Although BIP is one of the Karp's 21 NP-complete problems, a number of sophisticated techniques have been developed in recent years to speed up finding solutions, including:

Branch and bound The solutions of an integer programming problem can be represented as a tree structure. For example, Figure 2.3 shows the tree structure of all solutions of a BIP problem with three variables. The branch and bound algorithm avoids traversing the entire tree by exploring only branches that may produce a better solution than the best one found so far (*branching*). This is achieved by estimating the bounds on the best value that can be acquired by adding a node (*bounding*) [136].

Presolve Presolve is a collection of preprocessing techniques to reduce the size of feasible solutions of an integer programming problem. It is often applied before the branch and bound algorithm and typically involves fixing variables, removing redundant constraints, and minimising zero-one inequalities [137].

Cutting planes Cutting planes is a technique to reduce the search space by adding additional linear inequality constraints (*cuts*). It works by solving the linear relaxation of a given integer program, which is solvable in polynomial time, and iteratively modifying linear programming solutions until an integer solution is found [138].

Heuristics Integer programming heuristics aim to find a good feasible solution in a relatively fast time. However, it does not guarantee to solve a problem with provable optimality. Examples of these heuristics based on branch and bound algorithms include: *stopping with a guarantee of closeness to optimality*, *beam search*, and *depth-first search to first incumbent* [139].

We do not elaborate on these techniques here as this dissertation does not try to improve current theory in solving integer programming problems, but to build an application utilising existing solutions. In a modern integer programming solver, such as Gurobi[2] and

[2] `https://www.gurobi.com/`

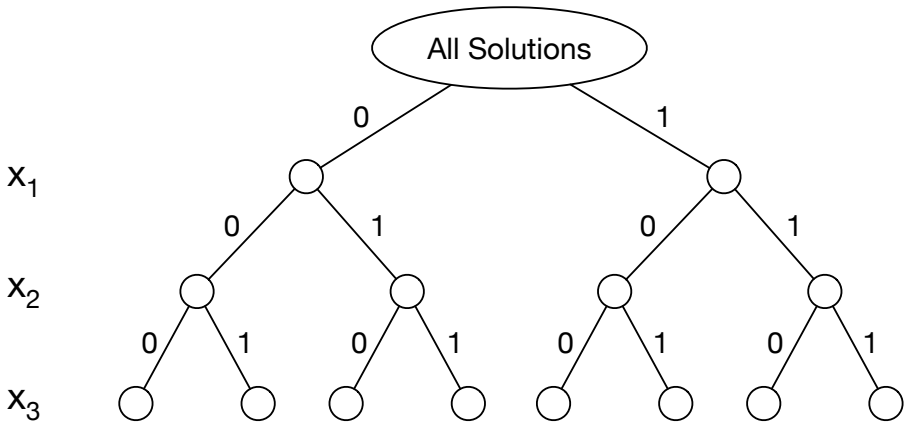


Figure 2.3: All solutions of a BIP problem with three variables

Optimization Toolbox[3], a combination of these techniques is often used to achieve better performance.

2.6 Simulated annealing

Simulated annealing is a stochastic approach to approximate the global optimum of a given function. It is most useful in the presence of large numbers of local optima [140]. Simulated annealing algorithms are inspired by the physical *annealing* procedure. In condensed matter physics, *annealing* denotes a physical process in which a metal is heated up to an annealing temperature and then slowly cooled down according to a specific schedule to decrease defects. Simulated annealing algorithms mimic this physical mechanism by temporarily accepting a worse solution with a specific probability. Thus, it allows the algorithm to escape from a local optimum.

Generally, a simulated annealing algorithm works as follows:

(1) Select a starting temperature T_0 and a starting point $\mathbf{x}_0$. The starting point $\mathbf{x}_0$ is also the best solution known at the start (i.e., $\mathbf{x}_* = \mathbf{x}_0$).

(2) Sample a new neighbour point $\mathbf{x}_t$. The point $\mathbf{x}_t$ is typically generated by applying a perturbation on the current best solution $\mathbf{x}_*$.

(3) Decide whether to accept or reject $\mathbf{x}_t$ with a certain probability. The acceptance criterion $P(\mathbf{x}_*, \mathbf{x}_t, T)$ depends on the current temperature and the difference in objective between the previous and new solution. If the algorithm decides to accept $\mathbf{x}_t$, the best solution $\mathbf{x}_*$ will be updated to $\mathbf{x}_t$.

(4) Finish the algorithm if the termination conditions are satisfied and output $\mathbf{x}_*$ as

[3] `https://www.mathworks.com/products/optimization.html`

the final 'optimal' solution. Otherwise, decrease the temperature T according to a cooling scheme and jump back to (2).

A commonly used acceptance criterion is based on the Metropolis algorithm [141]. In particular, it is defined as follows:

$$P(\mathbf{x}_*, \mathbf{x}_t, T) = \begin{cases} 1, & \text{if } f(\mathbf{x}_t) \geq f(\mathbf{x}_*) \\ e^{\frac{f(\mathbf{x}_t)-f(\mathbf{x}_*)}{T}}, & \text{otherwise} \end{cases}$$

where $f(\cdot)$ is the objective function and $P(\mathbf{x}_*, \mathbf{x}_t, T)$ gives the probability of accepting $\mathbf{x}_t$.

It is clear from this acceptance criterion that the algorithm always accepts the neighbour point $\mathbf{x}_t$ if it provides a better or equal objective than $\mathbf{x}_*$. Otherwise, it chooses to accept $\mathbf{x}_t$ with a probability of $e^{\frac{f(\mathbf{x}_t)-f(\mathbf{x}_*)}{T}}$ to avoid being trapped in local optima. Note that as the temperature T goes lower, it is less likely for $\mathbf{x}_t$ to be accepted if it has a worse objective than $\mathbf{x}_*$. In other words, the algorithm reduces the extent of its search to converge to an optimum as the temperature decreases.

In this dissertation we apply simulated annealing to attack Apple's fix of the factory calibration fingerprinting vulnerability. More details are described in §5.6.

2.7 Summary

This chapter summarised the background required for the rest of the dissertation and highlighted the relevant related work. There are three types of fingerprinting on mobile devices based on the target: *fingerprinting the user*, *fingerprinting the software*, and *fingerprinting the hardware*. This chapter looked broadly at all three types of fingerprinting. First, we gave an introduction to biometric fingerprinting. We showed that there are a variety of physiological and behavioural traits that can be used to fingerprint users. Although biometric fingerprints offer great convenience to mobile users, they have raised concerns with regard to user privacy. Biometric fingerprinting is a widely known concept and its introduction can help to better understand its software and hardware counterparts. We then discussed the topics of this dissertation: *library fingerprinting* to identify third-party software libraries, and *device fingerprinting*, to identify devices by their software and hardware components. We summarised existing studies on both topics and showed that current solutions failed to generate obfuscation-resilient library fingerprints and time-invariant device fingerprints in practical settings. Our work fills this gap and in doing so helps to improve platform security and protect user privacy.

The remainder of the chapter summarised a few key techniques used later in this dissertation. Specifically, we gave a brief introduction to integer programming and LSH that we use in Chapter 3 to formulate the library fingerprinting problem and accelerate our

algorithm. In addition, we described the general steps of a simulated annealing algorithm to support our device fingerprinting analysis.

CHAPTER 3

THIRD-PARTY ANDROID LIBRARY FINGERPRINTING

In this chapter we present a novel third-party Android library fingerprinting tool, LIBID, that can reliably identify the library version used in Android apps given the library and app binaries. LIBID is resilient to common code obfuscation techniques, including identifier renaming, code shrinking, control-flow randomisation, and package modification.

This chapter describes the design and evaluation of LIBID. First, we summarise well-related studies on library fingerprinting in §3.1. Then, we give details about the design of LIBID. We start by building library and app profiles directly from their binaries by fingerprinting obfuscation-resilient features. Based on these profiles, we match library classes with app classes. In particular, we formulate the constraints between matched classes and construct novel Binary Integer Programming (BIP) models to find the optimal match pairs that satisfy these constraints. LIBID also leverages a special Locality-Sensitive Hashing (LSH) technique to improve its efficiency and scalability by avoiding full pair-wise feature comparisons. In terms of computation complexity, we design two schemes for LIBID: LIBID-S (§3.2.1) and LIBID-A (§3.2.2), with a focus on scalability and accuracy, respectively. In addition, we develop a systematic approach to generate synthetic apps with different versions of libraries (§3.3.1). We use generated apps as the ground truth to tune the detection thresholds of LIBID and tentatively compare the results with several state-of-the-art library detectors. Furthermore, we apply all library detectors to open source apps on F-Droid and benchmark their performance (§3.3.2). We also conduct a large-scale study on popular Google Play apps and identify a high proportion of popular apps using vulnerable libraries (§3.3.3). Finally, we discuss the limitations in §3.4 and conclude this chapter in §3.5.

Contributions. We make the following academic contributions in this chapter:

1. We use recent software engineering methods, such as *LSH Ensemble*, to design and implement a state-of-the-art third-party library fingerprinting system for Android.
2. We are the first to formulate library matching constraints and convert library fingerprinting into a BIP problem.
3. We propose a method of generating synthetic apps, where the ground truth is known, in order to empirically determine appropriate detection thresholds.
4. We show LibID achieves higher F_1 score than other state-of-the-art methods both when no obfuscator is used and when ProGuard/Allatori/DashO is used.
5. We show that LibID successfully detects vulnerable versions of the `OKHttp` library in nearly 10% of popular Google Play apps; LibScout only finds 7.5%.
6. We make all source code written by us available for other researchers.[1]

3.1 Related work

The high prevalence of third-party libraries in Android apps has been an obstacle to app clone detection research for many years. Most of the existing studies employed naive whitelisting techniques to identify and exclude common libraries before analysis, either by comparing package names [142, 143] or the hash value of known libraries [144]. However, these approaches can miss many less-popular libraries, and the lack of granularity makes them incapable of detecting obfuscated libraries. More advanced work filters libraries by clustering. WuKong [6] chose the frequency of Android API calls as the feature to identify libraries by clustering. Andarwin [145] grouped similar *semantic blocks* from the Program Dependence Graph (PDG) to detect library and applied LSH techniques to accelerate the clustering process. LibDetect [146] identified and removed in-app library classes from an app using fuzzy hashing to increase the accuracy of app clone detection.

Grace et al. and Book et al. studied the presence and behavior of advertising libraries using a whitelist [147, 148]. Li et al. harvested 1,113 common libraries from 1.5 million Google Play apps by comparing the name of in-app methods and packages [149]. LibRadar [9] implemented and improved WuKong by providing an online detection platform and designing a better cluster algorithm. LibD [10] detected and classified library candidates based on dependencies between methods and packages. LibSift [150] performed library detection by comparing the primary components of the PDG. In addition, machine learning techniques have been applied to detect third-party libraries [151, 152]. Nevertheless, these proposals rely on the package hierarchies and can neither pinpoint library version nor handle popular obfuscation techniques such as code shrinking.

[1] `https://github.com/ucam-cl-dtg/LibID`

MobScanner [153] used weighted features to identify in-app library versions. However, it relies on string constants and is unreliable when intensive code shrinking is applied. Similar to LibID, LibScout [12] built library profiles from pre-collected library binaries. LibScout used a fuzzy descriptor to generate a method signature, and further obtain a class signature using the hash of in-class methods. Then, LibScout calculates the similarity of class signatures between a candidate package and a known library package to determine whether they are a match. Although LibScout is resilient against identifier renaming operations, it performs poorly if either dead-code elimination or class repackaging is enabled. OSSPolice [56] was designed to identify license violations. It extracted more features (e.g., string constants) from library binaries to better pinpoint library versions. Nevertheless, OSSPolice is only resilient to simple obfuscation techniques such as identifier renaming. Ordol [154] is another tool that can detect the library version in Android binaries. It assigned a weight to every method and class based on the number of instructions and tried to find the maximum weight bipartite matchings. Although it does not rely on any hierarchy information, Ordol requires pair-wise comparison and its threat model does not include code shrinking and package modifications.

Recently, LibPecker [155] utilized the class dependencies to perform obfuscation-resilient library matchings. Instead of matching the dependency graph, they encoded the graph into a set of fuzzy class signatures and calculate the Jaccard similarity between the library and app signatures to decide whether two classes are a match. Nevertheless, LibPecker relies on the package hierarchy and cannot handle drastic code shrinking. Orlis [13] was also designed to be resilient to code shrinking and class repackaging. The authors have shown that Orlis outperformed LibDetect when identifying obfuscated in-app library classes. However, as shown in §3.3, Orlis is not designed to detect specific library versions.

3.2 LibID design

The aim of LibID is to reliably identify third-party libraries from Android binaries against popular obfuscation techniques, which often requires fine-grained analysis that is time-consuming. To improve the usability of LibID, we introduce two library fingerprinting schemes: LibID-S (§3.2.1) and LibID-A (§3.2.2), which focus on scalability and accuracy, respectively.

The workflow of LibID is illustrated in Figure 3.1. It consists of two major steps: *fingerprinting* and *matching*. In particular, LibID first generates a profile for every library and app binary via fingerprinting obfuscation-resilient features. Then, these profiles are compared using a matching process. The output of the matching process allows LibID to report the confidence of any given library appearing in any given app. If there is a match, LibID additionally reports the in-app package names of all matching libraries in an app,

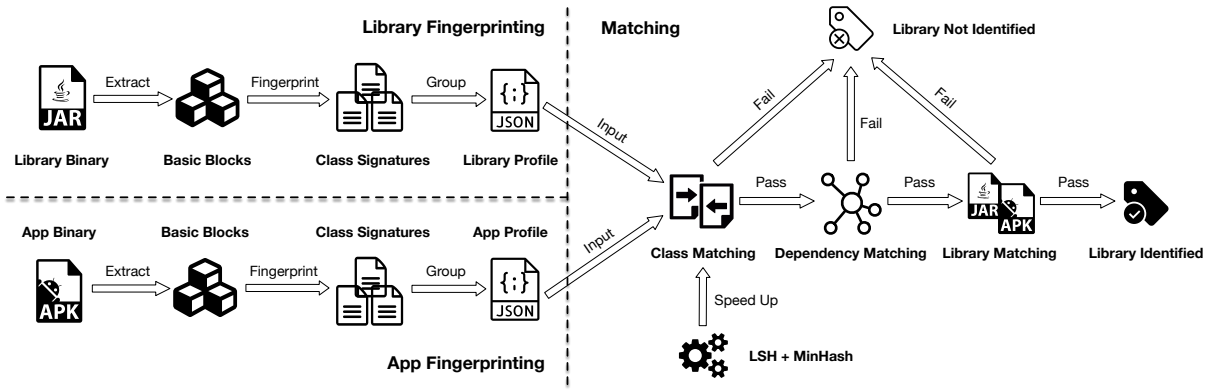


Figure 3.1: The workflow of LibID

which enables researchers to quickly find the library code in the app.

3.2.1 Scalability-focused design (LibID-S)

In this section we describe the design of our scalability-focused library fingerprinting tool, LibID-S. In particular, we first present our novel approach to generate both library and app profiles in §3.2.1.1. Then, we introduce our matching algorithm to decide whether a library is used in an app (§3.2.1.2). Last but not least, we summarise the pros and cons of LibID-S in §3.2.1.3.

3.2.1.1 Fingerprinting

As shown in Figure 3.1, the *fingerprinting* step can be divided into three stages. We do not differentiate the fingerprinting of apps and libraries here because they use the same techniques. In particular, LibID-S first constructs the Control Flow Graph (CFG) from a library binary and extracts all basic blocks from it. Then, LibID-S builds a collection of *basic block signatures* that are invariant to identifier renaming, code shrinking, control flow randomisation, and package modification techniques. The challenge is to do this while preserving as much information about the original code as possible. In this chapter, we use the features and grammar presented in Table 3.1 to build a textual representation of the *basic block signature*, which records, in order, all field-related operations (*read* or *write*), method calls, the existence of strings inside each block, etc. If there are multiple *field* or *method* instructions in a basic block, there will be an abbreviation for each instruction in the signature. For example, if there are two *field reads* in a basic block, we will record two instances of `F0`. The basic block signature also includes the name of called methods if they are from the Android Software Development Kit (SDK).

Furthermore, we associate the class features with each basic block signature by prefixing the basic block signature with four new components as shown in Figure 3.2. We elaborate on these appended fields below.

Table 3.1: Grammar for the basic block signature

Instruction Type		Abbreviation
Basic Block		`B`
Field	Read	`F0`
	Write	`F1`
Method Call	New Instance	`M0`
	Other	`M1`
String		`S`
If		`I`
Return		`R`
Goto		`G`

Class access flag This field keeps the native Java access flag of the class since obfuscators typically do not change these class access flags.

Superclass name This field records the name of the superclass if the current class is extended from a superclass in the Android SDK. Otherwise, this field is `[X]`.

Class interfaces This field only records class interfaces that are from the Android SDK. If there are multiple qualified interfaces, a separator `|` will be inserted in this field to separate them. This field is `[]` if the class does not implement any class from the Android SDK.

Method descriptor We use the same format as LibScout [12] and OSSPolice [56] to profile class methods. Each method parameter is represented by its type in Dalvik and non-framework types will be replaced with a placeholder `X`.

This augmented basic block signature represents a feature of the class, and thus we call it a *class signature*; a class typically has a set of class signatures. An example of a class signature is given in Figure 3.2. Then, each Android app or library can be characterised by a *class signature dictionary*, where the key is the name of its member class and the value is the set of class signatures of that class.

3.2.1.2 Matching

After generating both app and library profiles, LibID-S compares their similarity and calculates the confidence of the library being used in the app. We refer to this procedure as the *matching* process. As shown in Figure 3.1, the *matching* process can be broken down into the following three steps: *class matching*, *dependency matching* and *library matching*.

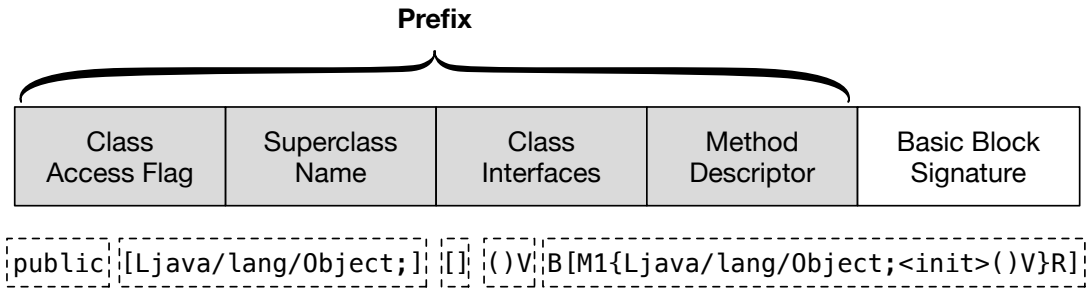


Figure 3.2: Format of a class signature with an example

Class matching. This stage aims to find the candidate matches between library and app classes. An intuitive way to achieve our goal is to check whether these classes have the same signatures. However, code obfuscators, such as ProGuard, can remove unused basic blocks from the original class or even delete the whole class. This process is called *code shrinking* or *dead-code elimination*. Thus, an obfuscated library class may have various signatures in different apps. In this regard, instead of looking for the same class signature sets, we define the *class signature containment* index as follows to quantify the similarity between two classes.

Definition 3.1 (Class Signature Containment)**.** For two classes c_1 and c_2, whose signature set is s_1 and s_2, respectively, the *class signature containment* of c_1 in c_2 is defined as:

$$S_C(c_1, c_2) = \frac{|s_1 \cap s_2|}{|s_1|} \tag{3.1}$$

In LibID, library profiles are extracted directly from their binaries, and thus they contain a complete set of library class signatures. However, an in-app version of a library class may contain only a small subset of signatures due to code shrinking applied by code obfuscators. LibID-S only considers an app class c_a as a candidate match to a library class c_l if $S_C(c_a, c_l)$ is equal to 1, since code shrinking does not change this index.

This containment index has been used to detect piggybacking apps [145, 156]. However, it requires pair-wise comparison which is time-consuming. Locality-Sensitive Hashing (LSH) provides an efficient solution to this problem [157]. It pre-processes the dataset by creating signature hashes such that similar items have a higher chance of sharing the same hash. Here, LibID applies a special LSH index technique named *LSH Ensemble* [134], which supports the containment query, to speed up the matching process.

Dependency matching. The *dependency matching* stage aims to find the true match pairs from the candidates. Based on Definition 3.1, it is likely that a single class in an app is matched to multiple candidate library classes, or vice versa. This is problematic since

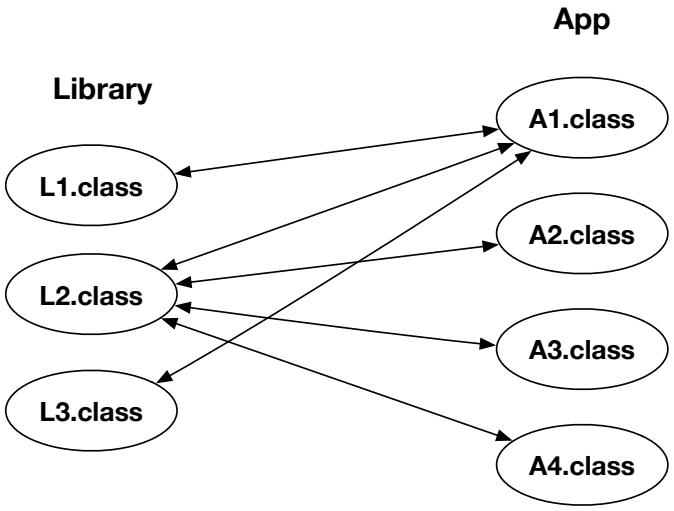


Figure 3.3: Class matching result (ellipses stand for classes and the connections between them indicate a candidate match)

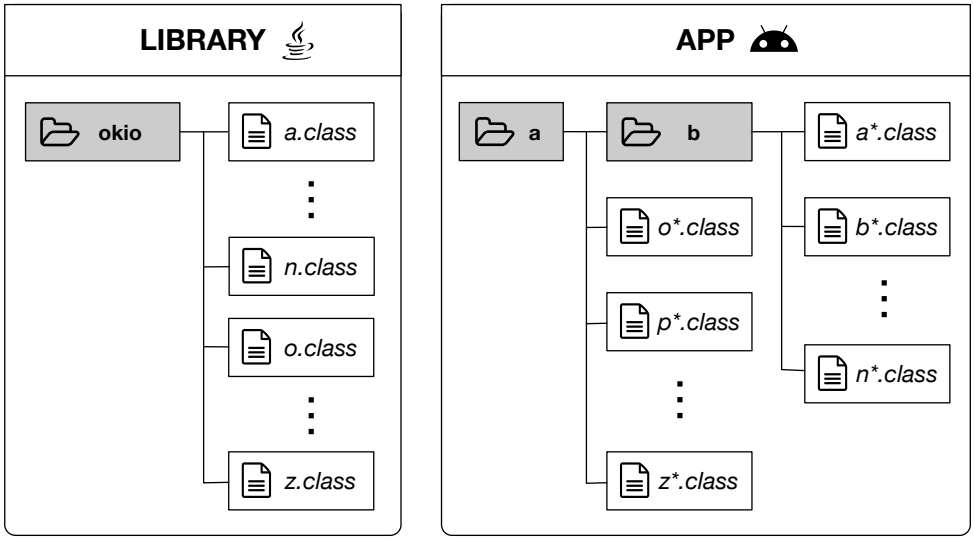
each library class can produce at most one matching in-app class, and vice versa. By way of an example, the classes `L1.class` and `L2.class` in Figure 3.3 cannot both be matched to `A1.class`. In fact, there can be at most two true matching pairs in Figure 3.3.

In general, suppose there are m library classes $\mathbf{C_L}$ that have candidate matches in the app and n app classes $\mathbf{C_A}$ that have candidate matches in the library. Then, we can generate a candidate match matrix $\mathbf{R} \in \{0,1\}^{m \times n}$, where $\mathbf{R}_{c_l,c_a} = 1$ if and only if $c_l \in \mathbf{C_L}$ and $c_a \in \mathbf{C_A}$ are a candidate match. A true class match matrix $\mathbf{C} \in \{0,1\}^{m \times n}$ ensures a unique match between library and app classes should have the following *uniqueness constraints*:

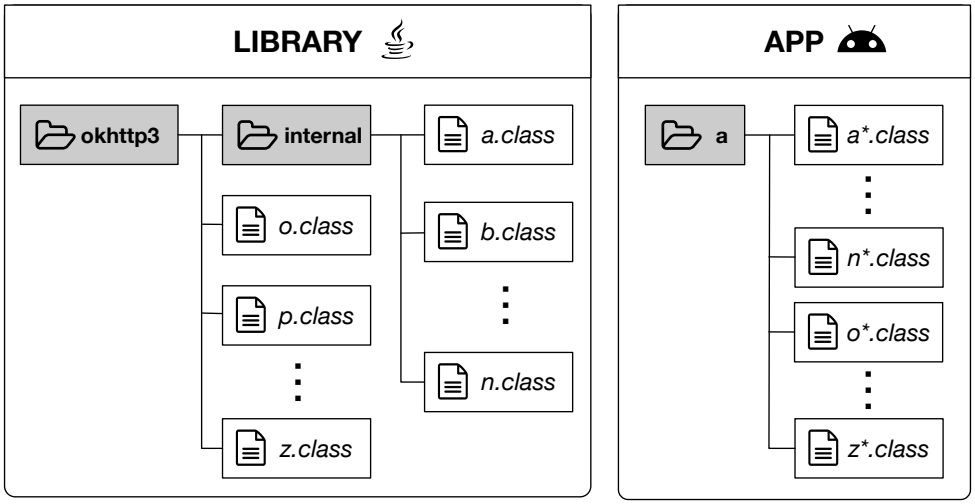
$$\begin{aligned} \mathbf{C}_{c_l,c_a} &\leq \mathbf{R}_{c_l,c_a}, \ \ \forall c_l \in \mathbf{C_L}, c_a \in \mathbf{C_A} \\ \sum_{c_a \in \mathbf{C_A}} \mathbf{C}_{c_l,c_a} &\leq 1, \ \ \forall c_l \in \mathbf{C_L} \\ \sum_{c_l \in \mathbf{C_L}} \mathbf{C}_{c_l,c_a} &\leq 1, \ \ \forall a_i \in \mathbf{C_A} \end{aligned} \tag{3.2}$$

Or in words, there should be at most one non-zero value in each row and column in $\mathbf{C}$ and $\mathbf{C}$ is a subset of $\mathbf{R}$.

In Android apps and libraries, each package may include a set of classes and several sub-packages. In most instances, obfuscators do not change the internal package structures. Therefore, LibID-S by default will only match an app class with a library class if they are at the same level in the matching package. For example, in Figure 3.4, suppose each library class is a candidate match to an app class. Here, we use the superscript * to denote candidate match pairs (e.g., `a.class` is a candidate match to `a`*`.class`). If we want to check whether the library package `Okio` and app package `a` is a match, then the match pair (`Okio/o.class`, `a/o`*`.class`) is *valid* while (`Okio/a.class`, `a/b/a`*`.class`) is



(a) Package flattening



(b) Class repackaging

Figure 3.4: Class hierarchy in app and library packages

invalid because classes in the latter pair are at a different level to the matching package. However, if we are comparing package `Okio` with sub-package `b` in Figure 3.4 (a), then the pair (`Okio/a.class`, `b/a*.class`) is *valid* because both `a.class` and `a*.class` are directly under the matching package.

To make sure the matched class pairs are *valid*, we need some additional constraints. Suppose there are m_p library packages $\mathbf{P_L}$ and n_p app packages $\mathbf{P_A}$ that contain matched classes or their sub-packages contain matched classes. For example, if $\mathbf{C_L}$ consists of two classes (`a/b/x.class` and `a/c/y.class`), then $\mathbf{P_L}$ would have three elements (`a/b`, `a/c`, `a`). Let $\mathbf{P} \in \{0,1\}^{m_p \times n_p}$ be the true package match matrix, then the hierarchical relation between classes can be ensured by the following *hierarchy constraints*:

$$\mathbf{P}_{p_l,p_a} \leq \mathbf{P}_{\text{parent}(p_l),\text{parent}(p_a)}, \quad \forall p_l \in \mathbf{P_L}, p_a \in \mathbf{P_A} \tag{3.3a}$$

$$\sum_{p_a \in \mathbf{P_A}} \mathbf{P}_{p_l,p_a} \leq 1, \ \ \forall p_l \in \mathbf{P_L} \tag{3.3b}$$

$$\sum_{p_l \in \mathbf{P_L}} \mathbf{P}_{p_l,p_a} \leq 1, \ \ \forall p_a \in \mathbf{P_A} \tag{3.3c}$$

$$\mathbf{C}_{c_l,c_a} \leq \mathbf{P}_{\text{package}(c_l),\text{package}(c_a)}, \ \ \forall c_l \in \mathbf{C_L}, c_a \in \mathbf{C_A} \tag{3.3d}$$

Constraint 3.3a ensures the app and library packages can only be a match if their parent package also matches. In particular, if a matching package is already a top-level package (no parent package), this constraint will be ignored. Constraint 3.3b and 3.3c make sure each library package can only be matched to at most one app package. Constraint 3.3d ensures that classes can only be matched if their packages match. Collectively, these hierarchy constraints guarantee that matched class pairs have the same hierarchy.

The objective of LibID-S is to find the maximum number of matched class pairs, which can be formulated to the following Binary Integer Programming (BIP) problem:

$$\begin{aligned} \textbf{maximise} \quad & \sum_{c_l \in \mathbf{C_L}, c_a \in \mathbf{C_A}} \mathbf{C}_{c_l,c_a} \\ \textbf{subject to} \quad & \text{Constraint 3.2 and 3.3} \end{aligned}$$

BIP problems have been extensively studied for decades and there are several sophisticated software packages available to help find solutions (§2.5).

Previous work has shown that 21% of apps using ProGuard have applied package modification operations, which includes *package flattening* and *class repackaging* [11]. If package flattening is enabled, rule-specified packages will be put into a single parent package, but the structure of the package will be preserved. Notably, we do not require the hierarchy of packages to be the same. For instance, it is possible to have a match between package `Okio` and `a/b` in Figure 3.4 (a). Therefore, LibID-S is robust to package flattening and package renaming tools that could change the hierarchy of root packages but not internal package structures.

By contrast, if class repackaging is applied, rule-specified classes will be moved to a single parent package, and thus the class hierarchies in the original package will no longer be preserved. An example is given in Figure 3.4 (b), where all classes in the library have been repackaged to package `a` in the app. If we still use Constraint 3.3, then many true class matches would be missed. To address this problem, we create a Class Repackaging Detection (CRD) mode. When LibID-S runs in the CRD mode, every app package that does not have sub-packages could be the parent package of repackaged classes. Let $\mathbf{P'_A}$ be the collection of these app packages, then LibID-S has the following hierarchy constraints

when the CRD mode is on:

$$\begin{gathered}\sum_{p_a \in \mathbf{P'_A}} \mathbf{P}_{p_l,p_a} \leq 1, \ \ \forall p_l \in \mathbf{P_L} \\ \mathbf{P}_{p_l^1,p_a} = \mathbf{P}_{p_l^2,p_a}, \ \ \forall p_l^1, p_l^2 \in \mathbf{P_L}, p_a \in \mathbf{P'_A} \\ \mathbf{C}_{c_l,c_a} \leq \mathbf{P}_{\text{package}(c_l),\text{package}(c_a)}, \ \ \forall c_l \in \mathbf{C_L}, c_a \in \mathbf{C_A}\end{gathered} \tag{3.4}$$

Or in words, all library packages could only be matched to a single app package that does not contain a sub-package. For example, in Figure 3.4 (b), app package `a` could be a candidate match to the library package `okhttp3`. Therefore, LibID-S is able to deal with class repackaging techniques if code shrinking is not applied at the same time.

Library matching. To pinpoint the third-party in-app library versions, an intuitive solution is to check the proportion of library classes that are present in the app. However, some classes may have no viable signature, which we call *unproductive classes*. An example of an unproductive class is a class that contains only abstract methods (i.e., no basic block). Unproductive classes contribute little to the uniqueness of the library and will never be matched to other classes. Therefore, we exclude them from the proportion calculation. Nevertheless, this evaluation index is not robust against code shrinking. As a fix, we also consider the proportion of matched classes in the matched app package and define the *library match index* as follows:

Definition 3.2 (Library Match Index)**.** Suppose the root package P_L of library L is matched to a package P_A in app A, and there are N_C true class match pairs between L and A, then the library match index of L in A is:

$$M(L, A) = \frac{N_C}{\min(N_{P_L}, N_{P_A})}$$

Where N_{P_L} and N_{P_A} are the number of productive classes in P_L and P_A, respectively.

The library match index is suitable for quantifying the confidence of the presence of a library in an app. On the one hand, apps may use several libraries that share the same root package. In this case, N_C/N_{P_A} can be pretty low even if the library is used by the app. On the other hand, if code shrinking is applied, then N_C/N_{P_L} could be very low but N_C/N_{P_A} would remain high. In both cases, $M(L, A)$ can still retain a high value. Therefore, we can decide whether app A uses library L using two thresholds:

$$M(L, A) > \Gamma_1 \text{ and } \frac{N_C}{N_{P_L}} > \Gamma_2 \tag{3.5}$$

The threshold Γ_2 ensures there is enough information about the library for us to make a meaningful decision. In most cases, $M(L, A)$ should be close to 100% if app A uses

library L. Although LSH will inevitably introduce false negatives and false positives, the relation constraints can help to reduce false alarms during the matching process. However, $M(L, A)$ can still be a small value if both class repackaging and code shrinking are applied. In this case, we can adjust the value of Γ_1 to balance between the detection of libraries and introducing false positives.

Since library updates typically modify only a small fraction of the code, there are likely multiple library versions that satisfy Constraint 3.5. To pinpoint the library version, LibID-S only keeps the library version(s) that produces the highest library match index.

3.2.1.3 Summary

LibID-S only relies on features that are consistent under code shrinking, control flow randomisation, and package modification. The use of the *LSH Ensemble* greatly reduces the matching time by avoiding pair-wise comparison between classes and all calculations in LibID-S are relatively light-weight. Therefore, LibID-S is a suitable tool when computing resources are limited. Nevertheless, although we have considered the *uniqueness* and *hierarchy* constraints during the matching process, this coarse-grained analysis can still result in false matches between classes. In addition, if multiple libraries share the same root package in the app and each library only has a few classes remained in the app, LibID-S may not be able to identify these libraries.

3.2.2 Accuracy-focused design (LibID-A)

Because LibID-S only considers coarse-grained features, it may not be suitable for applications that favour accuracy over scalability. In this section we present an extended design, LibID-A, to address this problem. Similar to LibID-S, the workflow of LibID-A consists of *fingerprinting* and *matching* steps.

3.2.2.1 Fingerprinting

On the basis of LibID-S, LibID-A employs finer-grained features (invocation, interface, and inheritance dependencies) during the library fingerprinting process, which enables it to not only find library matches for each app more precisely but also pinpoint class match pairs with higher accuracy. Overall, LibID-S does not make use of the information about dependencies between different classes, which may result in incorrect matching. To address this problem, LibID-A further records the following additional information:

Method calls If a method callee is obfuscatable, LibID-A will record the name of the called class and method under the key of the caller class in the *invocation dictionary*.

Class inheritance If a superclass is obfuscatable, LibID-A will record the name of the superclass under the key of the current class in the *inheritance dictionary*.

Class interfaces If a class interface is obfuscatable, LibID-A will record the name of the interface under the key of the current class in the *interface dictionary*.

Notably, we only keep records of the above information if the relating entities are obfuscatable (i.e., not from the Android SDK). Otherwise, they should have already been integrated into class signatures. Then, LibID-A builds *dependency graphs* based on the invocation, inheritance, and interface dictionaries. Here, we define a dependency graph as an undirected graph, where each node corresponds to a class and each edge represents an invocation, inheritance or interface dependency between two classes. Note that, these dictionaries are only used to construct the dependency graph. No static identifiers (e.g., method and class names) are utilised for string matching as they can easily be changed by identifier renaming.

3.2.2.2 Matching

The matching process in LibID-A also includes three major phases: *class matching*, *dependency matching*, and *library matching*.

Class matching. This process is the same as the *class matching* step in LibID-S. We use the class containment similarity index (see Definition 3.1) to find all candidate class matches while applying *LSH Ensemble* to speed up the matching process.

Dependency matching. Apart from considering the uniqueness and hierarchy constraints as in LibID-S, LibID-A also includes the following constraints:

Invocation constraints For any two app classes ($c_a^1, c_a^2 \in \mathbf{C_A}$), if c_a^1 invokes a method in c_a^2 with a formatted method descriptor of d, and their true class match in the library is c_l^1 and c_l^2, respectively, then there must be a method call from c_l^1 to c_l^2 whose descriptor is also d. It is not necessarily true the other way around.

Inheritance constraints For any two app classes ($c_a^1, c_a^2 \in \mathbf{C_A}$), if c_a^1 is the superclass of c_a^2, and their true class match in the library is c_l^1 and c_l^2, respectively, then c_l^1 must also be the superclass of c_l^2, and vice versa.

Interface constraints For any two app classes ($c_a^1, c_a^2 \in \mathbf{C_A}$), if c_a^1 is an interface of c_a^2, and their true class match in the library is c_l^1 and c_l^2, respectively, then c_l^1 must also be an interface of c_l^2. It is not necessarily true the other way around.

The formulation of these constraints are very similar to LibID-S, and thus we do not reiterate them here. Notably, the invocation and interface constraints do not work the other way around. The reason is that both method calls and class interfaces can be deleted during code obfuscation, but the superclass is always kept if the child class exists. LibID-A also has a CRD mode, which replaces the default hierarchy constraints with Constraint 3.4 while keeping other constraints the same.

The objective of LibID-A can be formulated as follows:

$$\textbf{maximize} \sum_{c_l \in \mathbf{C_L}, c_a \in \mathbf{C_A}} \mathbf{C}_{c_l,c_a} + w \sum_{c_a^1, c_a^2 \in \mathbf{C_A}} (\mathbf{V} + \mathbf{H} + \mathbf{T})_{c_a^1, c_a^2}$$

subject to uniqueness, hierarchy, invocation, inheritance, and interface constraints

where $\mathbf{V}, \mathbf{H}, \mathbf{T} \in \{0,1\}^{n \times n}$ are the invocation, inheritance, and interface matrices. Their element is 1 if and only if there is corresponding dependency between c_a^1 and c_a^2. The weighted parameter w should be small enough (e.g., 0.0001). In other words, LibID-A aims to find the optimal solution that has the largest number of class match pairs. If there are multiple solutions, it selects the one with the largest number of dependency matches.

Library matching. To reliably detect libraries when both class repackaging and shrinking are applied, LibID-A utilises the dependency graph to perform better library matching.

Relation pruning Although class repackaging can move classes from different packages into a single package, the dependency graphs stay the same. By way of an example, the dependency graph of the `Okio` library version 1.7.0 is shown in Figure 3.5. Therefore, we should only consider classes in the app package that are reachable from any matched classes. In other words, an app class needs to be in the same connected component with any matched class in the dependency graphs in order to be considered. We name this process *relation pruning*.

Ghost hunting Library classes may invoke methods from classes that exist in another library under the same package name. This is most common when libraries are developed by the same company. For instance, the `Leakcanary` library calls methods from the `Leakcanary-Watcher` library. In addition, they share the same root package `com/squareup/leakcanary`. In this case, we cannot eliminate the interference of other libraries through *relation pruning*. Instead, for each library class, if it depends on another class that is not in the library and not from Android SDK, we will record this connection and refer to the connected class as a *ghost class*. For each *ghost class*, we find its matched app class by *dependency matching* and remove it from the dependency graphs. We call this process *ghost hunting*.

For each library and app match pair, we first implement *ghost hunting* to eliminate the

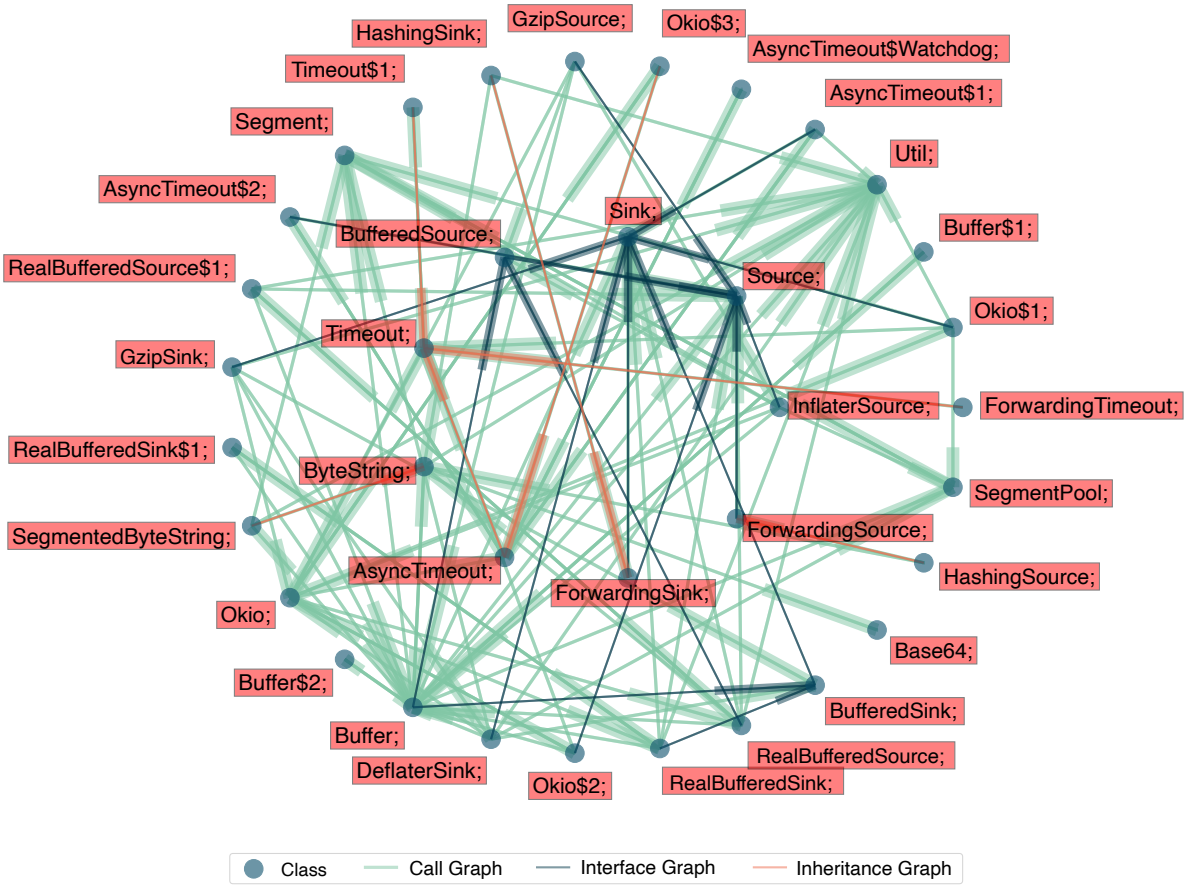


Figure 3.5: Dependency graph of Okio 1.7.0 (direction is represented by the thick line)

distraction of ghost classes and then apply *relation pruning* to avoid the interference of classes that are not from the library but in the same app package. Let $N_{P_A}^C$ be the number of qualified app classes, then the definition of library match index for LibID-A can be updated to:

$$M(L, A) = \frac{N_C}{\min(N_{P_L}, N_{P_A}^C)} \tag{3.6}$$

Where notations have the same meaning as in Definition 3.2. If *class matching* is accurate, $M(L, A)$ should be the same after applying class repackaging or when other classes are inside the same package. Similar to LibID-S, we can use the Constraint 3.5 to decide if library L is used in app A.

3.2.2.3 Summary

Overall, LibID-A has enhanced the capabilities of LibID-S. By introducing finer-grained constraints, LibID-A provides higher accuracy in terms of class-level matching, which further enables LibID-A to pinpoint the version of libraries with better accuracy. In addition, LibID-A utilises the dependency graphs to separate different library classes, which enables it to better deal with class repackaging and the interference of shared-package

libraries. The adoption of finer-grained features will inevitably slow down the execution of LibID-A. To achieve a balance between time efficiency and accuracy, LibID-A also uses *LSH Ensemble* to speed up the matching process and applies various constraints to minimise false matches. The execution of LibID-A is independent for different apps and thus can easily be parallelised.

3.3 Evaluation

LibID uses Androguard[2] version 3.0 with the DAD decompiler to construct the CFG of methods. We extend the original Androguard project to support multi-dex files and our signature algorithms. In addition, we make use of the datasketch[3] package to implement the *LSH Ensemble* algorithm; all LSH parameters are set to their default value tuned by benchmarks. The BIP models are implemented with the Gurobi optimiser[4]. We deploy LibID on a computing node with 32 Intel Xeon Gold 6142 CPUs with 12GB RAM each.

To evaluate the effectiveness of LibID, we collect 69 popular libraries with 1 444 versions from Maven Central, JCenter, Github, and official library websites. Since Androguard cannot directly parse `.jar` files, we convert them to `.dex` files using dex2jar[5]. The decompilation, however, does not work in all cases. During our experiment, around 94.17% of libraries are successfully converted. Then, we compare LibID with state-of-the-art work, including LibScout [12], Orlis [13], and LibPecker [155], under different settings.

3.3.1 Library fingerprinting on synthetic apps

To determine the detection threshold Γ_1 and Γ_2 in Constraint 3.5, we propose a novel way to generate synthetic apps with different versions of libraries under different obfuscation configurations. We use ProGuard as the obfuscator due to its popularity. A study has shown that about 88% of obfuscated apps that were popular on the Google Play store chose ProGuard as their only obfuscator, and 70% of them used the default configuration [11].

Our method works by creating an empty Android project. Then, for each library in our dataset, we manually collect code snippets that provide main library functions from either the official documentation or example projects. We store these snippets in a database, along with the import statements, boilerplate program, and dependencies for each library. In our dataset, the main APIs of most libraries are consistent for a wide range of versions. Therefore, a single generated project can be used with many library versions. We compile each app in the database under four different ProGuard settings, as shown in Table 3.2.

[2] `https://github.com/androguard/androguard`
[3] `https://github.com/ekzhu/datasketch`
[4] `https://www.gurobi.com`
[5] `https://github.com/pxb1988/dex2jar`

Table 3.2: Library detection results

(a) ProGuard disabled

Tool	# apps	# TP	# FP	# FN	F_1
Orlis	1045	359	250	436	0.5114
LibScout		969	33	43	0.9623
LibPecker		984	38	23	0.9699
LibID-S		1 044	1	0	0.9995
LibID-A		1 045	0	0	1

(b) Class repackaging enabled

Tool	# apps	# TP	# FP	# FN	F_1
Orlis	1045	499	276	270	0.6464
LibScout		7	2	1 036	0.0133
LibPecker		100	66	879	0.1747
LibID-S		1 029	5	11	0.9923
LibID-A		1 028	14	3	0.9918

(c) Shrinking enabled (default ProGuard setting)

Tool	# apps	# TP	# FP	# FN	F_1
Orlis	932	226	169	537	0.3903
LibScout		26	4	902	0.0543
LibPecker		126	90	716	0.2382
LibID-S		924	8	0	0.9957
LibID-A		932	0	0	1

(d) Shrinking and class repackaging enabled

Tool	# apps	# TP	# FP	# FN	F_1
Orlis	932	225	165	542	0.3889
LibScout		0	0	932	N/A
LibPecker		22	47	863	0.0461
LibID-S		305	145	482	0.4931
LibID-A		831	40	61	0.9427

Identifier renaming is enabled in all groups except when ProGuard is disabled.

We apply both LibID-S and LibID-A to detect in-app libraries and calculate the number of true positives (TP), false positives (FP) and negatives (FN). Here, a false negative means the tool does not report any version of the library, while a false positive is counted if the tool reported some versions of the library but not the correct one. Since we know in advance which library is used in each app, these evaluation indexes can be measured against ground truth. Then, we calculate the F_1 score to quantify the performance of

both tools. To determine Γ_1 and Γ_2, we first set them to 1 and 0, respectively, to observe the result and then gradually adjust them to appropriate values ($\Gamma_1 = 0.8, \Gamma_2 = 0.1$) to achieve the highest F_1 score.

Evaluation. We tentatively compare the performance of LibID with LibScout, Orlis, and LibPecker because they are state-of-the-art library fingerprinting tools which are designed to handle code shrinking. In particular, Orlis only reports matches between app and library classes. Therefore, we regard library candidates that have a class match reported by Orlis as possible matches. For LibPecker, we set all thresholds based on the original paper. LibPecker reports the similarity between the app and each library candidate, and thus we choose the library version(s) with the highest similarity, if above the threshold, as the matched in-app library version(s).

Table 3.2 presents the experiment results under four ProGuard settings. It shows that LibID achieves higher F_1 scores than other tools in all cases. When ProGuard is disabled, all tools except Orlis achieve high accuracy. Overall, Orlis is ineffective in identifying in-app library versions. Table 3.2 (b) shows that LibID is much more effective than LibScout and LibPecker when class repackaging is enabled. LibID accurately identifies libraries for more than 98.4% apps and reports fewer false negatives. In comparison, LibScout and LibPecker are correct for 0.7% and 9.6% apps, respectively, because they rely on the package hierarchy information.

Table 3.2 (c) demonstrates that other tools are also ineffective against code shrinking. In particular, if code shrinking is enabled, library code may be partially removed in the app. We define an index I_C to quantify the library information remaining in the app:

$$I_C = \frac{\#\text{ Signatures of library classes present in the app}}{\#\text{ Signatures of library classes in the SDK}} \tag{3.7}$$

Based on the definition, I_C will be 1 if code shrinking is disabled. We calculate I_C for every synthetic app that applied code shrinking. Because synthetic apps are relatively simple, most of them contain only a small portion of library classes after code shrinking. Figure 3.6 presents the relation between the recall and I_C when code shrinking is enabled. It reveals that LibID behaves well against code shrinking, while LibScout, LibPecker and Orlis only detect libraries when $I_C > 70\%$, and still fail to detect many in that case.

The weakness of LibID-S is clear when both code shrinking and class repackaging are enabled. LibID-S uses the library match index $M(L, A)$ in Definition 3.2 to quantify the confidence of a library L being used in an app A. However, this value can be very low if the majority of library classes have been deleted (low N_C/N_{P_L}) while there are many other non-library classes in the same package (low N_C/N_{P_A}). By contrast, LibID-A only considers app classes that are related to the library by eliminating other classes through *ghost hunting* and *relation pruning*, and thus its library match index in Equation 3.6 is

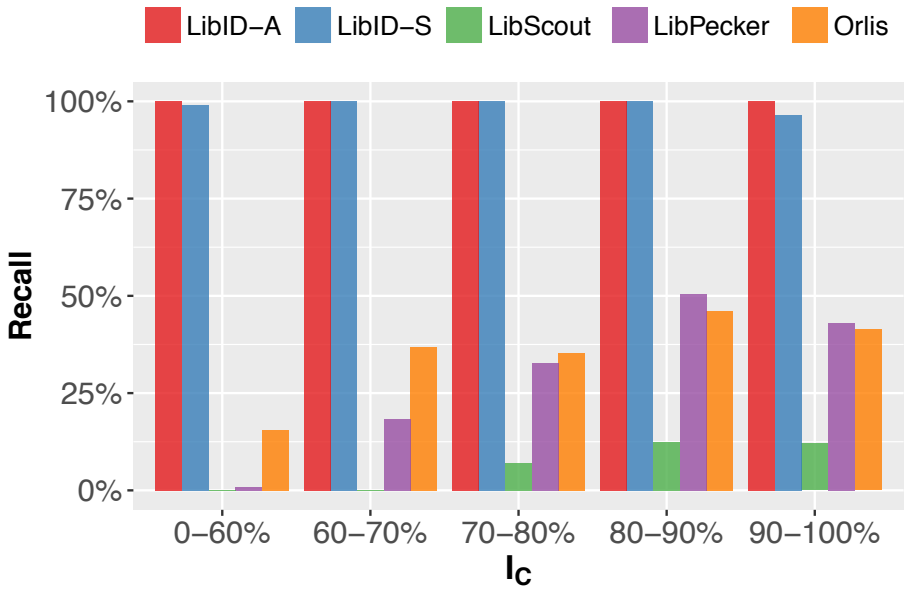


Figure 3.6: Recall under different shrinking range

still high for the right library. Nevertheless, LibID-A does produce several false alarms. There are two reasons for these errors. First, class-level matching is not always accurate due to lack of hierarchy information, which would further influence the *ghost hunting* and *relation pruning* processes. Second, in a few cases, ProGuard keeps some library classes in the original package to ensure normal functioning while moving others to a single parent package, which could degrade the accuracy of LibID-A.

3.3.2 Library fingerprinting on F-Droid apps

Wang *et al.* has compiled F-Droid projects under different obfuscators, including ProGuard, Allatori, and DashO, and published the dataset online[6]. The dataset contains around 200 app binaries for each obfuscation configuration with the ground truth of in-app library versions. However, apps in this dataset that should be obfuscated by ProGuard are, in fact, unobfuscated. Therefore, we manually compiled 215 F-Droid projects with ProGuard applied (both code shrinking and class repackaging are enabled).

We use both self-compiled apps and the dataset provided by Wang *et al.* as the ground truth to evaluate the performance of library detectors on commercial apps with different obfuscators. LibID-S and LibID-A are set with the thresholds determined in §3.3.1. Since all tools may report multiple library versions, we define two types of precision: *pinpoint precision* and *in-range precision*. In particular, the *pinpoint precision* of a tool is the proportion of correctly identified libraries over all the libraries reported. When calculating the *in-range precision*, we regard all reported versions of a library as a single entity. We say an entity is correctly identified if the correct library version is among all the versions

[6] `https://github.com/presto-osu/orlis-orcis`

Table 3.3: Library detection results

(a) No obfuscator

Tool	Precision*	Recall	F_1*
Orlis	49.82% (81.14%)	26.25%	0.3438 (0.3966)
LibScout	67.42% (69.35%)	94.09%	0.7855 (0.7984)
LibPecker	64.50% (65.26%)	91.68%	0.7573 (0.7625)
LibID-S	67.42% (93.91%)	88.35%	0.7648 (0.9105)
LibID-A	70.12% (88.16%)	96.30%	0.8115 (0.9205)

(b) ProGuard (both shrinking and class repackaging enabled)

Tool	Precision*	Recall	F_1*
Orlis	29.41% (62.50%)	15.00%	0.1987 (0.2419)
LibScout	65.22% (65.22%)	15.00%	0.2439 (0.2439)
LibPecker	20.00% (20.00%)	4.00%	0.0667 (0.0667)
LibID-S	58.67% (66.67%)	44.00%	0.5029 (0.5301)
LibID-A	64.29% (72.58%)	45.00%	0.5294 (0.5556)

(c) Allatori

Tool	Precision*	Recall	F_1*
Orlis	46.46% (76.67%)	19.37%	0.2734 (0.3092)
LibScout	87.23% (91.11%)	8.63%	0.1571 (0.1577)
LibPecker	55.28% (65.26%)	66.11%	0.6021 (0.6103)
LibID-S	73.72% (92.74%)	48.42%	0.5845 (0.6362)
LibID-A	75.86% (92.22%)	64.84%	0.6992 (0.7614)

(d) DashO

Tool	Precision*	Recall	F_1*
Orlis	54.43% (87.76%)	15.03%	0.2356 (0.2567)
LibScout	90.64% (92.81%)	54.20%	0.6783 (0.6843)
LibPecker	47.06% (47.06%)	55.94%	0.5112 (0.5112)
LibID-S	69.55% (97.45%)	53.50%	0.6047 (0.6907)
LibID-A	69.79% (91.11%)	57.34%	0.6296 (0.7039)

*: The value before the bracket is the pinpoint precision (or F_1); the value inside the brackets is the in-range precision (or F_1).

reported. The *in-range precision* of a tool is the proportion of correctly identified entities over all the entities reported. We also define the *pinpoint* and *in-range* recall by a similar approach and the respective F_1 score can be calculated based on the precision and recall.

Example 3.1. Suppose a tool reports "`ACRA-4.6.1`, `ACRA-4.6.2`, `Gson-2.5`" while the ground truth is `ACRA-4.6.1`. Then, the tool has a *pinpoint* precision, recall, and F_1 score of 1/3, 1, and 1/2, respectively. The tool also has an *in-range* precision, recall, and F_1

Table 3.4: Top 5 libraries identified by LibID

Library	# LibID	# LibScout	# Common
`Gson`	1419	1114	1093
`Facebook`	1243	1082	1058
`OkHttp`	1157	690	689
`Okio`	1139	563	561
`Bolts`	918	317	316

score of 1/2, 1, and 2/3, respectively;

Table 3.3 presents the experiment results. Overall, LibID behaves better than state-of-the-art tools in all four cases. The F_1 score (both *pinpoint* and *in-range*) of LibID is greater than 0.5 when ProGuard is applied, while that of other tools is lower than 0.25. In all cases, LibID has the highest *in-range* precisions among these tools. Nevertheless, results also reveal that the performance of LibID for these F-Droid apps is not as good as in §3.3.1. We investigate several cases where LibID-A does not identify the library correctly and find the following causes: (i) I_C is less than 5% for some libraries (e.g., Guava) after code shrinking, which is lower than the threshold; (ii) LibID-A fails to identify some libraries when both shrinking and class repackaging are enabled for the same reason as in §3.3.1; and (iii) the obfuscation in some apps changes the *class access flag* and *class interfaces* and splits a single library class into several classes, which could defeat the design of LibID-A.

3.3.3 Vulnerable library usage in popular Google Play apps

To study the library usage in production apps and further evaluate our approach, we downloaded 3 958 apps from the top-100 apps across 59 categories on the Google Play store in April 2017. We employ both LibID (using LibID-A) and LibScout to analyse these popular apps. On average, LibID detects one more library per app than LibScout. Table 3.4 gives the summary of top five most popular libraries identified by LibID. In particular, the columns respectively present the number of apps using each library identified by LibID, LibScout, and both of them. Table 3.4 shows that a large proportion of libraries have been missed by LibScout. We confirm the reason is that LibScout struggles to detect libraries after code shrinking.

Vulnerable library detection. A primary use case of LibID is to identify vulnerable in-app libraries. By way of an example, we choose `OkHttp`, a popular Android library used in 29% of apps in our dataset for managing HTTP-based network requests, as the target to further evaluate our library fingerprinting tool. There is a severe vulnerability

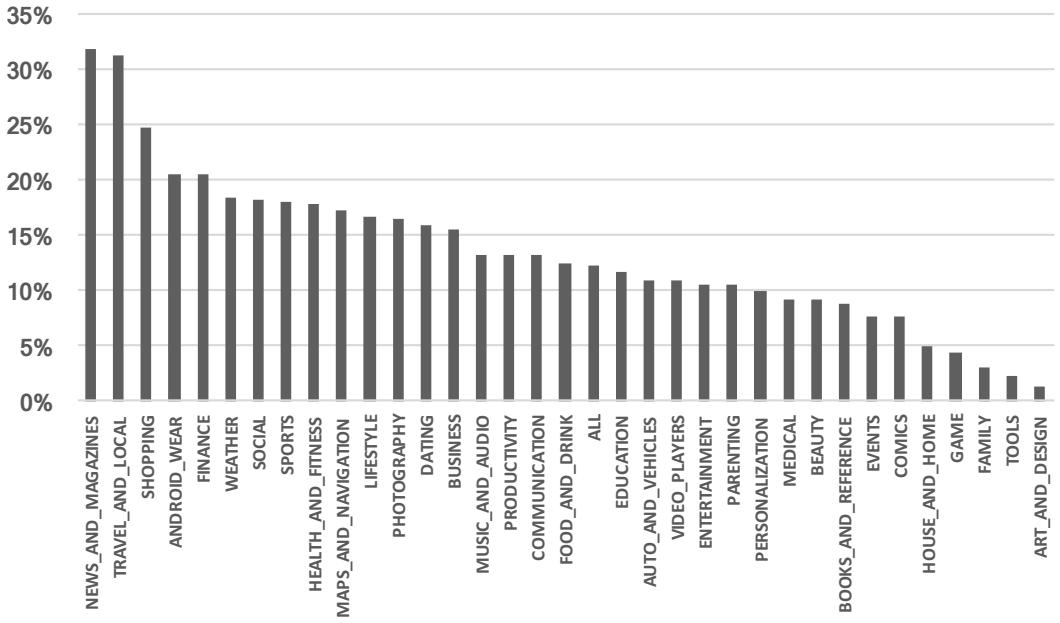
Table 3.5: Vulnerable `OkHttp` library detection result

Tool	# TP	# FP	# FN	F_1
Orlis	172	165	222	0.4706
LibScout	295	1	99	0.8551
LibPecker	339	9	55	0.9137
LibID	393	0	1	0.9987

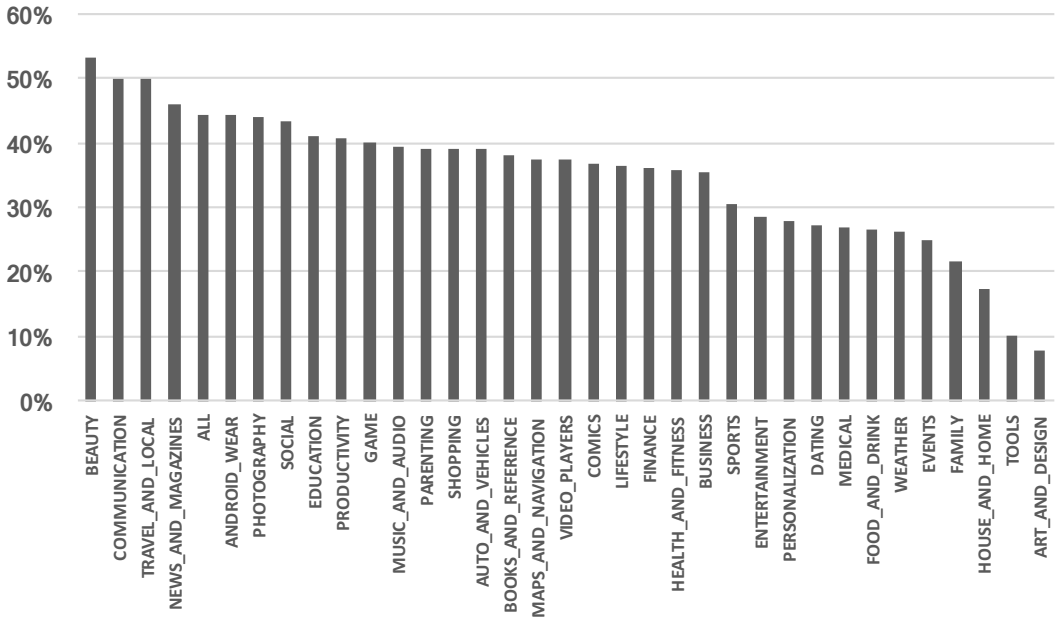
in `OkHttp` for versions 2.x before 2.7.4, and 3.x before 3.1.2, that allows an attacker to bypass the certificate pinning (CVE–2016–2402) [158].

We apply LibID, LibScout, Orlis, and LibPecker to check if a vulnerable version of the `OkHttp` library is used in popular apps. The results are presented in Table 3.5. All tools find many popular apps use at least one version of vulnerable `OkHttp` libraries. In particular, LibID identifies 393 affected apps, while the figure for LibScout and LibPecker is 296 and 339, respectively. All tools but Orlis agree on 294 apps, but the remaining list of affected apps differs. We notice that `OkHttp` contains a field marking the actual version of the library, and this field has not been obfuscated in the majority of our test apps. Therefore, we only need a little manual effort to verify the results. Note that LibID does not use string constants as a feature to detect libraries. Thus, the version information of `OkHttp` in the app binaries only helps us to confirm the result instead of assisting LibID directly. For those apps that LibID and LibScout disagree, all 99 additional apps reported by LibID are true positives. Meanwhile, LibScout uniquely reports 2 vulnerable apps, and one of them is a false positive; the other one is missed by LibID. We further analyse the app that LibID fails to detect. It turns out the app uses both an obfuscated version of `OkHttp` 2.4.0 (vulnerable) and an unobfuscated version of `OkHttp` 3.4.1 (patched). During the matching process, LibID mismatches the `OkHttp` 2.4.0 library to the `OkHttp` 3.4.1 package but the matching confidence does not reach the threshold. As a result, this vulnerable app is missed by LibID.

Overall, the results demonstrate the accuracy and robustness of LibID, but it also raises security concerns about the Android platform. Although the vulnerability of `OkHttp` was reported in February 2016, around 10% of popular apps on the Google Play dataset still use an insecure version of the `OkHttp` library. Figure 3.7 presents the detailed distribution of vulnerable apps in our dataset. More than 30% of apps in two categories (`NEWS_AND_MAGAZINES` and `TRAVEL_AND_LOCAL`) and at least 10% of apps in 25 categories use a vulnerable version of `OkHttp`. Meanwhile, a large proportion of `OkHttp` libraries used in popular apps are unpatched, including more than 44% of `OkHttp` libraries in the top 100 free apps and the other six categories. We also discover that three of Google's apps use vulnerable versions of `OkHttp` libraries, all of which are installed by millions of users. Although we confirm that these three apps do not invoke the vulnerable certificate pinning



(a) The proportion of apps using vulnerable `OkHttp` in each category



(b) The proportion of `OkHttp`-included apps using vulnerable `OkHttp` in each category

Figure 3.7: Categorical summary of apps using vulnerable `OkHttp`

method, it is still best practice to upgrade the library version to ensure vulnerable code is not introduced in a future release. Finally, we sent emails to the developer of the reported 393 vulnerable apps on December 23, 2017. As a result, we received 22 non-automatic replies. Among these replies, 17 said they will investigate, 3 have updated the `OkHttp` library, and 2 promised to update it in the future release.

3.4 Discussion

Comparison. In the previous section we have quantitatively evaluated the performance of LibID using three different datasets. However, the results may not be intuitive and could take some time to interpret. Here, we present an intuitive overview of the capabilities of LibID and other representative library fingerprinting tools in Table 3.6. The training and detection time in Table 3.6 are inferred qualitatively based on the complexity and categorised into three levels (Low, Medium, High). Future library fingerprinting applications can choose the most suitable tool according to their resources and requirements by referring to this table.

Compared with other work listed in Table 3.6, we have proposed a new library match index that takes into account the majority of library code could be deleted during code obfuscation. For accurate version pinpointing, LibID utilises fine-grained features such as the dependency graph that is often time-consuming. LibID novelly converts the dependency graph matching problems to BIP problems and applies LSH to speed up the matching process. Furthermore, we are the first to design the *relation pruning* and *ghost hunting* processes to support cases when both class repackaging and code shrinking are applied.

Limitations. LibID selects several features to profile and detect libraries. These features, however, are not robust against more advanced obfuscation techniques. For instance, the *class access flag* and *class interfaces* fields can be modified if either "allowaccessmodification" or "mergeinterfacesaggressively" option is enabled in ProGuard. API hiding techniques also allow hiding calls to the Android SDK through Java reflection. Nevertheless, these cases are relatively rare compared with code shrinking and package modification operations (§3.3.2). The documentation of ProGuard also suggests users exploit these advanced features cautiously because they can reduce the performance and may cause JVM problems.[7] The aim of LibID is to be the first to reliably detect libraries under the most popular obfuscation functions, including code shrinking, identifier renaming, control flow randomisation, and package modifications. We achieve this goal.

One major concern of LibID is the scalability. Using library fingerprinting on unobfuscated F-Droid apps as an example (§3.3.2), we calculate the average time spent on fingerprinting a library, fingerprinting an app, and matching an app with libraries for each tool. The results are presented in Table 3.7. Although LibID is more time-efficient than Orlis and LibPecker, it spends more time than LibScout. This is because LibID performs a finer-grained analysis compared with LibScout. Both the CFG construction and candidate class matching are time-consuming. The BIP solver could also take a long time

[7] `https://www.guardsquare.com/en/products/proguard/manual/usage`

Table 3.6: Comparison of different library fingerprinting tools

Tool	Version Identification	Obfuscation Resilience	Shrinking Resilience	Class Repackaging Resilience	Training Time	Detection Time	Open Source
Wukong	○	●	○	○	M	L	●
LibRadar	○	●	○	○	M	L	●
LibD	○	●	○	○	H	M	●
LibSift	○	●	◐	○	H	M	○
Ordol	●	●	◐	◐	–	M	○
LibScout	●	●	◐	○	–	L	●
LibID-S	●	●	◐	◐	–	M	●
LibID-A	●	●	●	●	–	H	●

● = provides property; ◐ = partially provides property; ○ = does not provide property

Table 3.7: Average time consumption

Tool	Library Fingerprinting	App Fingerprinting	Matching
Orlis	5.18 s	-	850.30 s
LibScout	1.01 s	-	1.63 s
LibPecker	-	-	509.40 s
LibID-S	4.13 s	38.36 s	2.26 s
LibID-A	4.13 s	38.36 s	11.25 s

to find a solution if there are too many constraints. As discussed in §2.5, BIP is a classic NP-complete problem. Although we did not encounter any situation where a solution could not be found in a day during our evaluation, we do not rule out the possibility that there might be some app and library pairs that could produce a way more complex matching graph and many more constraints than those in our evaluation datasets, in which case the BIP solver may not be able to find a solution in meaningful time. Nevertheless, the use of *LSH Ensemble* has greatly reduced the matching time by avoiding pair-wise comparison and we design two working schemes to prioritise either scalability or accuracy. As seen in Table 3.7, LibID-S spends much less time in the matching process because it does not make use of the dependency graph. In addition, for large-scale analysis, we can run LibID in parallel on a computing cluster since each task is independent. In general, LibID-A is preferred for small-scale analysis, or large-scale analysis when there are enough computing resources; LibID-S is favoured in cases where the computational cost of LibID-A is too high, or even infeasible.

3.5 Summary

In this chapter we presented LibID, a reliable tool that identifies third-party Android libraries and their version in app binaries. LibID includes two detection schemes: LibID-S and LibID-A, which focus on scalability and accuracy, respectively. We converted the abstract library fingerprinting problem into a BIP problem that has been well studied in the research community. Overall, LibID overcomes several limitations found in previous work and is able to determine the version of third-party libraries used in an app binary using identifier renaming, code shrinking, control flow randomisation, and package modification techniques. Evaluation was supported by our novel method that semi-automatically generates apps containing third-party libraries, as well as an analysis of hundreds of F-Droid apps. This provided valuable ground truth data to support accurate evaluation and comparison of our approach to previous work. Our experiments showed that LibID can detect a higher proportion of libraries than prior art, especially when code shrinking was enabled and the package hierarchy was modified. Finally, we demonstrated the utility

of LibID by detecting the use of a vulnerable version of the `OkHttp` library in nearly 10% of 3 958 most popular apps on the Google Play Store.

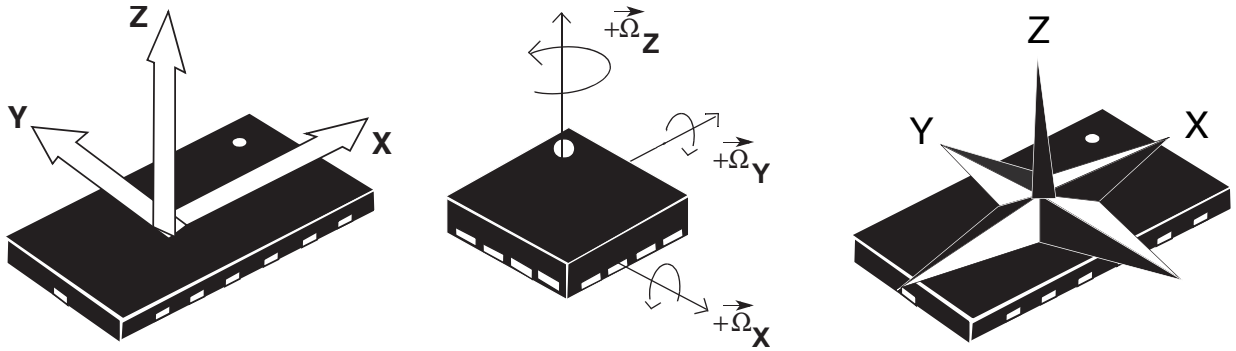
CHAPTER 4

Methodology of factory calibration fingerprinting attack

This chapter describes a new type of device fingerprinting attack, the *factory calibration fingerprinting* attack. Our attack uses data gathered from the accelerometer, gyroscope, and magnetometer sensors found in smartphones to construct a globally unique fingerprint. When we started this work, developers can access these sensors from both a native mobile app and a mobile website without asking for user permission. Therefore, they were good candidates for device fingerprinting. While we have focused our analysis on motion sensors, we anticipate that a calibration-based fingerprint can also be generated for other sensors across many different devices, including the camera, touchscreen, and battery.

The goal of the adversary is to obtain a reliable fingerprint from the built-in motion sensors of a smartphone. Our threat model is as follows. We assume an adversary is able to record motion sensor samples from a smartphone. The attacker can do this if the user installs an app, or visits a website (accelerometer and gyroscope only), under the control of the attacker. Furthermore, we assume that the software embedded in the app or web page is able to communicate with a remote server under the control of the attacker; this is typically the case for both apps and web pages.

This chapter explains this new type of attack using the gyroscope in iOS Devices as an example; the factory calibration fingerprinting for other sensors and devices is discussed in Chapter 5. In particular, we introduce the background of motion sensor calibration (§4.1) and propose a practical technique to identify if a sensor has been factory calibrated (§4.2). For sensors that are factory calibrated, we propose a method to estimate their nominal gain if the datasheet is unavailable (§4.3). We also analyse the motion sensor data representation in iOS and discuss the implications it has for the factory calibration fingerprinting attacks (§4.4). Then, we give details about the steps to generate the factory calibration fingerprint from within an app (§4.5) and from within a website (§4.6). Finally,



(a) Accelerometer **(b)** Gyroscope **(c)** Magnetometer

Figure 4.1: Illustration of MEMS motion sensors

we discuss practical options to launch the factory calibration fingerprinting attack (§4.7) and summarise this chapter (§4.8).

Contributions. This chapter makes the following academic contributions:

1. We describe steps to verify if a motion sensor is factory calibrated from its output.
2. We present an approach to estimate the nominal gain of motion sensors.
3. We show there are only 16 bits of data in the fractional part of motion sensor output in iOS devices.
4. We introduce a novel technique to estimate the gain matrix of motion sensors by studying the sensor output alone; we show the technique works on both mobile apps and websites.
5. We elaborate on options to generate a calibration fingerprint of a device in practical situations.

4.1 Motion sensor calibration

Motion sensors used in modern smartphones, including the accelerometer, gyroscope, and magnetometer, are based on MEMS (Micro-Electro-Mechanical Systems) technology and use microfabrication to emulate traditional mechanical parts. These sensors are pervasive in modern mobile devices. In earlier versions of iOS devices, such as iPhone 4, motion sensors are integrated into different MEMS chips. However, the latest Apple devices have a six-axis Inertial Measurement Unit (IMU), which comprises a triaxial accelerometer and a triaxial gyroscope in a small package. Figure 4.1 presents an illustration of MEMS motion sensors. Details about these sensors are listed below.

Table 4.1: List of iOS devices that do not have a gyroscope

Category	Model
iPhone	iPhone 2G, iPhone 3G, iPhone 3GS
iPad	iPad 1
iPod Touch	iPod Touch 1, iPod Touch 2, iPod Touch 3

Accelerometer An accelerometer measures the velocity change of a device in each of the axes. The value can be either positive or negative based on the direction of acceleration. All iOS devices and Apple Watches are equipped with at least one triaxial accelerometer [159]. In particular, there is an additional triaxial accelerometer in addition to the IMU in the iPhone 6 to minimise power consumption [160].

Gyroscope A gyroscope measures the rotation speed of a device in each of the axes. The value can be either positive or negative based on the direction of rotation. Most iOS devices and all Apple Watches are equipped with one triaxial gyroscope [161]. Concretely, Table 4.1 lists all the iOS devices that do not contain a gyroscope.

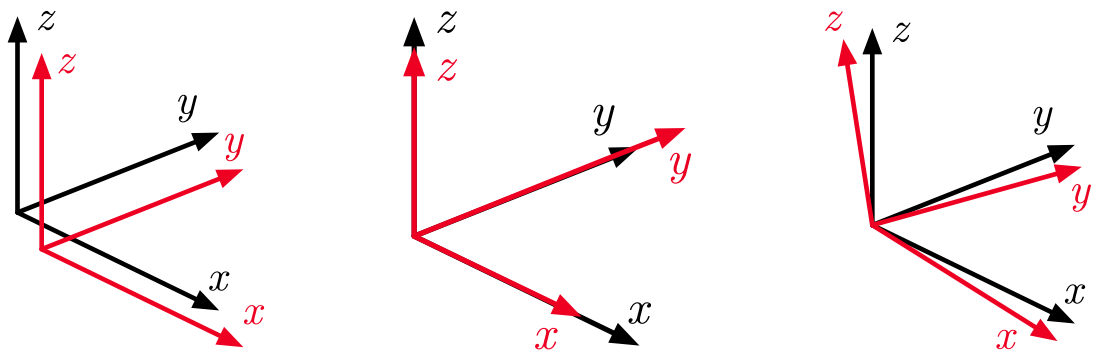
Magnetometer A magnetometer measures the Earth's magnetic field relative to the device. The value can be either positive or negative based on the direction of the magnetic field. All Apple Watches and iOS devices apart from the iPod Touch lineup, iPhone 2G, and iPhone 3G have a triaxial magnetometer [162].

Although MEMS technology has greatly reduced the size and cost of motion sensors, MEMS sensors are usually less accurate than their optical counterparts due to various types of error. In general, these errors can be categorised as *deterministic* and *random*: random errors are usually caused by electronic noise interfering with the output of sensors, which change over time and have to be modelled stochastically; deterministic errors are produced by manufacturing imperfections and can be classified into three categories: *bias*, *scaling*, and *nonorthogonality misalignment errors* [163, 164]. These deterministic errors can be defined as follows:

Bias error Bias error manifests as the deviation between the actual and nominal value at a known reference point when there is no external force applied to the sensor.

Scaling error Scaling error can be identified as the difference between the actual and nominal scale, or sensitivity, of each sensor axis.

Misalignment error Misalignment error occurs when the three axes of the sensor are not orthogonal to each other due to mounting point variations during manufacture.



(a) Bias error **(b)** Scaling error **(c)** Misalignment error

Figure 4.2: Illustration of deterministic errors

An illustration of these errors is presented in Figure 4.2, where black and red lines represent the nominal and actual values respectively. Calibration aims to identify and compensate for the deterministic errors from the sensor. Many commercial sensors are factory calibrated and their calibration parameters are stored in firmware or non-volatile memory, providing accurate measurements off-the-shelf [165]. In the context of mobile devices, the main benefit of per-device calibration is that it allows more accurate attitude estimation [166]. By contrast, sensors embedded in low-cost smartphones are usually poorly calibrated due to the high cost and complexity of factory calibration [167]. For an individual manufacturer, the choice of sensor calibration is, therefore, an engineering trade-off.

MEMS sensors usually convert and store the analogue measurement in a digital register through an Analogue-to-Digital Converter (ADC) module. For a triaxial motion sensor, let $\mathbf{A} = [A_x, A_y, A_z]^T$ be the sensor ADC output. Considering all three kinds of deterministic errors, the calibration of the motion sensor can be represented by the following equation [168]:

$$\begin{bmatrix} O_x \\ O_y \\ O_z \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix} \begin{bmatrix} 1 & N_{xy} & N_{xz} \\ N_{yx} & 1 & N_{yz} \\ N_{zx} & N_{zy} & 1 \end{bmatrix} \begin{bmatrix} A_x + B_x \\ A_y + B_y \\ A_z + B_z \end{bmatrix} \tag{4.1}$$

where $O_i \in \mathbf{O}$ is the calibrated output; $S_i \in \mathbf{S}$ is the scale factor; $N_{ij} \in \mathbf{N}$ represents the nonorthogonality between axis i and j; and $B_i \in \mathbf{B}$ is the bias. A sensor's *sensitivity*, or *gain*, is defined as the ratio between the output signal and measured property. A sensor's nominal gain is the intended operating sensitivity of the sensor. It is a single value that is usually documented in the sensor datasheet. We use F to denote a sensor's nominal gain in this dissertation. If a sensor is ideal, its scale matrix $\mathbf{S}$ and nonorthogonality matrix $\mathbf{N}$ should be $F \cdot \mathbf{I}$ and $\mathbf{I}$, respectively, where $\mathbf{I}$ is an identity matrix. However, due to the existence of errors, there may be a non-negligible difference between the actual and

nominal gain [169]. Equation 4.1 can be further simplified as:

$$\mathbf{O} = \mathbf{G}(\mathbf{A} + \mathbf{B}) \tag{4.2}$$

where $\mathbf{G} = \mathbf{SN}$ is referred to as the gain matrix.

A myriad of calibration techniques have been proposed to calculate the gain matrix and bias vector during manufacture [163]. Vendors can also choose to only calibrate the bias vector to lower the cost. Once factory calibration is completed, the calibration parameters of the sensor will be stored in non-volatile memory inside the device and do not change over time [170, 171]. Details of the calibration process used by manufacturers are generally trade secrets and are not made public.

4.2 Factory calibration in mobile devices

Both Android and iOS provide APIs to access the *raw* and *fused* motion sensor data. Web developers can also access the *fused* accelerometer and gyroscope data via JavaScript. According to the Android documentation, the *raw* or *uncalibrated* data is the sensor output after factory calibration and temperature compensation, while the *fused* or *calibrated* sensor data API applies bias compensation and noise correction on the raw measurements.[1] On iOS, it is less clear from the documentation whether the raw data provided by iOS APIs is factory calibrated. To investigate this, we collect a batch of raw gyroscope data from both an iPhone X and a Samsung Galaxy S8. Both handsets are placed on a flat desk and stay still during data collection.

Figure 4.3 (a) presents the raw gyroscope measurements collected from both devices. Notably, the quantisation in gyroscope outputs is clear in the figure. This is because the outputs of the gyroscope ADC module are integers. Taking the difference between two sensor readings directly reveals the gain of the sensor. According to Equation 4.2, the difference between two sensor outputs, $\Delta\mathbf{O}$, can be calculated by:

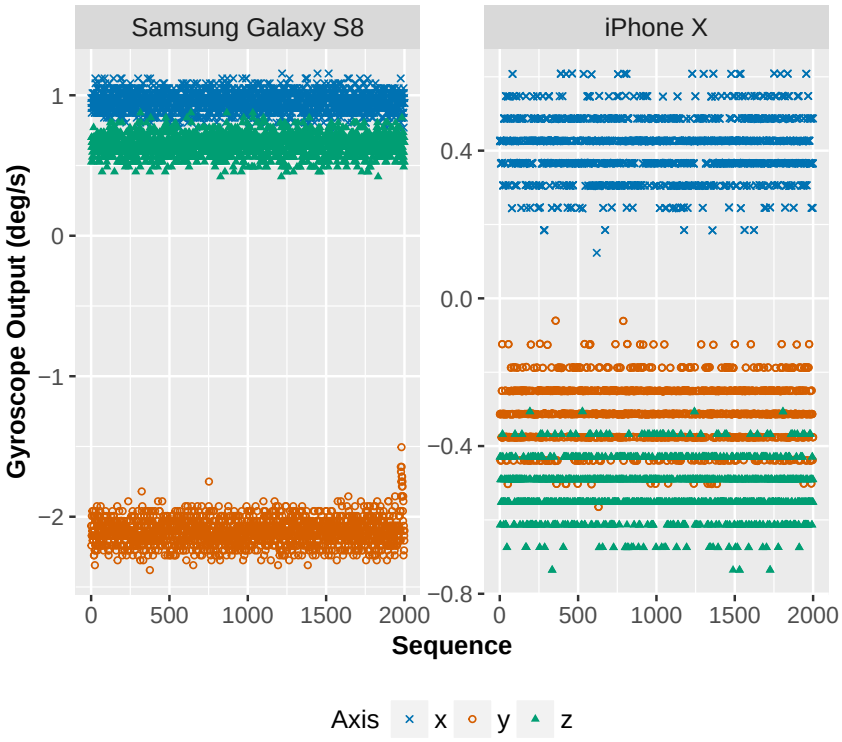
$$\Delta\mathbf{O} = \mathbf{G}\Delta\mathbf{A} \tag{4.3}$$

where $\Delta\mathbf{A}$ is the difference between the corresponding ADC outputs.

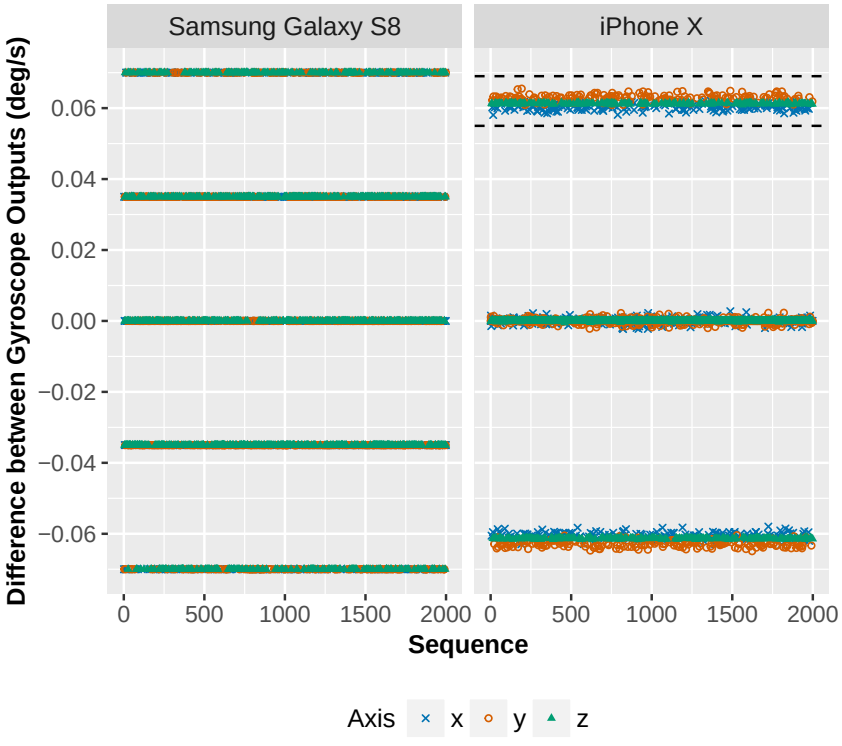
If there is no factory calibration, the gain matrix $\mathbf{G}$ has a value of $F_G \cdot \mathbf{I}$, where F_G is the nominal gain of the gyroscope and $\mathbf{I}$ is the identity matrix. In this case, we have:

$$\begin{bmatrix} \Delta O_x \\ \Delta O_y \\ \Delta O_z \end{bmatrix} = F_G \begin{bmatrix} \Delta A_x \\ \Delta A_y \\ \Delta A_z \end{bmatrix} \tag{4.4}$$

[1] `https://developer.android.com/guide/topics/sensors/sensors_motion.html`



(a) Raw gyroscope measurements



(b) Consecutive differences between gyroscope measurements

Figure 4.3: Raw gyroscope data collected from a Samsung Galaxy S8 and an iPhone X

Since ΔA_i is an integer, ΔO_i should be a multiple of the nominal gain. Figure 4.3 (b) shows the difference between consecutive measurements for each of the three axes. To clearly present the results, the figure only presents a small range of data. As seen in Figure 4.3 (b), the difference between gyroscope outputs of the Samsung Galaxy S8 is always a multiple of a constant value (nominal gain). By contrast, the difference for the iPhone X is not a single multiple. Figure 4.3 (b) also reveals that each axis of the gyroscope in the iPhone X has a different gain and bias. In particular, the `x` axis has a slightly lower gain than the `y` and `z` axes.

Overall, Figure 4.3 demonstrates two things: (i) the gain matrix of the gyroscope in the iPhone X is factory calibrated while the one in the Samsung Galaxy S8 is not; and (ii) the iOS API for accessing raw motion sensor data obtains the factory-calibrated data. We further implement the same experiment on other iOS device models and confirm their gyroscope is also factory calibrated. The factory calibration of motion sensors in Android devices is discussed in §5.3.

4.3 Nominal gain estimation

In general, manufacturing imperfections introduce idiosyncrasies across different sensors. Therefore, if factory calibration is carried out on a per-device basis, it is likely the calibration matrices, including the gain matrix $\mathbf{G}$ and the bias matrix $\mathbf{B}$, are also unique and we can use either of these matrices as a device fingerprint. In this dissertation we focus on recovering the gain matrix $\mathbf{G}$.

We first investigate fingerprinting devices from mobile apps, where raw sensor data is accessible. To estimate the gain matrix $\mathbf{G}$ of a sensor, we first need to know its nominal gain. For some earlier iOS devices, such as iPhone 4, the nominal gain of the gyroscope, 70 millidegrees per second (mdps), is specified in the datasheet. Although we could not find the gyroscope specification for recent iOS device models, we propose an approach to estimate the nominal gain from gyroscope measurements.

Equation 4.3 shows that the bias $\mathbf{B}$ can be effectively eliminated by taking the difference between two sensor outputs. From Figure 4.3 (b), we observe that the actual gain, or sensitivity, of each axis is in close proximity to each other (or to the nominal gain) and the fluctuation within each band is small compared with the large gap between different bands. This implies that the actual gain matrix is close, but not equal, to the ideal gain matrix, $F_G \cdot \mathbf{I}$. Since the iPhone X was resting on a desk during data collection, the difference between consecutive ADC outputs ΔA_i should be small (e.g., 0, ± 1). To estimate the nominal gain, F_G, of the iPhone X, we find the band with only positive values that is closest to 0 but its range does not include 0. In the case of Figure 4.3 (b), a qualified band is the one inside dashed black lines. Then, the average of all the values inside the band,

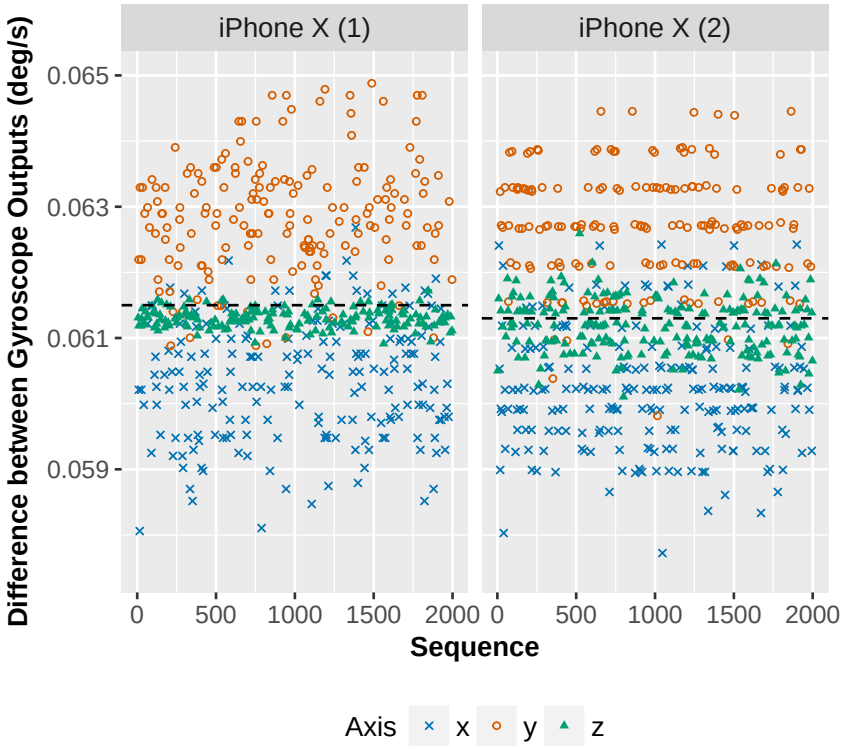


Figure 4.4: Estimation of the gyroscope nominal gain from two iPhone X devices. The dash line marks the estimated nominal gain

which includes data from all three axes, can be used as a reliable estimate of the nominal gain of the iPhone X. Note that this approach requires the device is stationary (e.g., at rest on a desk) during measurement so that we have enough data points with $\Delta A_i = \pm 1$. This is usually not a problem since we only need to estimate the nominal gain once for each device model.

Figure 4.4 presents the estimated gyroscope nominal gain from two iPhone X devices. Both devices were at rest on a desk during the data collection. For clarity, Figure 4.4 only presents the range of data inside a qualified band. As shown in Figure 4.4, the difference between the estimated gyroscope nominal gain from these two devices is tiny. Therefore, we can use the estimated nominal gain from one device to estimate the gain matrix of another device. It is also clear from Figure 4.4 that the degree of fluctuations around the nominal gain is different for these two devices, which indicates that their gain matrix is different and may be used as a fingerprint.

Table 4.2 lists the nominal gain (in mdps) of the gyroscope for all iOS devices that we have measured. Note that we have observed two possible nominal gain values for iPhone 5S devices. This may be because some iPhone 5S devices use a different IMU model from others. In addition, the estimated nominal gain of 61 mdps indicates that the sensor is likely configured to a measurement range of ± 2000 dps and a resolution of 16 bits ($4000/2^{16} \approx 0.061$).

Table 4.2: Estimated gyroscope nominal gain for iOS devices

Model	Nominal Gain (mdps)
iPhone 5S*/6/6 Plus/6S/6S Plus/7/7 Plus/8/8 Plus/SE iPhone X/XS/XS Max iPad Pro 9.7/10.5/12 inch	61
iPhone 4/4S/5/5C/5S*, iPad 3/Mini/Mini 4/Mini Retina/Air/Air 2	70

* iPhone 5S devices have two possible nominal gain values.

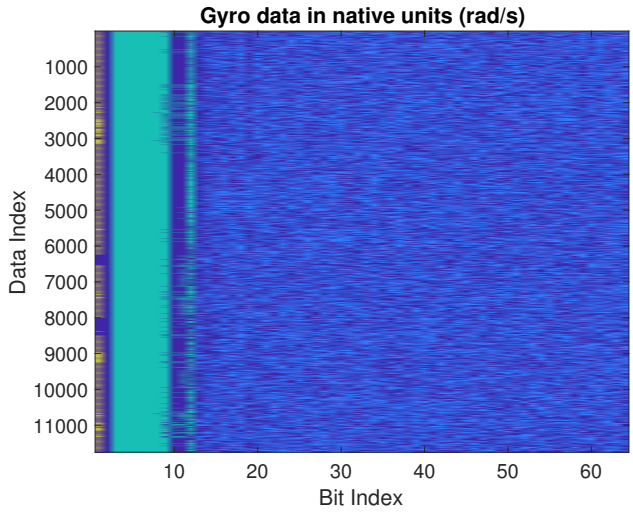
4.4 Data representation of sensor outputs

Before looking into the gyroscope fingerprinting technique, we first investigate the data representation format used by the hardware. We start by looking at the binary representation of the raw sensor data. In particular, we collect about 4K raw gyroscope samples from an iPad Air 2. Each sample is a 3-tuple consisting of triaxial measurements; the native unit provided by the iOS SDK is radians per second (rad/s). A visualization of the raw gyroscope data, in the IEEE 754 floating-point representation, is shown in Figure 4.5 (a).

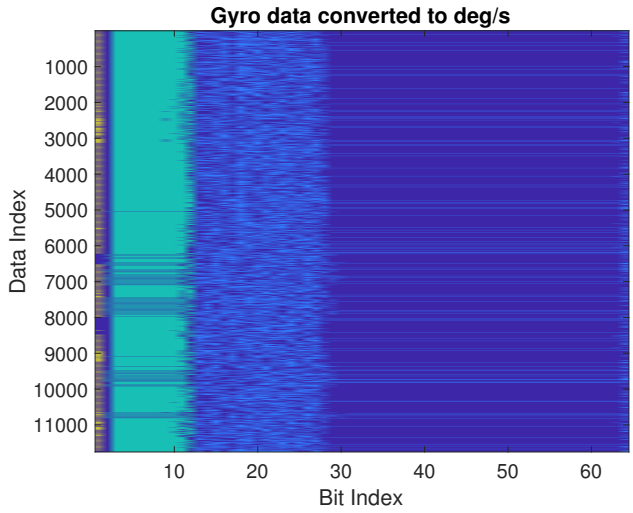
Here, the colours show the various parts of the IEEE 754 representation: bit 1 (yellow) is the sign bit, bits 2–12 (green) are the exponent, and bits 13–64 (blue) are the significand. Overall, no obvious structure is observed in this form. However, if we convert the data into units of degrees per second (deg/s), then we obtain the results shown in Figure 4.5 (b).

It is now obvious that the internal calculations have much less precision than is available in IEEE 754 representation and must be in the units of degrees per second. We gain some more information if we convert the double-precision numbers, in degrees per second (dps), into fixed-point Q32.32 form. Results are presented in Figure 4.5 (c).

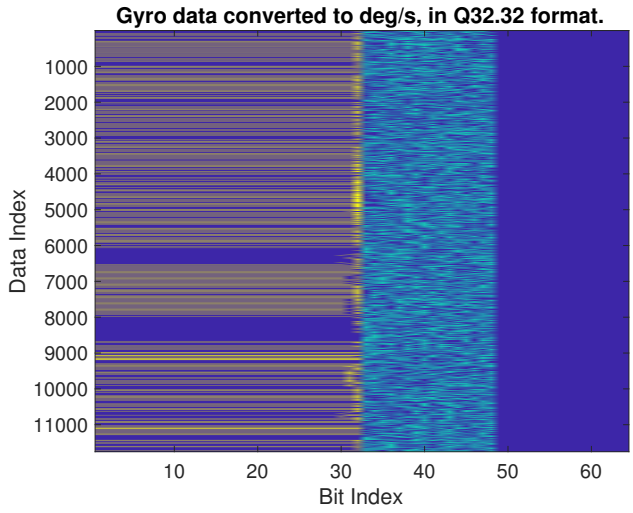
In Figure 4.5 (c), the first 32 bits contain the integer part of the data (two's complement) and the last 32 bits show the fractional part. It is obvious that there are only 16 bits of data in the fractional part. The 16-bit resolution of gyroscope outputs is observed on all the iOS devices we have tested. There are a few possible reasons for this, but the simplest is that the value in the gain matrix $\mathbf{G}$ is stored as a signed integer with a resolution of 2^{-16} dps. After investigation, we find that every device that uses an M-series motion coprocessor, which was released by Apple in September 2013 with the iPhone 5S, shows this pattern. The purpose of the motion coprocessor is to offload the collection and processing of sensor data from the CPU. However, for older devices such as iPhone 4 and iPhone 4S, gain matrix values are stored with more precision and the calibration involves truncation down to 2^{-16} dps after the gain is applied. The complete set of devices that



(a) rad/s, IEEE 754 format



(b) deg/s, IEEE 754 format



(c) deg/s, Q32.32 format

Figure 4.5: Different binary representations of the raw gyroscope data

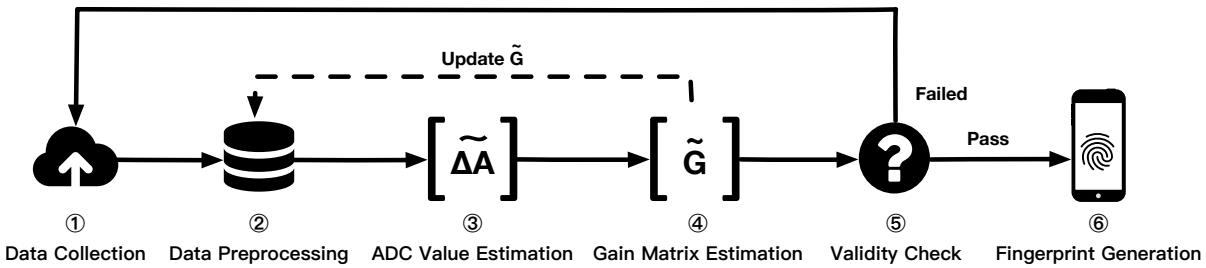


Figure 4.6: General steps to recover the factory calibration fingerprint

use the M-series motion coprocessor can be found online[2]. As discussed in §4.5, the lack of resolution in these devices allows us to precisely recover the gain matrix $\mathbf{G}$.

4.5 Sensor fingerprinting from a mobile app

The general process to generate the factory calibration fingerprint of a sensor is illustrated in Figure 4.6. It consists of six major steps: *data collection*, *data preprocessing*, *ADC value estimation*, *gain matrix estimation*, *validity check*, and *fingerprint generation*. In this section, we first present a basic approach to generate the gyroscope fingerprint (GyroID), which works well when the device is stationary. Then, we propose an improved scheme that can reliably generate the GyroID even when the device is moving.

4.5.1 Basic approach

We first consider the case where the device is stationary or moving slowly during sampling.

Data collection. We collect a small number of samples from the gyroscope through a mobile app at the maximum sampling frequency. We use $\mathbf{O} = [\mathbf{O}_1, \mathbf{O}_2, \cdots, \mathbf{O}_N]$ to denote the collected data, where $\mathbf{O}_i = [O_{i_x}, O_{i_y}, O_{i_z}]^T$ is a 3-by-1 vector. Empirically, we find $N = 100$ samples collected in less than 1 second is sufficient.

Data preprocessing. After collecting the data, we calculate $\Delta\mathbf{O}$ by differencing the consecutive outputs for all three axes. In other words, $\Delta\mathbf{O}$ is calculated by:

$$\Delta\mathbf{O} = [\mathbf{O}_2 - \mathbf{O}_1, \mathbf{O}_3 - \mathbf{O}_2, \cdots, \mathbf{O}_N - \mathbf{O}_{N-1}]$$

ADC value estimation. In this step, we aim to recover $\Delta\mathbf{A}$, which is the difference between consecutive ADC outputs. From Equation 4.3 we know:

$$\Delta\mathbf{A} = \mathbf{G}^{-1}\Delta\mathbf{O} \tag{4.5}$$

[2] `https://en.wikipedia.org/wiki/Apple_motion_coprocessors`

where $\mathbf{G}^{-1}$ is the inverse of the gain matrix $\mathbf{G}$. However, the value of $\mathbf{G}$ is unknown at the moment. Nevertheless, we can estimate $\mathbf{G}$ by its ideal value $\mathbf{G}_0 = F_G \cdot \mathbf{I}$, where F_G is the nominal gain of the gyroscope. Since the deterministic errors are comparatively small, $\mathbf{G}$ should be relatively close to $\mathbf{G}_0$ (§4.3). Because $\Delta\mathbf{A}$ only has integer values, we can estimate $\Delta\mathbf{A}$ by:

$$\widetilde{\Delta\mathbf{A}} = \text{round}(\mathbf{G}_0^{-1}\Delta\mathbf{O}) \tag{4.6}$$

where the round$(\cdot)$ function rounds each element to the nearest integer. However, since $\mathbf{G}_0$ is not equal to $\mathbf{G}$, the rounded value $\widetilde{\Delta\mathbf{A}}$ may not be the true value. Therefore, we further calculate the rounding error $\mathbf{E}^r_{\Delta\mathbf{A}} \in \mathbb{R}^{3\times(N-1)}$, which is defined as follows:

$$\mathbf{E}^r_{\Delta\mathbf{A}} = |\widetilde{\Delta\mathbf{A}} - \mathbf{G}_0^{-1}\Delta\mathbf{O}| \tag{4.7}$$

To ensure the estimated values are correct, we require that every value in $\mathbf{E}^r_{\Delta\mathbf{A}_i}$, which means the i-th column in $\mathbf{E}^r_{\Delta\mathbf{A}}$, be lower than a threshold Γ (e.g., 0.1). If not, we believe the rounding is ambiguous and remove both $\widetilde{\Delta\mathbf{A}}_i$ and $\Delta\mathbf{O}_i$ from the dataset. Once all ambiguous values are removed, we will regard $\widetilde{\Delta\mathbf{A}}$ as a safe estimate of the true ADC value matrix $\Delta\mathbf{A}$.

Note that, this is only true if the device is stationary or moving slowly, in which case the absolute values of $\Delta\mathbf{A}$ are small. Otherwise, any rounding error could be accumulated and results in rounding to an incorrect integer value.

Gain matrix estimation. After estimating the ADC value matrix $\widetilde{\Delta\mathbf{A}}$, we can estimate the gain matrix by:

$$\widetilde{\mathbf{G}} = \arg\min_{\mathbf{G}} \left\|\mathbf{G}\widetilde{\Delta\mathbf{A}} - \Delta\mathbf{O}\right\|_2^2$$

where $\|\cdot\|_2^2$ is the squared Euclidean 2-norm function. Or in words, we use the least squares solution to $\mathbf{G}\widetilde{\Delta\mathbf{A}} = \Delta\mathbf{O}$ as the gain matrix estimate.

Validity check. To quantify the deviation between $\widetilde{\mathbf{G}}$ and the true value of $\mathbf{G}$, we define the estimation error $\mathbf{E}^e \in \mathbb{R}^{3\times1}$ as follows:

$$\mathbf{E}^e = \frac{\left\|\Delta\mathbf{O} - \widetilde{\mathbf{G}}\widetilde{\Delta\mathbf{A}}\right\|_2}{N-1}$$

If the estimation error is small (i.e., $\max(\mathbf{E}^e) < \Theta$), then $\widetilde{\mathbf{G}}$ should be close to the true gain matrix $\mathbf{G}$.

By way of an example, here is the $\widetilde{\mathbf{G}}$ that we estimated from an iPhone XS in the

units of radians per second (rps):

$$\begin{bmatrix} 0.001068460229340 & -0.000009587379924 & -0.000002929477199 \\ 0.000002929477199 & 0.001073520235411 & 0.000005858954398 \\ -0.000001065264436 & -0.000006657902725 & 0.001069525493776 \end{bmatrix}$$

Recall that the gain matrix of the iPhone XS, which uses Apple M12 coprocessor, only has 2^{-16} resolution in the units of dps. If we represent $\widetilde{\mathbf{G}}$ in the units of 2^{-16} dps we can observe that its values are extremely close to whole integers:

$$\begin{bmatrix} 4012.000000000001 & -35.999999999999318 & -10.999999999999677 \\ 11.000000000000174 & 4030.999999999999 & 21.999999999999631 \\ -3.999999999999980 & -25.000000000000011 & 4016.000000000000 \end{bmatrix}$$

This indicates that the gain matrix is stored in the units of 2^{-16} dps. In fact, we find that all iOS devices with an M-series coprocessor store the gain matrix this way. For these devices, we can simply round $\widetilde{\mathbf{G}}$ to obtain the true gain matrix $\mathbf{G}$. From here on, whenever we refer to a gain matrix (e.g., $\mathbf{G}$, $\mathbf{G}_0$, and $\widetilde{\mathbf{G}}$), its values are in the units of 2^{-16} dps by default.

For devices with a motion coprocessor, we also calculate the rounding error $\mathbf{E}^r_{\mathbf{G}} \in \mathbb{R}^{3\times 3}$ to ensure the rounding is safe:

$$\mathbf{E}^r_{\mathbf{G}} = |\widetilde{\mathbf{G}} - \text{round}(\widetilde{\mathbf{G}})|$$

We believe that the rounding is reliable if the maximum value in $\mathbf{E}^r_{\mathbf{G}}$ is lower than a threshold, such as 0.01. Otherwise, if any of these checks fail, it is likely that the device was moving during data collection. For this basic approach, we need to repeatedly collect another batch of data until the estimation error and rounding error are small enough.

Fingerprint generation. The gyroscope calibration fingerprint, GyroID, is generated differently based on whether the device has an M-series coprocessor.

If the device does have an M-series coprocessor, the GyroID is defined as follows:

$$\text{GyroID} = \text{round}(\widetilde{\mathbf{G}}) \ - \ \text{round}(\mathbf{G}_0) \tag{4.8}$$

Or in words, the GyroID is the gain matrix $\mathbf{G}$ after subtracting the nominal gain in the

units of 2^{-16} dps. For instance, the GyroID of the iPhone XS in previous example is:

$$\text{GyroID} = \begin{bmatrix} 14 & -36 & -11 \\ 11 & 33 & 22 \\ -4 & -25 & 18 \end{bmatrix}$$

For devices that do not contain a motion coprocessor, their GyroID is calculated by:

$$\text{GyroID} = \widetilde{\mathbf{G}} \; - \; \text{round}(\mathbf{G}_0) \tag{4.9}$$

because values in the gain matrix are not simply integers in the units of 2^{-16} dps in this case.

Summary. The basic approach illustrates the general idea and procedure to generate the GyroID. Overall, the calculations are light-weight and are easy to implement. However, the basic approach requires the device to be stationary or moving slowly during measurement. To address this problem, we propose an improved approach which takes device movement into consideration.

4.5.2 Improved approach

The drawback of the basic approach is that it may take a long time to generate the GyroID, because the approach will keep trying until the device is almost stationary. In Equation 4.6, we use a bootstrap value $\mathbf{G}_0 = F_G \cdot \mathbf{I}$ to estimate the value of $\Delta\mathbf{A}$. However, since $\mathbf{G}_0$ is not equal to $\mathbf{G}$, the rounded value $\widetilde{\Delta\mathbf{A}}$ may not be the actual value $\Delta\mathbf{A}$. In general, the difference between $\Delta\mathbf{O}$ and $\mathbf{G}_0\Delta\mathbf{A}$ will increase as elements in $\Delta\mathbf{O}$ gets bigger, leading to incorrect rounding (i.e., $\widetilde{\Delta\mathbf{A}} \neq \Delta\mathbf{A}$).

In the improved approach, we use the same processing steps as in §4.5.1 except for *data preprocessing*, *ADC value estimation*, and *gain matrix estimation*. The general idea is that, instead of feeding $\Delta\mathbf{O}$ directly into the algorithm (which might result in incorrect rounding), we update $\widetilde{\mathbf{G}}$ iteratively using data with different ranges. The general steps of the improved approach are also illustrated in Figure 4.6.

Data preprocessing. In this step, we first generate more data from $\Delta\mathbf{O}$ with small values because smaller values are less likely to introduce rounding errors. To do so, we can sort elements in $\Delta\mathbf{O}$ and then take the difference between adjacent elements; the resulting values are then likely to be small. In more detail, suppose $\Delta\mathbf{O} = [\Delta\mathbf{O}_1, \Delta\mathbf{O}_2, \cdots, \Delta\mathbf{O}_N]$, where $\Delta\mathbf{O}_i = [\Delta O_{i_x}, \Delta O_{i_y}, \Delta O_{i_z}]^T$ is a 3-by-1 vector. We first sort $\Delta\mathbf{O}$ based on the value of ΔO_{i_x} into ascending order. Here, we use $[\Delta\mathbf{O}]_x$ to denote the sorted array. Similarly, we sort $\Delta\mathbf{O}$ by the value of ΔO_{i_y} and ΔO_{i_z}, and denote the results as $[\Delta\mathbf{O}]_y$ and $[\Delta\mathbf{O}]_z$,

respectively. Then, we calculate $\Delta\Delta\mathbf{O}$ as follows:

$$\Delta\Delta\mathbf{O} = [\text{diff}([\Delta\mathbf{O}]_x)\ \text{diff}([\Delta\mathbf{O}]_y)\ \text{diff}([\Delta\mathbf{O}]_z)] \tag{4.10}$$

where the diff$(\cdot)$ function differences consecutive column vectors in a matrix. For instance, diff$(\Delta\mathbf{O})$ is calculated by:

$$\text{diff}(\Delta\mathbf{O}) = [\Delta\mathbf{O}_2 - \Delta\mathbf{O}_1, \cdots, \Delta\mathbf{O}_N - \Delta\mathbf{O}_{N-1}]$$

By subtracting similar vectors, $\Delta\Delta\mathbf{O}$ contains more data with smaller values. From Equation 4.5 we have:

$$\mathbf{G}^{-1}\text{diff}(\Delta\mathbf{O}) = \text{diff}(\Delta\mathbf{A})$$

where values in diff$(\Delta\mathbf{A})$ are all integers. Combined with Equation 4.10, it is clear that the result of $\mathbf{G}^{-1}\Delta\Delta\mathbf{O}$ should only contain integer values. Therefore, we can directly add $\Delta\Delta\mathbf{O}$ to the $\Delta\mathbf{O}$ dataset and our basic algorithm still applies (i.e., $\Delta\mathbf{O} \leftarrow [\Delta\mathbf{O}\ \Delta\Delta\mathbf{O}]$). This expansion can be implemented multiple times to generate more data with small values.

Then, we generate several batches of data from the expanded dataset based on the value range and update $\widetilde{\mathbf{G}}$ iteratively. In particular, each batch, $\Delta\mathbf{O}^{\epsilon_i}$, is a subset of $\Delta\mathbf{O}$ where the absolute value of all its elements is lower than a multiplication of the nominal gain. That is to say,

$$\Delta\mathbf{O}^{\epsilon_i} = \{\Delta\mathbf{O}_j \in \Delta\mathbf{O} \mid \max(|\Delta\mathbf{O}_j|) < (\epsilon_i + 0.5)F_G\}$$

where ϵ_i is the threshold for batch i. We start ϵ_i from 1 and double its value for each batch (i.e., $\epsilon_{i+1} = 2\epsilon_i$) until the batch is the same as $\Delta\mathbf{O}$. Then, we progressively feed $\Delta\mathbf{O}^{\epsilon_i}$ to the next step and update the value of $\mathbf{G}$ from the first batch to the last one.

ADC value estimation. In this step, instead of using $\mathbf{G}_0$ to estimate $\Delta\mathbf{A}$ and calculate the rounding error $\mathbf{E}^r_{\Delta\mathbf{A}}$, we use $\widetilde{\mathbf{G}}$ whose initial value is $\mathbf{G}_0$. In other words, Equation 4.6 and 4.7 should be updated to:

$$\widetilde{\Delta\mathbf{A}}^{\epsilon_i} = \text{round}(\widetilde{\mathbf{G}}^{-1}\Delta\mathbf{O}^{\epsilon_i})$$
$$\mathbf{E}^r_{\Delta\mathbf{A}^{\epsilon_i}} = |\widetilde{\Delta\mathbf{A}}^{\epsilon_i} - \widetilde{\mathbf{G}}^{-1}\Delta\mathbf{O}^{\epsilon_i}|$$

Gain matrix estimation. This step updates $\widetilde{\mathbf{G}}$ from each batch of data, $\Delta\mathbf{O}^{\epsilon_i}$. As seen in Figure 4.6, one major difference between the improved approach and the basic approach is that the former may go back to the *data preprocessing* stage after the *gain matrix estimation* step. In particular, after each update of $\widetilde{\mathbf{G}}$, the algorithm will check

whether $\Delta\mathbf{O}^{\epsilon_i} = \Delta\mathbf{O}$ holds true. If true, it means we have processed all the output data, and we will pass the estimated $\widetilde{\mathbf{G}}$ to the *validity check* process. Otherwise, the algorithm will go back to the *data preprocessing* stage with an updated $\widetilde{\mathbf{G}}$, and a new batch of data will be processed with $\epsilon_{i+1} = 2\epsilon_i$.

Summary. The improved approach updates the estimation of $\mathbf{G}$ iteratively from data within a small range to the whole dataset. By using the iteratively updated $\mathbf{G}$ to estimate ADC outputs, the improved approach reduces the error of each estimation. Thus, it is able to generate a GyroID even when the device is moving modestly. In general, the basic approach is useful to illustrate the idea. The improved approach introduces few additional computations and is much more reliable. Therefore, it is the preferred way to fingerprint devices in practical situations.

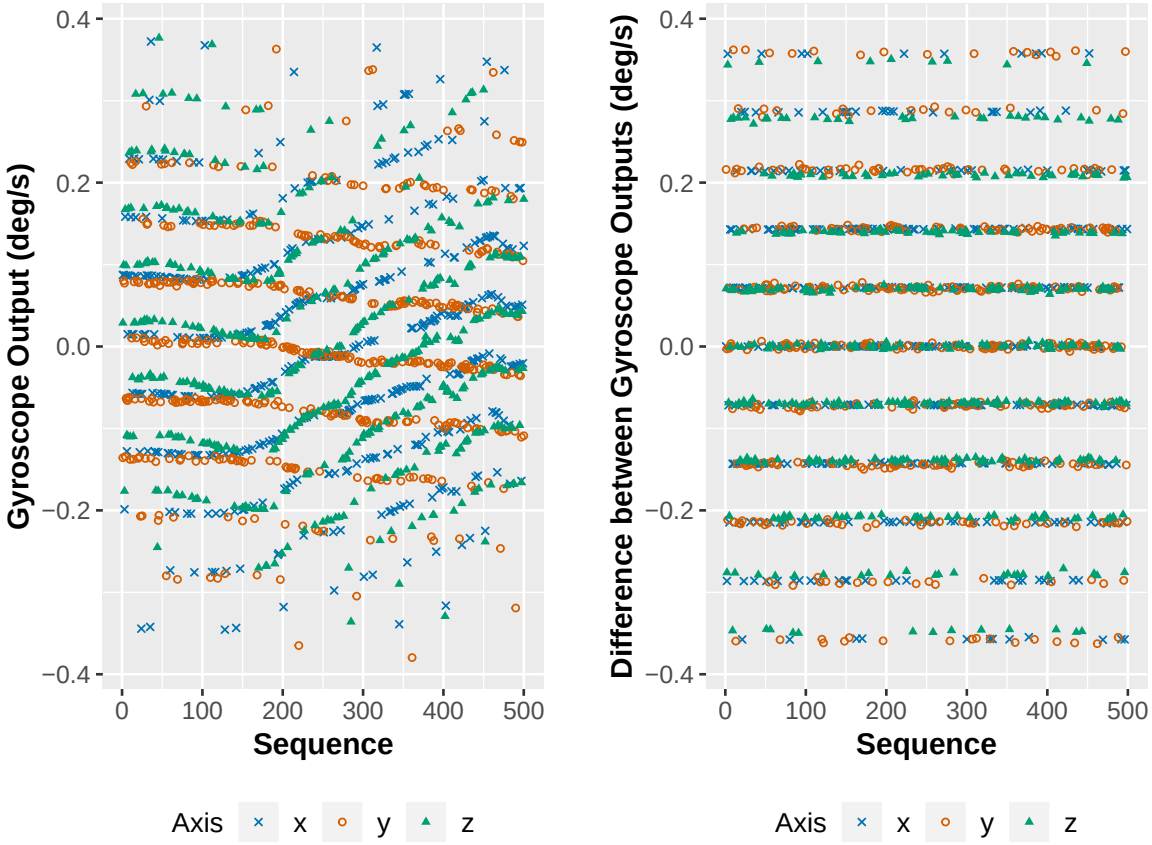
4.6 Sensor fingerprinting from a mobile website

JavaScript also provides APIs for web developers to access the *fused* gyroscope data. In this dissertation, we state the sensor data is *fused* if its value is a result of applying a time-variant bias correction to the raw sensor data. In practice, it is common to use a Kalman filter that combines, or *fuses*, the accelerometer and gyroscope inputs to calculate the corrected bias values. Figure 4.7 (a) presents the 500 sequential gyroscope samples collected from an iPad Air through mobile Safari when the device is at rest on a desk. As shown in Figure 4.7 (a), quantisation in the fused data is still visible because the gyroscope ADC outputs are integers. However, there is a slowly varying continuous component added to the bias due to the bias correction. Figure 4.7 (b) shows that the bias part can be nearly eliminated by subtracting consecutive samples. Therefore, we can apply the same technique described in §4.5 to recover the gain matrix.

4.7 Practical factory calibration fingerprinting

To launch a factory calibration fingerprinting attack, an adversary can collect gyroscope samples from any device using an app written by the adversary or that visits any website under the adversary's control. The adversary can then generate a device fingerprint (i.e., GyroID) from the samples and store it in a database. Later on, the adversary can query the database to determine when a particular physical device uses a particular app or visits a particular website. The details of the generation and query of the GyroID in the database differ depending on the collecting source (app or web) and device model.

Specifically, for apps running on a device with an M-series motion coprocessor, the adversary can follow the steps presented in §4.5.2 to recover the exact GyroID. Otherwise,



(a) Fused gyroscope data

(b) After differencing

Figure 4.7: Gyroscope data collected via JavaScript (iPad Air)

if the device does not have an M-series coprocessor (e.g., iPhone 4, iPhone 4S, and iPhone 5) or if adversaries only have access to fused gyroscope data (e.g., via JavaScript), there are two options to determine whether two GyroID entries in the database represent the same physical device:

Clustering An adversary can directly calculate the GyroID by Equation 4.9 and store it in a database. For every new device with the same model, the attacker can calculate its GyroID and compare the Euclidean distance between the GyroID and the ones in the databases. If they are close, then it is likely that they are the same device (the entropy of **G** is discussed in §5.5.1).

Rounding For devices without an M-series coprocessor, we find that if we still use Equation 4.8 to estimate the GyroID from different batches of data, there could be at most ±1 fluctuation for each of the 9 values in the GyroID. This is because the *validity check* process ensures that the estimation error is small. By way of an example, the estimated GyroID of an iPhone 5 from a mobile app is given in Figure 4.8. Here, values in each column are corresponding to the nine GyroID values estimated from one batch of data (five batches in total) using Equation 4.9

$$\begin{bmatrix}
-39.862363137944158 & -39.815935280377744 & -39.822619103792022 & -39.807082121390522 & -39.817704006855820 \\
36.746838108661962 & 36.725833955351305 & 36.751195821364234 & 36.752406611083359 & 36.751266990517692 \\
20.321937017717612 & 20.276563465382658 & 20.302206619012743 & 20.304280744909118 & 20.293839012434400 \\
-25.775903878528148 & -25.826990327452950 & -25.865651673449992 & -25.824470693110083 & -25.830236948855958 \\
19.070418546381916 & 19.087074948988629 & 19.124401486623356 & 19.120464786337834 & 19.117788287737312 \\
-29.059409132023614 & -29.082230352356362 & -29.029881179984351 & -29.043929037666555 & -29.056049474283022 \\
31.344396673396361 & 31.364984433906649 & 31.347831819184396 & 31.359581874031452 & 31.366435779636262 \\
-35.131464999438137 & -35.144073074775363 & -35.126381870264737 & -35.138736688905318 & -35.140984092442856 \\
-74.091971345986167 & -74.128105301813775 & -74.093537886235026 & -74.123911828207383 & -74.131598201240195
\end{bmatrix}$$

Figure 4.8: GyroID of an iPhone 5 estimated from five batches of data

(unit: 2^{-16} dps). Therefore, an adversary can simply store the estimated GyroID and perform a fuzzy query (i.e. accept a ± 1 fluctuation for each element), albeit this option provides less entropy.

4.8 Summary

In this chapter we introduced the factory calibration fingerprinting attack: a new method of fingerprinting devices with embedded motion sensors by careful analysis of the sensor output alone. We elaborated on the steps to recover the factory calibration parameters of motion sensors and demonstrated the effectiveness of this attack on the gyroscope in iOS devices. In addition, we studied the data representation of motion sensor output in iOS devices and found the lack of precision in the M-series coprocessor helps the generation of the calibration fingerprint. Our attack is easy to conduct by a website or an app in under 1 second, requires no special permissions, does not require user interaction, and is computationally efficient. Our attack can also be applied retrospectively to a historic archive of sensor data.

CHAPTER 5

Factory calibration fingerprinting for smartphones

In Chapter 4 we have focused exclusively on the gyroscope found in iOS devices to better explain the methodology of factory calibration fingerprinting attack. In this chapter, we discuss related work on motion sensor-based device fingerprinting (§5.1); we present our findings on the factory calibration of the accelerometer and magnetometer on iOS devices (§5.2); we conduct a large-scale factory calibration behaviour analysis on popular Android device models (§5.3); we compare the calibration fingerprint with the Fingerprintjs2 fingerprint for iOS devices (§5.4.1) and for Google Pixel phones (§5.4.2); we analyse the calibration fingerprint for vulnerable iOS and Google Pixel devices and estimate their entropy in §5.5.1 and §5.5.2, respectively.

We followed a coordinated disclosure procedure and reported the vulnerability to Apple on 3rd August 2018 and Google on 10th December 2018. In iOS 12.2, Apple adopted our suggestion and added random noise to sensor outputs (CVE-2019-8541). In addition, Apple removed access to motion sensors from Mobile Safari by default and in later versions also removed motion sensor access from WebKit. However, in this chapter we show that Apple's fix is imperfect by extracting the exact calibration fingerprint using more sensor data (§5.6).

Contributions. This chapter makes the following academic contributions:

1. We describe how factory calibration data can be extracted from the accelerometer, gyroscope, and magnetometer found on recent smartphones.

2. We demonstrate that the magnetometer and gyroscope calibration data together form a reliable fingerprint for iOS devices that does not change after factory reset or operating system update.

3. We collect motion sensor data from 870 iOS devices and show that our approach can generate a globally unique identifier; we show that the calibration fingerprint of iPhone 6S has about 67 bits of entropy.

4. We implement our approach as an iOS app and find the approach is lightweight and efficient: data collection and processing typically take less than one second in total.

5. We conduct a large-scale analysis on motion sensor calibration in 146 Android device models from 11 vendors. We find that all Google Pixel phones except for Pixel 1/1 XL can be fingerprinted by our attack; we show that the calibration fingerprint of Pixel 4/4 XL has about 57 bits of entropy.

6. We analyse Apple's fix and show that it is imperfect: with ~50K samples the exact fingerprint can be extracted.

5.1 Related work

Due to manufacturing imperfections, the same motion stimulus may result in different responses across motion sensors of the same make and model. Dey *et al.* found that the differences in accelerometer responses are consistent and can be used as a device fingerprint [16]. In particular, they used the embedded vibration motor in a smartphone to generate consistent motion stimulus. They then applied supervised learning to differentiating different smartphones using features extracted from the accelerometer response in both time and frequency domains. Their experiment demonstrated that such a fingerprint is viable. However, their approach is less effective in an uncontrolled environment and is susceptible to several external factors, such as the smartphone casing, CPU loads, and device resting position.

Unlike our device fingerprinting work, many existing fingerprinting solutions for motion sensors require carefully designed external stimuli. In particular, Goethem *et al.* proposed fingerprinting accelerometers through a browser by triggering scheduled device vibrations via JavaScript because it introduces some distinctive patterns in the collected accelerometer trace [172]. Baldini *et al.* placed a magnet on a rotating platform to stimulate the magnetometer of nearby smartphones and applied an SVM (Support Vector Machine) model to classify these smartphones using features extracted from stimulated magnetometer outputs [173]. In addition, fingerprinting motion sensors via their calibration parameters have also been suggested. Son *et al.* proposed using the *power-on offset calibration* of the gyroscope embedded in a drone to serve as its identity [174]. However, this *power-on offset calibration* is not factory calibration. The calibrated offsets are dynamically calculated every time the drone is turned on. Thus, it changes over time and varies with temperature.

By comparison, our work is the first to recover the factory calibration parameters stored in non-volatile memory inside the device which do not change after leaving the factory.

Existing motion sensor fingerprinting techniques are mostly based on machine learning approaches. Bojinov *et al.* demonstrated it is possible to fingerprint the accelerometer from a webpage using typical clustering approaches [110]. However, they only correctly identified 53% of the devices in their dataset even after integrating the UA string into their model. Das *et al.* applied several supervised machine learning models to make a distinction between devices based on the gyroscope and accelerometer readings [175]. To increase accuracy, they used inaudible sound to stimulate the motion sensors. As a countermeasure, they suggested that manufacturers should perform better calibration of motion sensors. However, the calibration process could leak information if not properly implemented. More recently, they further improved its accuracy by introducing a voting scheme among different classifiers [17]. Nevertheless, their approach requires a lot of compute resource, which is not available on a handset. Even then, their approach achieved less than 60% F_1 score in an open-world setting when devices were held in hand. They also applied their approach to making a distinction between 85 iPhone 6 devices. When devices were held in hand, only 60% of these devices produced unique fingerprints, which, by reference to the birthday problem, indicates that their approach provides around 13 bits of entropy. Using the motion sensor data they collected through JavaScript, we correctly identified all iOS devices in the dataset based on the factory calibration behaviour without prior knowledge about the device model. Researchers have also attempted to apply deep learning to fingerprinting motion sensors. In particular, Liu *et al.* designed a deep neural network based on multiple LSTM (Long Short Term Memory) models to auto-discover suitable features for device fingerprinting from accelerometer and gyroscope outputs [176]. They demonstrated that their model works better than those based on handcrafted features and achieved 93% F_1 score when fingerprinting 117 heterogeneous smartphones using one-second motion data. Nevertheless, their approach is computationally expensive and requires knowing the number of unique devices in advance. The performance of their model on devices of the same make and model is also unclear. Most recently, Das *et al.* studied the sensor API usage in popular websites [177]. They showed that 2 653 of the Alexa top 100K websites accessed motion sensor data and 63% of the scripts for accessing motion sensors also engaged in browser fingerprinting.

5.2 Fingerprinting iOS devices

In Chapter 4 we have described the factory calibration fingerprinting of the gyroscope found in iOS devices. In this section, we look at other types of motion sensors, including magnetometer and accelerometer, and present our findings on their factory calibration

Table 5.1: Magnetometer type of different iOS device models

Type	Model	Nominal Gain (μT)
Type I	iPhone 4S/5/5C/5S/6/6 Plus All iPad models	0.35/0.28/0.17*
Type II	iPhone 6S/6S Plus/7/7 Plus/SE	0.075
Type III	iPhone 8/8 Plus	0.075
Type IV	iPhone X/XS/XS Max	0.075

* Type I devices have three possible nominal gain values.

behaviours. We also give a formal definition of SensorID that can be used as a reliable device fingerprint (§5.2.3).

5.2.1 Magnetometer

We find the magnetometer in iOS devices can also be fingerprinted. Similar to the gyroscope, the raw readings from the magnetometer only have a resolution of 2^{-16} μT (microtesla). After subtracting consecutive raw magnetometer measurements for every device model in our dataset, we observe four types of pattern:

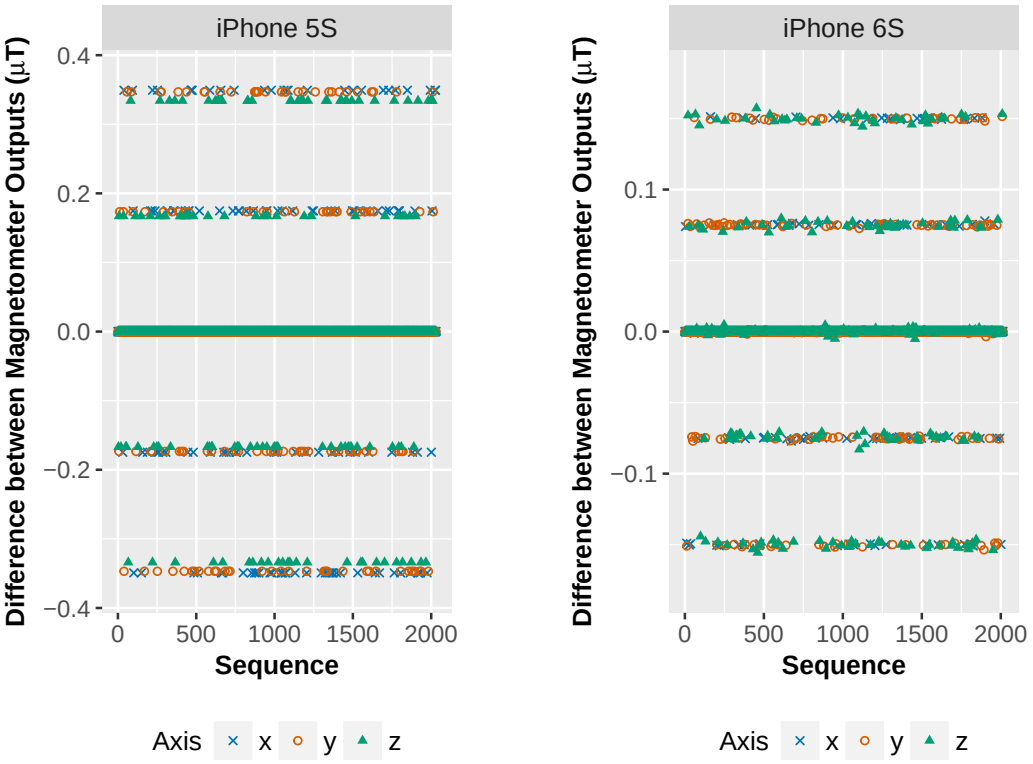
Type I (different sensitivity, negligible fluctuation) Type I devices have a slightly different sensitivity for each axis.

Type II (fixed sensitivity, moderate fluctuation) Type II devices have the same sensitivity for every axis but there is a moderate fluctuation within each band.

Type III (different sensitivity, moderate fluctuation) Type III devices have a slightly different sensitivity for each axis and there is a moderate fluctuation within each band; the quantisation of the data is evident in this case.

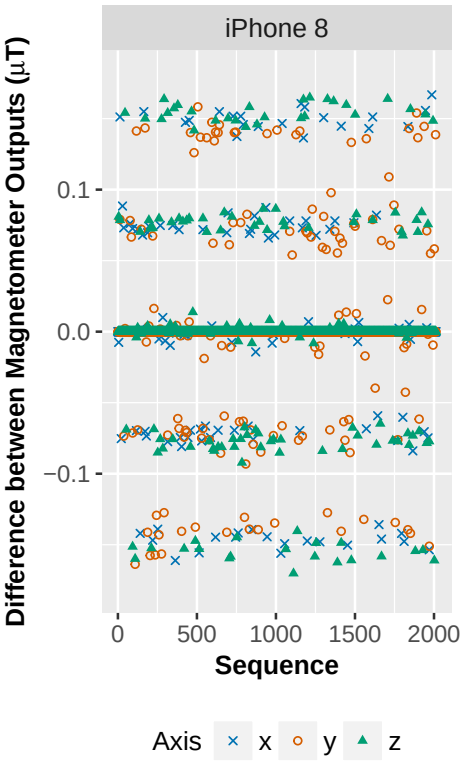
Type IV (different sensitivity, intense fluctuation) Type IV devices have an intense fluctuation on the magnetometer output. In this case, the quantisation of the data is not as evident as in other cases.

By way of an example, we collect 2K magnetometer samples at 200 Hz for a device of each type and present the data in Figure 5.1; we do not restrict how participants use the phone during the data collection. We summarise the magnetometer type of different iOS device models and their estimated nominal gain in Table 5.1. Overall, the observation of the four patterns reveals the different underlying calibration procedures. For all four types of devices, we can use the same approach described in §4.5 to obtain the magnetometer fingerprint (i.e., MagID). Although the gain matrix of the magnetometer is not stored at

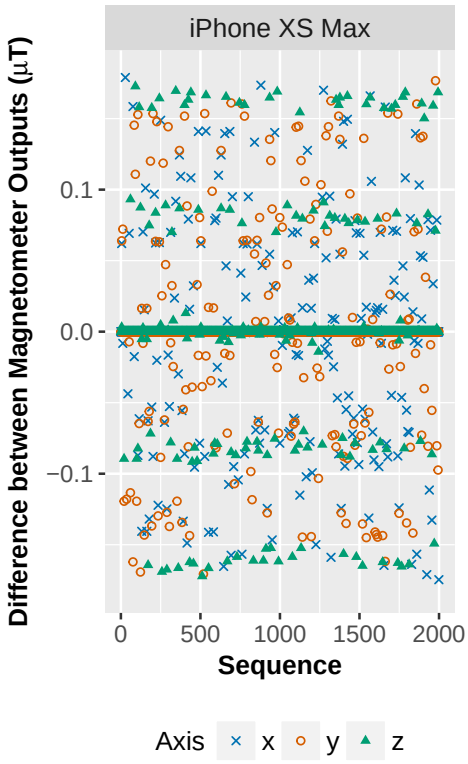


(a) Type I

(b) Type II



(c) Type III



(d) Type IV

Figure 5.1: Consecutive difference between raw magnetometer measurements for iOS devices

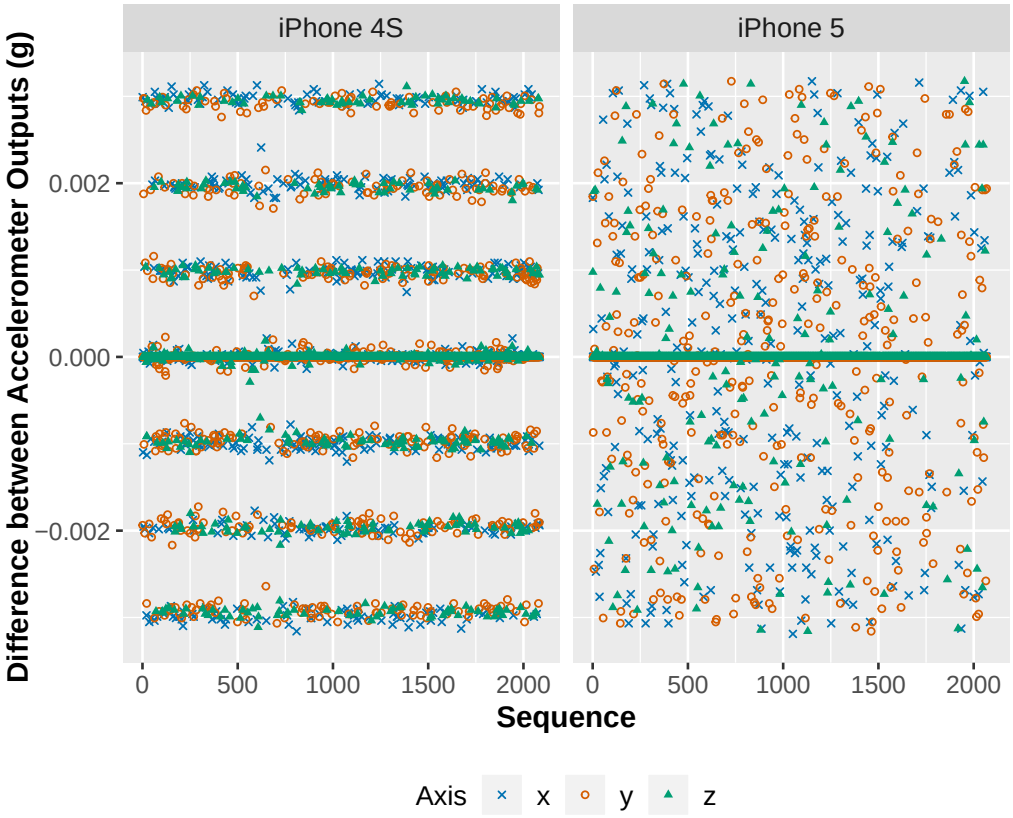


Figure 5.2: Comparison between iPhone 4 and 5 (Accelerometer)

2^{-16} µT resolution, adversaries can use the same techniques discussed in §4.7 to launch an attack by either clustering or rounding. Currently, the magnetometer data is not accessible in major browsers. Nevertheless, the MagID provides additional entropy to the GyroID. Thus, we can combine them as a finer-grained fingerprint when analysing apps.

5.2.2 Accelerometer

We also observe a distinctive quantisation pattern in accelerometer outputs from older generations of iOS devices, including the iPhone 4S and iPad Mini. For example, Figure 5.2 shows the consecutive differences between 2K accelerometer outputs from an iPhone 4S. The quantisation is clear, and thus we can apply the same approach described in §4.5 to recover the accelerometer fingerprint. Note that these devices do not use an Apple motion coprocessor. Therefore, adversaries would need to use techniques described in §4.7 to launch the attack. For newer versions of iOS devices, such as the iPhone 5 also shown in Figure 5.2, Apple uses a higher-resolution accelerometer that conceals the quantisation in accelerometer outputs. Our attack currently does not directly apply to these devices.

Table 5.2: Android devices tested

Vendor	Device Model
BQ	Aquaris M5/X2
Google	Pixel 1/1 XL/2/2 XL/3/3 XL/3a/3a XL/4/4 XL/C
HTC	10/U11/U11+/U12+/U12 Life/U Ultra/Desire 12/Nexus 9
Huawei	Mate 9/10 Pro/20 Lite/20 Pro/30 Honor 8/9/9X/20 Lite/Play MediaPad T3 10/M3 Lite 10/M5 Lite P10/P20/P20 Lite/P20 Pro/P30/P30 Pro/P Smart Y5/Nova 2/Nova 5i/Nexus 6P
LG	G5/G5 SE/G6/G6Fit/G7 ThinQ/G8S Thin Q K10/K40/V20/V30/V40 ThinQ/Q6/Stylo 4/Nexus 5X
Motorola	G4/G5 Plus/G6/G6/G6 Play/G7/G7 Play X4/X Play/One Action/One Vision/One Zoom/Nexus 6
Nokia	1/2.1/4.2/6.1/7 Plus/7.1/8 Sirocco/8.1
OnePlus	3/3T/5T/6/6T/7/7T/7T Pro
Samsung	Galaxy S7/S7 Edge/S8/S8+/S9/S9+/S10/S10+/S10e Galaxy S20/S20+/S20 Ultra 5G/Fold/Z Flip/M20 Galaxy Note8/Note9/Note10/Note10+ Galaxy A5/A6/A7/A8/A9/A10/A30/A40/A50/A51/A70/A80 Galaxy Tab S3/S4/S5e/S6
Sony	Xperia 1/5/XZ2/XZ2 Compact/XZ3/Z5/Z5 Premium/XA/XA2
Xiaomi	Mi 5s/6/8/8 Lite/8 Pro/9/Mix 2/Max 3/A2/A3 Redmi 5/7/Note 4/Note 5

5.2.3 SensorID

In this dissertation, we define the SensorID as a combination of distinctive sensor calibration fingerprints. In the case of iOS devices, the SensorID includes both the GyroID and MagID. For older generations of iOS devices (e.g., iPhone 4S and iPad Mini), the SensorID also includes the accelerometer fingerprint (i.e., AccID).

5.3 Fingerprinting Android devices

To study whether Android devices are susceptible to similar fingerprinting attacks, we use four automated testing platforms, including AWS Device Farm[1], Firebase Test Lab[2],

[1] `https://aws.amazon.com/device-farm/`
[2] `https://firebase.google.com/docs/test-lab`

App Centre[3], and Sauce Labs[4], to collect data from 146 Android device models from 11 vendors. Table 5.2 lists all device models we have tested. Among all Android devices we have tested, we find that all Google Pixel phones, other than the Pixel 1/1 XL, can be fingerprinted by our approach; we do not observe per-device calibration behaviour on the Pixel C tablet. In addition, we notice that the calibration process applied to motion sensors varies across device models. In particular, we find the full gain matrix of both the accelerometer and gyroscope in Google Pixel 4 and 4 XL are per-device calibrated, while only the main diagonal of the gain matrix for the accelerometer is calibrated in other Pixel devices.

Figure 5.3 shows the consecutive difference between 2K accelerometer outputs collected from a Pixel 3 and a Pixel 4 XL when they are at rest on a desk. The quantisation in accelerometer outputs is clear in both cases. In addition, the figure suggests that only the scale matrix, or the main diagonal of the gain matrix, of the accelerometer in the Pixel 3 is calibrated; the same pattern is also observed in Pixel 2/2 XL/3 XL/3a/3a XL devices. Figure 5.3 also suggests that all nine values in the gain matrix of the accelerometer embedded in the Pixel 4 XL are calibrated; the same pattern is also observed in the Pixel 4. Apart from the accelerometer, the gyroscope found in Google Pixel 4 and 4 XL devices is also per-device calibrated.

In addition to Pixel devices, we have noticed that the accelerometer in Huawei Honor 20 Lite, MediaPad M3 Lite 10, and MediaPad T3 10 and the gyroscope in BQ Aquaris X2 have different sensitivity for each axis. However, we only have one device for each of these models and thus cannot confirm whether they are per-device calibrated. So far we have tested 321 unique devices (146 unique device models); we have not observed per-device calibration behaviour on any other Android device models we tested. We therefore focus on Google Pixel devices in the rest of this chapter. The choice of factory calibration is up to individual manufacturers.

SensorID. For vulnerable Google Pixel phones, we can extract the SensorID from both an Android app and a website running on an Android browser. For the Google Pixel 4 and 4 XL, the SensorID includes both the AccID and GyroID. For other Pixel phones excluding Pixel 1, the SensorID only includes the AccID.

In summary, Table 5.3 lists all device models whose SensorID includes the AccID, GyroID, and MagID, respectively.

[3] `https://appcentre.ms/`
[4] `https://saucelabs.com/`

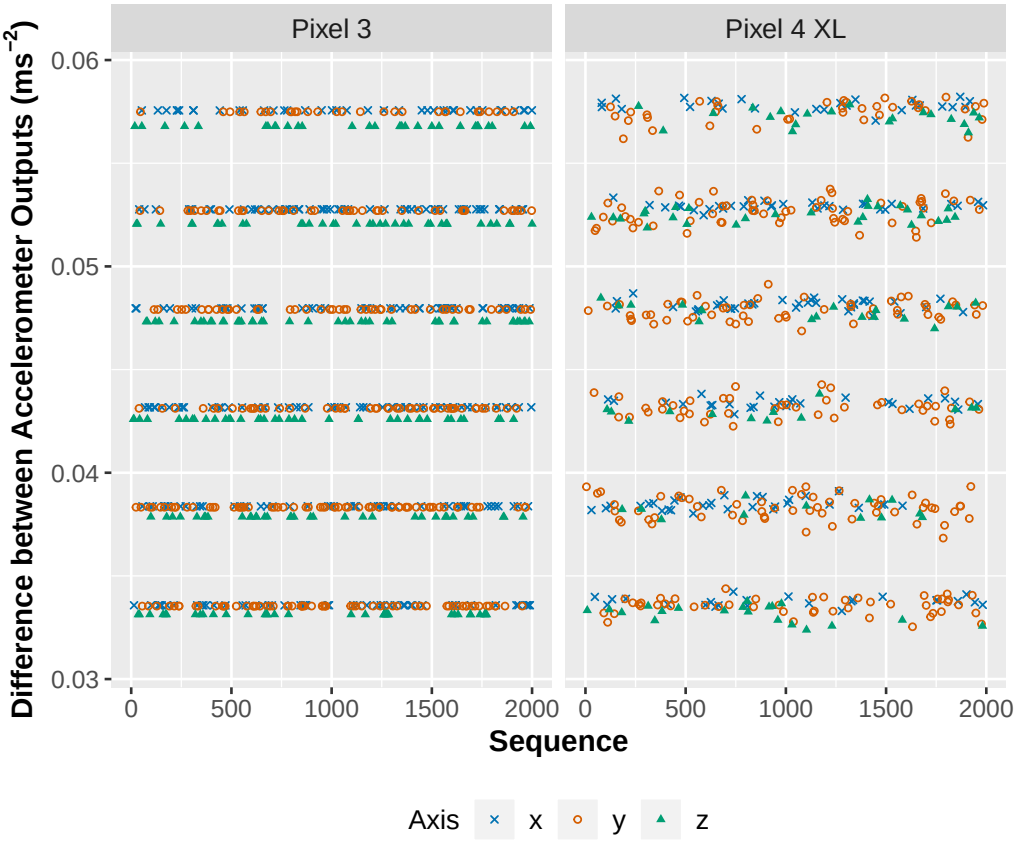


Figure 5.3: Comparison between Pixel 3 and 4 XL (Accelerometer)

5.4 Evaluation

Since we have found iOS and Google Pixel devices are vulnerable to the factory calibration fingerprinting attack, we evaluate the performance of the attack for these devices in §5.4.1 and §5.4.2, respectively.

5.4.1 Fingerprinting iOS devices

To evaluate the factory calibration fingerprinting attack, we have developed both a website and an iOS app to collect sensor data. The iOS app collects raw data from the motion sensors (accelerometer, gyroscope, and magnetometer) at 200 Hz and does not ask users to put the device in any particular position. The app also embeds a WebView; the embedded WebView and the separate website both collect fused accelerometer and gyroscope data via JavaScript.

For both the app and website, we use the Fingerprintjs2[5] library in the default configuration to generate a browser fingerprint for evaluation purposes. In addition to

[5] `https://github.com/fingerprintjs/fingerprintjs2`

Table 5.3: SensorID composition

Fingerprint	Device Model
AccID	iPhone 4S, iPad Mini Pixel 2/2 XL/3/3 XL/3a/3a XL/4/4 XL
GyroID	iOS devices, Pixel 4/4 XL
MagID	iOS devices

volunteers, we have recruited participants from Amazon Mechanical Turk[6] and Prolific[7] to download the app and contribute sensor data. The public data collection exercise has been approved by the ethics committee of the Department of Computer Science and Technology at the University of Cambridge.

To date, the SensorID app has collected data from 795 unique iOS devices; 761 of them contain an M-series motion coprocessor. In addition, the website has collected fused data from another 75 devices. Some users choose to participate in this study multiple times. Thus, there might be more than one record for each unique device. On both the app and the website, we ask users to tell us whether they have submitted the data from this device before.

GyroID. Using the raw gyroscope data collected from the 761 iOS devices with an M-series coprocessor, we are able to recover the exact GyroID. For the other 34 devices that do not contain an M-series coprocessor, we use the rounding option in §4.7 to generate the GyroID due to the small sample size. Based on the GyroID, we successfully identify multiple records that are submitted by the same device. This is confirmed by user-supplied data about whether they have submitted samples from this device before and the device IP address when they submit. The GyroID of each device is distinct.

Since the website only collects fused gyroscope data, we choose the rounding option in §4.7 to generate the GyroID. Then, we compare it with the GyroID of the 795 devices that we recovered from the raw data. As a result, we identify 3 devices submitted through both the website and the app. The app also collects fused sensor data from the built-in WebView. For this data, we use the clustering approach to generate a group of gain matrix estimates. Then, we apply the Multivariate ANalysis Of VAriance (MANOVA) technique to analyse these estimates and successfully identify all 795 unique devices in the dataset. In particular, we also identify 6 devices that submitted multiple times through the app. The results are the same as we obtained from the raw data.

[6] `https://www.mturk.com`
[7] `https://prolific.ac`

Table 5.4: Comparison of iOS device fingerprints

# Devices	Fingerprint	Group Size	# Groups
870	GyroID	1	870
	Fingerprintjs2	1	391
		2	43
		3	22
		4	12
		5	2
		6	4
		7	2
		9	1
		10	1
		11	2
		13	2
		14	1
		19	1
		22	1
		28	1
		36	1
		45	1
795	MagID	1	775
		2	10
	Fingerprintjs2	1	308
		2	44
		3	20
		4	13
		5	3
		6	3
		7	2
		9	2
		10	1
		11	2
		13	2
		14	1
		19	1
		22	1
		28	1
		36	1
		45	1
10	AccID	1	10
	Fingerprintjs2	1	8
		2	1

MagID. We also apply the improved approach to fingerprint the magnetometer with the rounding option. After generating the MagID, we group devices by their MagID and present the results in Table 5.4. In the table, the group size records the number of different devices sharing the same MagID. Therefore, a group of size 1 means the device has a unique MagID in our dataset. We find that the 10 groups of size 2 are all old device models with a Type I magnetometer, indicating they have a higher chance of collision on MagID than others. The reason is that the entropy of the MagID for Type I devices is only provided by the scale matrix (i.e., main diagonal of the gain matrix). Nevertheless, the MagID is orthogonal to the GyroID. Thus, they can be combined together to provide additional entropy.

AccID. Similar to the analysis of data collected from the built-in WebView, we use the clustering option in §4.7 to analyse the accelerometer fingerprint and apply MANOVA to identify unique devices. As discussed in §5.2, our attack only applies to older generations of iOS devices. In our dataset, that includes 9 iPhone 4S devices and an iPad 3. We apply our attack on these 10 devices and find that all of them have a unique AccID. It is likely that other iOS devices prior to iPhone 4S can also be attacked by our approach, but we do not have data from these devices to confirm it.

Comparison. Finally, we compare the GyroID, MagID, AccID with the default configuration of Fingerprintjs2, which utilises font detection, canvas, WebGL, etc. to fingerprint devices. Table 5.4 presents the results and demonstrates GyroID, MagID, and AccID provide more entropy than traditional browser fingerprinting techniques. While GyroID is unique for every device in our dataset, 45 out of 135 iPhone 7 devices have the same Fingerprintjs2 fingerprint; these 45 devices are all from the UK. In the case of AccID, it identifies all 10 unique devices, while Fingerprintjs2 generates the same fingerprint for two iPhone 4S devices; both devices are from Germany. The results indicate that the Fingerprintjs2 fingerprint may be correlated with a particular handset configuration.

We have also developed a proof-of-concept app for iOS devices with an M-series motion coprocessor. Screenshots of the app is presented in Figure 5.4. The app implements our attack to generate the GyroID of the test device. The code is written in Swift 4.1 with XCode 9.4.1. The app collects 100 raw gyroscope samples and attempts to generate the GyroID. If it fails (due to intense shaking of the phone) the app automatically collects another 100 raw samples and repeats the process. Overall, it takes about 0.5 seconds to collect 100 gyroscope samples and another 0.01 seconds to generate the GyroID. Vigorous movement during extraction may require additional samples, but the task nevertheless completes within a few hundred samples and takes a few seconds. In any case, the

(a) **(b)**

Figure 5.4: Screenshots of the GyroID proof of concept app

generated GyroID always stays the same. A proof-of-concept webpage and demo videos can be found on the website: `https://sensorid.cl.cam.ac.uk/`.

5.4.2 Fingerprinting Google Pixel devices

In general, it is difficult to find many people with a Pixel device via crowdsourcing platforms due to the relatively small market share. Nevertheless, we find most online app testing platforms provide access to Pixel devices. Therefore, we develop an Android app to collect raw motion sensor data and send it back to our server. In a method similar to the one used in our iOS app, our app also embeds a WebView to collect fused sensor data via JavaScript as well as a fingerprint generated by Fingerprintjs2. In addition, the app records the `ANDROID_ID` that is unique to each combination of the app-signing key, user, and device to identify unique devices in our dataset.

We deploy the Android app on four app testing platforms (listed in §5.3) to collect data from various Android device models. Since we have only confirmed per-device calibration on Pixel phones other than the Pixel 1/1 XL, we focus our analysis on these devices. By the end, we have collected data from 152 unique Pixel devices.

Table 5.5: Comparison of Google Pixel device fingerprints

Device Model	Fingerprint	Group Size	# Groups
Pixel 2/2 XL	AccID	1	46
	Fingerprintjs2	1	17
		2	4
		4	1
		5	2
		7	1
Pixel 3/3 XL/3a/3a XL	AccID	1	61
	Fingerprintjs2	1	25
		2	8
		3	5
		5	1
Pixel 4/4 XL	AccID	1	45
	GyroID	1	45
	Fingerprintjs2	1	10
		2	1
		9	1
		10	1
		14	1

Using the raw accelerometer data, we generate the AccID for each device and use clustering to determine whether two devices have the same AccID. We compare the results with Fingerprintjs2 in Table 5.5. For Pixel 4 and 4 XL devices, we also calculate their GyroID and compared it with other fingerprints. As shown in the table, both AccID and GyroID uniquely identifies every Pixel device while multiple devices have the same fingerprint generated by Fingerprintjs2. In particular, 14 out of 45 Pixel 4/4 XL devices have the same Fingerprintjs2 fingerprint. This is likely because the embedded WebView with default configuration does not expose many unique characteristics if two devices are running the same Android version. The fingerprint is likely to have more entropy if Fingerprintjs2 is running in an Android browser. Nevertheless, unlike SensorID, which is a hardware identifier, Fingerprintjs2 cannot track users as they move across Android browsers.

In addition, we have tried to generate the SensorID for each device using the web sensor data collected via JavaScript in the embedded WebView. We then differentiate devices based on their SensorID, generated from the web data, using clustering. We successfully identify all unique Pixel devices by their AccID. For Pixel 4 and 4 XL phones, we are also able to make a distinction between individual devices by their GyroID alone.

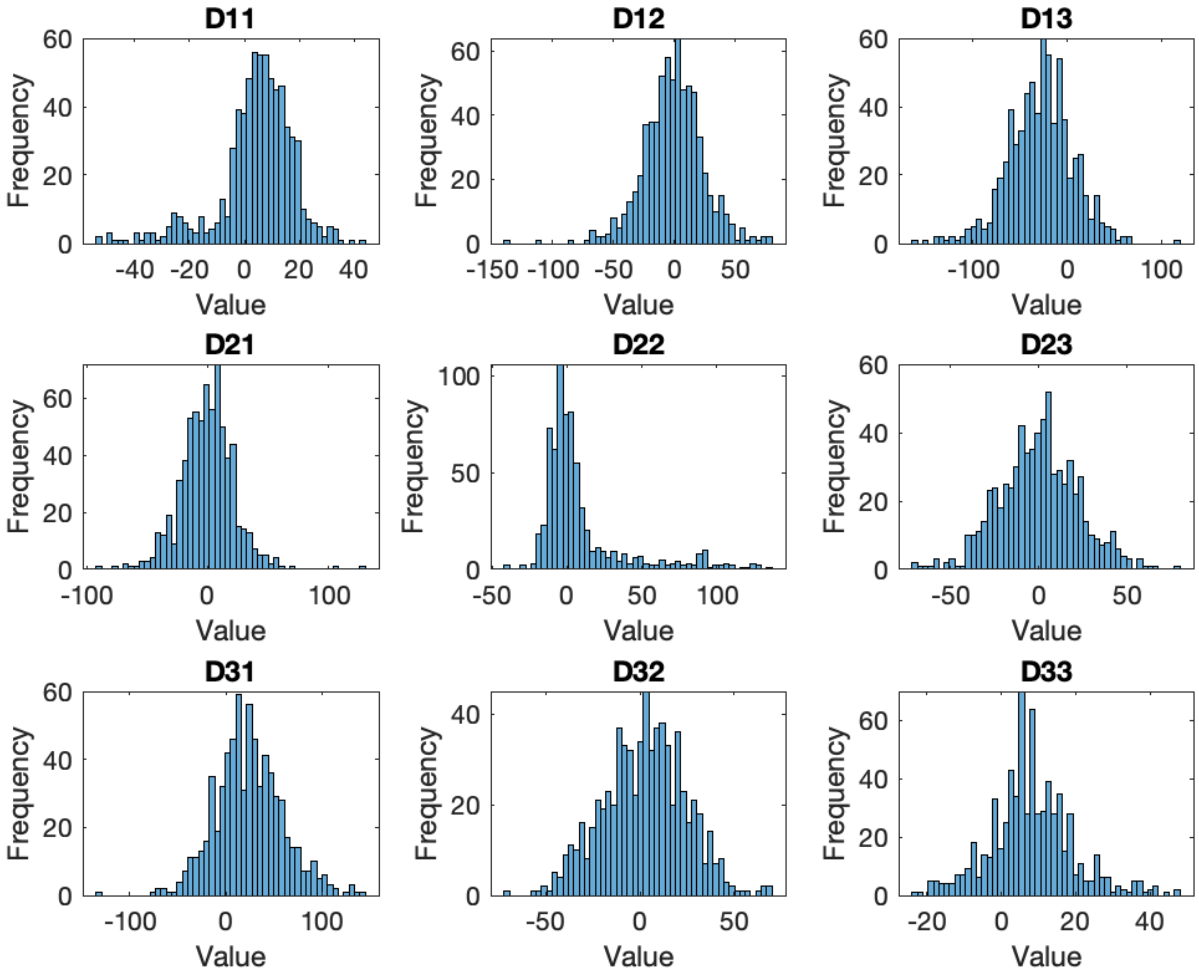


Figure 5.5: Distribution of each element in the GyroID (nominal gain = 61 mdps)

5.5 Discussion

In this section, we discuss some possible concerns regarding the validity of this research.

5.5.1 Is SensorID unique for iOS devices?

To study how unique is SensorID, we first study the GyroID of all iOS devices with an estimated nominal gain of 61 mdps. Device models included in this category can be found in Table 4.2. We choose this category for two reasons. First, all device models in this category are modern devices which contain an M-series motion coprocessor and this makes it possible to extract their exact gain matrix. Second, devices with different default gain may have a different GyroID distribution, so we select the larger size group, which contains 693 devices in total. Figure 5.5 presents the distribution of each element in GyroID for the 693 devices. For simplicity, we denote the GyroID as $\mathbf{D} \in \mathbb{Z}^{3\times3}$ in the following analysis.

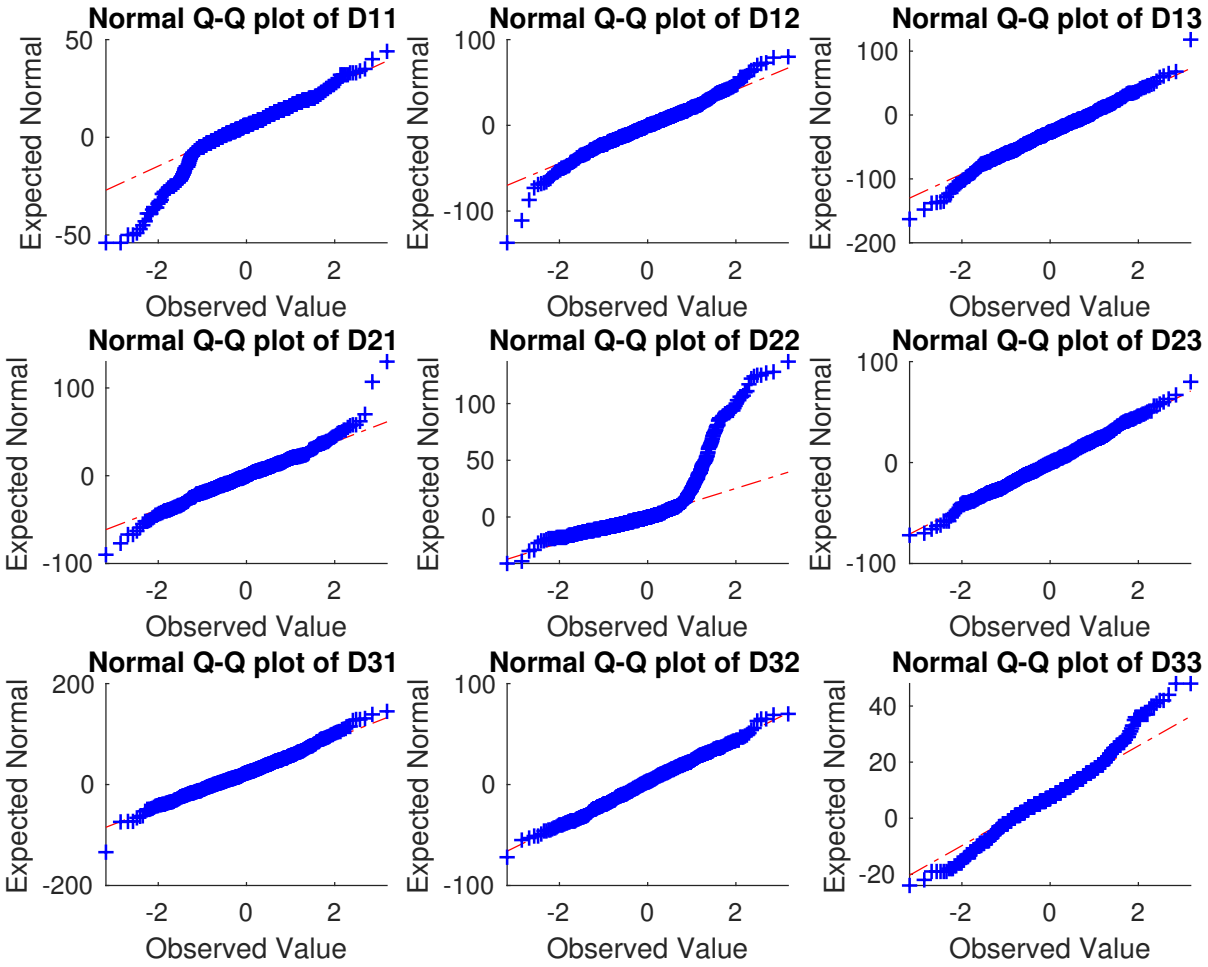


Figure 5.6: Q-Q plot of each element in the GyroID (nominal gain = 61 mdps). If the distribution of a variable is normal, its Q-Q plot should appear linear

Normality analysis. To test for normality, we first draw the Q-Q plot of each $\mathbf{D}_{ij} \in \mathbf{D}$ in Figure 5.6. In general, a Q-Q plot would appear linear if the distribution of a variable is normal. As seen in Figure 5.6, 6 out of 9 Q-Q plots do appear linear, which gives us some confidence that these elements are likely normally distributed. We applied both the Kolmogorov-Smirnov test and the Shapiro-Wilk test of normality for each element in $\mathbf{D}$. Results show that the off-diagonal elements in $\mathbf{D}$ have strong normality, while elements in the main diagonal ($\mathbf{D}_{11}$, $\mathbf{D}_{22}$, $\mathbf{D}_{33}$) are rejected by both tests at the 0.05 significance level. Since main-diagonal elements in $\mathbf{D}$ correspond to sensor sensitivities, the non-normality is likely because the sensor sensitivities of different device models are centred at different values due to different manufacturing pipelines. Thus, the distribution of these elements should be analysed on a per device model basis. To confirm this, we run a normality test on data from each device model separately we find that the main diagonal elements also show strong normality.

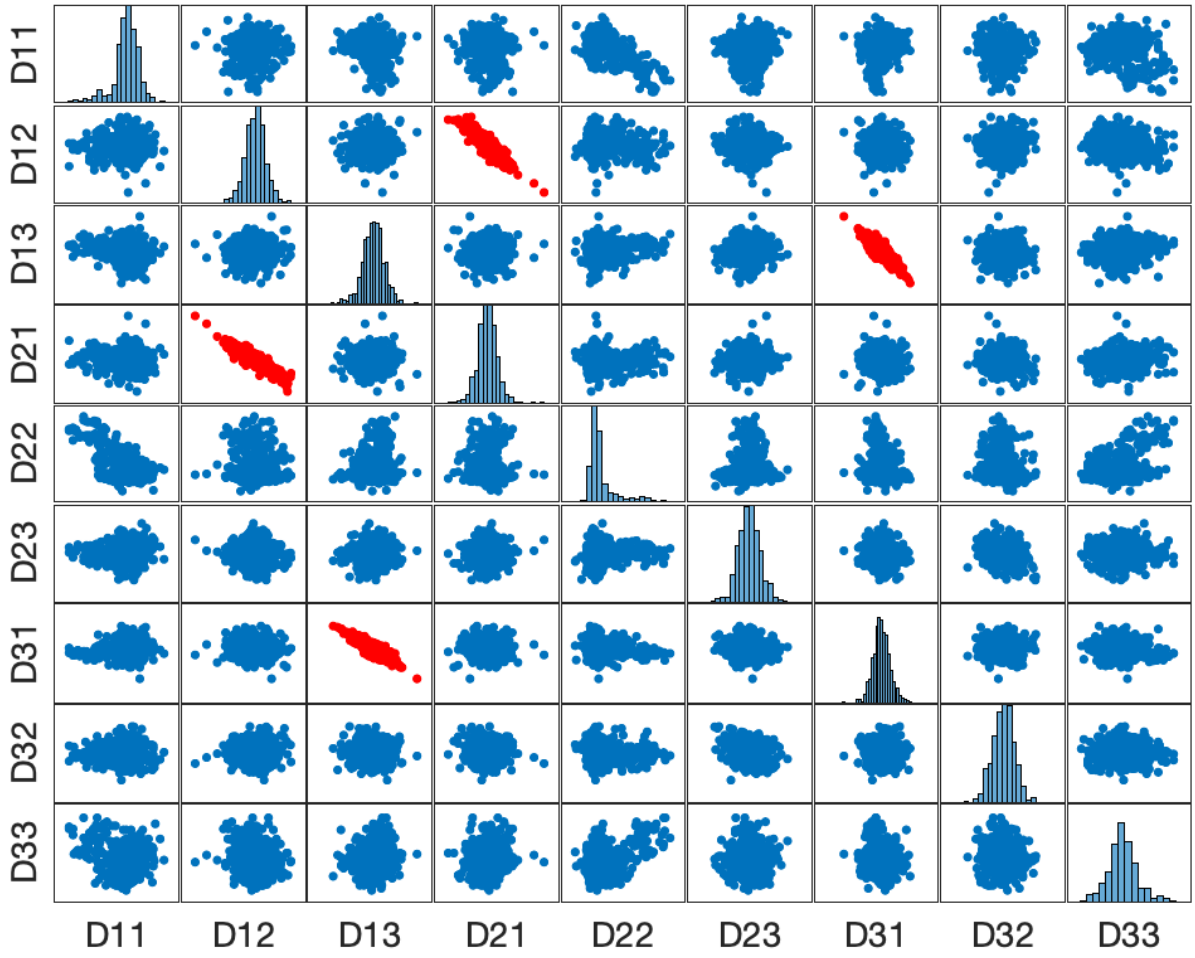


Figure 5.7: Scatter plot matrix of elements in the GyroID (nominal gain = 61 mdps). Each scatter plot shows the relationship between two variables. Pairs with significant correlation are highlighted in red

Correlation analysis. To test for correlation, we run the Pearson correlation test on each $\mathbf{D}_{ij}$ and find that $\mathbf{D}_{12}$ and $\mathbf{D}_{13}$ are strongly correlated with $\mathbf{D}_{21}$ and $\mathbf{D}_{31}$, respectively, at the 0.01 significance level. The scatter plot in Figure 5.7 gives a more intuitive view of the relation between different elements in $\mathbf{D}$, where pairs with significant correlation are highlighted in red. Main-diagonal and off-diagonal elements in $\mathbf{D}$ respectively correspond to the sensitivity of sensor axes and the non-orthogonality factor between sensor axes. Since non-orthogonality and sensitivity are independent, none of the main-diagonal elements shows a strong correlation to any off-diagonal elements in Figure 5.7. The deviation to nominal sensitivity in one axis due to manufacturing imperfections should also not affect the sensitivity of other axes, and thus main-diagonal elements are not strongly correlated to each other. However, Figure 5.7 shows that $\mathbf{D}_{21}$ and $\mathbf{D}_{31}$ are strongly correlated to $\mathbf{D}_{12}$ and $\mathbf{D}_{13}$, respectively. Therefore, we exclude $\mathbf{D}_{21}$ and $\mathbf{D}_{31}$ from our entropy calculation to avoid over-estimation.

Entropy calculation. We first calculate the entropy of off-diagonal elements in $\mathbf{D}$, excluding $\mathbf{D}_{21}$ and $\mathbf{D}_{31}$. For each off-diagonal element, we estimate the parameters of the normal distribution, including the mean μ and standard deviation σ, from the dataset. Technically, it is not a strict normal distribution since each element can only be an integer. Nevertheless, it is a result of rounding, and thus we can still use the normal distribution to estimate the entropy.

In general, the entropy of a discrete random variable X, which is denoted as $\mathrm{H}(X)$, can be calculated by:

$$\mathrm{H}(X) = -\sum_{x_i \in X} \mathrm{P}(x_i) \log_2 \mathrm{P}(x_i) \tag{5.1}$$

where $\mathrm{P}(x_i)$ is the probability of X being equal to x_i. In our case, we regard the element $\mathbf{D}_{ij}$ as the variable X. Then, we have $x_i \in \{-65535, \ldots, 65535\}$ because of the 16-bit resolution. Suppose $X \sim \mathcal{N}(\mu, \sigma^2)$ with the density function $f(x)$, then we can calculate $\mathrm{P}(x_i)$ as follows:

$$\mathrm{P}(x_i) = \begin{cases} \int_{x_i-0.5}^{x_i+0.5} f(x)\ dx, & \text{if } x_i \in (-65535, 65535) \\ \int_{-\infty}^{-65534.5} f(x)\ dx, & \text{if } x_i = -65535 \\ \int_{65534.5}^{+\infty} f(x)\ dx, & \text{if } x_i = 65535 \end{cases} \tag{5.2}$$

By this equation, we calculate the entropy of $\mathbf{D}_{12}$, $\mathbf{D}_{13}$, $\mathbf{D}_{23}$, and $\mathbf{D}_{32}$. For main diagonal elements (i.e., $\mathbf{D}_{11}$, $\mathbf{D}_{22}$, and $\mathbf{D}_{33}$), we calculate their entropy on a per device model basis. Here, we use the iPhone 6S as an example to calculate the GyroID entropy because it is the most popular device model in our dataset (127 devices). For these iPhone 6S devices, we adopt a similar approach and apply both Equation 5.1 and Equation 5.2 to calculate the entropy. As a result, we estimate the GyroID for iPhone 6S has about 42 bits of entropy.

By the same analysis, we estimate the entropy of the MagID for iPhone 6S. If adversaries launch the attack using the rounding option (§4.7), each element could have ± 1 uncertainty. In this case, we estimate that the MagID contains about 25 bits of entropy. The MagID should have more entropy if adversaries choose the clustering option. Since we only have 10 old-generation iOS devices that have a distinctive AccID, we do not include AccID into the SensorID entropy calculation. We observe no evidence of a strong correlation between the MagID and GyroID. Therefore, we estimate the SensorID for iPhone 6S has around 67 bits of entropy.

Uniqueness analysis. There were 728M active iPhones worldwide in April 2017 and the iPhone 6S devices accounted for 18% of them [178]. Therefore, there were around 131M iPhone 6S devices. From the birthday problem, we know that the chance of two

iPhone 6S devices having the same SensorID is around 0.0058%, suggesting it is a globally unique device fingerprint. In addition, the SensorID is orthogonal to other fingerprinting techniques. Therefore, adversaries can combine the SensorID with other metadata (e.g., system language) or other fingerprinting techniques (e.g., canvas fingerprinting) to further increase the fingerprint entropy.

Limitations. Results from both the Kolmogorov-Smirnov and Shapiro-Wilk normality tests suggest that values in $\mathbf{D}$ are consistent with a normal distribution, but it is possible that the actual distribution is not normally distributed in the tail regions. For example, manufacturers may discard sensors that have an extreme value in the gain matrix; this would reduce the available entropy. We therefore need to consider whether non-normal distributions might invalidate our entropy calculations.

Firstly, it is worth mentioning that this kind of rejection policy is unlikely in practice; one of the key benefits of factory calibration is that sensors with anomalous physical gains will still perform well when calibrated. More importantly, the calculation of entropy is dominated by the core shape of the distribution, where we have abundant data. Non-Gaussianity may affect the tails of the distribution, but this would have a negligible effect on the calculated entropy. To give a concrete example: we find all values in $\mathbf{D}$ fall inside the range $(\mu - 4\sigma, \mu + 4\sigma)$. If we make the assumption that values outside this range are discarded, we still estimate the SensorID provides around 67 bits of entropy for the iPhone 6S.

A related concern is there could be undetected higher-order correlations between values in $\mathbf{D}$. A similar argument applies in this case: the entropy calculation is dominated by the core of the (now multivariate) distribution, where we have abundant data, and where we see no evidence of non-independence. In the tail regions, non-independence might go undetected, but this would have little impact on the calculated entropy.

Ultimately, the calculation of entropy cannot be done with absolute rigour given a finite number of samples from an unknown distribution, but it is still possible to perform a thorough analysis, and significant errors in the estimated entropy of SensorID due to non-Gaussianity or non-independence are very unlikely.

5.5.2 Factory calibration in Android devices

Rooted Android handsets provide access to the gain matrix values for the accelerometer and gyroscope in the local file system on boot. We therefore confirm that our distinct estimates for the AccID on two Pixel 3 devices are correct as well as our estimated AccID and GyroID values for a Pixel 4 device. While the magnetometer in the Pixel 3/4 also has a full gain matrix, its values appear to be the same for all devices of the same model and thus does not provide any entropy. The motion sensors in other Android devices may

Table 5.6: SensorID entropy estimation (Google Pixel devices)

Fingerprint	Device Model	# Devices	Entropy (bits)
AccID	Pixel 2/2 XL	46	~14
	Pixel 3/3 XL/3a/3a XL	61	~12
	Pixel 4/4 XL	45	~25
GyroID	Pixel 4/4 XL	45	~45

also be factory calibrated. If the calibration is restricted to offsets (i.e., bias compensation) then our approach is ineffective since it targets the gain matrix and cannot recover bias compensation.

We group the Google Pixel devices into three categories: Pixel 2 series (Pixel 2/2 XL), Pixel 3 series (Pixel 3/3 XL/3a/3a XL), and Pixel 4 series (Pixel 4/4 XL). To estimate the entropy of SensorID for vulnerable Pixel devices, we analyse the normality of each value and the correlation between values in the fingerprint for each category. For Pixel 2 and 3 series, the AccID only has non-zero values in the main diagonal ($\mathbf{D}_{11}$, $\mathbf{D}_{22}$, $\mathbf{D}_{33}$), and thus it provides less entropy than that for Pixel 4 series. For devices in the same category, we find all non-zero elements in their calibration fingerprint show strong normality.

We estimate the entropy based on the assumption that adversaries choose to attack using the rounding option, which has ± 1 uncertainty for each element. Results are presented in Table 5.6. In particular, we find off-diagonal elements in GyroID are strongly correlated with off-diagonal elements in AccID in Pixel 4 series devices. This is likely because the accelerometer and gyroscope are integrated into the same chip (`LSM6DSR`). Therefore, to estimate the entropy of SensorID, the combination of AccID and GyroID, we simply add the entropy provided by the main diagonal variables in the AccID to the entropy provided by the GyroID. As a result, we estimate the SensorID provides around 57 bits of entropy; a precise estimate of entropy is difficult since we only have data from 45 Pixel 4/4 XL devices whereas in our previous study with iOS we used data from 127 iPhone 6S devices. Assuming that our entropy estimate is accurate, and analysing this as an example of the birthday problem, if there are 100 million Pixel 4/4 XL devices in the market, the probability that every device has a globally unique SensorID is around 97%.

5.5.3 Is SensorID correlated with the manufacturing batch?

To answer this question, we first study the correlation between the SensorID and the country of the device, which is inferred from the IP address when a user submits data. We do not find any evidence of strong correction at the 0.05 significance level. In addition, we collect gyroscope data from 25 iOS devices in an Apple Store. Some of these devices have similar serial numbers, which suggests they may come from the same manufacturing

batch. However, the GyroID of these devices differs significantly. Furthermore, there is no significant difference in the GyroID distribution for devices from the Apple Store and for devices that we collect otherwise.

5.5.4 Consistency of SensorID

We have not observed any change in the SensorID of our test devices in the past 16 months. Our dataset includes iOS devices running iOS 9/10/11/12 and Pixel devices running Android 8/9/10. We have tested compass calibration, factory reset, and updating the operating system (before the vulnerability is fixed); the SensorID always stays the same. We have also tried measuring the sensor data at different locations and under different temperatures; we confirm that these factors do not change the SensorID either.

5.5.5 Impact and responsible disclosure

We followed a responsible disclosure procedure and reported this vulnerability to Apple on 3rd August 2018 and Google on 10th December 2018. In particular, we suggested two possible countermeasures. The first is to add a random noise $\epsilon \in \mathbb{R}^{3\times 1}$, from the uniform distribution in the range $[-0.5, 0.5]$, to each ADC output. The added noise obfuscates the effect of quantisation, making the attack much harder. The second approach we proposed is to round the calibrated sensor output to the nearest multiple of the nominal gain. This approach is more practical to apply since it does not require access to the ADC values.

In iOS 12.2, Apple adopted our suggestion and added random noise to sensor outputs (CVE-2019-8541). In addition, Apple removed access to motion sensors from Mobile Safari by default and in later versions removed motion sensor access from WebKit as well. In Android 11, Google deployed a fix that rounds the motion sensor outputs to the nearest multiple of the nominal gain.

When running an iOS version prior to iOS 12.2, all iOS devices that have motion sensors can be fingerprinted by this approach, including the iPhone XS and iPhone XS Max. A SensorID can be generated by both apps and mobile websites and requires no user interaction. Both mainstream iOS browsers (Safari, Chrome, Firefox, and Opera) and privacy-enhanced browsers (Brave and Firefox Focus) were vulnerable to this calibration-based fingerprinting attack, even with the fingerprinting protection mode turned on. For Google Pixel phones running Android 10, we notice that some privacy-enhanced browsers, including Brave and Tor Browser, do block access to motion sensors by default while others (Chrome, Firefox, Firefox Focus, Opera, and DuckDuckGo) do not. Using a browser that blocks motion sensor access could protect Pixel phone users from this attack when browsing online. A recent study shows that motion sensor data is accessed by 2 653 of the Alexa top 100K websites, including more than 100 websites exfiltrating motion sensor

data to remote servers [177]. This is troublesome since it is likely that the SensorID can be calculated with exfiltrated data, allowing retrospective device fingerprinting. Although the factory calibration fingerprinting attack does not directly apply to iOS devices running iOS 12.2 or later, a dedicated attacker may still be able to extract the fingerprint (§5.6).

5.6 Apple's fix

Apple declined to share the details of the fix deployed in iOS 12.2. Therefore, we reverse-engineer Apple's fix by studying the gyroscope output in iOS devices with iOS 12.2 or later; we show that the random noise applied does not fully conceal the calibration fingerprint.

5.6.1 Analysis of Apple's fix

Figure 5.8 presents the histogram of the y-axis data of raw gyroscope output, collected from the same iPhone X running two different iOS versions. In both cases, the device is at rest on a desk. When the device is running iOS 11.4.1, the quantisation in the histogram is clear and we can recover the exact gain matrix (§4.5). To mitigate our attack, Apple added random noise to the gyroscope outputs in iOS 12.2, concealing the quantisation information. Figure 5.8 suggests that the added noise follows a uniform distribution, which is one of the countermeasures we proposed to Apple. However, we notice there are some peculiarities in the noise added.

To figure out the range of the uniform noise applied, we first obtain the gyroscope gain matrix of an iOS device when it is running an iOS version before iOS 12.2. Then, we update the device to the latest iOS version and take 20K gyroscope measurements from the device at 200 Hz when the device is at rest on a desk. We denote the gain matrix as $\mathbf{G}$ and these gyroscope outputs as $\mathbf{O}$. Then, the underlying bias-corrected ADC outputs $\mathbf{I}$ (i.e., $\mathbf{I} = \mathbf{A} + \mathbf{B}$) can be estimated by:

$$\widetilde{\mathbf{I}} = \text{round}(\mathbf{G}^{-1}\mathbf{O}) \tag{5.3}$$

Although the estimated ADC values, $\widetilde{\mathbf{I}}$, may not be accurate due to perturbation, it still gives us useful insights about the added noise. Furthermore, we can get the noise estimate, $\widetilde{\mathbf{N}}$, by:

$$\widetilde{\mathbf{N}} = \mathbf{O} - \mathbf{G}\widetilde{\mathbf{I}}$$

By way of an example, Figure 5.9 presents the histogram of the estimated gyroscope noise of an iPhone XS. As shown in the Figure 5.9, the majority of the estimated noise is distributed uniformly in the range $[-1997, 1997] \times 2^{-16}$ dps. The small number of outliers are likely produced due to the inaccurate estimation of the ADC values in Equation 5.3.

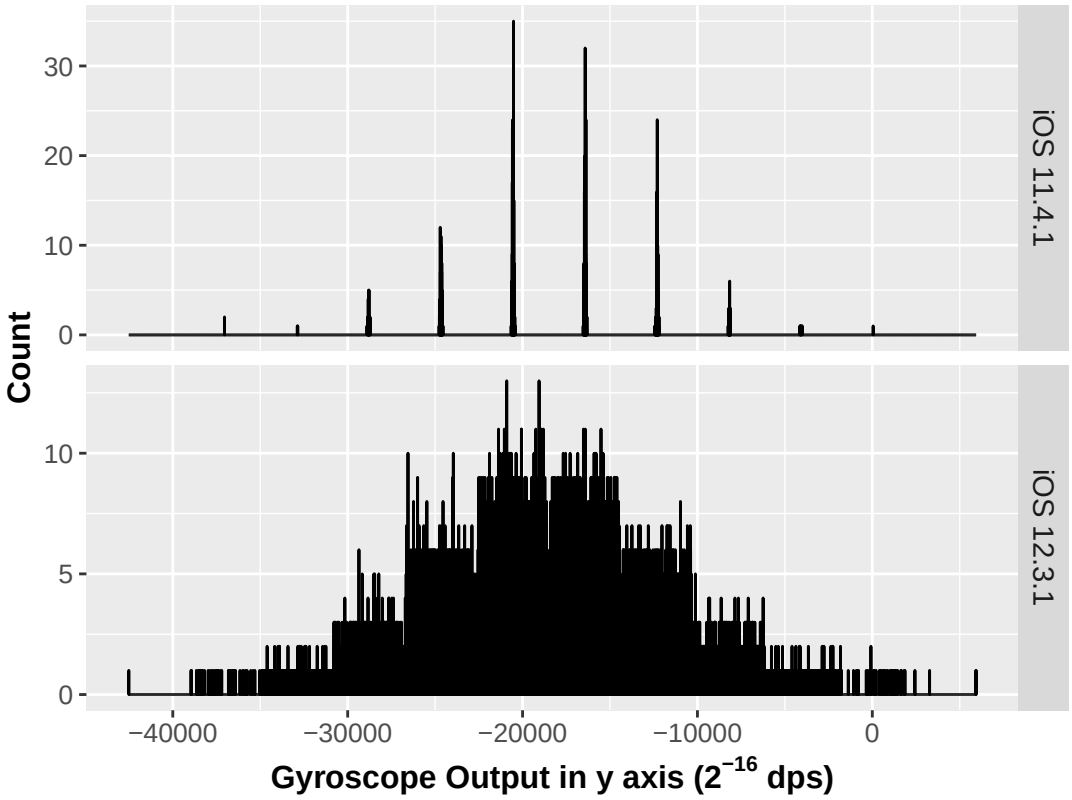


Figure 5.8: Histogram of raw gyroscope data of an iPhone X

We have also tested other iOS devices and observed the same result. This confirms that Apple did not add random noise in the range $[-0.5, 0.5]$ to the ADC values as we proposed but instead added random noise in the range $[-1997, 1997] \times 2^{-16}$ dps to the calibrated signal. Because the width of the random noise is slightly narrower than the sensitivity of some gyroscope axes, it leaks more information than our original proposal. In the case of the iPhone XS in Figure 5.9, the width of the perturbation, 3995, is lower than the sensitivity of all three axes (4012, 4031, and 4016, respectively). Here, we show that we can recover the gain matrix from the noisy data by performing a maximum likelihood search using simulated annealing.

5.6.2 Attack on Apple's fix

In this section, we first define the objective function to quantify the likelihood of observing the outputs given a gain matrix. Then, we show that the exact gain matrix can be estimated from the objective function using simulated annealing.

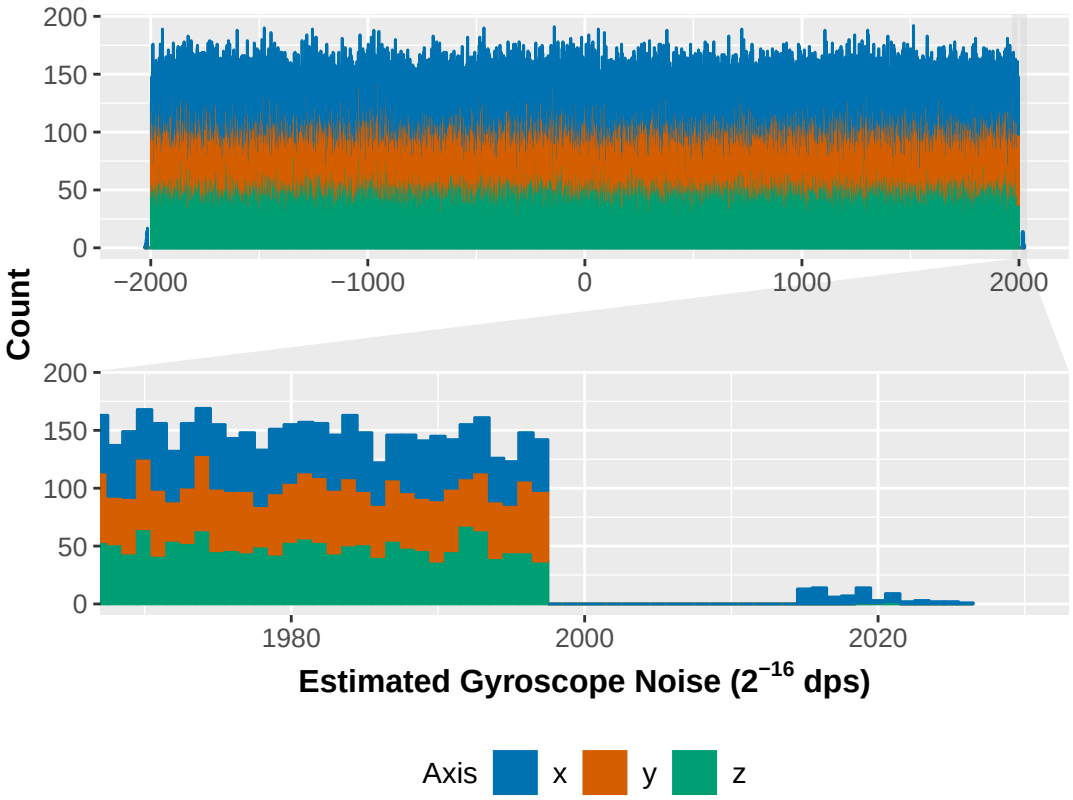


Figure 5.9: Histogram of estimated gyroscope noises (iPhone XS)

5.6.2.1 Objective function

For a candidate gain matrix $\widetilde{\mathbf{G}}$, we estimate the corresponding bias-corrected ADC outputs $\mathbf{I}$ by:

$$\widetilde{\mathbf{I}} = \text{round}(\widetilde{\mathbf{G}}^{-1}\mathbf{O})$$

Here, we do not subtract sensor outputs to remove the bias as we did in §4.5 because it would spread the noise (double the noise range) and make the ADC value estimation less accurate. Both $\mathbf{O}$ and $\widetilde{\mathbf{G}}$ are in the units of 2^{-16} dps so they only contain integer values. Nevertheless, the estimated $\widetilde{\mathbf{I}}$ is not guaranteed to be correct due to the perturbation, which is why we observed a few outliers in Figure 5.9.

Then, we estimate the clean gyroscope outputs (i.e., without added noise) by:

$$\widetilde{\mathbf{O}} = \widetilde{\mathbf{G}}\widetilde{\mathbf{I}}$$

And the offset between the estimated outputs $\widetilde{\mathbf{O}}$ and observed outputs $\mathbf{O}$ can be calculated by:

$$\boldsymbol{\Delta} = \mathbf{O} - \widetilde{\mathbf{O}}$$

Since the added noise is uniformly distributed in the range $[-1997, 1997] \times 2^{-16}$ dps, any offset beyond this range is caused by either an incorrect gain matrix (i.e., $\widetilde{\mathbf{G}} \neq \mathbf{G}$) or

incorrect ADC estimations (i.e., $\widetilde{\mathbf{I}} \neq \mathbf{I}$). To quantify the error in each data sample, we define the following error function:

$$f(\mathbf{\Delta}_i) = \sum_{\delta \in \mathbf{\Delta}_i} \max(|\delta| - 1997, 0)$$

Furthermore, we define the range-related likelihood function for each data sample as follows:

$$l_r(\mathbf{\Delta}_i) = \exp(-\frac{f(\mathbf{\Delta}_i)}{T})$$

where $\exp(\cdot)$ is the natural exponential function and T is the temperature variable used in simulated annealing that decreases over iterations.

Although the likelihood function $l_r(\mathbf{\Delta}_i)$ penalises data samples with an offset beyond the added noise range, it does not give us any information about the distribution of data within the range. In general, the ADC outputs of the gyroscope in every axis follow a normal distribution when the device is resting on a platform such as a desk (as shown in Figure 5.8). If the device has moved during the data collection, we can use a stationary position filter to get the segments with stationary measurements. Therefore, we can fit a normal distribution $N(\mu_a, \sigma_a)$ to $\widetilde{\mathbf{I}}_a$ for each axis $a \in \{x, y, z\}$. Then, based on the normal distribution, we can calculate the distribution-related likelihood function, $l_d(\mathbf{I}_i)$, by:

$$l_d(\mathbf{I}_i) = \prod_{a \in \{x,y,z\}} p_a(I_{i_a})$$

where $p_a(\cdot)$ is the probability density function of the normal distribution $N(\mu_a, \sigma_a)$ and I_{i_a} is the output $\mathbf{I}_i$ in the axis a.

Finally, we define our objective function $L(\mathbf{O}|\widetilde{\mathbf{G}})$ as a negative log likelihood function:

$$L(\mathbf{O}|\widetilde{\mathbf{G}}) = -\sum_{i=1}^{N} \log(l_r(\mathbf{\Delta}_i) l_d(\mathbf{I}_i))$$

where N is the number of gyroscope outputs. Then, the true gain matrix can be estimated by the following equation:

$$\mathbf{G} = \arg\min_{\widetilde{\mathbf{G}}} L(\mathbf{O}|\widetilde{\mathbf{G}}) \tag{5.4}$$

5.6.2.2 Simulated annealing

Simulated annealing is a probabilistic technique for solving optimisation problems and is often used in the presence of large numbers of local optima (§2.6). Since the objective function in Equation 5.4 is highly non-smooth, we use simulated annealing to solve this optimisation problem.

First, we set the initial state of the candidate gain matrix to the nominal value (i.e.,

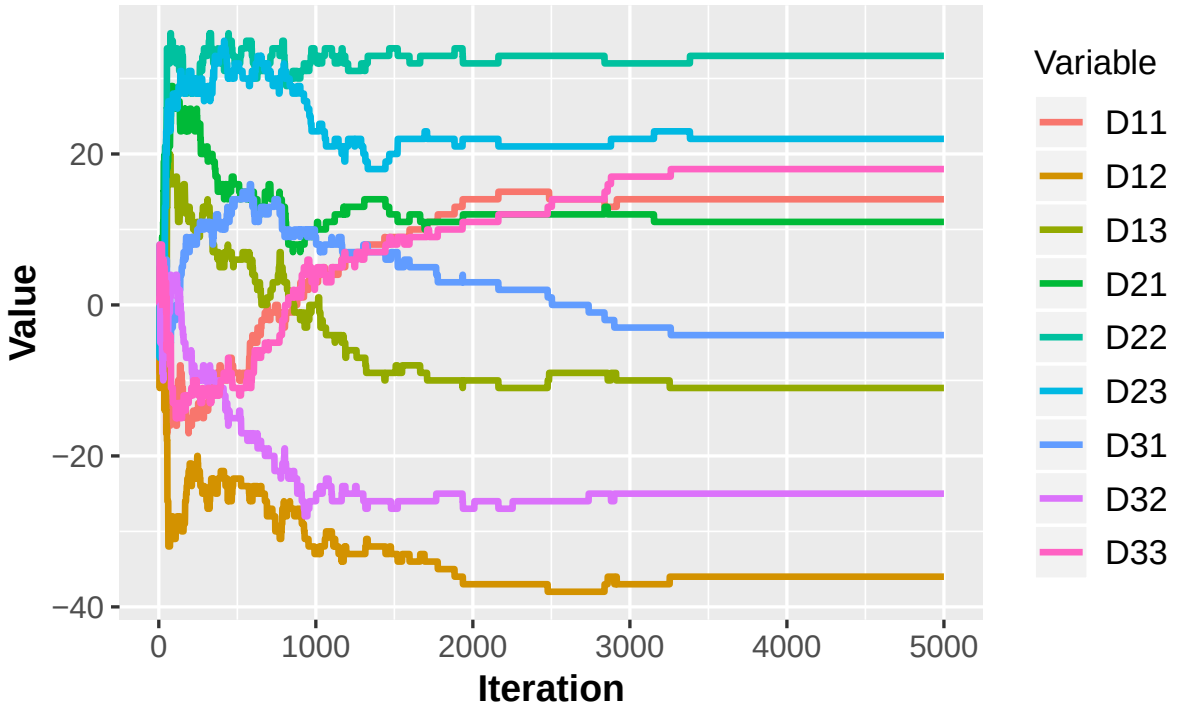


Figure 5.10: Estimated GyroID at each iteration (iPhone XS)

$\widetilde{\mathbf{G}}^0 = \mathbf{G}_0$) and the temperature parameter T to 10. The temperature T will decrease in each iteration and finishes at 0.1. Then, we calculate the objective function $L(\mathbf{O}|\widetilde{\mathbf{G}}^0)$ and denote its value as L^0.

In each following round t, we propose a new candidate gain matrix by adding a random perturbation to the previous state:

$$\widetilde{\mathbf{G}}^t = \widetilde{\mathbf{G}}^{t-1} + \text{round}(\sigma^t \mathbf{R}^t) \tag{5.5}$$

Here, $\mathbf{R}^t$ is a 3-by-3 matrix that contains random floating-point values sampled from a standard uniform distribution at round t. We also use a step parameter σ^t to control the step size at each round; its initial state is set to 5 (i.e., $\sigma^0 = 5$) to allow bigger steps in the beginning.

Then, we calculate the objective value L^t and compare it with the previous objective value L^{t-1}. If L^t is lower than L^{t-1}, we always accept the proposal and keep $\widetilde{\mathbf{G}}^t$ as the latest estimate of gain matrix. Otherwise, we choose to accept $\widetilde{\mathbf{G}}^t$ or keep $\widetilde{\mathbf{G}}^{t-1}$ probabilistically to prevent getting stuck in a local minimum. If the proposal is accepted, we also keep the corresponding object value (i.e., $L^t = L^{t-1}$) and increase the step size slightly (e.g., $\sigma^{t+1} = \sigma^t \times 1.05$) to allow faster exploration. Otherwise, the step size will be decreased slightly (e.g., $\sigma^{t+1} = \sigma^t / 1.05$) to help finding the minimum. We also set a lower bound for σ^t at 0.7 to prevent the step size being too small to update the candidate gain matrix in Equation 5.5.

5.6.3 Results and discussion

We test the algorithm on our iPhone XS handset; the GyroID of this iPhone XS is:

$$\begin{bmatrix} 14 & -36 & -11 \\ 11 & 33 & 22 \\ -4 & -25 & 18 \end{bmatrix}$$

In particular, we first collect 50K gyroscope samples from the device at 200 Hz when it is stationary. We run our algorithm on this data for 5K iterations and present the result in Figure 5.10. The figure shows the estimated GyroID stabilises after round 3383; the stabilised GyroID is the same as the one we estimated before noise was added. We have also tested the algorithm on an iPhone X and we are also able to recover the exact gain matrix using the same approach and setup.

The experiments show that Apple's fix is still susceptible to probability-based attacks. However, we find that the attacker would need at least 50K data samples. Since the sampling frequency of the gyroscope in recent iOS devices is 200 Hz, this means the attacker would need to collect gyroscope measurements for at least 4.2 minutes when the device is resting on a platform. In addition, the attack we proposed is also computation-intensive and thus it is unlikely to be implemented directly inside a mobile app. These restrictions make the attack less practical. Since Apple has also removed access to motion sensors from Safari and Webkit, such an attack can now only be conducted via an app.

Even if Apple had adopted our proposed mitigation of adding uniform noise in the range [-0.5, 0.5] to ADC outputs, this maximum likelihood estimation based attack would still work. However, the attacker would need even more samples. A better-designed noise scheme may enhance security further, but it might be harder to implement in mobile devices and could degrade user experience.

5.7 Summary

In this chapter we investigated the performance of factory calibration fingerprinting attack on both iOS and Android devices. Using iPhone 6S as an example, we showed that the GyroID contains about 42 bits of entropy and the MagID provides an additional 25 bits of entropy. Furthermore, we demonstrated that the combination of the MagID and GyroID is very likely to be globally unique for the iPhone 6S and does not change on factory reset or after a software update. For older generations of iOS devices, such as the iPhone 4S and iPad Mini, we can further extract the AccID and use it to provide extra entropy. In addition to iOS devices, we also conducted a large study of popular Android device models in the market and found that all Google Pixel phones except for the

Pixel 1/1 XL can be fingerprinted by our attack. We estimated the SensorID entropy for each vulnerable Pixel model and showed that it provides approximately 57 bits of entropy for the Pixel 4/4 XL. Recently, we analysed Apple's fix to our attack and showed it is still possible to extract the GyroID even after the fix, although doing so would require significantly more data and computation power. Since it is no longer possible to access the motion sensors in iOS browsers, the opportunity to launch this attack is restricted to installed apps.

Although this work mainly targets the motion sensors found in mobile devices, the concept of a factory calibration fingerprinting attack is widely applicable. The contribution of this work lies not just in discovering and fixing a vulnerability, but in raising awareness of the privacy implications of a widely deployed industrial practice and encouraging future studies in this area. Mobile devices are integrated with an increasing number of sensors. Future researchers can use a similar approach to this work to determine whether a sensor is factory calibrated and design corresponding algorithms to recover these calibration parameters. In addition, our entropy analysis of the calibration fingerprint provides a reference for future entropy estimation applications. We expect future research will successfully perform factory calibration fingerprinting attacks on other types of sensor.

CHAPTER 6

Conclusion

The unique identification of software and hardware components has important implications for the privacy and security of mobile platforms. This dissertation makes significant contributions to this research topic by providing novel and practical algorithms to generate reliable software and hardware fingerprints. The focus was on two aspects: a reliable software library fingerprinting tool that is resilient against code obfuscation, and a practical hardware device fingerprint that allows tracking users across apps and websites.

We focused our study of library fingerprinting on the Android platform due to the availability of well-established analytical tools. This stream of work can not only help to improve the accuracy of app clone detection but also to detect the use of vulnerable libraries and software license violation. In reference to the research questions about library fingerprinting listed in Introduction, our work provided the following insights:

- *How to reliably identify third-party Android libraries when a large portion of library code is deleted and when package modifications are applied?*

  In Chapter 3 we presented LibID, a library fingerprinting tool that is more resilient to code shrinking and package modification than state-of-the-art tools. This is achieved by exploiting class dependencies and designing novel processes such as *relation pruning* and *ghost hunting* to alleviate the interference of non-library code. We showed that the library fingerprinting problem can be formulated using binary integer programming models. LibID is able to identify specific versions of third-party libraries in candidate apps through static analysis of app binaries coupled with a database of third-party libraries.

- *How does our library fingerprinting solution compare with existing studies?*

  Several library fingerprinting approaches have been proposed to detect third-party libraries used in an Android app. However, in Chapter 3 we demonstrated these techniques are not robust against popular code obfuscators, such as ProGuard that

is used in nearly half of popular apps on the Google Play store. For evaluation, we proposed a novel approach to generate synthetic apps to tune the detection thresholds. Then, we used F-Droid apps as the ground truth to evaluate LibID under four different obfuscation settings. LibID achieved an F_1 score of more than 0.5 in all cases while prior work is below 0.25.

- *How common do popular Android apps use known vulnerable libraries?*

  By way of an example, we applied LibID to detecting the use of vulnerable `OkHttp` libraries in 3 958 most popular apps on the Google Play Store. After a year of the vulnerability being disclosed, we found 393 apps still used a vulnerable version of the `OkHttp` library, including three of Google's apps. We sent emails to relevant app developers and only 22 non-automatic replies have been received.

We also looked for a better device fingerprinting approach. Mobile devices typically contain a vast array of sensors from accelerometers and GPS units, to cameras and microphones. Data from these sensors is accessible to application programmers to build context-aware applications. Good sensor accuracy is often crucial, and therefore manufacturers often use per-device factory calibration to compensate for systematic errors introduced during manufacture. In Introduction, we have raised a few research questions about fingerprinting devices by the factory calibration. In summary, our work provided the following answers to these questions:

- *How to practically recover the factory calibration parameters of sensors from their output?*

  In Chapter 4 we presented a new type of fingerprinting attack on sensor data: *factory calibration fingerprinting*. A factory calibration fingerprinting attack infers the per-device factory calibration data from a device by careful analysis of the sensor output alone. Such an attack does not require direct access to any calibration parameters since these are often embedded inside the firmware of the device and are not directly accessible by application developers. We demonstrated the potential of this new class of attack by performing factory calibration fingerprinting attacks on motion sensors found in iOS and Android devices. These sensors were good candidates because access to these sensors did not require any special permissions, and the data can be accessed via both a native app installed on a device and also by JavaScript when visiting a website on a smartphone. We were able to perform a very effective factory calibration fingerprinting attack: our approach requires fewer than 100 samples of sensor data and takes less than one second to collect and process into a device fingerprint that does not change over time or after a factory reset.

- *What smartphone models are vulnerable to the factory calibration fingerprinting attack?*

  In Chapter 4 we have proposed an approach to check if a motion sensor is factory calibrated based on its output. Applying this approach, we surveyed the accelerometer, gyroscope, and magnetometer in iOS devices and summarised device models that are vulnerable to the factory calibration fingerprinting attack for each sensor type in Chapter 5. We found that our attack applies to at least one type of motion sensors for every iOS device we have tested. In addition, we collected motion sensor data from 146 popular Android device models and confirmed that the accelerometer and gyroscope in several Google Pixel models can also be fingerprinted by our attack.

- *How unique are the calibration parameters and how does it compare with existing motion sensor-based device fingerprinting studies?*

  In Chapter 5 we demonstrated that our approach is very likely to produce a globally unique fingerprint for iOS devices, with an estimated 67 bits of entropy in the fingerprint for iPhone 6S devices. In addition, Google Pixel phones can also be fingerprinted by our approach and the fingerprint has about 57 bits of entropy for the Pixel 4/4 XL. In both cases, the factory calibration-based fingerprint provides more entropy than the device fingerprint generated by previous work, which has at most 13 bits of entropy.

To protect users from the factory calibration fingerprinting attack, we suggested two countermeasures to Apple and Google that have little impact on user experience. We later analysed Apple's fix and found that while it is still possible to extract the exact fingerprint after the fix, the attack is less practical due to its cost. This analysis also shows it is difficult to remove unique device fingerprints without compromising user experience. For instance, blocking apps' access to motion sensors would stop our attack but it would also affect the normal functioning of many apps. Manufacturers could also choose to not factory calibrate their sensors but then the sensor readings will be less accurate. Moreover, because there are idiosyncrasies across different sensors due to manufacturing imperfections, adversaries can still exploit these differences to identify individual sensors even if the factory calibration has not been applied. Apart from motion sensors, studies have shown that a distinctive device fingerprint can also be generated from other types of hardware (§2.3.3). Nevertheless, as demonstrated in this dissertation, it is often viable to increase the cost of launching a fingerprinting attack without sacrificing much usability. We recommend manufacturers and vendors integrate mitigation methods to thwart calibration fingerprinting attacks in current and future products.

Our work shows that reliable software and hardware fingerprints can be generated through careful mathematical formulation and reasoning. Compared with prior work, our

methods are easy to explain and interpret and have better performance when code obfuscation and factory calibration apply. By disclosing the factory calibration fingerprinting attack to relevant vendors and notifying app developers of the use of vulnerable third-party libraries, our work has made practical contributions to enhancing the privacy and security of mobile platforms.

6.1 Future research

Despite its merits, our work is not without its limitations. For example, our library fingerprinting approach may not be robust against more advanced code obfuscation techniques and our method for estimating the entropy of the calibration fingerprint does not detect higher-order correlations. Future research can improve the work presented in this dissertation by addressing these problems. Based on our research, future work may also want to investigate the following topics:

Library fingerprinting in pre-installed Android apps. Android device vendors often include a set of pre-installed apps in the firmware. Although some of these apps are essential for the normal functioning of the device, many of them are not. These apps typically have system privileges and cannot be uninstalled by the user. Despite being pervasive, pre-installed apps have largely avoided the scrutiny of researchers because they are often not available on Google Play and are typically assumed to be trusted as they are chosen by the device vendor. However, recent studies have shown that there are a variety of privacy and security issues in these apps. In particular, Gamba *et al.* collected and analysed pre-installed Android apps from more than 200 vendors via crowdsourcing [179]. Their analysis uncovered that user tracking is pervasive in pre-installed Android apps. Elsabagh *et al.* extracted pre-installed apps directly from 2 017 Android vendor firmware images and their study uncovered 850 unique privilege-escalation vulnerabilities in these apps [180]. In this dissertation we have shown that third-party libraries are popular in Android apps and many apps use a vulnerable version of these libraries. Nevertheless, the usage of third-party libraries in pre-installed apps is largely unexplored. Future work can apply library fingerprinting tools, such as LibID, to study the popularity of third-party libraries in pre-installed apps. Most of the pre-installed apps can only be updated via Over-The-Air (OTA) updates which are less frequent. Future work can also investigate how often do developers update the third-party libraries in these apps.

Factory calibration fingerprinting other sensors. So far we have only investigated the possibility of fingerprinting mobile devices using the factory calibration parameters of their motion sensors. We prioritized these sensors because they are pervasive in modern

mobile devices and require no special permission to access. In addition, the calibration of these sensors follows a linear model which makes it possible to recover the exact calibration matrix. However, the idea of a calibration fingerprint attack is widely applicable. We anticipate factory calibration information used in other embedded sensors may also be recovered and used as a fingerprint, and therefore we expect future research will successfully perform factory calibration fingerprinting attacks on other types of sensor. For example, digital cameras have a rich set of factory calibration information, including distortion coefficients and intrinsic and extrinsic matrices. If this information can be recovered precisely, it may be able to uniquely fingerprint a digital camera. On Google Pixel 3 and 4 devices, we found these factory calibration parameters are also recorded in files that require root permission to access. Factory calibration fingerprinting may also be used to identify the source camera of a particular image if these calibration parameters can be extracted directly from digital images.

Fingerprinting uncalibrated sensors. As discussed in §5.3, we have found that motion sensors in many Android devices are not per-device factory calibrated. Therefore, our factory calibration fingerprinting does not apply to these sensors. Nevertheless, it is still possible to fingerprint Android devices using a calibration-based scheme. In particular, since manufacturing imperfections are inevitable, uncalibrated sensors are likely to give a more diverse range of outputs given the same input than a calibrated sensor. This diversity may allow us to calibrate these sensors in a mobile app. For example, Grammenos *et al.* have proposed a calibration scheme for the accelerometer and gyroscope in Android devices that does not require any user interaction [164]. Once the calibration matrices are estimated, they can be used as a sensor fingerprint to identify the device. However, due to the lack of high-precision calibration equipment, estimating the calibration matrix of a sensor at different times may produce slightly different results. Therefore, practical attacks would need to use either a clustering- or a rounding-based approach to identify a particular device uniquely.

Bibliography

[1] We Are Social. Digital in 2020. https://wearesocial.com/digital-2020, 2020. Accessed: 14 July 2020.

[2] Statista. Mobile advertising spending worldwide 2007-2022. https://www.statista.com/statistics/303817/mobile-internet-advertising-revenue-worldwide, 2019. Accessed: 14 July 2020.

[3] Statista. Distribution of free and paid apps in the Apple App Store and Google Play as of June 2020. https://www.statista.com/statistics/263797/number-of-applications-for-mobile-phones, 2020. Accessed: 15 July 2020.

[4] Fyber. 2019 state of in-app advertising and monetization. https://blog.fyber.com/2019-state-of-in-app-advertising-and-monetization/, 2020. Accessed: 28 October 2020.

[5] Avi Goldfarb and Catherine E Tucker. Privacy regulation and online advertising. *Management Science*, 57(1):57–71, 2011.

[6] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. WuKong: A Scalable and Accurate Two-Phase Approach to Android App Clone Detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 71–82. ACM, 2015.

[7] Shashi Shekhar, Michael Dietz, and Dan S Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the 21st USENIX Security Symposium*, pages 553–567. USENIX Association, 2012.

[8] Wenbo Yang, Juanru Li, Yuanyuan Zhang, Yong Li, Junliang Shu, and Dawu Gu. APKLancet: Tumor Payload Diagnosis and Purification for Android Applications. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 483–494. ACM, 2014.

[9] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. LibRadar: Fast and Accurate Detection of Third-Party Libraries in Android Apps. In *Proceedings of the*

38th International Conference on Software Engineering Companion (ICSE-C), pages 653–656. ACM, 2016.

[10] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. LibD: Scalable and Precise Third-Party Library Detection in Android Markets. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, pages 335–346. IEEE, 2017.

[11] Yan Wang and Atanas Rountev. Who Changed You? Obfuscator Identification for Android. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 154–164. IEEE, 2017.

[12] Michael Backes, Sven Bugiel, and Erik Derr. Reliable Third-Party Library Detection in Android and Its Security Applications. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 356–367. ACM, 2016.

[13] Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. Orlis: Obfuscation-Resilient Library Detection for Android. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 13–23. ACM, 2018.

[14] Apple. Details for app privacy questions now available. `https://developer.apple.com/news/?id=hx9s63c5&1599152522`, 2020. accessed: 26 October 2020.

[15] Giles Hogben. Changes to device identifiers in Android O. `https://android-developers.googleblog.com/2017/04/changes-to-device-identifiers-in.html`, 2017. Accessed: 27 July 2020.

[16] Sanorita Dey, Nirupam Roy, Wenyuan Xu, Romit Roy Choudhury, and Srihari Nelakuditi. AccelPrint: Imperfections of Accelerometers Make Smartphones Trackable. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2014.

[17] Anupam Das, Nikita Borisov, and Edward Chou. Every Move You Make: Exploring Practical Issues in Smartphone Motion Sensor Fingerprinting and Countermeasures. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2018(1):88–108, 2018.

[18] Jiexin Zhang, Alastair R Beresford, and Ian Sheret. SensorID: Sensor Calibration Fingerprinting for Smartphones. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*, pages 638–655. IEEE, 2019.

[19] Alec J Jeffreys, Victoria Wilson, and Swee Lay Thein. Individual-specific 'fingerprints' of human DNA. *Nature*, 316(6023):76–79, 1985.

[20] Mark Hawthorne. *Fingerprints: Analysis and Understanding*. CRC Press, 2008.

[21] Paul Rascagneres and Vitor Ventura. Fingerprint cloning: myth or reality? `https://blog.talosintelligence.com/2020/04/fingerprint-research.html`, 2020. Accessed: 26 September 2020.

[22] Zoe Kleinman. Politician's fingerprint 'cloned from photos' by hacker. `https://www.bbc.com/news/technology-30623611`, 2014. Accessed: 26 September 2020.

[23] Darknetlive. Dream vendor "canna_bars" sentenced to prison. `https://darknetlive.com/post/dream-vendor-canna-bars-sentenced-to-prison/`, 2019. Accessed: 26 September 2020.

[24] Giulio Lovisotto, Henry Turner, Simon Eberz, and Ivan Martinovic. Seeing red: Ppg biometrics using smartphone cameras. In *Proceedings of the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 818–819. IEEE/CVF, 2020.

[25] LG Electronics. LG G8 ThinQ air motion: learn how touchless works. `https://www.lg.com/us/mobile-phones/g8-thinq/air-motion`, 2019. Accessed: 29 September 2020.

[26] Vincent Chang. LG G8 ThinQ: unlock this phone with your veins. `https://www.straitstimes.com/tech/smartphones/unlock-this-phone-with-your-veins`, 2019. Accessed: 29 September 2020.

[27] Apple. About Face ID advanced technology. `https://support.apple.com/en-us/HT208108`, 2020. Accessed: 28 September 2020.

[28] Tom Warren. Apple's Face ID struggles detailed in new iPhone X report. `https://www.theverge.com/2017/10/25/16542614/apple-iphone-x-face-id-manufacturing-issues-report`, 2017. Accessed: 28 September 2020.

[29] Chris Smith. Whoops! Apple Face ID can be fooled with glasses and tape, researchers find. `https://www.trustedreviews.com/news/apple-face-id-can-be-fooled-with-glasses-and-tape-researchers-find-3929324`, 2019. Accessed: 28 September 2020.

[30] Chaos Computer Club. Hacking the Samsung Galaxy S8 irisscanner. `https://media.ccc.de/v/biometrie-s8-iris-en`, 2017. Accessed: 28 September 2020.

[31] Takayuki Arakawa, Takafumi Koshinaka, Shohei Yano, Hideki Irisawa, Ryoji Miyahara, and Hitoshi Imaoka. Fast and Accurate Personal Authentication Using Ear Acoustics. In *Proceedings of the 8th Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pages 1–4. IEEE, 2016.

[32] James M Kates. A Computer Simulation of Hearing Aid Response and the Effects of Ear Canal Size. *The Journal of the Acoustical Society of America*, 83(5):1952–1963, 1988.

[33] Pim T Tuyls, Evgeny Verbitskiy, Tanya Ignatenko, Daniel Schobben, and Ton H Akkermans. Privacy-protected biometric templates: acoustic ear identification. In *Biometric Technology for Human Identification*, volume 5404 of *Proceedings of SPIE*, pages 176–182. International Society for Optics and Photonics, 2004.

[34] Yang Gao, Wei Wang, Vir V Phoha, Wei Sun, and Zhanpeng Jin. EarEcho: Using Ear Canal Echo for Wearable Authentication. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)*, 3(3):1–24, 2019.

[35] Takashi Amesaka, Hiroki Watanabe, and Masanori Sugimoto. Facial Expression Recognition Using Ear Canal Transfer Function. In *Proceedings of the 23rd International Symposium on Wearable Computers (ISWC)*, pages 1–9. ACM, 2019.

[36] Oldrich Plchot, Lukas Burget, Hagai Aronowitz, and Pavel Matejka. Audio enhancing with DNN autoencoder for speaker recognition. In *Proceedings of the 41st International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5090–5094. IEEE, 2016.

[37] Chao Shen, Yuanxun Li, Yufei Chen, Xiaohong Guan, and Roy A Maxion. Performance Analysis of Multi-Motion Sensor Behavior for Active Smartphone Authentication. *IEEE Transactions on Information Forensics and Security (TIFS)*, 13(1):48–62, 2017.

[38] Chen Wang, Junping Zhang, Jian Pu, Xiaoru Yuan, and Liang Wang. Chrono-Gait Image: A Novel Temporal Template for Gait Recognition. In *Proceedings of the 11th European Conference on Computer Vision (ECCV)*, pages 257–270. Springer, 2010.

[39] Simon Eberz, Kasper B Rasmussen, Vincent Lenders, and Ivan Martinovic. Looks like eve: Exposing insider threats using eye movement biometrics. *ACM Transactions on Privacy and Security (TOPS)*, 19(1):1–31, 2016.

[40] Simon Eberz, Giulio Lovisotto, Kasper B Rasmussen, Vincent Lenders, and Ivan Martinovic. 28 blinks later: Tackling practical challenges of eye movement bio-

metrics. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1187–1199. ACM, 2019.

[41] Artur Janicki, Federico Alegre, and Nicholas Evans. An assessment of automatic speaker verification vulnerabilities to replay spoofing attacks. *Security and Communication Networks*, 9(15):3030–3044, 2016.

[42] Tomi Kinnunen, Md Sahidullah, Héctor Delgado, Massimiliano Todisco, Nicholas Evans, Junichi Yamagishi, and Kong Aik Lee. The ASVspoof 2017 Challenge: Assessing the Limits of Replay Spoofing Attack Detection. *Interspeech 2017*, pages 2–6, 2017.

[43] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. DolphinAttack: Inaudible Voice Commands. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 103–117, 2017.

[44] Qiben Yan, Kehai Liu, Qin Zhou, Hanqing Guo, and Ning Zhang. SurfingAttack: Interactive Hidden Attack on Voice Assistants Using Ultrasonic Guided Waves. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2020.

[45] Henry Turner, Giulio Lovisotto, and Ivan Martinovic. Attacking speaker recognition systems with phoneme morphing. In *Proceedings of the 25th European Symposium on Research in Computer Security (ESORICS)*, pages 471–492. Springer, 2019.

[46] Patrizio Campisi. *Security and Privacy in Biometrics*. Springer, 2013.

[47] Antitza Dantcheva, Petros Elia, and Arun Ross. What Else Does Your Biometric Data Reveal? A Survey on Soft Biometrics. *IEEE Transactions on Information Forensics and Security (TIFS)*, 11(3):441–467, 2015.

[48] the hacker news. Facebook sdk vulnerability puts millions of smartphone users' accounts at risk. `http://thehackernews.com/2014/07/facebook-sdk-vulnerability-puts.html`, 2014. accessed: 31 August 2020.

[49] jesse wilson. OkHttp certificate pinning vulnerability. `https://publicobject.com/2016/02/11/okhttp-certificate-pinning-vulnerability/`, 2016. accessed: 31 August 2020.

[50] the hacker news. Warning: 18,000 Android apps contains code that spy on your text messages. `https://thehackernews.com/2015/10/android-apps-steal-sms.html`, 2015. accessed: 31 August 2020.

[51] the hacker news. Backdoor in Baidu Android SDK puts 100 million devices at risk. https://thehackernews.com/2015/11/android-malware-backdoor.html, 2015. accessed: 31 August 2020.

[52] Takuya Watanabe, Mitsuaki Akiyama, Fumihiro Kanei, Eitaro Shioji, Yuta Takata, Bo Sun, Yuta Ishi, Toshiki Shibahara, Takeshi Yagi, and Tatsuya Mori. Understanding the Origins of Mobile App Vulnerabilities: A Large-scale Measurement Study of Free and Paid Apps. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, pages 14–24. IEEE, 2017.

[53] 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. *ACM SIGPLAN Notices*, 49(6):259–269, 2014.

[54] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining Apps for Abnormal Usage of Sensitive Data. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 426–436. IEEE, 2015.

[55] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1329–1341. ACM, 2014.

[56] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying Open-Source License Violation and 1-Day Security Risk at Large Scale. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2169–2185. ACM, 2017.

[57] Wei Tang, Ping Luo, Jialiang Fu, and Dan Zhang. LibDX: A Cross-Platform and Accurate System to Detect Third-Party Libraries in Binary Code. In *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 104–115. IEEE, 2020.

[58] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2011.

[59] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, 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. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[60] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 611–622. ACM, 2013.

[61] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 241–250. ACM, 2011.

[62] Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 37–48. ACM, 2015.

[63] Anand Paturi, Patrick Gage Kelley, and Subhasish Mazumdar. Introducing Privacy Threats from Ad Libraries to Android Users through Privacy Granules. In *Proceedings of the 2015 NDSS Workshop on Usable Security (USEC)*. Internet Society, 2015.

[64] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What Mobile Ads Know About Mobile Users. In *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2016.

[65] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, and Yuri Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *Proceedings of the 22nd USENIX Security Symposium*, pages 399–314. USENIX Association, 2013.

[66] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2014.

[67] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. FLEX-DROID: Enforcing In-App Privilege Separation in Android. In *Proceedings of the 23th Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2016.

[68] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS. In

Proceedings of the 37th IEEE Symposium on Security and Privacy (SP), pages 357–376. IEEE, 2016.

[69] Saksham Chitkara, Nishad Gothoskar, Suhas Harish, Jason I Hong, and Yuvraj Agarwal. Does this App Really Need My Location? Context-Aware Privacy Management for Smartphones. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(3):42, 2017.

[70] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song. NetworkProfiler: Towards Automatic Fingerprinting of Android Apps. In *Proceedings of the 2013 IEEE International Conference on Computer Communications (INFOCOM)*, pages 809–817. IEEE, 2013.

[71] Qinglong Wang, Amir Yahyavi, Bettina Kemme, and Wenbo He. I Know What You Did On Your Smartphone: Inferring App Usage Over Encrypted Data Traffic. In *Proceedings of the 2015 IEEE Conference on Communications and Network Security (CNS)*, pages 433–441. IEEE, 2015.

[72] Vincent F Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. AppScanner: Automatic Fingerprinting of Smartphone Apps From Encrypted Network Traffic. In *Proceedings of the 2016 IEEE European Symposium on Security and Privacy (EuroSP)*, pages 439–454. IEEE, 2016.

[73] Vincent F Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. Robust Smartphone App Identification Via Encrypted Network Traffic Analysis. *IEEE Transactions on Information Forensics and Security (TIFS)*, 13(1):63–78, 2017.

[74] Giuseppe Aceto, Domenico Ciuonzo, Antonio Montieri, and Antonio Pescapé. Multi-Classification Approaches for Classifying Mobile App Traffic. *Journal of Network and Computer Applications*, 103:131–145, 2018.

[75] Giuseppe Aceto, Domenico Ciuonzo, Antonio Montieri, and Antonio Pescapé. Mobile Encrypted Traffic Classification Using Deep Learning: Experimental Evaluation, Lessons Learned, and Challenges. *IEEE Transactions on Network and Service Management (TNSM)*, 16(2):445–458, 2019.

[76] Jeremy Martin, Travis Mayberry, Collin Donahue, Lucas Foppe, Lamont Brown, Chadwick Riggins, Erik C Rye, and Dane Brown. A Study of MAC Address Randomization in Mobile Devices and When it Fails. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017(4):365–383, 2017.

[77] Samy Kamkar. Evercookie – virtually irrevocable persistent cookies. `https://samy.pl/evercookie`, 2010. Accessed: 11 September 2020.

[78] Jonathan Schmidt. Does the dark side still have (ever) cookies? https://faui1-files.cs.fau.de/public/publications/df/df-whitepaper-18.pdf, 2020. Accessed: 2 October 2020.

[79] Google. Best practices for unique identifiers. https://developer.android.com/training/articles/user-data-ids, 2020. Accessed: 27 July 2020.

[80] Apple. User privacy and data use. https://developer.apple.com/app-store/user-privacy-and-data-use/, 2020. Accessed: 27 July 2020.

[81] William Enck, Damien Octeau, Patrick D McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, pages 315–330. USENIX Association, 2011.

[82] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 116–127. ACM, 2007.

[83] Jeremy Martin, Douglas Alpuche, Kristina Bodeman, Lamont Brown, Ellis Fenske, Lucas Foppe, Travis Mayberry, Erik Rye, Brandon Sipes, and Sam Teplov. Handoff All Your Privacy – A Review of Apple's Bluetooth Low Energy Continuity Protocol. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2019(4):34–53, 2019.

[84] Johannes K Becker, David Li, and David Starobinski. Tracking Anonymized Bluetooth Devices. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2019(3):50–65, 2019.

[85] Erik Sy, Christian Burkert, Hannes Federrath, and Mathias Fischer. A QUIC Look at Web Tracking. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2019(3):255–266, 2019.

[86] A Selcuk Uluagac, Sakthi V Radhakrishnan, Cherita Corbett, Antony Baca, and Raheem Beyah. A Passive Technique for Fingerprinting Wireless Devices with Wired-side Observations. In *Proceedings of the 1st IEEE Conference on Communications and Network Security (CNS)*, pages 305–313. IEEE, 2013.

[87] Christoph Neumann, Olivier Heen, and Stéphane Onno. An empirical study of passive 802.11 Device Fingerprinting. In *Proceedings of the 32nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 593–602. IEEE, 2012.

[88] Tim Stöber, Mario Frank, Jens Schmitt, and Ivan Martinovic. Who do you sync you are? Smartphone Fingerprinting via Application Behaviour. In *Proceedings of the 6th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, pages 7–12. ACM, 2013.

[89] Peter Eckersley. How Unique Is Your Web Browser? In *Proceedings of the 10th International Conference on Privacy Enhancing Technologies (PETS)*, pages 1–18. Springer, 2010.

[90] Keaton Mowery and Hovav Shacham. Pixel Perfect: Fingerprinting Canvas in HTML5. In *Proceedings of the 2012 SP Workshop on Web 2.0 Security and Privacy (W2SP)*, pages 1–12. IEEE, 2012.

[91] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 674–689. ACM, 2014.

[92] Lily H Newman. Apple just made Safari the good privacy browser. `https://www.wired.com/story/apple-safari-privacy-wwdc`, 2018. Accessed: 2 October 2020.

[93] Brave. Brave, fingerprinting, and privacy budgets. `https://brave.com/brave-fingerprinting-and-privacy-budgets`, 2019. Accessed: 4 September 2020.

[94] Qiang Xu, Rong Zheng, Walid Saad, and Zhu Han. Device Fingerprinting in Wireless Networks: Challenges and Opportunities. *IEEE Communications Surveys & Tutorials*, 18(1):94–104, 2015.

[95] Adam C Polak, Sepideh Dolatshahi, and Dennis L Goeckel. Identifying Wireless Users via Transmitter Imperfections. *IEEE Journal on Selected Areas in Communications*, 29(7):1469–1479, 2011.

[96] Sepideh Dolatshahi, Adam Polak, and Dennis L Goeckel. Identification of wireless users via power amplifier imperfections. In *Proceedings of the 44th Asilomar Conference on Signals, Systems and Computers*, pages 1553–1557. IEEE, 2010.

[97] Adam C Polak and Dennis L Goeckel. Wireless Device Identification Based on RF Oscillator Imperfections. *IEEE Transactions on Information Forensics and Security*, 10(12):2492–2501, 2015.

[98] Vladimir Brik, Suman Banerjee, Marco Gruteser, and Sangho Oh. Wireless Device Identification with Radiometric Signatures. In *Proceedings of the 14th ACM International Conference on Mobile Computing and Networking (MobiCom)*, pages 116–127. ACM, 2008.

[99] Seth Andrews, Ryan M Gerdes, and Ming Li. Towards Physical Layer Identification of Cognitive Radio Devices. In *Proceedings of the 5th IEEE Conference on Communications and Network Security (CNS)*, pages 1–9. IEEE, 2017.

[100] Kevin Merchant, Shauna Revay, George Stantchev, and Bryan Nousain. Deep Learning for RF Device Fingerprinting in Cognitive Communication Networks. *IEEE Journal of Selected Topics in Signal Processing*, 12(1):160–167, 2018.

[101] Soorya Gopalakrishnan, Metehan Cekic, and Upamanyu Madhow. Robust Wireless Fingerprinting via Complex-Valued Neural Networks. In *Proceedings of the 38th IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2019.

[102] Tadayoshi Kohno, Andre Broido, and Kimberly C Claffy. Remote Physical Device Fingerprinting. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2(2):93–108, 2005.

[103] Steven J Murdoch. Hot or Not: Revealing Hidden Services by their Clock Skew. In *Proceedings of the 13th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 27–36. ACM, 2006.

[104] Swati Sharma, Alefiya Hussain, and Huzur Saran. Experience with Heterogenous Clock-Skew based Device Fingerprinting. In *Proceedings of the 2012 Workshop on Learning from Authoritative Security Experiment Results (LASER)*, pages 9–18. ACM, 2012.

[105] Jun Huang, Wahhab Albazrqaoe, and Guoliang Xing. BlueID: A Practical System for Bluetooth Device Identification. In *Proceedings of the 33rd IEEE Conference on Computer Communications (INFOCOM)*, pages 2849–2857. IEEE, 2014.

[106] Gabi Nakibly, Gilad Shelef, and Shiran Yudilevich. Hardware Fingerprinting Using HTML5. *arXiv preprint arXiv:1503.01408*, 2015.

[107] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. Clock Around the Clock: Time-Based Device Fingerprinting. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1502–1514. ACM, 2018.

[108] William Banks Clarkson. *Breaking Assumptions: Distinguishing Between Seemingly Identical Items Using Cheap Sensors*. PhD thesis, Princeton University, 2012.

[109] Anupam Das, Nikita Borisov, and Matthew Caesar. Do You Hear What I Hear? Fingerprinting Smart Devices Through Embedded Acoustic Components. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 441–452. ACM, 2014.

[110] Hristo Bojinov, Yan Michalevsky, Gabi Nakibly, and Dan Boneh. Mobile Device Identification via Sensor Fingerprinting. *arXiv preprint arXiv:1408.1416*, 2014.

[111] Matthew Dekker and Vimal Kumar. Using Audio Characteristics for Mobile Device Authentication. In *Proceedings of the 13th International Conference on Network and System Security (NSS)*, pages 98–113. Springer, 2019.

[112] Zhe Zhou, Wenrui Diao, Xiangyu Liu, and Kehuan Zhang. Acoustic Fingerprinting Revisited: Generate Stable Device ID Stealthily with Inaudible Sound. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 429–440. ACM, 2014.

[113] Daniel Arp, Erwin Quiring, Christian Wressnegger, and Konrad Rieck. Privacy Threats through Ultrasonic Side Channels on Mobile Devices. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroSP)*, pages 35–47. IEEE, 2017.

[114] Dajiang Chen, Ning Zhang, Zhen Qin, Xufei Mao, Zhiguang Qin, Xuemin Shen, and Xiang-Yang Li. S2M: A Lightweight Acoustic Fingerprints-Based Wireless Device Authentication Protocol. *IEEE Internet of Things Journal*, 4(1):88–100, 2016.

[115] Gianmarco Baldini, Irene Amerini, and Claudio Gentile. Microphone Identification Using Convolutional Neural Networks. *IEEE Sensors Letters*, 3(7):1–4, 2019.

[116] Sevinc Bayram, Husrev Sencar, Nasir Memon, and Ismail Avcibas. Source camera identification based on CFA interpolation. In *Proceedings of the 2005 International Conference on Image Processing (ICIP)*, pages 69–72. IEEE, 2005.

[117] Shang Gao, Guanshuo Xu, and Rui-Min Hu. Camera Model Identification Based on the Characteristic of CFA and Interpolation. In *Proceedings of the 2011 International Workshop on Digital Watermarking (IWDW)*, pages 268–280. Springer, 2011.

[118] Kai San Choi, Edmund Y Lam, and Kenneth KY Wong. Source Camera Identification Using Footprints from Lens Aberration. In *Digital Photography II*, volume 6069 of *Proceedings of SPIE*, pages 172–179. International Society for Optics and Photonics, 2006.

[119] Jaroslaw Bernacki. Digital camera identification based on analysis of optical defects. *Multimedia Tools and Applications*, 79(3):2945–2963, 2020.

[120] Jessica Fridrich. Digital Image Forensics. *IEEE Signal Processing Magazine*, 26(2):26–37, 2009.

[121] Jan Lukas, Jessica Fridrich, and Miroslav Goljan. Digital Camera Identification from Sensor Pattern Noise. *IEEE Transactions on Information Forensics and Security (TIFS)*, 1(2):205–214, 2006.

[122] Diego Valsesia, Giulio Coluccia, Tiziano Bianchi, and Enrico Magli. Compressed Fingerprint Matching and Camera Identification via Random Projections. *IEEE Transactions on Information Forensics and Security*, 10(7):1472–1485, 2015.

[123] Ambuj Mehrish, A Venkata Subramanyam, and Sabu Emmanuel. Robust PRNU Estimation From Probabilistic Raw Measurements. *Signal Processing: Image Communication*, 66:30–41, 2018.

[124] Chang-Tsun Li. Source Camera Identification Using Enhanced Sensor Pattern Noise. *IEEE Transactions on Information Forensics and Security (TIFS)*, 5(2):280–287, 2010.

[125] Zhongjie Ba, Sixu Piao, Xinwen Fu, Dimitrios Koutsonikolas, Aziz Mohaisen, and Kui Ren. ABC: Enabling Smartphone Authentication with Built-in Camera. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2018.

[126] Zhongjie Ba, Zhan Qin, Xinwen Fu, and Kui Ren. CIM: Camera in Motion for Smartphone Authentication. *IEEE Transactions on Information Forensics and Security (TIFS)*, 14(11):2987–3002, 2019.

[127] David Freire-Obregón, Fabio Narducci, Silvio Barra, and Modesto Castrillón-Santana. Deep learning for source camera identification on mobile devices. *Pattern Recognition Letters*, 126:86–91, 2019.

[128] Lukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Diaz. The Leaking Battery: A Privacy Analysis of the HTML5 Battery Status API. In *Data Privacy Management, and Security Assurance*, pages 254–263. Springer, 2016.

[129] Jose Carlos Norte. Advanced Tor browser fingerprinting. `http://jcarlosnorte.com/security/2016/03/06/advanced-tor-browser-fingerprinting.html`, 2016. Accessed: 4 September 2020.

[130] Piotr Indyk and Rajeev Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the 30th ACM Symposium on Theory of Computing (STOC)*, pages 604–613. ACM, 1998.

[131] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2011.

[132] Yushi Jing and Shumeet Baluja. VisualRank: Applying PageRank to Large-Scale Image Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 30(11):1877–1890, 2008.

[133] Jian Lu. Video fingerprinting for copy identification: from research to industry applications. In *Media Forensics and Security*, volume 7254, pages 1–15. International Society for Optics and Photonics, 2009.

[134] Erkang Zhu, Fatemeh Nargesian, Ken Q Pu, and Renée J Miller. LSH Ensemble: Internet-Scale Domain Search. *Proceedings of the VLDB Endowment*, 9(12):1185–1196, 2016.

[135] Laurence A Wolsey. *Integer Programming*. John Wiley & Sons, 1998.

[136] Ailsa H Land and Alison G Doig. An Automatic Method of Solving Discrete Programming Problems. In *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*, pages 105–132. Springer, 2010.

[137] Tobias Achterberg, Robert E Bixby, Zonghao Gu, Edward Rothberg, and Dieter Weninger. Presolve Reductions in Mixed Integer Programming. *INFORMS Journal on Computing*, 32(2):473–506, 2020.

[138] Hugues Marchand, Alexander Martin, Robert Weismantel, and Laurence Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*, 123(1-3):397–446, 2002.

[139] John W Chinneck. Practical optimization: a gentle introduction. `http://www.sce.carleton.ca/faculty/chinneck/po.html`, 2018. Accessed: 30 July 2020.

[140] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[141] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[142] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks (SecureComm)*, pages 86–103. Springer, 2013.

[143] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 175–186. ACM, 2014.

[144] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS)*, pages 37–54. Springer, 2012.

[145] Jonathan Crussell, Clint Gibler, and Hao Chen. AnDarwin: Scalable Detection of Android Application Clones Based on Semantics. *IEEE Transactions on Mobile Computing (TMC)*, 14(10):2007–2019, 2015.

[146] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. CodeMatch: obfuscation won't conceal your repackaged app. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, pages 638–648. ACM, 2017.

[147] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, pages 101–112. ACM, 2012.

[148] Theodore Book, Adam Pridgen, and Dan S Wallach. Longitudinal Analysis of Android Ad Library Permissions. *arXiv preprint arXiv:1303.0857*, 2013.

[149] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An Investigation into the Use of Common Libraries in Android Apps. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 403–414. IEEE, 2016.

[150] Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, Annamalai Narayanan, and Lipo Wang. LibSift: Automated Detection of Third-Party Libraries in Android Applications. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 41–48. IEEE, 2016.

[151] Annamalai Narayanan, Lihui Chen, and Chee Keong Chan. AdDetect: Automated Detection of Android Ad Libraries using Semantic Analysis. In *Proceedings of the 9th International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–6. IEEE, 2014.

[152] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps. In *Proceedings of the 13th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 89–103. ACM, 2015.

[153] Salman A Baset, Shih-Wei Li, Philippe Suter, and Omer Tripp. Identifying android library dependencies in the presence of code obfuscation and minimization. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 250–252. IEEE, 2017.

[154] Dennis Titze, Michael Lux, and Julian Schuette. Ordol: Obfuscation-Resilient Detection of Libraries in Android Applications. In *Proceedings of the 2017 IEEE Trustcom/BigDataSE/ICESS*, pages 618–625. IEEE, 2017.

[155] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. Detecting Third-Party Libraries in Android Applications with High Precision and Recall. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 141–152. IEEE, 2018.

[156] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017.

[157] Alexandr Andoni and Piotr Indyk. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. In *Proceedings of the 47th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 459–468. IEEE, 2006.

[158] computer security resource center. CVE-2016-2402 detail. `https://nvd.nist.gov/vuln/detail/CVE-2016-2402`, 2016. accessed: 1 October 2020.

[159] Apple. Getting raw accelerometer events. `https://developer.apple.com/documentation/coremotion/getting_raw_accelerometer_events`, 2020. Accessed: 27 July 2020.

[160] Chipworks. Comparing the InvenSense and Bosch accelerometers found in the iPhone 6. `http://www.chipworks.com/about-chipworks/overview/blog/`

comparing-invensense-and-bosch-accelerometers-found-iphone-6, 2014. Accessed: 27 July 2020.

[161] Apple. Getting raw gyroscope events. https://developer.apple.com/documentation/coremotion/getting_raw_gyroscope_events, 2020. Accessed: 27 July 2020.

[162] Apple. Required device capabilities. https://developer.apple.com/support/required-device-capabilities, 2020. Accessed: 27 July 2020.

[163] Shashi Poddar, Vipan Kumar, and Amod Kumar. A Comprehensive Overview of Inertial Sensor Calibration Techniques. *Journal of Dynamic Systems, Measurement, and Control*, 139(1):011006, 2017.

[164] Andreas Grammenos, Cecilia Mascolo, and Jon Crowcroft. You Are Sensing, but Are You Biased? A User Unaided Sensor Calibration Approach for Mobile Sensing. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)*, 2(1):11, 2018.

[165] David Tedaldi. *IMU calibration without mechanical equipment*. PhD thesis, University of Padova, 2013.

[166] Thibaud Michel, Pierre Geneves, Hassen Fourati, and Nabil Layaïda. On Attitude Estimation with Smartphones. In *Proceedings of the 2017 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 267–275. IEEE, 2017.

[167] David Tedaldi, Alberto Pretto, and Emanuele Menegatti. A Robust and Easy to Implement Method for IMU Calibration without External Equipments. In *Proceedings of the 2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3042–3049. IEEE, 2014.

[168] Wei Ren, Tao Zhang, Haiyun Zhang, Leigang Wang, Yongjie Zhou, Mengkai Luan, Huifeng Liu, and Jingwei Shi. A Research on Calibration of Low-Precision MEMS Inertial Sensors. In *Proceedings of the 25th Chinese Control and Decision Conference (CCDC)*, pages 3243–3247. IEEE, 2013.

[169] Iuri Frosio, Federico Pedersini, and N Alberto Borghese. Autocalibration of MEMS Accelerometers. *IEEE Transactions on Instrumentation and Measurement (TIM)*, 58(6):2034–2041, 2009.

[170] STMicroelectronics. L3G4200D: three axis digital output gyroscope. https://www.elecrow.com/download/L3G4200_AN3393.pdf, 2014. Accessed: 27 July 2020.

[171] STMicroelectronics. LIS331DLH: MEMS digital output motion sensor. `http://www.st.com/resource/en/datasheet/lis331dlh.pdf`, 2009. Accessed: 27 July 2020.

[172] Tom Van Goethem, Wout Scheepers, Davy Preuveneers, and Wouter Joosen. Accelerometer-Based Device Fingerprinting for Multi-factor Mobile Authentication. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems (ESSoS)*, pages 106–121. Springer, 2016.

[173] Gianmarco Baldini, Franc Dimc, Roman Kamnik, Gary Steri, Raimondo Giuliani, and Claudio Gentile. Identification of Mobile Phones Using the Built-In Magnetometers Stimulated by Motion Patterns. *Sensors*, 17(4):783, 2017.

[174] Yunmok Son, Juhwan Noh, Jaeyeong Choi, and Yongdae Kim. GyrosFinger: Fingerprinting Drones for Location Tracking Based on the Outputs of MEMS Gyroscopes. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):1–25, 2018.

[175] Anupam Das, Nikita Borisov, and Matthew Caesar. Tracking Mobile Web Users Through Motion Sensors: Attacks and Defenses. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2016.

[176] Xiang-Yang Li, Huiqi Liu, Lan Zhang, Zhenan Wu, Yaochen Xie, Ge Chen, Chunxiao Wan, and Zhongwei Liang. Finding the Stars in the Fireworks: Deep Understanding of Motion Sensor Fingerprint. *IEEE/ACM Transactions on Networking (TON)*, 27(5):1945–1958, 2019.

[177] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. The Web's Sixth Sense: A Study of Scripts Accessing Smartphone Sensors. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2018.

[178] Jeff Dunn. It looks like Apple will have plenty of iPhone owners that could use an upgrade this holiday season. `http://uk.businessinsider.com/apple-iphone-most-popular-model-newzoo-chart-2017-7`, 2018. Accessed: 27 July 2020.

[179] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. An Analysis of Pre-installed Android Software. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP)*, pages 1039–1055. IEEE, 2020.

[180] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. FIRMSCOPE: Automatic Uncovering of Privilege-Escalation

Vulnerabilities in Pre-Installed Apps in Android Firmware. In *Proceedings of the 29th USENIX Security Symposium*, pages 2379–2396. USENIX Association, 2020.