Technical Report

UCAM-CL-TR-909
ISSN 1476-2986

Number 909

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Exploring new attack vectors for the exploitation of smartphones

Laurent Simon

July 2017

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 2017 Laurent Simon

This technical report is based on a dissertation submitted April 2016 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Homerton College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Summary

Smartphones have evolved from simple candy-bar devices into powerful miniature computing platforms. Today's smartphones are complex multi-tenant platforms: users, OS providers, manufacturers, carriers, and app developers have to co-exist on a single device. As with other computing platforms, smartphone security research has dedicated a lot of effort, first, into the detection and prevention of ill-intentioned software; second, into the detection and mitigation of operating system vulnerabilities; and third, into the detection and mitigation of vulnerabilities in applications.

In this thesis, I take a different approach. I explore and study attack vectors that are specific to smartphones; that is, attack vectors that do not exist on other computing platforms because they are the result of these phones' intrinsic characteristics.

One such characteristic is the sheer number of sensors and peripherals, such as an accelerometer, a gyroscope and a built-in camera. Their number keeps increasing with new usage scenarios, e.g. for health or navigation. So I show how to abuse the camera and microphone to infer a smartphone's motion during user input. I then correlate motion characteristics to the keyboard digits touched by a user so as to infer PINs. This can work even if the input is protected through a Trusted Execution Environment (TEE), the industry's preferred answer to the trusted path problem.

Another characteristic is their form factor, such as their small touch screen. New input methods have been devised to make user input easier, such as "gesture typing". So I study a new side channel that exploits hardware and software interrupt counters to infer what users type using this widely adopted input method.

Another inherent trait is that users carry smartphones everywhere. This increases the risk of theft or loss. In fact, in 2013 alone, 3.1M devices were stolen in the USA[1], and 120,000 in London[2]. So I study the effectiveness of anti-theft software for the Android platform, and demonstrate a wide variety of vulnerabilities.

Yet another characteristic of the smartphone ecosystem if the pace at which new devices are released: users tend to replace their phone about every 2 years, compared to 4.5 years for their personal computers. For already 60% of users today[3], the purchase of a new smartphone is partly funded by selling the previous one. This can have privacy implications if the previous owner's personal data is not properly erased. So I study the effectiveness of the built-in sanitisation features in Android smartphones, lifting the curtains on their problems and their root causes.

[1] http://www.consumerreports.org/cro/news/2014/04/smart-phone-thefts-rose-to-3-1-million-last-year/index.htm

[2] http://www.london.gov.uk/media/mayor-press-releases/2013/07/mayor-challenges-phone-manufacturers-to-help-tackle-smartphone

[3] https://www.gartner.com/newsroom/id/2986617

This thesis demonstrates that smartphone platforms have, by their nature, enabled new and specific avenues of attack. I hope these findings provide new insights for all stakeholders involved in the development of smart platforms.

Acknowledgements

I thank my supervisor, Ross Anderson, for never losing faith in me. He has given me invaluable insights, advice, and support.

My research was supported by Samsung.

I thank my wife, Ann, who has been instrumental in keeping me positive in times of self-doubt; and my parents for supporting me throughout my education.

I thank all the members of the Mobile Reading Group for fostering valuable discussions, and my first-year viva examiners Alastair Beresford and Robert Watson for the suggestions they gave.

I thank all my fellow students at the lab who have helped me, including Rubin Xu, Wei-Ming Khoo, Dongting Yu, Kumar Sharad, Ilias Marinos, Marios Omar Choudary, Sheharbano Khattak, Khaled Baqer, Christian O'Connell, Stephan Kollmann, Simon Baker, Yiannos Stathopoulos, Wenduan Xu, Sandro Bauer, David Llewellyn-Jones, Alice Hutchings, Graeme Jenkinson, Bjoern A. Zeeb, Jeunese Payne, David Modic, and Sophie Van Der Zee.

Contents

1 Introduction **11**

1.1 Chapter Outline . . . 11

1.2 Publications . . . 12

2 Background **15**

2.1 Mobile Platforms . . . 15

2.1.1 Windows . . . 18

2.1.2 iOS . . . 18

2.1.3 Android . . . 19

2.2 Security Enabler . . . 21

2.3 Platform Vulnerabilities . . . 22

2.3.1 Smartphone OS Vulnerabilities . . . 22

2.3.2 OEM-Introduced Vulnerabilities . . . 25

2.3.3 Carrier-level Vulnerabilities . . . 27

2.4 App Vulnerabilities . . . 29

2.5 Malware . . . 30

2.5.1 Malware Categories . . . 30

2.5.2 Malware Distribution Channels . . . 31

2.5.3 Mitigation Solutions . . . 32

2.6 Side Channels . . . 33

2.7 Forensics . . . 34

2.7.1 Data Acquisition . . . 35

2.7.2 Data Reconstruction . . . 36

2.7.3 Secure Deletion . . . 36

2.8 This Dissertation . . . 37

3 **Inferring PINs Through The Camera and Microphone** **39**

3.1 Introduction 39

3.2 Attack Principles 41

3.2.1 Attack Flow 43

3.2.2 Stealthiness 44

3.3 Implementation Details 46

3.3.1 Collecting Mode 46

3.3.2 Feature Extraction 46

3.3.3 Learning Mode 48

3.3.4 Logging Mode 48

3.4 Evaluation 51

3.4.1 Setup 51

3.4.2 Single-Digit Prediction 51

3.4.3 PIN Predictions 54

3.5 Limitations 58

3.6 Possible Countermeasures 59

3.6.1 Non-TEE devices 59

3.6.2 TEE-enabled devices 60

3.6.3 Other Considerations 60

3.7 To Patch or Not to Patch 61

3.8 Summary 62

4 **Interrupt-based Side Channel on Android** **63**

4.1 Introduction 64

4.2 Background and Threat Model 65

4.2.1 Android Soft-keyboards & Gesture Typing 65

4.2.2 Android & procfs 66

4.2.3 Attack Overview 67

4.3 Evaluation 76

4.3.1 Methodology 76

4.3.2 Word Prediction 78

4.3.3 Detection of Sentences of Interest . . . 78
4.3.4 De-Anonymization of Users . . . 81
4.4 Countermeasures . . . 85
4.5 To Patch or Not to Patch . . . 89
4.6 Discussion . . . 90
4.7 Summary . . . 92

5 Security Analysis of Android Factory Resets 95
5.1 Introduction . . . 95
5.2 Technical Background . . . 98
5.2.1 Flash & File Systems . . . 98
5.2.2 Secure Deletion Levels . . . 99
5.2.3 Linux Kernel Deletion APIs . . . 100
5.2.4 Data Partitions . . . 100
5.3 Analysis of Android Factory Reset . . . 101
5.3.1 Methodology . . . 101
5.3.2 Results and Discussion . . . 105
5.4 Data Recovery in Practice . . . 110
5.4.1 General Results . . . 112
5.4.2 Case Study: Hijacking Google Accounts . . . 112
5.4.3 Possible Attackers . . . 112
5.5 Alternative Sanitisation Methods . . . 113
5.6 Recommendations . . . 115
5.7 Encryption Requirements . . . 116
5.8 Summary . . . 117

6 Security Analysis of Android Anti-Theft Apps 119
6.1 Introduction . . . 119
6.2 Background . . . 121
6.2.1 Bootloader, Recovery and Safe Modes . . . 121
6.2.2 Mobile Anti-Virus (MAV) Apps and Device Admin API . . . 122
6.3 Methodology . . . 124

6.4 Account Authentication 125
6.5 App Configuration & User Interface 126
6.6 Lock Implementations and Effectiveness 128
6.6.1 Removal of MAVs & API Misuse 128
6.6.2 Rate Limiting 132
6.6.3 MAV Response 133
6.6.4 Network-level Attacks 133
6.6.5 Vendor and Customised Android Failures 134
6.6.6 Misc 136
6.6.7 Encryption to the Rescue 137
6.7 Wipe Implementations 137
6.7.1 General Results 137
6.7.2 Case Studies of Lookout and Avast 139
6.7.3 Inherent Problem of Remote Wipe 140
6.8 Discussion 140
6.9 Summary 141

7 Conclusions 143

Bibliography 147

Chapter 1

Introduction

1.1 Chapter Outline

Smartphones have become ubiquitous, with almost two billion people owning one at the time of writing. These hand-held devices are used for everything: banking, games, Internet, emails, chat, pictures, etc. With every new technology we get new security challenges and paradigms, and these are the topic of my dissertation. My thesis is that smartphones enable new attack vectors that are specific to them. It is paramount to study and understand these specific vulnerabilities to better protect users.

Therefore, in this work, I explore attack vectors inherent in mobile platforms. I have identified the following (non-exhaustive) characteristics:

Sensors and Peripherals: Smartphones come with a wide variety of built-in sensors and peripherals. In Chapter 3, I devise a novel side channel attack that uses the built-in camera to recover sensitive PINs entered by a user. This attack is possible because of the way users type on their smartphones: a user jiggles his phone by typing. The attack reduces the entropy of PINs and highlights some of the challenges that designers must be aware of when implementing a trusted input path inside a Trusted Execution Environment (TEE).

Form factor: Smartphones are portable hand-held devices. As such, they are small in size and have a virtual keyboard implemented on top of a touch screen rather than a physical keyboard. To improve the typing experience, new input methods have been devised. The most popular is "gesture typing", which is shipped on newer Android versions by default. With gesture typing, users swipe their finger from one character to another rather than tapping each key individually. In Chapter 4, I study a novel side-channel attack against this input method to recover sentences entered by users. I show that malicious (permissionless) apps installed on an Android device can monitor the system-wide screen's hardware interrupt counter and software interrupt counter, and then correlate these to

text entered by users through supervised machine-learning techniques. These findings highlight a new way in which system-wide resources can threaten user privacy.

Personal: Smartphones are personal devices. Users keep them nearby all the time and carry them everywhere. But this increases the risk of devices being lost or stolen. In 2012 alone, smartphone robberies represented almost 50% of all robberies in San Francisco, 40% in New York City and were up 27% in Los Angeles[1]. In Chapter 6, I study the effectiveness of anti-theft software for the Android platform. I discover a wide variety of vulnerabilities that allow a thief to recover a user's personal data, even if the phone is remotely locked or erased through anti-theft software. I highlight the erroneous assumptions made by app developers and certain limitations of the Android API.

Development pace: The life cycle of a typical smartphone is short: users replace their smartphone on average every 2 years, compared to 4.5 years for their desktop computers. Often the purchase of a new smartphone is partly funded by selling the previous one. Gartner anticipates the refurbished phone market being worth 140M by 2017[2]. This can have privacy implications if personal data is not properly erased. In Chapter 5, I study the so-called Factory Reset function in Android smartphones. I provide a detailed analysis of when, why and how it fails, highlighting the complexity of Android's multi-tenant ecosystem. I also provide concrete guidelines for Google and other vendors to improve the reliability of the Factory Reset function.

1.2 Publications

During the course of my PhD, I have published the following:

- L. Simon, W. Xu, and R. Anderson, "Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards," in *Proceedings of Privacy Enhancing Technologies Symposium (PETS)*, 2016.
- S. Khattak, T. Elahi, L. Simon, C. M. Swanson, S. J. Murdoch, and I. Goldberg, "SoK: Making Sense of Censorship Resistance Systems," in *Proceedings of Privacy Enhancing Technologies Symposium (PETS)*, 2016.
- L. Simon, "A Gentle Introduction to Side Channel Attacks on Smartphones," book chapter, in *The Book of Payments: Historical and Contemporary Issues in the Cashless Economy BBL and LE, editors (Palgrave).*
- L. Simon and R. Anderson, "Security Analysis of Android Factory Resets," in *Proceedings of 4th Workshop on Mobile Security Technologies (MoST)*, 2015.

[1] `gizmodo.com/5953494/hold-on-tight-smartphone-mugging-is-more-popular-than-ever`
[2] `https://www.gartner.com/newsroom/id/2986617`

- L. Simon and R. Anderson, "Security Analysis of Android Factory Resets," in *Black Hat Mobile Security Summit*, 2015.
- L. Simon and R. Anderson, "Security analysis of consumer-grade anti-theft solutions provided by android mobile anti-virus apps," in *4th Mobile Security Technologies Workshop (MoST)*, 2015.
- S. Khattak, L. Simon, and S. J. Murdoch, "Systemization of pluggable transports for censorship resistance," *arXiv preprint arXiv:1412.7448*, 2014.
- L. Simon and R. Anderson, "PIN Skimmer: Inferring PINs Through The Camera and Microphone," in *Proceedings of 3rd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2013.

Chapter 2

Background

2.1 Mobile Platforms

The cellphone market has changed radically over the last ten years. A decade ago, Symbian was the leading operating system with more than 50% market share[1], and Nokia was the uncontested hardware market leader with 50% market share[2]. These two giants of the ecosystem at the time are nowhere to been seen today. The last decade has seen vicious competition, with many unsuccessful attempts by OEMs and carriers to capture mobile market share. For example, MeeGo OS was Nokia's last attempt to get back in the game, after iOS and Android had stolen its lead. Ubuntu Touch was an attempt by Canonical to launch an innovative open-source OS, but it failed to gain traction. Firefox Mobile was led by Telefonica and targeted developing countries, but the project was discontinued in 2015. Blackberry phones, once leaders in the enterprise sector, now represent less than 0.3% of shipments in 2015[3]. Tizen is an ongoing effort by Samsung to break free from dependency on Android. CyanogenMod is a customized version of Android which OEMs and vendors can use without having their hands tied to Google[4,5]. From this battle, two operating systems have emerged victorious: Google's Android OS, and Apple's iOS. Together they represent more than 95% of shipped devices in 2015 (Fig. 2.1).

The smartphone ecosystem comprises many entities (or stakeholders), and this has direct implications on the security of mobile platforms. One major stakeholder of any mobile platform is the **OS provider**: it develops, designs, and maintains the main components of the smartphone OS with which users interact. The provider controls the overall strategy of the OS, such as new features to incorporate in future releases, security

[1]http://www.internetnews.com/wireless/article.php/3584431

[2]http://www.canalys.com/newsroom/64-million-smart-phones-shipped-worldwide-2006

[3]http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[4]http://uk.businessinsider.com/cyanogen-taking-over-android-2015-1

[5]http://www.techtimes.com/articles/46820/20150418/microsoft-partners-cyanogen-bring-windows-apps-android-phones-google-worry.htm

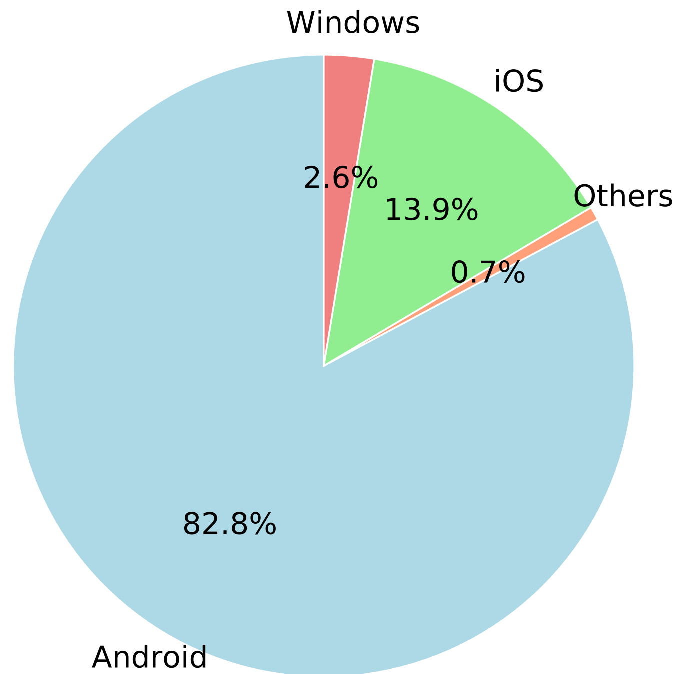


Figure 2.1: Smartphone OS market share in Q2 2015[1]

mechanisms, etc. By controlling Android, Google can also align the OS development with its overall business strategy.

The **Trusted Execution Environment (TEE)** is a headless, invisible OS used for performing sensitive tasks and storing sensitive data. It can be used for enforcing a trusted boot, for storing encryption keys and payment credentials, etc. The TEE is generally shielded from the smartphone OS through hardware features. The set of requirements that a TEE implementation must meet (e.g. APIs to communicate with the smartphone OS, the security requirements, etc.) are defined by a not-for-profit industry association called GlobalPlatform[2]. In practice, vendors tend to implement their own TEE from scratch, either because they want special features or because available code-reviewed TEE implementations are not free. This creates vulnerabilities.

The **vendors**, a.k.a. as **Original Equipment Manufacturers (OEMs)**, design the smartphones themselves. They decide what hardware goes into which device (e.g. camera resolution, screen size), and control the marketing strategy for each device they release. For business reasons, OEMs must differentiate their offer from their competitors'. Customisation is visible through the user interface and additional software pre-installed. The conception of a phone must happen fast because users upgrade their device every 2 years; this pace has implications for the security of devices.

Carriers, a.k.a. as **mobile network operators (MNOs)**, provide access to the

[1] `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`
[2] `https://www.globalplatform.org`

mobile network for both voice and data usage. Historically, they have subsidized devices from OEMs in many countries and this continues today. Many of their contracts run over 1.5 or 2 years, and bundle mobile network access with a brand new device. This may partly explain why most users change their phone every 2 years. Carriers may also pre-install software on devices to offer additional services and for differentiation.

To connect to a mobile network (e.g. GSM, 3G), a phone must speak the relevant wireless protocol. So all smartphones come with a "hidden" OS called the baseband OS (or simply the baseband) that implements the protocol and radio stack. The baseband runs on a dedicated processor that is different from the one where the smartphone OS runs. To differentiate the two processors, it is common to refer to the smartphone OS one as the "application processor", and to the baseband one as the "baseband (radio) processor". The **baseband provider** is the entity that provides the baseband OS, it is distinct from the smartphone OS provider. Cryptographic keys used by the baseband to authenticate to the mobile network are stored on a SIM card, a tamper-resistant smart card. The SIM card is owned by the carriers, but is designed and implemented by **SIM providers** (e.g. Gemalto). The baseband OS comes with its own set of vulnerabilities and privacy issues.

End users are people who use devices. In particular, they can seamlessly install software, a.k.a. apps, from an online app store. Apps are developed by **app developers** through an SDK provided by the OS provider. In addition to the default OS app store, certain OEMs also have their own online store. But there does not seem to be much traction for those since app developers would rather release their app on the OS store to reach a greater number of users.

App developers do not work for free. Sometimes a user must pay at installation time or "in-app" for additional features. But most of the time, developers resort to embedding ad libraries to monetize their app. These libraries are provided by **advertisers**, and automatically display ads to users. The more an ad network knows about a user, the more targeted the ads will be, the more likely a user is to click it, and the more likely the app developer will be to make money. The incentives are perverse as users may want to disclose as little information about themselves, while app developers generally want the opposite.

Even with two dominant OSes, maintaining different code bases for each can be a daunting task for app developers. Therefore, there exist 3rd-party **SDK providers** that make life easier for developers by providing a unified API across different OSes.

In the next sections, we zoom in each of the dominant mobile platforms and highlight how they differ from the above general picture.

2.1.1 Windows

Windows Mobile is a group of mobile operating systems developed by Microsoft. The first release dates back to 2000. Originally, the devices principally targeted were enterprise pocket PCs and smartphones. They were based on Windows-CE, an OS for embedded devices. To remain competitive with iOS and Android, Microsoft shifted its offer towards Windows Phone (WP), with the goal of making it a consumer product rather than a corporate-only device. The first version, WP7, was released in 2010 and was still based on Windows-CE. WP8 moved away from Windows-CE and uses a Windows-NT kernel instead; it was released in 2012, and support for WP7 ended in 2014. WP8 apps are developed with a phone version of the Windows Runtime a.k.a. Windows Phone Runtime; it supports apps written in C#, VB.NET and C/C++. In 2013, Microsoft acquired Nokia's smartphone division; and as a result, most Windows phones are manufactured by Nokia today. It costs USD 19 for an individual and USD 99 for an enterprise to create a developer account and publish apps on the store. Microsoft takes a 30% cut of developers' revenue.

WP8 supports a range of security features[1]. Secure boot mitigates the risk of rootkits. Code signing is mandatory for all OS components including drivers and pre-installed apps. This is used to verify the integrity of the OS during secure boot. Apps also require code signing and go through a vetting process before they are made available on the Windows store for end users to download. Apps run in a sandboxed environment called a "chamber" with a minimum set of "capabilities". A capability gives an app the right to access a resource, such as the phone's current location, the phonebook, etc. The capabilities required by an app must be listed by its app developer at compilation time. The list is approved by end users at installation time in an all-or-nothing fashion. If a user installs an app, this grants it access to the entire list of capabilities requested, and no fine-grained revocation is possible afterwards. Full disk encryption (FDE) is also supported through the Bitlocker technology. However the encryption keys are backed up in the Microsoft cloud[2]. It is possible for enterprises to implement internal security policies on employees' devices through Exchange ActiveSync.

2.1.2 iOS

iOS is Apple's mobile OS. The first iPhone was released in 2007 and marks a milestone in the evolution of smartphones[3]. It democratised touch screen displays while the leader Symbian was still banking on physical keyboards or a stylus. Unlike Windows and Android, Apple is the only OEM that ships iOS devices. In fact, there is only a single line of devices

[1] http://blogs.msdn.com/cfs-filesystemfile.ashx/__key/communityserver-blogs-components-weblogfiles/00-00-01-55-06/8272.20_2C00_206.01_5F00_WP-8_5F00_SecurityOverview_5F00_102912_5F00_CR.pdf

[2] http://windows.microsoft.com/recoverykey

[3] http://blog.gotchamobi.com/mobile-marketing/a-brief-history-of-smart-phones

known as the "iPhone". As a result, Apple acts both as the OS provider and the OEM, and controls both software and hardware end-to-end. This has obvious advantages both in terms of product integration, security, reactivity, and time-to-market. Like Windows, there is a single app store. It costs USD 99 annually for a developer account, and Apple takes a 30% cut on developers' revenue. Apps are written in objective C, C or swift.

Apple phones have a wide range of security features. Code signing is mandatory for every executable page, with the exception of the browser for JIT (this is in contrast to Android where foreign code can be loaded at runtime in various ways [1]). iOS implements Mandatory Access Control to shared resources such as the file system. By installing an app, a user does not grant it any permissions, except for basic ones such as Internet access. At runtime, when an app attempts to access sensitive resources (e.g. phone location, address book, microphone), iOS prompts the user with a runtime dialog asking the user to confirm or deny the access. Even once granted, permissions can individually be revoked in the phone's Settings menu at the user's discretion. iOS also comes with secure boot with cryptographic keys stored in hardware, encrypted partitions, and an integrated fingerprint scanner[1]. For corporate users, Apple has partnered with IBM to boost its app offering[2].

2.1.3 Android

Android OS is Google's mobile OS. Its first release dates back to 2008. At the time, iOS's walled garden platform was the dominant player. So Google took a different approach by promising a completely open source platform which OEMs and carriers alike can customise to meet their needs. In return for providing the OS for free, OEMs and carriers must pre-install Google services on it. There seems to be growing resistance from carriers and OEMs towards Google's not-so-open-source practices[3]. In turn, this creates space for competitive OSes (CyanogenMod and Tizen) that promise truly open-source stacks. Google Play is the official app store, but users are allowed to install apps from third-party app stores not endorsed by Google. This is referred to as "sideloading" an app. This is necessary for users in countries where the Google store is not accessible (e.g. China). It costs a one-time USD 20 fee to create a developer account. Apps are written in Java and C with JNI bindings.

The Android OS uses a Linux kernel with patches to improve battery life. Google created Android with the idea that individual app components could talk to each other. So Google incorporated a new IPC mechanism called Binder that apps can use to communicate. Essentially, one can consider Android as a collection of "locally-networked" apps. And this means that app developers must be careful about the confidentiality and integrity of inter-app messages they transmit.

[1] https://www.apple.com/business/docs/iOS_Security_Guide.pdf
[2] https://www.apple.com/uk/business/mobile-enterprise-apps/
[3] http://uk.businessinsider.com/google-should-ditch-android-open-source-2015-4

Android's security model is based on the concept of *application sandboxes*. Prior to Android 4.3, application sandboxes were implemented on top of Linux discretionary access control (DAC), with different applications having different userids, and the application-layer permission model relied on this application sandbox. In Android 4.3, Android added Mandatory Access Control (MAC) through SELinux to tighten its security. Permissions are declared by app developers at compilation time, and are used to restrict access to system resources at run time. As for Windows, permissions were initially accepted in an all-or-nothing fashion at installation by users. But starting from Android 6, this has changed: individual permissions can now be revoked on a per-app basis in the phone settings menu. Unlike on iOS, there is no runtime prompt and permissions are granted by default. It is a user's responsibility to "opt-out". To compensate for its liberal approach, Android ships with an on-device AV called "VerifyApps" that scans apps at installation. Furthermore, Google scans apps in the Play store, and has the ability to scan the majority of Android apps available on third-party stores through its web infrastructure. There is also a vast amount of data that Google can mine for each Google developer account. Thanks to surveillance and blacklisting, the number of infected devices has dropped sharply over the last few years [2, 3]. But even this is not always sufficient, and malicious apps regularly get through. In 2015 for example, Lookout found 20K apps containing rooting functionalities in third-party stores[1]. In addition, there is a great deal of fragmentation resulting from the numerous customizations applied by carriers and OEMs to different phones. Combined with the pace at which devices are released, this creates vulnerabilities that Google cannot mitigate alone.

Although customisation does introduce some vulnerabilities, it can also provide security benefits. Samsung's KNOX platform provides additional security through a special ARM CPU mode called TrustZone. This mode prevents tagged resources (CPU caches, filesystem, RAM, peripherals) from being accessed by the smartphone OS. Android also offers Mobile Device Management (MDM) features through the Admin APIs, and a dual-persona option called "Android for Work" was added in Lollipop (Android 5). Android supports FDE through dm-crypt but only recently has it been integrated with the phone hardware.

Out-of-the-box, Android can also be customised by users themselves if they wish too, through a process known as "rooting". Rooting essentially gives a user root access to a device. This is in sharp contrast to Apple and Microsoft's approach. In practice however, many OEMs try to prevent this and some even void the device warranty if they detect it. Still, hackers often find ways to root a device through OS vulnerabilities.

[1] `https://blog.lookout.com/blog/2015/11/04/trojanized-adware/`

2.2 Security Enabler

Besides the new risks associated with smartphones, one must also acknowledge the security benefits they can provide. In other words, smartphones are also "security enablers". The major area of research in this space is user authentication beyond character-based passwords, for example by leveraging smartphone sensors and the touch screen [4].

The most-widely known is Android's "gesture pattern" a.k.a. "graphical passwords". It is more usable than a traditional password, but its entropy is not always better than a PIN because users do not select patterns randomly enough [5]. Picture passwords also suffer from the same issue, which allows attackers to guess a significant number of them in practice [6]. Some research has also gone into making passwords shoulder-surfing resistant [7] and smudge resistant [8, 9]. Shake-On-It (Shot) [10] is a key exchange system that uses the built-in vibrators and accelerometers to exchange data between two phones with physical contact, thereby preventing relay attacks.

Two-Factor Authentication (2FA) may be the main practical security use of smartphones. It has been common via SMS for a decade, and smartphones make it easier to deploy app-based versions of the same idea. Sound-proof [11] is a 2FA system that verifies the proximity of a computer and a smartphone by comparing the ambient noise recorded through their respective microphones. SigVerify [12] uses the accelerometer and gyroscope to authenticate users and provide a proof of liveliness, based on a pattern a user draws by waving the device in the air. Smartphones can also be used as a replacement to smartcards by implementing a PKCS#11 token.

Another research direction to improve on passwords is called "continuous authentication". In this paradigm, users are continuously authenticated through behavioural features. These can be based on the way users interact with common UI elements [13], how they touch their screen [14–19], how their hands move and hold a device [20], as well as other characteristics such as when emails come in, which apps are used and when [21], etc. There is a London startup[1] banking on how users enter their PIN for authentication. As continuous authentication provides only a certain level of guarantee, sometimes even these systems must resort to a "strong" authentication mechanism such as a password. Deciding when to strongly authenticate users incurs the same usability penalty that continuous authentication tries to solve, so there are usability-security trade-offs [22]. Pico [23] proposes an authentication platform based on a cloud of smart wearable devices. The insight is that the combination of these wearable devices can uniquely identify you and it is hard for an adversary to steal all of them.

Another line of research is to add biometrics to devices. A fingerprint scanner is already shipping with iPhones, and Google has recently added support to Android. Besides fingerprints, face recognition has also been supported on certain phones but this does not

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

appear to be so reliable[1]. There is also a startup[2] developing ear-based biometrics. The FIDO[3] alliance has produced a flexible authentication standard that is being implemented in recent devices. The big Internet giants such as Facebook and Google also use data-driven behavioural biometrics to detect account compromise, and smartphones play an important role in these. For example, if someone's phone is currently in Paris but a login attempt is made from Brazil, this can raise red flags.

2.3 Platform Vulnerabilities

For each stakeholder presented in Section 2.1, sub-classes of vulnerabilities have been discovered. These may be platform-specific or generic, and we present them in the following sections. The majority of mobile security research has been dedicated to evaluating the security and privacy of the smartphone OS. So not surprisingly, there is a rich literature on this. More specifically, the majority of papers are about Android so this is naturally reflected in the following sections. There are probably four major reasons why this is the case. First, Android has the largest market share (Section 2.1). Second, Android is open source: anyone can review the code, compile it and run it, which reduces the barrier to entry for most researchers. Academic researchers often do not have the resources and skills to reverse-engineer binary code. It is time-consuming and risky, as they may find nothing they can publish. Third, Android is more liberal in what it allows apps to do (e.g. runtime code loading, background services, etc.), which increases the attack surface. Fourth, unlike iOS, Google allows third-party OEMs. This means that the numerous entities involved during the conception of a smartphone increase the risk of mistakes and bugs.

2.3.1 Smartphone OS Vulnerabilities

Vulnerabilities introduced by the OS provider directly affect the smartphone OS. Android has seen its share of such problems, although no mass-scale exploitation has been uncovered to date.

The most criticised part of Android has been its permission system. Users must accept permissions on an all-or-nothing basis at installation time. No revocation is possible afterwards. This has generated a large body of work. As odd as it might seem, Google itself did not originally seem to know exactly which APIs needed which permissions. So some early papers were just about clarifying this mapping [24]. Not surprisingly, app developers

[1] http://www.pcmag.com/article2/0,2817,2396321,00.asp

[2] http://www.cta.tech/Blog/Articles/2015/July/STARTUP-STORIES-Unlock-Your-Phone-with-Your-Ears.aspx

[3] https://fidoalliance.org/fido-alliance-announces-72-certified-authentication-products/

did not understand the system either, and often requested more permissions than their app needed [25]. Another issue with the permission system is whether permissions are presented to users in an intelligible way. In 2012, people still did not understand them well enough [26, 27] to make educated privacy decisions about apps. Certain permissions were better understood than others though, for example the location one [28]. The only way to gauge why an app needs sensitive permissions is to read its description and the reviews in the app store. Some research has indeed tried to automate this process by designing tools that assess the match between an app's natural-language description and the permissions it requests [29, 30]. But this can only take you so far. Furthermore, there is also an underground economy of trading mobile app reviews [31].

There have been various efforts to re-think the permission model (*i*) to make it more intuitive to users [32–36], (*ii*) to make permissions revocable after installation and at runtime [37–39], and (*iii*) to grant permissions based on OS-controlled UI elements that convey implicit user consent [40]. Google added a post-installation revocation feature in Android 6 (Marshmallow).

Another related issue with permissions is the way they are granted during installation and upgrade. Researchers found that malicious apps could request non-existing system permissions. But when these become defined in a future version of the OS, the upgrade process would automatically grant them [41, 42]. Other problems with the installation process include a TOCTTOU vulnerability during app binary verification because the latter is saved on a shared partition[1]. This allows an app to change the permissions displayed to users during installation. Installation issues also include the (in)famous series of so-called MasterKey vulnerabilities that allow tampering with an application's code without altering its original signature[2,3]. Stagefright is a recent series of vulnerabilities affecting the parsing routines of multimedia files[4] [43]. There is a clear trend towards looking at the part of the OS written in native (i.e. C/C++) code today. This has received a lot less attention than the Java parts to date.

Another feature of Android we touched upon in Section 2.1.3 is the network-like communication that apps use to (*i*) request services and (*ii*) subscribe to asynchronous events from system and third-party apps. The IPC mechanism is called Binder, on top of which messages called "Intents" are transmitted. Intents are comprised of subelements including the message itself and "routing" information used by the OS to deliver it to the expected application. The IPC mechanism is actually trickier that it originally looked, so some early papers documented how it works and the common pitfalls to avoid [44]. In

[1] `http://researchcenter.paloaltonetworks.com/2015/03/android-installer-hijacking-vulnerability-could-expose-android-users-to-malware/`

[2] `https://bluebox.com/uncovering-android-master-key-that-makes-99-of-devices-vulnerable/`

[3] `http://www.saurik.com/id/19`

[4] `https://blog.zimperium.com/zimperium-zlabs-is-raising-the-volume-new-vulnerability-processing-mp3mp4-media/`

practice, this means that certain messages may be intercepted and/or tampered with by a malicious app [45], that the origin of an intent can be spoofed [46], and that supposedly-private app functions may be exposed to other apps – the so called confused deputy problem [47]. This does not solely affect third-party apps but also system apps and vendor apps[1] [48]. On a similar note, there has also been concern that multiple benign-looking apps may communicate so as to share their individual permissions. So runtime systems have been devised to help spot these [49].

Another widespread issue is the webview javascript bridge bug. It was apparently disclosed by a blogger in 2012[2,3] (although we recall reading about it a year earlier on a Chinese blog). All mobile platforms support web-based apps. These apps run javascript code loaded inside a web engine. On Android, this engine is called a "webview". To interface javascript code with native APIs, so-called "javascript bridge" functions must be implemented. These are entry points that are callable from javascript. Within these bridge functions, an app developer can query native APIs to access the contact list, the geolocation, etc. so long as the app has the relevant permissions. However, the Android engineers overlooked the fact that any Java object inherits from the *Object* class, which itself provides methods to construct new objects through reflection. This means that an attacker who can load malicious code into a web app can use an innocuous-looking bridge function to access all sensitive information allowed by the permissions granted to the app.

There have been few attempts to break the crypto in Android, but one major incident is the (in)famous OpenSSL's PRNG bug. It was found to be predictable because all apps inherit from a common process (the Zygote) and the PRNG state was simply duplicated across forks [50]. The same work also identified that the PRNG was not properly seeded in certain cases. The theft of USD 5700 worth of Bitcoins was attributed to this bug in 2013[4].

There has also been some research into the user interface side of things. Early on in 2010, a wave of UI-redressing attacks hit Android [51]. They involved a UI component called a *Toast* that can be overlaid on top of a foreground app by a permissionless app running in the background. More recently, this area has regained interest, with researchers discovering more ways to implement UI-hacking attacks to spoof UI elements, DoS users or monitor what they do[5,6] [52, 53]. On a higher level, researchers have also studied what a "secure" mobile UI should look like [40, 54].

Another line of research has studied the interaction of the smartphone OS with the

[1] `http://www.cvedetails.com/cve/CVE-2015-3843/`

[2] `http://d3adend.org/blog/?p=314`

[3] `https://labs.mwrinfosecurity.com/blog/2013/09/24/webview-addjavascriptinterface-remote-code-execution/`

[4] `http://arstechnica.com/security/2013/08/google-confirms-critical-android-crypto-flaw-used-in-5700-bitcoin-heist/`

[5] `http://drops.wooyun.org/papers/9769`

[6] `http://www.modzero.ch/modlog/archives/2015/04/01/android_apps_in_sheeps_clothing/`

physical world, such as secondary physical devices connected via short range wireless technology. Early research looked at phone exploitation through its NFC stack [55–57]. Since then, researchers have looked at the access controls between an Android phone and a Bluetooth-paired device. The take away message is that the Android OS does not provide a reliable framework for a phone app and an IoT device to authenticate each other [58]. Therefore in practice, it is often possible to intercept messages between the two devices, or even spoof one of them. More robust access control frameworks have been devised to mitigate these risks [59]. More recently, research has also demonstrated how to control a phone's voice interface by playing inaudible sounds from a distance [60].

A few papers have looked at iOS security too. Researchers found that runtime user confirmation popups ran in the same address space as apps, therefore an ill-intentioned app could "skip" the user confirmation to access sensitive information without user approval [61]. Flaws in the iTunes syncing process have also been showcased: once a computer is compromised, it can install malicious apps on a connected iPhone [62]. Vulnerabilities called "cross-app resource access attacks (XARA)" have been discovered recently. They are caused by a lack of app authentication when accessing sensitive resources such as the Keychain. As a result, malicious apps may steal credentials stored by other apps [63].

In this dissertation, we also look at platform vulnerabilities. For example in Chapter 4, we show that some information about what users type in their keyboard is inadvertently leaked to other apps by the OS. In Chapter 5, we present flaws in the Factory Reset of Android smartphones.

2.3.2 OEM-Introduced Vulnerabilities

Besides vulnerabilities introduced by the smartphone OS provider, customisations by OEMs can also affect the smartphone OS. This mostly applies to Android devices. iPhones are exempt from this problem since Apple does not license its OS to other OEMs.

The first paper about OEM customisations set out to look for confused-deputy attacks on 8 stock android phones [64]. Using static dataflow analysis techniques, researchers found that for the 13 permissions they examined, 11 were leaked by pre-installed apps. Each phone leaked up to 8 permissions. A following paper the next year (2013) showed that 85% of OEM-installed apps had more permissions than they needed to run, and concluded that between 65% and 80% of permission leaks were caused by OEM customisations [65]. The following year, researchers moved lower in the stack and looked at driver-level customisations. They found that the permissions of certain device files (e.g. */dev/camera*) were read/write-able by permissionless apps on the phone [66]. Other research has

showcased vulnerabilities in Qualcomm drivers[1,2]. More recently, researchers have also found that OEMs inadvertently relax the default SELinux policy when customising their phones [67].

There is a long list of OEM-introduced vulnerabilities reported by industry researchers too. Samsung hit the headlines several times. Some of its phones suffered from the so-called "USSD vulnerability", which allowed a malicious webpage to push commands to the Dial app. This led to premium number charges or factory reset of the device[3]. Samsung was caught removing bound checks in its GPU driver to improve performance, giving permissionless apps the ability to read/write arbitrary memory pages [68]. Users' fingerprints collected by the Samsung S5 were not processed securely on the device[4]. The default Samsung keyboard did not properly validate input it received through HTTP, which allowed a network attacker to gain remote code execution[5] [69]. Google showcased 11 security vulnerabilities affecting the latest Samsung S6[6]. A recent path traversal attack (i.e. improper validation of filenames) led to remote code execution from a malicious webpage[7,8].

Other OEMs have also had their share of problems. One can simply look at the number of root exploits found by hackers to customise their phones[9,10] [70]. The Trusted Execution Environment (TEE) is also a source of problems. It has been little studied but vulnerabilities have still emerged, often leading to root exploits and/or secure boot bypass on various phone models[11,12,13]. The diversity of TEE implementations in industry eventually leads to a greater attack surface and risk of vulnerabilities[14,13] [71].

It is hard to release bug-free software, hence it is essential for devices to receive regular and timely updates. Unfortunately, these do not occur often enough in practice on the Android platform [72]. This is probably due to the many stakeholders slowing down the

[1] http://mlsec.org/joern/docs/2014-inbot.pdf
[2] http://androidvulnerabilities.org/by/manufacturer/Qualcomm
[3] https://www.nowsecure.com/blog/2012/09/25/remote-ussd-code-execution-on-android-devices/
[4] http://www.theguardian.com/technology/2015/apr/23/samsung-investigating-fingerprint-hack-galaxy-s5
[5] https://www.nowsecure.com/blog/2015/06/15/a-pattern-for-remote-code-execution-using-arbitrary-file-writes-and-multidex-applications/
[6] http://googleprojectzero.blogspot.co.uk/2015/11/hack-galaxy-hunting-bugs-in-samsung.html
[7] http://blog.quarkslab.com/remote-code-execution-as-system-user-on-android-5-samsung-devices-abusing-wificredservice-hotspot-20.html
[8] https://code.google.com/p/google-security-research/issues/detail?id=489
[9] http://wiki.cyanogenmod.org/w/Devices
[10] http://androidvulnerabilities.org/by/manufacturer/
[11] http://blog.azimuthsecurity.com/2013/04/unlocking-motorola-bootloader.html
[12] http://blog.azimuthsecurity.com/2013/05/exploiting-samsung-galaxy-s4-secure-boot.html
[13] http://theroot.ninja/disclosures/TRUSTNONE_1.0-11282015.pdf
[14] http://bits-please.blogspot.com.ar/2015/08/exploring-qualcomms-trustzone.html

process, as well as the lack of incentives to maintain older devices – OEMs and carriers make money when they sell devices, so maintaining older devices would be economically unsound. There may also be a resource issue, as deadlines cause a developer to move its customisation team from the current product to the next one, leaving no-one to maintain legacy versions.

In this dissertation, we also present OEM vulnerabilities. For example in Chapter 5, we highlight flaws in the way vendors have patched the Factory Reset of Android smartphones. In Chapter 6, we show how certain vendor customisations may allow thieves to retrieve personal information on a PIN-locked device.

2.3.3 Carrier-level Vulnerabilities

Carriers maintain the mobile network and provide SIM cards. Phones access the network through the baseband OS which is provided by a third party. Security issues can therefore emerge (*i*) at the mobile network level, (*ii*) at the SIM level, and (*iii*) in the baseband OS.

SIM-based attacks are almost unheard of since the SIM incorporates hardware protection mechanisms that require lab attacks in order to be bypassed. Such attacks do not scale easily. Two prominent attacks are worth mentioning though. The first was demonstrated by Karsten Nohl in 2013. He demonstrated how he could recover the 56-bit DES encryption key of a SIM by sending a text message that spoofed the identity of the carrier[1] [73]. The hack worked because of the small keyspace. The second was the recovery of SIMs' cryptographic keys through Differential Power Analysis (DPA)[2] [74] at low cost. The takeaway message is that SIM cards are becoming so cheap that certain manufacturers no longer implement known side-channel countermeasures.

Baseband attacks have been gaining interest too[3]. They have been studied by Weinmann [75], Mulliner [76] and Solnik and Blanchou [77]. In particular, Solnik and Blanchou [77] demonstrated a complete bypass of the main OS lock screen on Android and iOS devices. They exploited Over-The-Air (OTA) update mechanisms used by carriers. More recently, researchers managed to hack into the Samsung Galaxy S6 baseband and reflash it remotely[4].

The mobile network has its own set of wireless protocols such as GSM, 3G, 4G and LTE. These protocols have had their share of problems. A5/1, a stream cipher used to provide over-the-air communication in GSM, has a key length of 56/64 bits only. This was a deliberate choice to allow government agencies to break it. Not surprisingly, breaking it

[1] `https://srlabs.de/rooting-sim-cards/`
[2] `https://www.youtube.com/watch?v=x8exHMhGy1Q`
[3] `https://www.youtube.com/watch?v=D59oYs3wUFA`
[4] `http://www.theregister.co.uk/2015/11/12/mobile_pwn2own1/`

has also become feasible for less funded attackers with just USD 1000[1,2] [78–80]. Breaking more recent versions of the cipher is also an area of interest [81]. An even bigger issue with GSM is its lack of authentication of the base station. It is therefore possible for anyone with a radio transmitter to act as a base station and trick mobile devices into connecting to it. Such MiTM devices are called Stingers or IMSI-catchers [82]. They can capture the IMSI and IMEI for tracking purposes, as well as intercept audio data of outgoing calls. Mechanisms to detect them have been proposed[3,4] [83]. Vulnerabilities in GSM's paging procedure can also lead to communication interception [84]. With the rise of femtocells, attacking a carrier's network using temporary infrastructure has also become a realistic threat to consider [85].

IMSI-catchers require an attacker to be in the vicinity of its targets. But there exist other techniques to achieve the same remotely. The Signalling System No.7 (SS7) is a signalling protocol for public switched telephone network (PSTN). It is used for call setup and teardown. Through the standardization of the SIGTRAN protocol, it has become possible to transfer SS7 messages over IP networks such as carriers' networks. SS7 messages are neither encrypted nor authenticated. Therefore, they allow attackers to remotely track mobile subscribers as well as redirect communications for interception[5,6,7,8] [86].

There has been a renewal of interest in the privacy implications of carriers' network protocols, such as the limitation of temporary IMSI (TIMSI) to prevent IMSI-catcher-based tracking [87, 88] and the feasibility of tracking on LTE networks [89]. By leveraging network delays resulting from the radio state of a mobile device (e.g. idle state), it is even possible for a remote device to identify a victim IP by sending the victim a few messages [90]. More recently, LTE implementation vulnerabilities have also been showcased. Voice-over-LTE uses the smartphone OS and the SIP protocol to handle call initialization and tear down. This, combined with the lack of access control implemented by carriers, means that a malicious app running on the smartphone OS can implement caller-spoofing, over-billing and denial-of-service attacks by sending rogue SIP packets onto the network[9] [91]. There is also a permission mismatch on Android, since voice services – usually protected through a CALL_PHONE permission, can be accessed with the innocuous internet permission – for sending SIP packets. Similar issues are reported in [92]. Battery-draining and DoS attacks were also demonstrated by exploiting NAT and firewall rules implemented by cellular middleboxes [93].

[1] https://srlabs.de/decrypting_gsm/
[2] https://www.youtube.com/watch?v=0hjn-BP8nro
[3] https://secupwn.github.io/Android-IMSI-Catcher-Detector/
[4] https://opensource.srlabs.de/projects/snoopsnitch
[5] http://secuinside.com/archive/2015/2015-2-7.pdf
[6] http://www.ptsecurity.com/upload/ptcom/SS7_WP_A4.ENG.0036.01.DEC.28.2014.pdf
[7] http://blog.ptsecurity.com/2015/01/mobile-eavesdropping-via-ss7-and-first.html
[8] https://www.washingtonpost.com/news/the-switch/wp/2014/12/18/german-researchers-discover-a-flaw-that-could-let-anyone-listen-to-your-cell-calls-and-read-your-texts/
[9] https://www.kb.cert.org/vuls/id/943167

2.4 App Vulnerabilities

A large proportion of app vulnerabilities result from design choices in the smartphone OS of the kind discussed in Section 2.3.

One prominent issue on Android is that many apps improperly authenticate and parse incoming IPC messages[1] [45–47, 94] (Section 2.3.1). More subtle IPC problems arise when dealing with web-based apps. Because of web mashups, it becomes impossible for a receiving app (e.g. Facebook) to identify the origin of a request [95, 96]. Prominent apps such as Facebook create custom handlers for url schemes (e.g. *fb://*) which could be subverted to access supposedly-private functions in the app – a special case of improper "intent" validation. Similar issues have been identified in third-party mobile development SDKs [97]. More generally, mobile development SDKs add an extra layer where things can go wrong[2].

At the network level, apps sometimes improperly validate SSL certificates. Both static [98] and dynamic [99] analysis tools have been devised to detect such problems – the latter being far more accurate. Researchers have also demonstrated that they can infer what users do on their phone by monitoring app traffic [100]. It is even possible for an Android app to work out if a user is home by monitoring the characteristics of traffic between an app and its associated home surveillance/motion detection system [101].

Certain apps do not properly authenticate the origin of code they load at runtime [1]. Web-based apps do not properly validate their input, leading to cross-site scripting vulnerabilities [102]. Apps do not always follow the best practices for storing authentication tokens [103] (e.g. static hard-coded credentials). As early Android versions did not support a secure storage necessary to do this properly without killing usability, it may be unfair to blame app developers. Moreover, the impact of such problems is limited in practice.

Most of the studies cited above attempt to identify a certain class of vulnerabilities in a large amount of apps, often through static analysis. Some papers take a different approach, by looking for a wide variety of vulnerabilities within a smaller set of apps. This usually requires manual analysis. For example, many popular Chinese apps were found to implement authentication improperly to their remote server [104], and this can be as stupid as hardcoding credentials in the app [105]. Popular banking apps were found to do not always follow best security practices either [106].

In this dissertation, we perform a thorough security analysis of the prominent Android anti-theft apps [107]. We highlight a set of problems such as improper API use, SSL validation issues, etc. (Chapter 6).

[1] `http://www.cvedetails.com/cve/CVE-2015-3843/`

[2] `http://blog.trendmicro.com/trendlabs-security-intelligence/setting-the-record-straight-on-moplus-sdk-and-the-wormhole-vulnerability/`

2.5 Malware

2.5.1 Malware Categories

Research on identifying malicious apps has generated the greatest number of papers in the mobile security literature. But it is often not easy to compare approaches taken by different researchers because there is no consensus on what should be considered to be malware. So in the following, we lay out a list of malware "types" that are prevalent in the literature.

In the early days of mobile security research (2007-2011), malware principally referred to an app that could **gain root access** to a device. The reason for this simplified view may have been the immaturity of the field. Within the first three years after Android's initial release, so many root vulnerabilities were discovered[1] that there was little point looking beyond root malware anyway.

Since then, the community has sharpened its understanding of what can be considered malware. Google itself maintains a list of 17 different types of malware in its yearly security report [108] – probably the most granular and comprehensive classification available. Google goes further than most other research endeavours by even considering DoS and malicious websites in its classification. This level of granularity is not essential to understand mobile malware, so we keep the description simpler in the following paragraphs.

Adware and **spyware** are not malware in the ordinary sense, since they do not trick users into installing software; neither do they exploit vulnerabilities in devices. They play and abide by the same rules as other apps, except that they abuse them somewhat. Adware is a major problem on mobile platforms because ad libraries are packaged into legitimate apps by developers. So ad libraries enjoy the same level of permissions and rights as the applications themselves do. This increases the attack surface both on Android[2] [109] and iOS[3]. There have been proposals to implement privilege separation between apps and their ad components [110–114], but none have been deployed. Furthermore, these do not prevent exploitation of OS vulnerabilities by ad libraries[2]. Spyware abuses the permission system and siphons out more information than strictly needed, or simply does not alert users when and why it collects personal information. Data collected may include device identifiers, users' email addresses, contacts, user location, etc. Spyware has emerged as the biggest threat to the mobile ecosystem, and this may explain in part why Google has introduced permission management features in Android 6. A lot of papers have looked at the detection of spyware, through field studies [115, 116], static analysis [117, 118] and dynamic analysis [119, 120].

[1] http://androidvulnerabilities.org/by/year/

[2] https://blog.lookout.com/blog/2015/11/04/trojanized-adware/

[3] https://www.fireeye.com/blog/threat-research/2015/11/ibackdoor_high-risk.html

Another type of malware is **chargeware**. These apps attempt to make money by charging users. They used to involve sending SMS to premium numbers. Google actively scans for these now. Recent Android versions also prompt users with a confirmation dialog if they detect an app sending a text to a premium number. Chargeware has also been WAP-based: in this scenario, attackers exploit WAP billing by automating website interactions. According to Google, chargeware declined by 60% between the first and last quarters of 2014 thanks to Google's VerifyApps system [108].

Another emerging type of malware is so-called **ransomware**. These apps lock up the phone screen or user files. Then they ask users to pay a ransom to get it unlocked. There is little academic research on this yet, but the technique has been gaining in popularity recently among malware authors [108].

There is such a rich spectrum of malware type that the term "Potentially Harmful Application (PHA)" has been coined to cover them all. PHA encompasses any app or behaviour that could, potentially, elicit undesired behaviours. As a result, many papers consider the detection of only a subset of PHA, be it through static [121–125] or dynamic [126–129] analysis. Overall, various sandbox frameworks have emerged to help researchers study malware, e.g. [130, 131].

A recurring question about mobile security in general is the extent to which malware is a problem. Mobile Anti-Virus companies claim Android is riddled with malware, whereas Google believes 99% of users do not benefit from installing a mobile AV app[1]. In 2015, Symantec again claimed that 17% of all Android apps were malware[2]. Even though Android surely could not withstand targeted attacks by well funded attackers, research has repeatedly found little evidence of malware in the Google Play store. Infection rates reported have been below 0.5% [121, 122, 132, 133]. By the end of 2015, Google [108] and Alcatel-Lucent [3] independently reported around 0.1% infection rates for PHA. So this raises an other question: are we good enough at detecting mobile malware? Perhaps not quite; in 2015, a root app on the Google Play store was installed by more than 100K users[3]!

2.5.2 Malware Distribution Channels

One prominent way to distribute malware is through repackaged apps in third-party app stores. Static and dynamic analysis approaches have been devised to detect them [134–137].

Another avenue to distribute malware is through ad libraries and networks[4]. More interestingly, attackers are increasingly going after developers themselves, e.g. by distribut-

[1] https://nakedsecurity.sophos.com/2014/07/09/googles-android-security-chief-dont-bother-with-anti-virus-is-he-serious/

[2] https://www.symantec.com/security_response/publications/threatreport.jsp

[3] https://securelist.com/blog/mobile/71981/taking-root/

[4] https://blog.lookout.com/blog/2015/11/04/trojanized-adware/

ing trojanized IDEs on third-party websites[1], through phishing techniques and by trying to buy the data app developers collect.

Another, little-studied way to install malware on devices is by infiltrating the supply chain. In January 2014, it was hypothetised that a newly-discovered android bootkit was inserted into devices by retailers in an IT mall in Zhongguancun, Beijing[2]. Later in March, Marble security firm claimed having discovered a pre-installed malicious NetFlix app in phones from various vendors[3]. One month later, Kaspersky also detected pre-installed malware supposedly installed with a kit sold by Chinese company Goohi[4]. In 2015, the German company G Data also found evidence of Android devices pre-loaded with malware[5]. The company believes the malware was introduced by middlemen who operate in China. Other companies like mSpy[6] openly sell smartphones from various vendors pre-loaded with spyware.

2.5.3 Mitigation Solutions

Mitigation techniques against malware can be broadly classified into (*i*) those that require OS changes and (*ii*) those that only require an app to be installed. App-based mitigations are usually less robust than OS-based ones, but they can be widely deployed.

The first work to suggest using a normal app to prevent spyware is Aurasium [138]. This takes as input an Android installation file, injects a library into it, and outputs a new installation file. The injected library is launched at runtime and hooks into all syscalls. This allows Aurasium to intercept accesses to sensitive resources such as contacts, etc. A similar system called I-Arm-Droid [139] re-writes the Java bytecode so as to instrument sensitive APIs at runtime. AppCage [140] uses Software Fault Isolation (SFI) to sandbox untrusted code. NativeWrap [141] confines different web domains into different security domains by re-writing webview apps. Boxify [142] is an Android app that intercepts all a non-trusted app's access attempts to shared resources. This is achieved by running the untrusted app in a confined process through Android's *isolated process* feature. PrivacyGuard [143] runs a VPN server on a smartphone to proxy and filter other apps' traffic.

Another approach to mitigating malware is through OS changes. There is a wide spectrum of work. The first approach improves Android's permission model by adding flexible permission management features [37–39]. As we mentioned in Section 2.3.1, a subset of these ideas was eventually incorporated by Google into Android 6. Another

[1] `https://www.fireeye.com/blog/threat-research/2015/11/xcodeghost_s_a_new.html`
[2] `http://blogs.360.cn/360mobile/2014/01/17/oldboot-the-first-bootkit-on-android/`
[3] `http://www.pcadvisor.co.uk/news/security/3505208/pre-installed-malware-turns-up-on-new-phones/`
[4] `https://www.securelist.com/en/blog/208213028/Caution_Malware_pre_installed`
[5] `https://securelist.com/blog/mobile/71981/taking-root/`
[6] `http://www.mspy.com/spy-phone/`

line of work mitigates system and/or privileged application compromise by retrofitting Mandatory Access Control (MAC) into the existing OS, such as with SELinux [144] or Tomoyo [145]. Google incorporated SELinux into Android 4.3. A few projects take isolation further through so-called lightweight virtualization. These leverage container-like ideas to create virtual namespaces [146–148]. More reliable still, we find solutions based on actual microkernels [149] and virtualization [150] that further reduce the attack surface. Using hardware features of ARM CPUs, ARMLock [151] implements SFI to run untrusted code.

2.6 Side Channels

Side-channel leaks can affect a wide range of communication layers such as protocols, applications, the OS, the CPU, etc. Side-channel attacks abuse subtle information leakage in design and/or implementation. In the following paragraphs, we give a brief overview of side channels in general, then we focus on those that are specific to smartphones.

On smart cards, power-analysis side channels were introduced by Kocher [152] and can recover keys by monitoring power consumption during cryptographic operations. Cryptographic keys can also be recovered by unprivileged software through cache-timing attacks if the code path or data access is data-dependent, e.g. unprotected RSA modular exponentiation [153] or AES table lookups [154, 155]. With the rise of virtualization and cloud computing, interest in this area has blossomed [156–158]. Approaches for cache-based attacks include the L3-cache Flush+Reload [159–161] and Flush+Flush [162], the L1-cache Prime+Probe [163, 164], and Evict+Time [163] techniques. These attacks have been hampered on ARM, and by extension on smartphones, because of a lack of (*i*) unprivileged flush instruction and (*ii*) support for inclusive shared last-level caches. The literature on cache-based side channels on ARM is therefore sparse [165–168]. Some recent attempts to port existing x86 techniques on to newer ARM CPUs with support for inclusive shared last-level cache (ARM Cortex-A53/A57) have recently succeeded [169], so these are becoming a practical threat.

Another general category of side channels are those based on protocols. Cache [170] statistically fingerprints 802.11 implementations through their duration field. Nmap[1] sends a series of network packets to a machine and infers the OS it runs based on its network stack behaviour.

Another general category of side channels are those based on radio. Perta *et al.* [171] abuse the different radio states of cellular phones to infer the phone IP address; the time it takes a phone to reply to incoming traffic depends on its level of radio activity. Brik *et al.* [172] identify the source network card of an IEEE 802.11 frame through passive radio-frequency analysis.

[1] `https://nmap.org/`

Another category of side-channels are those based on traffic. Stöber *et al.* [173] identify smartphones based on their traffic. Conto *et al.* [100] infer a smartphone user's activity based on its internet traffic. More generally, internet packet length and timing characteristics are also used for webpage fingerprinting [174] and identifying protocols such as Tor [175]. Traffic analysis can also be used to infer voice content in encrypted VoIP traffic [176, 177].

More specific to smartphones, we find attacks that abuse sensors and/or peripherals. Mäntyjärvi *et al.* [178] use gait recognition through the accelerometer to identify users. Michalevsky *et al.* [179] infer a person's gender by recovering spoken words through smartphone accelerometers. Sarfraz *et al.* [180] infer a user's location through their smartphone accelerometer. Michalevsky *et al.* [181] also infer a person's location by monitoring their phone's power consumption – it's correlated with the distance to the base station. TapLogger [182], TouchLogger [183], and TapPrint [184] infer a PIN entered on a smartphone by monitoring phone motion inferred from real-time accelerometer data. Dey *et al.* [185] fingerprint devices based on their accelerometer characteristics. Our work, presented in Chapter 3, works out the PIN through the phone camera, which is used to infer device motion during user input.

At the API level, Zhou *et al.* [186] infer the location of a user's smartphone through its software voice assistant. Specifically, they leverage the mutually exclusive access control of the audio API to infer the length of spoken words, and from these deduce the directions taken by a user.

The last category of side channels are those based on virtual filesystems such as *procfs*. Zhang and Wang [187] demonstrate how an app's stack pointer exposed by *procfs* can be used to fingerprint keystroke events and infer a user's password. Jana and Shmatikov [188] infer information contained in a webpage by monitoring the memory footprint of the web browser while it loads the page. More recently, Zhou *et al.* [186] show how traffic information gleaned by a smartphone app through virtual files can be used to fingerprint Twitter and identify the user.

In this dissertation, we explore two novel side channel attacks. In Chapter 3, we estimate the motion of a smartphone during user input through the front camera; and we use this motion as a side channel to infer PINs. In Chapter 4, we exploit the timing of the screen's hardware interrupts and of the context switch to infer what users type on their phone.

2.7 Forensics

There are three reasons why smartphone forensics matter. First, forensics are valuable for law enforcement agencies. They must support different platforms and a myriad of

different apps and phone models. This, coupled with the lack of privileges that normal apps have, makes forensic software challenging to write. Second, thieves could also extract personal information from devices they steal. Unlike the police, thieves need not support all models because they can target the most common phones only. Third, data could also be extracted from second-hand devices, and even if users have sanitised their device, as we show through our study of Android Factory Reset in Chapter 5.

2.7.1 Data Acquisition

Cannon [189], Osborn[1], and Ossmann and Osborn [190] attempted to access personal data on smartphones with physical access to them, by exploiting Android's debug options and bootloader mis-configurations. Vulnerabilities in the trusted boot, TEE and Bootloader implementations can also be used to gain OS-level access to the chip[2,3,4,5,6,7,8] [71]. Breaking into the device through the baseband OS [75–77] also gives enough privilege to perform data acquisition. Pereira *et al.* [191] discuss the security implications of non-standard AT commands exposed by certain Samsung phones via USB; these could be also used to access personal data even on screen-locked devices. Mahajan *et al.* [192] use commercial software on 5 different devices to recover Viber and Whatsapp chats from unlocked Android smartphones. The forensic software they use requires Android's debug option to be enabled for storage acquisition.

Müller *et al.* [193] use a cold boot attack on Samsung Nexus devices to recover FDE keys from RAM. Their hack, however, requires an open (or unlocked) Bootloader. To mitigate cold boot attacks, Copl *et al.* [194] use ARM-specific mechanisms to keep application code and data on the System-on-Chip (SoC) rather than in DRAM. Tresor [195] implements AES using x86's debug registers. AESSE [196] achieves similar goals through SSE instructions. CleanOS [197] is a customised OS that keeps plaintext key and data in memory only when necessary. Encryption keys are not stored on the smartphone but in the cloud: they are requested on a need-to-know basis and erased from RAM after use.

[1] `http://www.irongeek.com/i.php?page=videos/derbycon2/1-2-9-kyle-kos-osborn-physical-drive-by-downloads`

[2] `http://blog.azimuthsecurity.com/2013/05/exploiting-samsung-galaxy-s4-secure-boot.html`

[3] `http://blog.azimuthsecurity.com/2013/04/unlocking-motorola-bootloader.html`

[4] `http://bits-please.blogspot.com.ar/2015/08/exploring-qualcomms-trustzone.html`

[5] `https://www.codeaurora.org/projects/security-advisories/lk-insufficient-verification-tagaddr-when-loading-device-tree-cve-2014-0974`

[6] `https://www.codeaurora.org/projects/security-advisories/incomplete-signature-parsing-during-boot-image-authentication-leads-to-signature-forgery-cve-2014-0973`

[7] `https://www.codeaurora.org/projects/security-advisories/fastboot-boot-command-bypasses-signature-verification-cve-2014-4325`

[8] `https://www.codeaurora.org/projects/security-advisories/lk-improper-partition-bounds-checking-when-flashing-sparse-images-cve`

Hardware attacks through unprotected JTAG[1,2] and UART [190] can also be used for data acquisition on smartphones. Attaching a bus monitor to monitor data transfers between a CPU and main memory is also feasible, and commercial solutions already exist[3,4]. DMA attacks have been reported by Becher *et al.* [198], Boileau [199], Witherden[5], and Piegdon [200], although not on smartphones. Breeuwsma *et al.* [201] present low-level acquisition techniques to copy all a flash's content through flasher tools, JTAG, or physical extraction.

2.7.2 Data Reconstruction

Algorithms to descramble data after acquisition are paramount, especially to make sense of scattered pieces of data. GUITAR [202] is an algorithm that attempts to reconstruct the user interface through data remnants found in RAM. Zandwijk and Peter [203] devise methods to make sense of this raw data: this involves overcoming LFSR-based scrambling and leveraging binary cyclic codes used by the chip firmware. Luke and Stokes [204], Billard and Hauri [205], and Lewis and Kuhn [206, Chapter 5] devise various techniques to recover video files and even generic files from flash-based media. Walls *et al.* [207, 208] present algorithms for forensic triage of non-sanitised feature phones and smartphones.

2.7.3 Secure Deletion

Some studies have looked at users' practices upon device disposal. For example, Garfinkel and Shelat [209] studied sanitisation practices of second-hand magnetic hard disks and found no standard practice in 2003, with only 9% of disk properly sanitised. How to perform "secure deletion" has also been a topic of interest. Wei *et al.* [210] empirically assessed the reliability of hard drive techniques and of the SSDs' built-in sanitisation commands. They found that built-in commands are often effective, but sometimes implemented incorrectly by manufacturers. Because flash-based media do not support in-place data update, they found that all existing hard drive techniques for individual file sanitisation fail. This was also reported by Freeman and Woodward [211]. Gutmann [212] looked at techniques to achieve analog sanitisation on hard drives, and devised the well-known 35-pass overwrite technique. File sanitisation techniques generally rely on data encryption: storage sanitisation is then performed by securely erasing keys using built-in commands or raw flash access [213–215].

In this dissertation, we study the Factory Reset of Android smartphones (Chapter 5). We present its flaws and reveal the drivers behind them.

[1] http://www.forensicswiki.org/wiki/JTAG_Samsung_Galaxy_S3_%28SGH-I747M%29
[2] https://copgeek018.wordpress.com/2012/04/03/157/
[3] http://www.epnsolutions.net/ddr.html
[4] http://www.futureplus.com/download/datasheet/fs2334_ds.pdf
[5] https://freddie.witherden.org/pages/ieee-1394-forensics.pdf

2.8 This Dissertation

We now discuss where this dissertation stands in comparison with the overall mobile security research field.

In Chapter 3, we explore a novel side channel that uses the built-in camera and microphone in Android to infer PINs entered by a user in a secure app running in the Trusted Execution Environment (TEE). Previous researchers have shown that the accelerometer and gyroscope are a source of side channel. In this chapter, we broaden the scope of attacks and show that all shared resources (e.g. peripherals, sensors, wearable devices) that allow an incoming flow of information may be used as side channel. This chapter raises awareness of the difficulty of properly designing a trusted path, and the limitations of the current GlobalPlatform guidelines.

In Chapter 4, we explore a new side channel based on the timings of the screen's hardware interrupts and software interrupts that the Android OS exposes to permissionless apps. We use machine learning techniques to correlate the timings to what a user types on their phone. This is a vulnerability introduced by the OS provider, Google; and it could be abused to create spyware.

In Chapter 5, we present vulnerabilities in the way the Android OS erases personal data during the Factory Reset. We also highlight OEM failures to patch these flaws. People with access to second-hand devices could recover personal data from the previous owner even if the phone has been "wiped".

In Chapter 6, we conduct a security analysis of anti-theft apps for the Android platforms. We encover a wide variety of vulnerabilities in these apps; highlight the difficulty of developers to comprehend security APIs; and present some flaws introduced by OEMs. This chapter highlights the complexity of the Android ecosystem.

Chapter 3

Inferring PINs Through The Camera and Microphone

Previous researchers have studied the use of the phone accelerometer and gyroscope as side channel data to infer PINs. Here, we describe a new side-channel attack that makes use of the video camera and microphone to infer PINs entered on a number-only soft keyboard on a smartphone. The microphone is used to detect touch events, while the camera is used to estimate the smartphone's orientation. We then correlate the orientation to the digit tapped by a user. We hope to raise awareness of side channel attacks even when strong isolation such as TrustZone is used to protect sensitive applications.

For the evaluation, we use a Nexus S and a Galaxy S3. When considering a set of 50 4-digit PINs, the correct PIN entered by a user is one of the 2 most likely predicted PINs 30% of the time; and one of the 5 most likely predicted ones 50% of the time. When selecting from a set of 200 8-digit PINs, the correct PIN is one of the 5 most likely predicted PINs 45% of the time; and one of the 10 most likely predicted ones 60% of the time.

It turns out to be difficult to prevent such side-channel attacks, so we provide guidelines for developers to mitigate present and future side-channel attacks on PIN input.

The work presented in this chapter was published in the 3rd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM) [216], and is in collaboration with Ross Anderson. The idea of using the camera as side channel was suggested by Viktor Konstantinov, an undergrad at Cambridge from the Mathematics department. I worked on the implementation and evaluation. Ross helped with the writing and reviewing of the final paper.

3.1 Introduction

To provide stronger isolation for sensitive applications (e.g. DRM, payment, banking, corporate emails), some smartphone vendors employ additional isolation mechanisms.

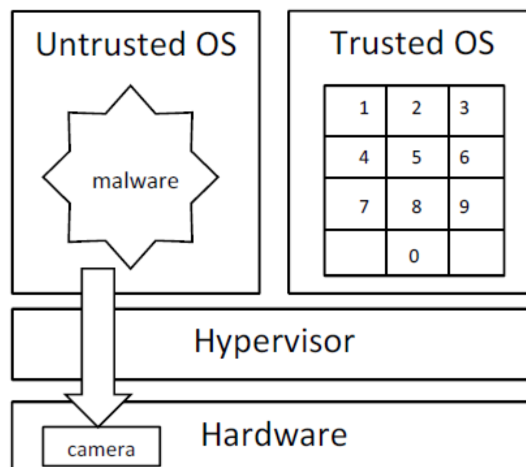


Figure 3.1: Attack scenario.

Samsung's KNOX[1] and BlackBerry's Balance[2] use so-called "containers" running on top of a single kernel. Users access their corporate documents in the "Work container" which is shielded from the "Home container" where third-party apps are installed. Theoretically, malware running in "Home" can exploit vulnerabilities in the shared kernel to get access to "Work". So to enhance protection even further, hardware-based isolation primitives are also deployed. The current trend is to run two entire OSes in parallel on the application CPU. The default OS (e.g. Android, WP, BB, etc.) runs as usual and can be customized with third-party apps; while the other OS only runs sensitive apps (e.g. corporate emails, banking apps, etc.). This separation can be achieved through virtualization, a microkernel, or ARM's TrustZone technology[3]. TrustZone is the solution that is currently being deployed by vendors. The default OS (e.g. Android) is usually referred to as the *Rich Execution Environment (REE)*, the *Insecure OS* or the *Untrusted OS*. The separate OS hosting sensitive apps is usually referred to as the *Trusted Execution Environment (TEE)*, the *Secure OS*, or the *Trusted OS*. Fig. 3.1 illustrates the concept with one Untrusted OS (left) and one Trusted OS (right).

While a user types sensitive information within the Trusted OS, the Insecure OS cannot access the screen, thereby providing an allegedly secure input path. This can be used to protect sensitive user input such as the unlock PIN, banking PINs or payment PINs for NFC. However, the sheer amount of shared hardware between the Trusted OS and the Untrusted OS opens up the possibility of side channel attacks. We already mentioned that through accelerometer readings, previous researchers showed that PINs can be inferred (Section 2.6). The insight is that device orientation is correlated to the digit touched by a user; and the orientation can be approximated through accelerometer data.

In this work, we evaluate the feasibility of using the video camera and microphone

[1] `https://www.samsungknox.com/en/support/knox-workspace/white-papers`
[2] `http://crackberry.com/blackberry-balance`
[3] `http://www.arm.com/products/processors/technologies/trustzone.php`

Figure 3.2: User holding & typing with one hand.

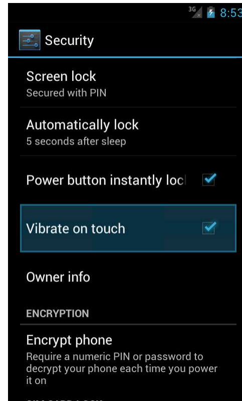


Figure 3.3: Screen lock vibration option in Android ICS.

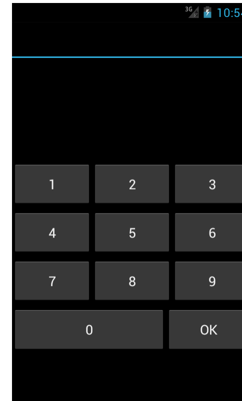


Figure 3.4: PIN pad.

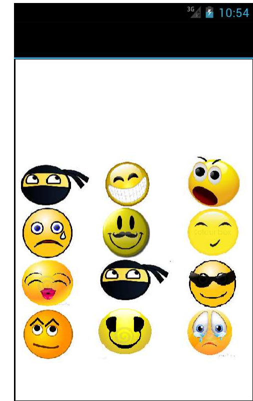


Figure 3.5: Game played by users in Collecting Mode. Users touch the identical icons.

to infer device orientation. By recording audio during PIN input, we can detect touch events (Section 3.3.4). By recording video from the front camera during PIN input, we can retrieve the frames that correspond to touch events and extract their relative orientation to a reference position. We show that this can be used to infer which part of the screen is touched.

The main contributions of this work are:

- The design, implementation and evaluation of a new side-channel attack that recovers PINs through the front video camera and microphone;
- Raising awareness of the difficulty of properly designing a trusted path. Specifically, all shared resources need careful consideration when reasoning about its security.
- Practical OS-level countermeasures to mitigate sensors-based side-channel attacks on sensitive input.

3.2 Attack Principles

Infection

We assume that the user has naively installed a malicious application from an app store (Google's or a third-party's), or maliciously been tricked into installing it via social engineering techniques. We initially assumed that the malicious application had exploited a vulnerability in Android and had gained root access on the device. We later discovered that, with some ingenuity, the attack could be performed by any app with camera and microphone permission (Section 3.6.1).

Phone Architecture

We assume a smartphone with two OSes running in parallel. A good example of such a phone is the Samsung Galaxy S3 which concurrently runs the Android OS and the TrustZone OS. Even though the malicious app has gained root access in the Android OS, the architecture of the phone ensures that it cannot access sensitive data in the TrustZone OS. As a plausible attack scenario, we assume the Trusted OS runs a banking application protected by a PIN. When a user wants to transfer money via the banking app, he opens the app in the Trusted OS and enter his PIN. When a user interacts with the Trusted OS, the Android OS (and hence the rootkit) cannot access the screen, hence providing a trusted path between the Trusted OS and the user. However, the rootkit still has access to certain shared resources like the accelerometer, the camera, the microphone, the GPS, etc. that can be used as side channels (Fig. 3.1). We will explain in Section 3.6.1 that if the banking application runs in the Android OS (rather than in TrustZone OS), other Android apps can also perform the attack without requiring root access.

PIN Settings

We focus the investigation on (digit-only) PINs because they are commonly used on phones (e.g. NFC payments, SIM PIN, unlock PIN, "dialed" PINs for banking services). We assume that a user types a PIN by touching the screen with the thumb of the hand holding the smartphone, as depicted in Fig. 3.2. Furthermore, we assume that the user touches the OK-button after entering his PIN in order to validate it. The PIN pad presented by the banking app is depicted in Fig. 3.4: it is identical to the one used to unlock Android smartphones. As user feedback, we assume the Trusted OS provides short vibrations upon each of the user's inputs. This is a common feature of smartphones' virtual keyboards and it is also one of the available options for the Android PIN lock screen, as depicted in the security settings view in Fig. 3.3.

Objective

Fig. 3.4 depicts the PIN pad displayed by the banking app running in the Trusted OS. Each button maps to a specific part of the screen. Our objective is to guess which part of the screen is touched by a user, through the front video camera and microphone data accessible to the Android OS while a user interacts with the banking app in the Trusted OS.

Cashing Out

Once the trojan in the Untrusted OS has inferred the PIN used to unlock the banking app running in the trusted OS, attackers need to cash out. We imagine that real miscreants

would advertise the PINs of phones they have compromised in underground forums along with the location of the devices. Remember that the trojan has root access in Android so has access to the GPS at will. Since smartphone theft is a growing problem, we imagine that smartphone thieves would "optimize" their theft by selectively tracking potential victims for whom the banking app's PIN is advertised in an underground forum.

3.2.1 Attack Flow

The attack has four modes of operation depicted in Fig. 3.6.

Monitoring Mode

In Monitoring Mode, the rootkit in the Untrusted OS monitors the user's behaviour to decide when to acquire data from the camera and microphone. In this mode, the rootkit can make use of all sensors (e.g. GPS, accelerometer, gyroscope) available to the Android OS to ascertain that necessary conditions are met before recoding data from the camera and microphone. For example, if the victim is in motion (e.g. walking), the data are noisy so it is important to filter them. We have not implemented this mode but previous research suggests this could be possible [217, 218]. This mode is left for future research.

Collecting Mode

In Collecting Mode, the user interacts with the malware, for example in the form of a malicious game running in the Android OS (Fig. 3.5). In this mode, a malicious game can legitimately receive all touch events from a user and simultaneously record data from the front camera. Every time a user touches the screen, the malicious app takes a picture with the front camera and saves the image to disk along with the digit it represents. Later, when WiFi becomes available (Section 3.2.2), the pictures can be uploaded to a remote server for processing. As presented later in the paper, the overall size of saved data is less than 2.5MB. Note that the number of smartphones with an embedded front camera is steadily growing as it enables the development of enhanced services like video calls.

Learning Mode

In Learning Mode, the remote server extracts relevant features from the collected data. The features are then fed to a learning algorithm in order to build a prediction model. In the future, we imagine that the Collecting and Learning phases could be skipped by building a generic model "offline" from a large set of users. For the current investigation though, we train each user individually.

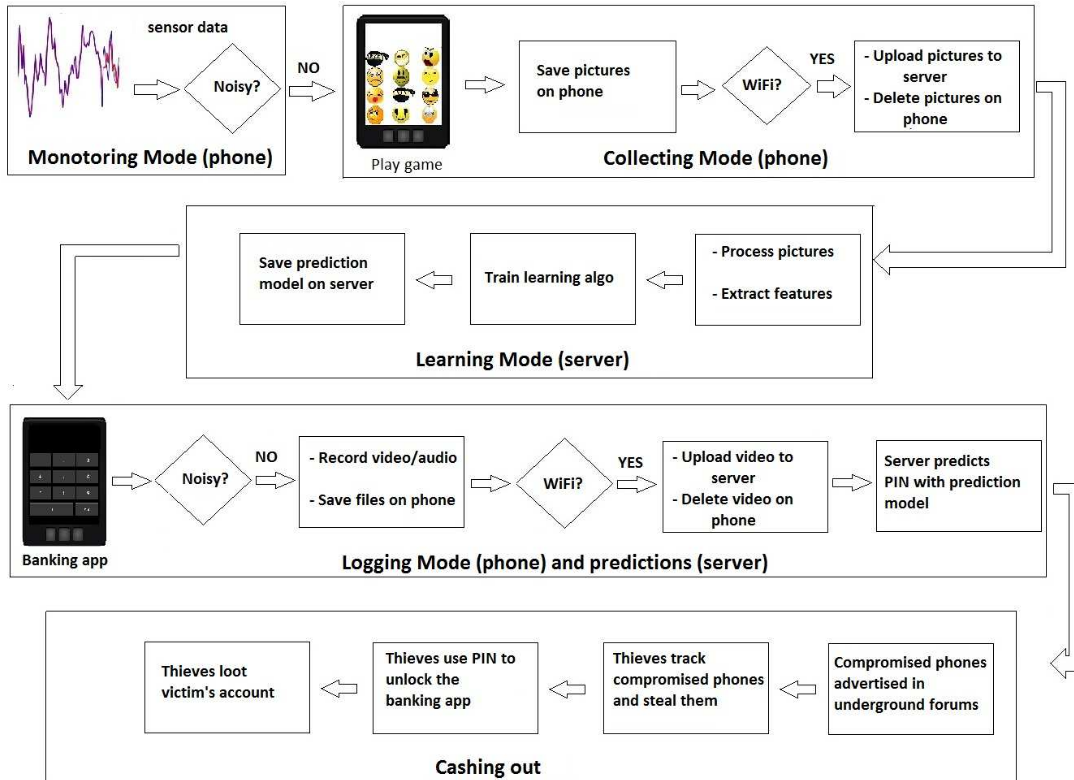


Figure 3.6: Modus operandi of trojan.

Logging Mode

In Logging Mode, i.e. when a user enters his PIN in the banking app in the Trusted OS, the malicious app turns on the front camera. It stores the video file (which contains the audio) on the phone before opportunistically uploading it to a remote server for processing. The server runs the algorithm trained in the Learning Mode to predict the PIN. At this point, the miscreants have a list of possible PINs for the banking app running on the compromised phones; which they could sell in the underground market. As presented later in the paper, the overall size of the video is less than 400KB.

3.2.2 Stealthiness

To be successful, malware must hide their behavior.

Overhead/Battery

Image processing algorithms are run on a remote server, not on the phone; so there is no noticeable battery drain noticeable by the victim. Similarly, the extraction of features, the training of the learning algorithm, and the predictions are performed server side so they do not cause battery drain on the phone. Because the data to upload does not exceed 2.5MB

for each mode of operation, there is no noticeable effect on battery when transferring data to the remote server.

Camera Use

The use of the camera must also be stealthy in order not to raise the victim's suspicion. Some phones have a LED that is automatically turned on when the camera is in use. The LED can be disabled via an API exposed by the Android OS. One possible drawback of this method is that it might not be supported by all android phones because of manufacturer customizations of the OS. Some phones also have a shutter sound when pictures are taken. The shutter sound could be disabled by temporarily muting the speakers while taking pictures. This could be an issue if the user is listening to music on his phone while malware mutes the phone. Our initial threat model considers malware with root access to the Android OS, where a robust way to disable both the LED and the shutter sound could be to tamper with the OS drivers.

Data Saved on Phone

As previously described, the trojan stores pictures in the Collecting Mode and a few seconds of video in the Logging Mode. As presented later in the paper, the data does not exceed 2.5MB for each mode of operation. We believe real malware would tamper with OS libraries to hide such files from users (as PC rootkits do).

Network Activity

The trojan requires Internet in order to upload the pictures stored during the Collecting Mode and a few seconds of video stored during the Logging Mode. To remain undetected by the victim, it can tamper with OS libraries to hide the number of packets sent to and received from a remote server. It is equally important to ensure not to incur charges to users for data usage. This is a genuine problem because some users have a limited amount of data they can spend per month. To this end, we believe a real trojan would opportunistically wait for a WiFi network to be in range to upload the data. Most users have a WiFi router at home to access the Internet, so a trojan would be able to upload data stealthily over WiFi every night when victims are home. If the trojan wants to hide network activity from the ISP, it could also make use of Domain Generation Algorithms (DGN) to be more covert[1].

[1] `http://www.pcworld.com/article/2038893/pushdo-botnet-is-evolving-becomes-more-resilient-to-takedown-attempts.html`

Permissions

The permissions needed by the trojan at installation time could arouse user suspicion. The stealthiness depends on the components exploited to root the phone. For example, some Android root exploits[1] do not need any permissions while others might. After exploitation, the rootkit could tamper with the relevant OS components to hide itself entirely from the victim.

3.3 Implementation Details

To test the feasibility of this attack, we ideally need a Trusted OS-enabled smartphone to run the PIN pad of Fig. 3.4. The Samsung Galaxy S3 features a TrustZone-based TEE called MobiCore developed by G&D[2]. Developing applications for MobiCore requires certification by G&D, so this is not possible at the moment.

So, to test the attack, we built the PIN pad of Fig. 3.4 as an Android application and ran it on the Google Nexus S and Samsung Galaxy S3 smartphones. While a user enters a PIN, the application also records the video stream from the front camera.

3.3.1 Collecting Mode

In order to build a prediction model, the trojan first needs to interact with users. We collected samples from users interacting with an Android application we developed. In practice, the application could be disguised as a game to trick users into touching certain areas of the screen (Fig. 3.5). The game takes a picture with the front camera when the user touches an icon and saves the images to disk. For the Nexus S, images have a resolution of 176×144 pixels and are of size 6.5KB each. For the Galaxy S3, images have a resolution of 320×240 pixels and are of size 24KB each.

3.3.2 Feature Extraction

After collecting mode, the server extracts relevant features from the images. Fig. 3.7 illustrates what happens when the user touches digit #1. In order for the thumb to touch the button, the supporting fingers push the phone upward towards the thumb. This has the effect of making the orientation of the smartphone change slightly. Note that the change of smartphone orientation is not the effect of the thumb touching the smartphone, but the necessary condition for the thumb to reach the button. This is different from accelerometer-based attacks which exploit the resulting orientation changes due to taps

[1] http://thesnkchrmr.wordpress.com/2011/03/24/rageagainstthecage/

[2] http://www.gi-de.com/en/trends_and_insights/mobicore/mobicore_1/mobicore.jsp

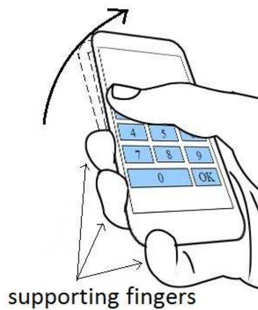


Figure 3.7: Supporting fingers push the phone upwards to touch a digit.

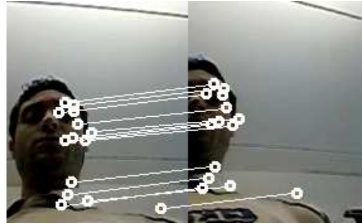


Figure 3.8: Key points of OK-button frame (left) and digit #1 frame (right).

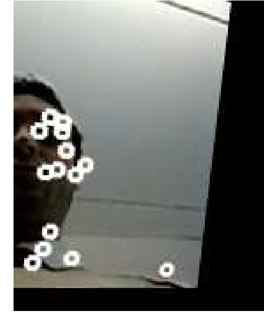


Figure 3.9: Image of OK-button frame by Homography.

$$HM_{OK,\#1} = \begin{pmatrix} 1.23589 & -0.08510 & 27.60422 \\ 0.25094 & 1.20318 & -34.89863 \\ 0.00155 & 0.00048 & 1 \end{pmatrix}$$

Figure 3.10: Homography Matrix between an OK-frame and a digit-#1 frame.

on the screen. As features, we use the relative rotation between two images taken at the same position. As point of reference on the screen, we use the OK button because it is always touched last and we know its position beforehand. Fig. 3.8 illustrates the relative rotation between an OK-button image and a digit-#1 image as seen by the front camera.

The relative rotation between an OK-button image and a digit image is computed as follows. First, given a digit image and an OK-button image, we extract the common key points using the RANdom Sample Consensus (RANSAC) method [219]. Key points are depicted as white circles in Fig. 3.8. They mostly correspond to the chest and the face of the user. This observation highlights the importance of using the front camera rather than the rear camera. The rear camera usually points to the floor. The floor often exhibits a homogeneous color and/or repetitive patterns, which makes the extraction of key points more difficult.

Secondly, given the common key points, we determine the relative rotation from the OK-button image (the reference) to the digit image. To this end, we compute the Homography Matrix (HM) [220], a 3×3 matrix that represents the rotation between two images taken at the same position. Here, we implicitly assume that the focal point of the camera is contained into the phone. This is a fair assumption: the focal distance is 0.9mm for the Nexus S and 2.5mm for the Galaxy S3. Fig. 3.10 shows the Homography Matrix representing the rotation between the two images of Fig. 3.8. The precision has been reduced to 5 digits to ease the reading. Fig. 3.9 shows the image of the OK-button by the Homography transformation of Fig. 3.10: it is rotated in such a way that it is identical to digit #1 image (apart from the missing pixels replaced by a black background).

We consider each element of the HM as a feature, and therefore use 9 (3×3) features. The open-source library OpenCv[1] is used for all these image manipulations.

3.3.3 Learning Mode

The objective in the Learning Mode is to use the features extracted from the images (i.e. the components of the HM) and train a learning algorithm in order to build a prediction model.

We store each feature as a numerical value with a maximum precision of 14 digits. As a learning algorithm, we use a Support Vector Machine (SVM) implemented with the open-source libraries LibSVM[2] and Weka[3]. When predicting a digit, the classifier outputs a probability for each of the possible digits, and we select the digit with the highest probability as the prediction. To build the model, we proceed in two phases. In the first "pre-experiment" phase, we get data from two users. The training data is used to experiment with the data and try different SVM parameters. We repeatedly (*i*) split the data into a 70% sub-training set and a 30% sub-test set, (*ii*) train the SVM on the sub-training set with different parameters, (*iii*) test the trained SVM on the sub-test set; until we find a configuration that leads to good predictions on the sub-test set. This process leads us to select the *nu-SVC* classifier with *linear kernel*, the normalized-feature option and $nu = 0.5$. Between 35% and 50% of the digits are correctly predicted on the sub-test. In the "experiment phase", users' training data is split into a 70% sub-training set and a 30% sub-test set (Section 3.4.1). The SVM is trained on the sub-training set while the sub-test set is used to evaluate how the trained SVM performs. Both the extraction phase and the Learning phase are performed server-side, not on the phone.

3.3.4 Logging Mode

In Logging Mode, the objective is to use the prediction model constructed in the Learning Mode in order to predict the PIN entered by a user in the PIN pad of Fig. 3.4.

While a user enters his PIN, the application turns on the front camera and records the video stream to a file. The video file contains one video stream and one audio stream. On the Google Nexus S, the video stream is composed of consecutive image frames sampled at 15Hz, of resolution 176×144 pixels, encoded with UYV420p and compressed with H.264. The 16-bit mono audio stream is sampled at a much faster frequency of 16KHz and encoded with AAC. The Samsung Galaxy S3 has similar video properties but frames are sampled twice as fast (30Hz) and have a higher resolution of 640×480 pixels. Three seconds of video represent about 75KB for the Nexus S and 390KB on the Galaxy S3.

[1] `http://opencv.org`
[2] `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`
[3] `http://www.cs.waikato.ac.nz/ml/weka/`

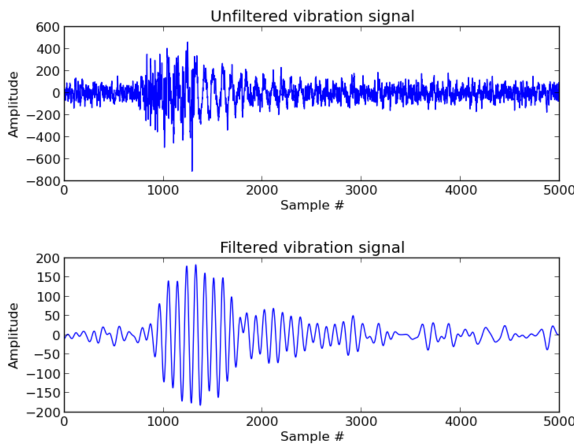


Figure 3.11: Audio signal during vibrations in a controlled environment on Nexus S.

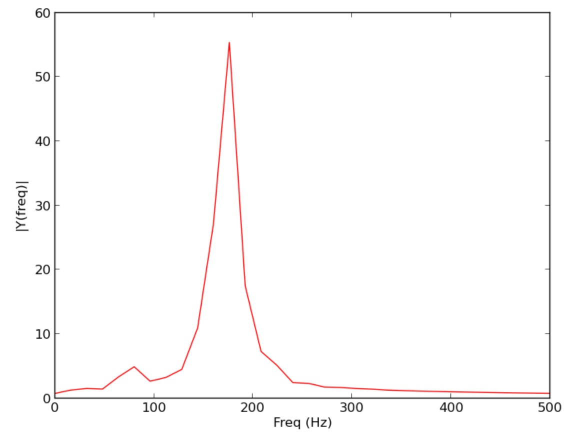


Figure 3.12: FFT coefficients of vibration signal on Nexus S.

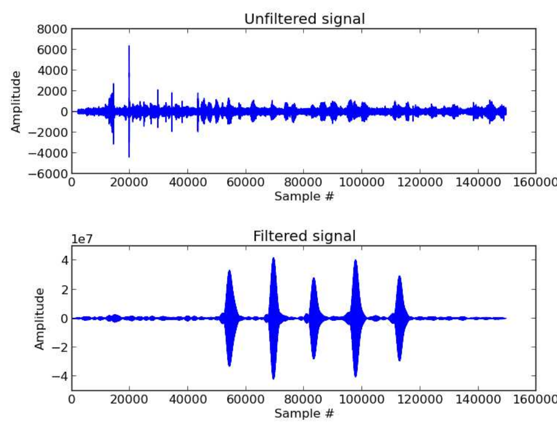


Figure 3.13: Audio signal while watching youtube video during PIN input.

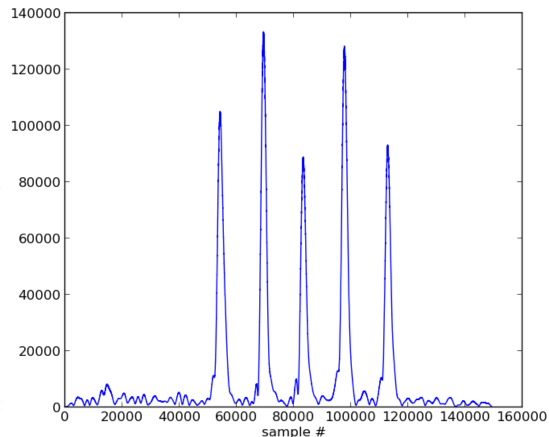


Figure 3.14: Convolution product of absolute value of filtered signal with 30ms-wide rectangular window.

Unlike in collecting mode, malware cannot legitimately receive touch events because the architecture of the phone prevents it (Section 6.1). Fortunately, the Trusted OS provides short vibrations upon each of the user's inputs (Section 4.2.3). When the vibrations occur, they loop back into the microphone, and we try to extract them from the audio stream contained in the video file to detect touch events. Hence, it is the microphone that allows us to detect touch events in this phase. Once we have touch events, we can extract the features of the corresponding frames (Section 3.3.2) and use them as input to the prediction model (Section 3.3.3) to predict PINs.

Vibration Characteristics

Fig. 3.11 shows an audio sample in a controlled (silent) environment captured by the Nexus S's microphone during a vibration. The top image represents the raw audio sample; the bottom image represents the same sample passed through a lowpass filter to remove the noise. In the filtered signal, the vibrations start at sample 1000 with the maximum amplitude reached at sample 1500, representing 30ms. After sample 1500, the signal progressively attenuates and finishes at sample 3000.

The spectrum of the filtered signal within the sample range $[1000, 2000]$ is shown in Fig. 3.12. We can deduce that the vibrations occur at a frequency of 180Hz. Hence, in order to de-noise the audio stream on the Nexus S, we decided to use a butterworth bandpass filter of order 3 with lowcut 175Hz and highcut 185Hz. A similar analysis shows that the vibrations occur at a frequency of 205Hz on the Galaxy S3; so we use a bandpass filter with lowcut 200Hz and highcut 210Hz to de-noise the audio signal.

Vibrations in Noisy Environment

Fig. 3.13 represents the audio signal extracted from a video recorded by the Nexus S while a user types a PIN and watches video from a nearby desktop computer: the top image represents the unfiltered signal, the bottom image represents the same signal passed through the bandpass filter (Section 3.3.4). In the filtered signal, the touch events become apparent. Essentially, the de-noising is reliable in environments which exhibit few harmonics in the passband ($[175, 185]$ for Nexus S; $[200, 210]$ for Galaxy S3).

Improving Vibration Detection in Noisy Environments

If the noisy environment contains harmonics in the passband, the de-noising method is not fully reliable. So to further protect against false positives (i.e. a touch event is detected when it is not actually present), we convolve the absolute value of the filtered signal with a rectangular window of duration 30ms. Recall from Section 3.3.4 that the vibrations actually last 30ms. Fig. 3.14 shows the result of the convolution product on the filtered signal of Fig. 3.13. The 5 peaks indicate frames corresponding to a touch event. Given the assumptions made in Section 4.2.3, the first 4 peaks correspond to a 4-digit PIN, while the 5th corresponds to the OK button.

Table 3.1: Renamed digits.

	Meaning	digit for LHU	digit for RHU
F_{top}	**F**ar from thumb, **top** position	#3	#1
N_{top}	**N**ear thumb, **top** position	#1	#3
F_{mid}	**F**ar from thumb, **mid**dle position	#6	#4
N_{mid}	**N**ear thumb, **mid**dle position	#4	#6
F_{bot}	**F**ar from thumb, **bot**tom position	#9	#7
N_{bot}	**N**ear thumb, **bot**tom position	#7	#9

3.4 Evaluation

3.4.1 Setup

In collecting mode, we collect samples from four users: two right-handed, the other two left-handed. Users are seated and interact with an app we developed for 2 minutes by touching digits on a screen. We obtain an average of 10 touch events for each digit and for the OK-button. This is each user's training set, about 650KB to save on disk for the Nexus S and 2.4MB for the Galaxy S3.

For each image in the training set, we extract the HM using all the OK-button images as a reference (Section 3.3.2). Then we use the HM data to build the prediction model. On a Linux machine with a 2.4GHz CPU and 1GB of RAM, it takes less than 1.5 minutes and 3 minutes to compute all the HMs for the Nexus S and Galaxy S3 respectively. The resulting HM data is split into a 70% sub-training set and a 30% sub-test set. The SVM is trained on the sub-training set while the sub-test set is used to evaluate how well the trained SVM performs. It takes about 8s to build the prediction model on the sub-training set.

To evaluate the predictions, we randomly select 100 8-digit PINs. Users enter each PIN once in the PIN pad of Fig. 3.4. This represents the evaluation set, and is different from the training set collected in collecting mode. We run the prediction model on the evaluation set, and we discuss the results in the following sections.

3.4.2 Single-Digit Prediction

To present the results, we first needed to understand the influence of a user's handedness on digit prediction. Re-consider Fig. 3.7. For a right-handed user, the left part of the screen is "far" from the thumb while the right part is "near". For the thumb to reach a "far" screen position (e.g. digit #1), the supporting fingers need to "lift" the phone more than for a "near" screen position (e.g. digit #9). Inversely, for a left-handed user, the right part of the screen is "far" from the thumb and the left part is "near". Hence, to

Table 3.2: Confusion matrix for Nexus S.

	F_{top}	#2	N_{top}	F_{mid}	#5	N_{mid}	F_{bot}	#8	N_{bot}	#0
$\mathbf{F_{top}}$	**49.8**	7.0	1.8	**18.9**	3.2	1.6	**12.7**	2.0	1.3	1.9
#2	3.4	21.0	8.7	12.8	13.3	6.1	15.4	8.1	3.4	7.9
N_{top}	1.6	15.2	12.2	4.5	13.1	10.5	8.4	12.4	9.9	12.3
F_{mid}	16.6	16.8	4.1	26.3	6.0	3.0	17.5	3.6	1.9	4.4
#5	2.3	18.3	11.4	6.9	14.4	9.0	9.1	12.4	6.2	9.9
$\mathbf{N_{mid}}$	1.7	**10.8**	**13.4**	3.6	**12.6**	**11.9**	4.8	**12.8**	**12.9**	**15.4**
F_{bot}	6.0	21.1	6.8	19.4	9.1	4.5	19.3	5.5	2.4	6.1
#8	2.0	17.4	12.4	5.3	13.6	10.2	7.5	12.3	7.8	11.6
N_{bot}	1.1	8.9	14.5	1.7	11.8	13.4	3.9	16.5	16.3	12.0
#0	1.5	12.9	12.7	4.3	13.9	10.6	7.5	14.3	8.6	13.7

Table 3.3: Confusion matrix for Galaxy S3.

	F_{top}	#2	N_{top}	F_{mid}	#5	N_{mid}	F_{bot}	#8	N_{bot}	#0
F_{top}	57.3	8.8	3.0	13.4	2.8	1.4	7.4	2.4	1.6	1.9
#2	9.3	20.1	11.7	10.6	10.0	5.8	11.3	7.7	7.3	6.1
N_{top}	2.5	13.9	14.5	5.6	13.4	11.1	7.5	9.4	14.3	7.8
F_{mid}	26.1	15.4	6.4	19.6	5.8	2.9	12.3	4.6	2.9	4.0
#5	3.8	16.4	14.1	6.9	11.7	8.2	9.6	10.6	10.3	8.4
N_{mid}	1.4	9.9	15.8	4.0	12.5	14.2	5.7	10.1	17.8	8.6
F_{bot}	15.1	13.2	8.8	19.4	8.3	4.7	12.8	6.7	4.7	6.4
#8	3.5	15.4	13.9	10.2	11.2	8.5	9.5	10.3	8.6	8.9
N_{bot}	2.1	7.0	14.6	3.8	12.4	14.9	4.1	10.7	20.7	9.7
#0	2.6	14.3	15.0	8.1	11.9	9.1	9.9	11.2	8.3	9.7

present the predictions independently of users' handedness, we rename digits according to the "role" they play. For example, F_{top} represents the digit in the **top** position of the screen which is "**F**ar" from the thumb. For a right-handed user, F_{top} is digit #1; for a left-handed user it is digit #3. Digits #2, #5, #8 and #0, being in the middle of the screen, play the same "role" regardless of users' handedness, so we do not rename them. Table 3.1 gives the renamed digits with their associated "real" digit for left-handed users (LHU) and right-handed users (RHU). For a visual representation, Fig. 3.16 depicts a pad with renamed digits for a right-handed user.

For each digit entered by a user, the prediction model outputs the list of predicted digits sorted by probability from the highest to the lowest. We aggregate and normalize the probabilities to obtain the confusion matrices. Table 3.2 and Table 3.3 represent the confusion matrices for the Nexus S and Galaxy S3 respectively. Each row of the matrix represents the actual digit entered by a user, while each column represents the predicted digit. Ideally, if all predictions were correct, the matrix would have 100 (100%) on its

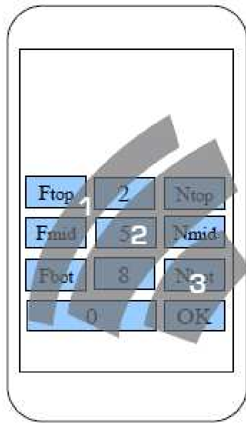


Figure 3.15: Areas reachable by the thumb with little help from supporting fingers.

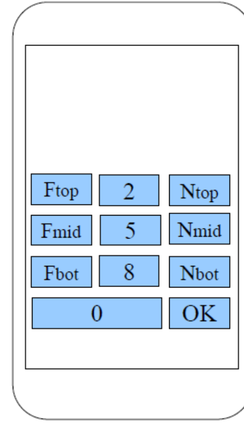


Figure 3.16: PIN-pad with renamed digit for a right-handed user.

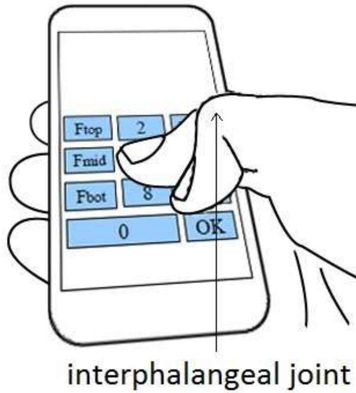


Figure 3.17: Rotation of thumb with interphalangeal joint.

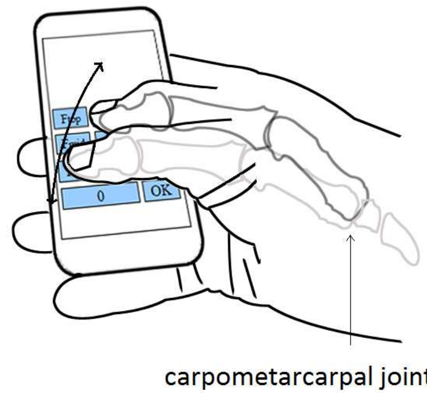


Figure 3.18: Rotation of thumb with carpometarcarpal joint.

diagonal and 0 anywhere else.

Consider the matrix for Nexus S (Table 3.2). For digit F_{top} (row 1), F_{top} (column 1) obtains 49.8% of the aggregated probability, F_{mid} (column 4) obtains 18.9% of the probabilities, digit F_{bot} (column 7) obtains 12.7% of the probability and other digits obtain negligible probability. On the other end, for digit N_{mid} (row 6), N_{mid} (column 6) obtains only 11.9% of the probability (just better than a random guess), while digits #0, #2, N_{top}, #5, #8 and N_{bot} all get more than 10% of probability. Digit N_{mid} is located near digit #2, N_{top}, #5, N_{bot} and #8 so it is not surprising that N_{mid} is mispredicted as one of them (Fig. 3.16). More surprisingly and less intuitively, digit #0 obtains more than 15% of the probability while it is not a neighbour of digit N_{mid}. We explain why in the following paragraphs.

As previously mentioned, when the thumb reaches for a digit, the fingers supporting the phone bring it forward. The thumb itself can also rotate around two axes, which greatly

influences the results. The first rotation is with the interphalangeal joint as depicted in Fig. 3.17. When touching screen positions “near” the thumb, the thumb can use its interphalangeal joint to reach neighbouring digits without help from the supporting fingers. However for “far” digits, the thumb needs to be in a “stretched” position and the interphalangeal joint does not help. Hence, one expects more “noise” and worse predictions for digits that are “near the thumb” (because pictures taken from the camera are similar).

The second rotation is with the carpometacarpal joint as depicted in Fig. 3.18. In Fig. 3.15, we have drawn areas that can be reached by the thumb without (or with little) help from the supporting fingers. The areas represent part of a circle, centered at the carpometarcarpal joint. The areas become thicker as we move towards the thumb to account for the noise due to the rotation with the interphalangeal joint. Digits in these areas should be mis-predicted as each other more often.

Given the explanations above and the approximate areas drawn in Fig. 3.15, one can understand why digit #0 obtains more than 15% for digit N_{mid}: #0 and N_{mid} can be reached by the thumb with the carpometacarpal joint rotation. One can also predict that digit #2 and F_{bot}, though not neighbours, should be mispredicted as each other; this is confirmed by the confusion matrices. Similarly, digits N_{top}, #8 and #0 should also be mispredicted as one another; this is also confirmed.

We also noticed little difference between the predictions on the Nexus S and Galaxy S3, despite the fact that the Galaxy S3 has a larger screen than the Nexus S. We expected that the larger the screen, the better the results would be, but the only noticeable improvement is the F_{top} digit which is better predicted on the Galaxy S3.

3.4.3 PIN Predictions

To evaluate PIN predictions, one can vary both the PIN length (i.e. the number of digits) and the size of the set. Varying the set size is motivated by the fact that users do not select their PIN randomly [221]. According to [222], with the 20 most common 4-digit PINs representing about 27% of all user-selected PINs. To predict a PIN, we first sort the PIN predictions by probability and keep the 30 most likely. For example, for a 50-PIN set, we obtain a list of the 50 possible PINs sorted by probability; which we truncate to keep the top 30 PINs only. If the correct PIN appears in first position, then the PIN is correctly predicted in 1 attempt. If the PIN appears in position n, the PIN is correctly predicted after n attempts. Intuitively, the larger the set, the greater the number of attempts to correctly guess a PIN.

Influence of the Phone

For simplicity, here we only present the results for 4-digit PINs. We consider subsets of the entire PIN space of size 50, 75, 150, and 200. Fig. 3.19 shows the prediction results for

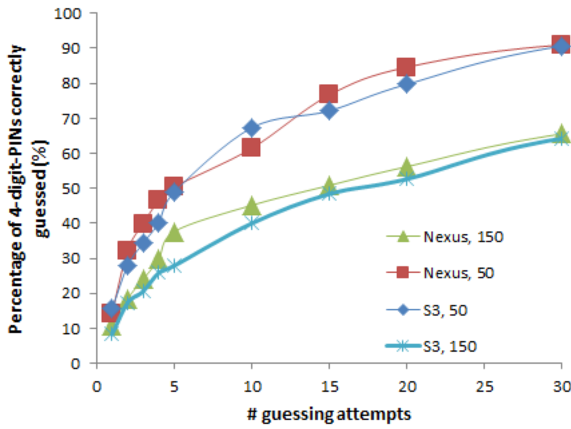


Figure 3.19: Percentage of 4-digit PINs correctly guessed for Nexus S and Galaxy S3 for PIN sets of size 50 and 150.

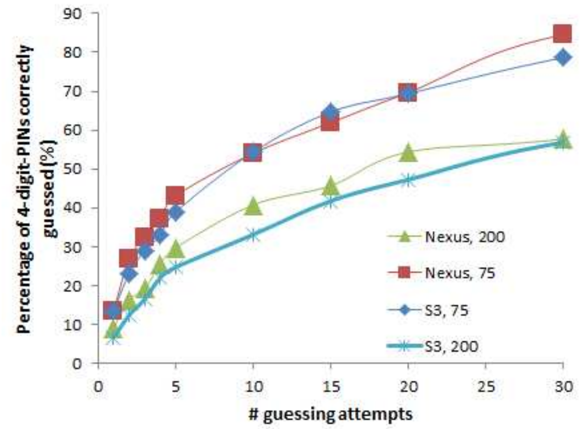


Figure 3.20: Percentage of 4-digit PINs correctly guessed for Nexus S and Galaxy S3 for PIN sets of size 75 and 200.

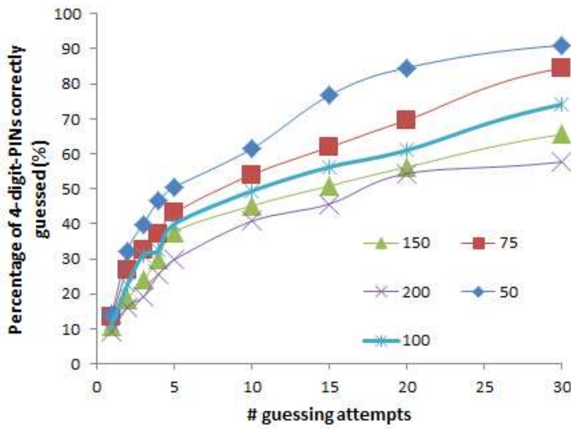


Figure 3.21: Percentage of 4-digit PINs correctly guessed for Nexus S for different sizes of PIN set.

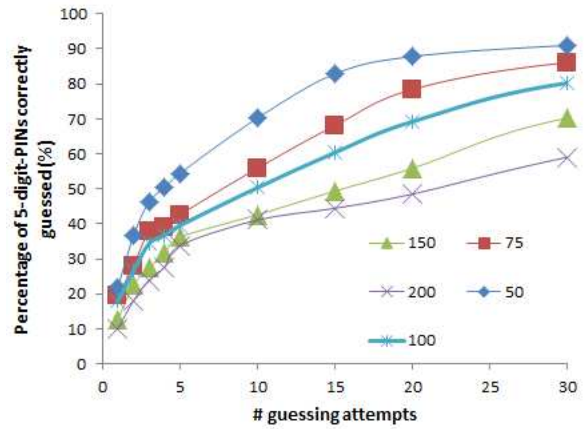


Figure 3.22: Percentage of 5-digit PINs correctly guessed for Nexus S for different sizes of PIN set.

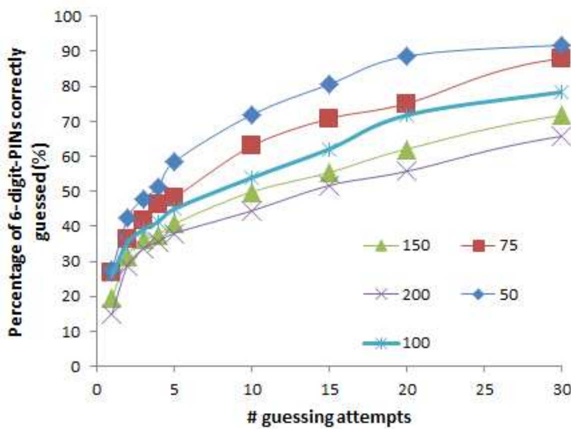


Figure 3.23: Percentage of 6-digit PINs correctly guessed for Nexus S for different sizes of PIN set.

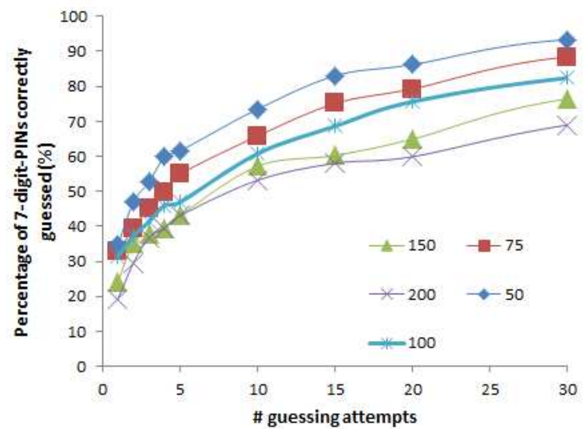


Figure 3.24: Percentage of 7-digit PINs correctly guessed for Nexus S for different sizes of PIN set.

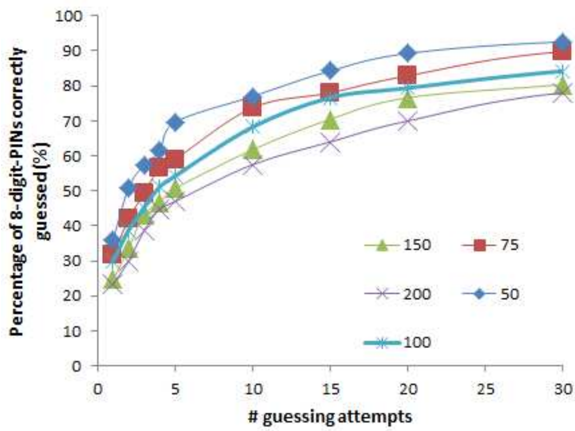


Figure 3.25: Percentage of 8-digit PINs correctly guessed for Nexus S for different sizes of PIN set.

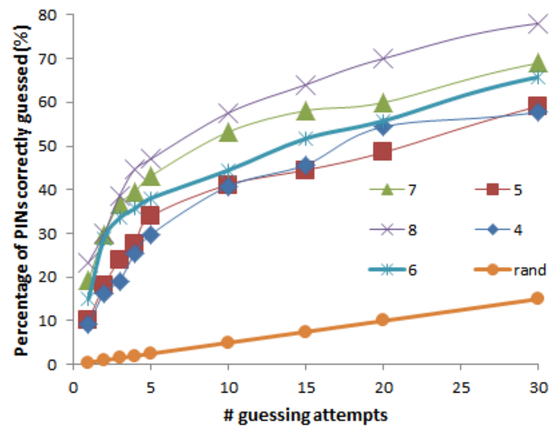


Figure 3.26: Percentage of PINs correctly guessed for Nexus S for a 200-PIN set for different PIN lengths.

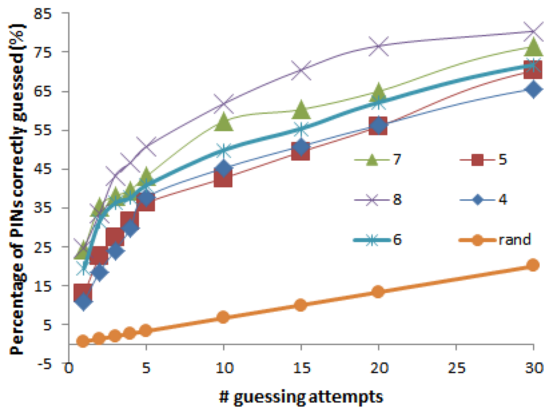


Figure 3.27: Percentage of PINs correctly guessed for Nexus S for a 150-PIN set for different PIN lengths.

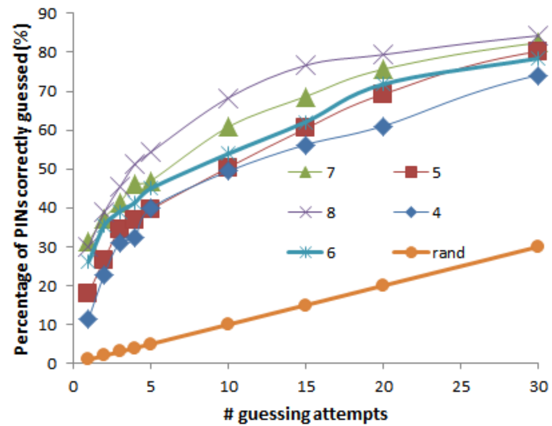


Figure 3.28: Percentage of PINs correctly guessed for Nexus S for a 100-PIN set for different PIN lengths.

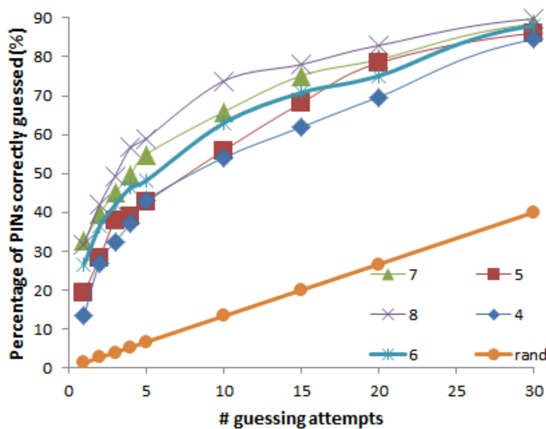


Figure 3.29: Percentage of PINs correctly guessed for Nexus S for a 75-PIN set for different PIN lengths.

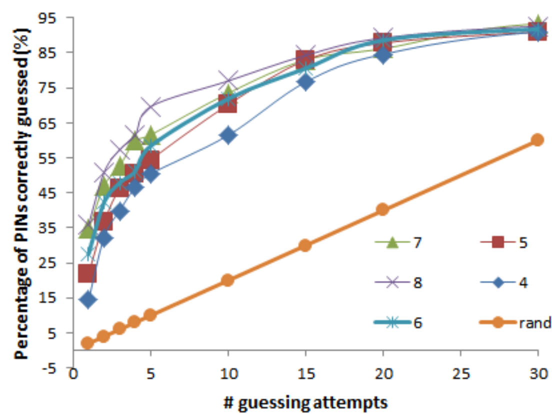


Figure 3.30: Percentage of PINs correctly guessed for Nexus S for a 50-PIN set for different PIN lengths.

PIN sets of size 50 and 150; Fig. 3.20 shows the results for PIN sets of size 75 and 200. For instance, for a 50-PIN set (Fig. 3.19), 50% of the PINs are correctly guessed after 5 attempts. As in Section 3.4.2, we noticed that the size of the phone has little influence on the results. Therefore, in the following sections, we only present the prediction results for the Nexus S phone.

Influence of Set Size

Here we vary the size of the PIN set we consider. Intuitively, the larger the size of the PIN set, the worse the predictions. We present the results for PINs of size 4 to 8 digits in Fig. 3.21 to Fig. 3.25.

Consider 4-digit PINs (Fig. 3.21). For a set of 50 PINs, about 30% of the PINs are correctly guessed after 2 attempts, and 50% after 5 attempts. For a set of 200 PINs, near 30% of the PINs are correctly predicted after 5 attempts. After 10 attempts (the maximum number of attempts allowed to unlock an iPhone), about 60% of a set of 50 PINs are correctly inferred, and about 40% of a set of 200 PINs. After 20 attempts (the maximum number of attempts allowed on Android devices), more than 80% of a set of 50 PINs are guessed and about 50% of a set of 200 PINs.

Influence of PIN Length

Here we vary the length of the PINs from 4 to 8 digits. The results are presented in Fig. 3.26 to Fig. 3.30. The straight line "rand" represents the prediction results of random guessing.

Consider a set of 200 PINs (Fig. 3.26). For 4-digit PINs, 30% are correctly guessed after 5 attempts. As the length of the PIN increases, the predictions improve: for 8-digit PINs, more than 45% of the PINs are correctly guessed after 5 attempts. This seems counter-intuitive at first. However, there is a reasonable explanation. First, the more the digits in a PIN, the more information one has and hence the greater the "distance" between them. Second, by keeping the size S of the PIN set constant, increasing the PIN length L is equivalent to decreasing the ratio $R = S/N$ where $N = 10^L$ is the size of the entire PIN space. For instance, for a set of 200 4-digit PINs, $R = 200/10^4 = 2\%$ of the entire PIN space, while for 8-digit PINs, $R = 200/10^8 is only 0.002\%$. Keeping the size of the set constant seems unfair because increasing the PIN length is supposed to increase the PIN space and make guessing more difficult. However, studies [221] show that users do not select their PINs randomly, so there is no direct correlation between the theoretical size of the PIN space and the one resulting from users' selection. As a convincing example, consider the space of passwords we use to access our email accounts. In theory, infinitely-long passwords of randomly-selected bytes are possible so the password

space has an infinite size. In practice however, we select finite passwords containing mainly ASCII characters; and often passwords are predictable English words.

3.5 Limitations

First, the relative rotation changes between two frames are calculated through a homography matrix, which in turn relies on finding key points between two video frames. In some cases, the number of key points obtained is not large enough to calculate the HM. Fortunately, such cases are rare and can be discarded in Collecting Mode. In logging mode, however, this problem leads to missing digit predictions. Detecting key points also relies on good pictures. This could be hampered by poor light and other lighting effects. More advanced image processing could be applied to overcome these problems. The speed at which the user types a PIN can also influence the quality of extracted frames: if he types a PIN too fast, it may render frames blurred and hamper the detection of the key points.

Second, this attack considers people who hold their phone and type PINs with the same hand. We do not known the percentage of the population that falls into this category. Most side-channel papers we mentioned in Section 2.6 assume that users use one hand to hold the phone and the other to enter their PIN. In this regard, our study is complementary.

Third, the attack currently considers a user who types an L-digit PIN followed by the OK-button ($L + 1$ touch events). It does not account for a user who unintentionally types a wrong digit, deletes it and continues typing. Other side channel papers do not consider this either (Section 2.6).

Fourth, in real-life scenarios, the collecting mode may be itself subject to noise. It is debatable whether a practical trojan would be able to acquire non-noisy information during collecting mode. In Section 3.2.1, we called this mode the 'monitoring mode' and mentioned that it is left to future research. The use of other sensors (e.g. accelerometer, GPS) could perhaps be used for this mode, as suggested by [217].

Fifth, the detection of touch events in logging mode relies on the assumption that few noise-frequencies are present in the passband ($[175Hz, 185Hz]$ for Nexus S, and $[200Hz, 210Hz]$ for Galaxy S3). This assumption is not guaranteed to hold every time a user enters a PIN. The human voice is assumed to have a spectrum between 300Hz and 3000Hz, so the de-noising should filter out most of people's conversations. But in our tests, we found that some male voices actually have a wider spectrum with low frequencies reaching down to 100Hz. In case false positives are detected (i.e. more than $L + 1$ touch events; with L the PIN length), the trojan could simply discard the data.

3.6 Possible Countermeasures

In this section, we present possible countermeasures to mitigate side-channel attacks based on sensors and peripherals. We also take into account previous papers that use accelerometer and/or gyroscope readings to infer PINs (Section 2.6). We also consider mitigations for devices that do not have a TEE because low-end devices may not support it.

3.6.1 Non-TEE devices

For devices that do not have a TEE, the PIN-pad application runs in the default OS and can be attacked by other apps. On Android, an application can record video only if it has screen focus. But there are ways around this restriction. First, using the SYSTEM_ALERT_WINDOW permission, apps can create *floating activities*, i.e. windows that display on top of other apps. Using a transparent 1×1-pixel *floating activity*, a malicious app could gain constant screen focus whilst remaining invisible to users, enabling it to capture video at will. Second, even without the former permission, applications with the CAMERA permission could take pictures even when running in background. Burst mode achieves 15Hz on both the Nexus S and Galaxy S3, so the attack remains possible. One caveat with the burst mode is that the time between two consecutive pictures is not always constant, so greater care must be taken to select the correct pictures for each touch event.

At the application level, mitigation options are limited. In Android, access to the microphone is exclusive so malware cannot access it if the PIN application does. However, access to other resources like accelerometer and gyroscope is always shared. An OS-level mitigation is more appealing because it centralizes the changes in one place and benefits all applications. In Android, there are mainly two ways to prompt a user to enter a PIN. The first is to use an *AlertDialog*[1] with the option *android:password="true"* in the manifest file. Upon displaying an *AlertDialog* with this option, we suggest the OS also deny access to shared sensors/peripherals resources to other user-installed applications. The second way to prompt for a PIN is via a GUI component (*Activity*). In this case, we suggest the OS expose a *PasswordActivity* which inherits from the *Activity*. The sole use of the *PasswordActivity* would be to inform the OS that the activity is used to collect sensitive information from users. When displayed, the OS should deny access to shared resources to other user-installed applications.

[1] `http://developer.android.com/reference/android/app/AlertDialog.html`

3.6.2 TEE-enabled devices

Here, the PIN-pad runs in the TEE. In this case, we also suggest an OS-level solution. The OS should provide a PIN/password GUI component callable by TEE applications. When the component is displayed, the OS (or the hypervisor) should deny access to shared resources to the untrusted OS and other TEE applications. The component should provide a default customizable PIN/password layout, but should also allow the application developers to write their own layout from scratch.

3.6.3 Other Considerations

In 2013 (when this work was published), the camera, the microphone, the accelerometer, and the gyroscope had been identified as side channels to infer PINs on smartphones. Unfortunately, it is not possible to anticipate which other shared resources can be used for side-channels. It is also likely that new sensors will be added to phones in the future; and this raises the question of which resources should remain accessible during PIN input. A naive solution would deny access to all resources, but this may affect usability. For instance, when a call comes in, a user needs to hear the ring-tone while unlocking his phone; otherwise he would miss the call. For these reasons, we think the use of a whitelist is more appropriate: deny access to all shared hardware resources except those explicitly allowed. The shared hardware resources we consider are the (video) camera, microphone, speakers, screen, BT, NFC, and on-board sensors[1]. Essentially, any incoming input should be prohibited as it may contain side-channel information about the PIN. The speakers could be in the white-list since they output sound but do not allow incoming information in. A more restrictive white-list could allow speakers only when used by the default "call application" or other system services. All the above, however, does not protect against side channels carried out by wearable devices such as smartwatches that are connected to the phone. Malware on a smartphone could instruct a smartwatch to record accelerometer readings during user input. Even if Bluetooth was disabled during the input, smartphone malware could wait till the end of user input to request the collected data be sent over by the watch. Therefore, to thwart multi-device side-channel attacks, all devices should be aware of a common policy. In the smartwatch example, this would mean disabling reading from all sensors and incoming data channels.

An orthogonal countermeasure to mitigate side-channel attacks is to use longer PINs (or passphrases) to increase the guessing entropy [223, 224], but this affects memorability and usability. Another additional countermeasure could be to enforce a maximum number of PIN attempts, as with bank cards. Unfortunately, the number of smartphone applications requiring a PIN will increase over time, forcing users to re-use them across applications and services. This makes it more difficult to enforce a maximum number of PIN attempts.

[1] `http://developer.android.com/reference/android/hardware/Sensor.html`

Randomising the position of the digits of the PIN pad is also an option to consider. Some online banking websites already use this on desktop applications. However, we believe this would cripple usability on phones. Banks have deployed randomised PIN pads mainly for money transfer; and a typical user may transfer money just a few times a month. In comparison, users need to unlock their phone throughout the day and make payments several times every day. If many applications randomised their PIN pad, it would further aggravate usability. Clearly, a randomised pad is not acceptable for all applications: for instance "payment companies" (e.g. Visa, PayPal) rely on building "frictionless" payment systems to maximize the number of users' purchases. With a randomised pad, users can no longer make payments reflexively. Last but not least, some users tend to remember their PIN by the position of the digits on the screen rather than by the digits themselves.

A more drastic solution is to get rid of passwords entirely. This could be achieved through biometrics, electronic devices which the phone can sense [23] and/or progressive authentication [225].

3.7 To Patch or Not to Patch

In the previous section, we concluded that a sound trusted path should stop any incoming flow of information (e.g. sensors, network, peripherals) to a user's personal devices (smartphone, wearable devices, etc.). But is the countermeasure worth being implemented in practice? We discuss this now.

There are several aspects to take into account. First, the reliability of the attack. In its current form, the attack is probabilistic. Hence to be exploited, it may require a large number of infected devices. The second aspect is monetisation. An attacker may collude with thieves to acquire a victim's device and physically unlock the banking app with the stolen PIN. For this, an attacker would need to recruit a network of thieves: this is not impossible, but it requires effort. A potential problem with the mitigation is that it may break certain apps that need continuous access to resources, e.g. a pedometer app. It is hard to assess how many apps would break and make people unhappy. So unless Google investigates this more carefully, it may be desirable to not patch now: there are easier ways to attack smartphones at present (Section 2). Attackers also tend to bank on reliable techniques rather that probabilistic ones in practice.

This does mean the findings of this chapter should be ignored. In fact, as the saying goes, "attacks only get better". For high-value targets such as politicians for example, we suggest working towards implementing the countermeasure. Often times, attackers exploit a series of smaller vulnerabilities to poke holes in a system.

3.8 Summary

In this chapter, we investigated the feasibility of inferring PINs entered by users with the use of the front video camera and microphone. The orientation of the smartphone during PIN input is extracted from the video stream and correlated to the position of the digit on the touchscreen. We presented the design, implementation and evaluation through an app we developed for the Android platform. We demonstrated that the camera, usually used for conferencing or face recognition, can be used maliciously to infer users' touch events too. Sensors and peripherals are not the only sources of side channels though, as we will illustrate in the next chapter.

Chapter 4

Interrupt-based Side Channel on Android

We present a new side-channel attack against soft keyboards that support gesture typing on Android smartphones. An application without any special permissions can observe the number and timing of the screen hardware interrupts and system-wide software interrupts generated during user input, and analyze this information to make inferences about the text being entered by the user. System-wide information is usually considered less sensitive than app-specific information, but we provide concrete evidence that this may be mistaken. Our attack applies to all Android versions, including Android M where the SELinux policy is tightened.

We present a novel application of a recurrent neural network as our classifier to infer text. We evaluate our attack against the "Google Keyboard" on Nexus 5 phones and use a real-world chat corpus in all our experiments. Our evaluation considers two scenarios. First, we demonstrate that we can correctly detect a set of pre-defined "sentences of interest" (with at least 6 words) with 70% recall and 60% precision. Second, we identify the authors of a set of anonymous messages posted on a messaging board. We find that even if the messages contain the same number of words, we correctly re-identify the author more than 97% of the time for a set of up to 35 sentences.

Our study demonstrates a new way in which system-wide resources can be a threat to user privacy. We investigate the effect of rate limiting as a countermeasure but find that determining a proper rate is error-prone and fails in subtle cases. We conclude that real-time interrupt information should be made inaccessible, perhaps via a tighter SELinux policy in the next Android version.

4.1 Introduction

Around 100 billion mobile apps were downloaded by users in 2014[1]. As users, we expect a certain level of isolation between these apps. For example, we expect that a benign permissionless app such as a weather app cannot read sensitive information entered in a messaging app. In this paper, we show that this basic assumption does not hold on soft keyboards that support "gesture typing". This new mode of acquisition can compromise users' privacy. The "gesture typing" mode has been introduced to improve usability on small-sized touch screen smartphones. In this mode, users swipe their finger from one character to another rather than tap each key individually (Section 4.2). This feature is enabled by default on Samsung and Nexus devices.

The attack leverages supposedly harmless information exposed by the OS to every process on the device, namely the system-wide screen's hardware interrupt counter and the system-wide software interrupt (a.k.a. context switch) counter. Intuitively, when a user interacts with the screen, such as (*i*) touching it or (*ii*) moving the finger on the screen, the Android kernel receives a hardware interrupt from the interrupt controller, which it can act upon to retrieve the current finger location. The number of hardware interrupts leaks some information about what a user types (Section 4.2). Second, a soft-keyboard app must keep track of a user's finger position on the screen in order to infer the word entered. Conceptually, this requires a soft-keyboard app to retrieve information from kernel-land into user-land. This effectively requires performing context switches. The number of context switches (a.k.a. software interrupts) the OS undergoes leaks some information about what a user types (Section 4.2), even when other processes run – our test phones have 200 running processes on average and 60 apps installed (Section 4.3).

The attack monitors the system-wide interrupt counters and uses supervised machine learning to infer information about text entered by users. For this, we borrow techniques from the NLP community; through the use of a sequence model based on a Recurrent Neural Network (RNN) (Section 4.2.3). We evaluate the attack in two different scenarios:

1. **Detection of pre-defined sentences:** Given a set of sentences of interest, we ask if we can detect when a user enters them. This could be used by curious advertising libraries embedded in benign apps to infer personal information entered e.g. in messaging apps. For example, an ad library could detect a search term such as "how to lose weight" into a search engine or messaging app. We are able to correctly detect sentences containing at least 5 words 60% of the time with 55% accuracy (Section 4.3).

2. **User identification:** Given a set of sentences and users, we ask if we can identify which users typed which sentences. This could be used to de-anonymize users of

[1]`http://www.statista.com/statistics/263794/number-of-downloads-from-the-apple-app-store/`

"anonymous" messaging apps such as YikYak. Even if sentences have the same number of words, we correctly re-identify their author 97% of the time (Section 4.3).

To mitigate this attack, we investigate rate limiting, but find it is more cautious to prohibit access to interrupt timing data entirely (Section 4.4).

In summary, the contributions are as follows:

- We present, design and evaluate a new side-channel attack against soft-keyboards that support gesture-typing. These keyboards have been downloaded hundreds of millions of times from Google Play, and they come pre-installed in Samsung and Nexus devices.
- We highlight the limitation of the current SELinux policy in all Android versions, including the latest stock Android M and customized versions used in the Samsung KNOX security container. This allows a permissionless benign app installed on an Android smartphone to breach a user's privacy.
- We propose practical enhancements to the OS platform. After highlighting the imitations of rate limiting, we suggest prohibiting access to interrupt timing data – as well as other global statistical resources – altogether in the next Android version.
- On the scientific front, this is the first work to apply a Recurrent Neural Network (RNN) to a side-channel problem.

4.2 Background and Threat Model

4.2.1 Android Soft-keyboards & Gesture Typing

The Android OS lets users install "keyboard apps"[1] to replace the default soft keyboard. The Android OS allows only one keyboard app to be enabled at any time, and this is configurable by a user. We refer to the keyboard app that is currently enabled as simply "the keyboard app" in the rest of the paper. When an app requires user input (e.g. through displaying an *EditText* Java object), the Android OS launches the keyboard app. This runs in a different process under a different user ID. A user effectively enters text in the keyboard app, which in turn sends it back to the callee app via IPC (APIs are standard and defined by the Android framework). The keyboard app can be used to provide new features, such as encryption [226] or novel input methods which are the focus of this paper.

Over the years, keyboards with a "gesture typing" mode have emerged to ease user input on small-sized touch-screen devices. At the time of writing, two keyboard apps with

[1]Technically, these keyboard apps are built on top of the Android Input Method Editor (IME) API.

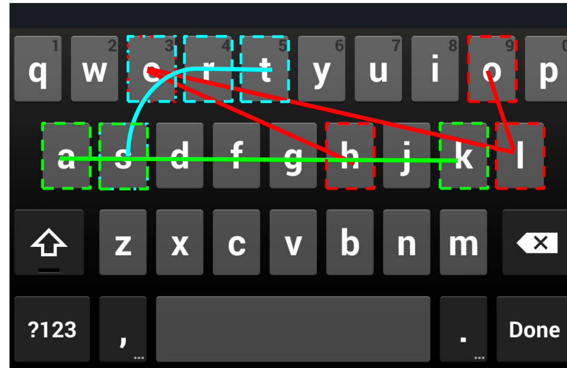


Figure 4.1: Path of finger during input of word “hello” (red), “ask” (green) and “très” (light blue).

gesture typing are prevalent. The Swiftkey app (10M-100M downloads) is the default keyboard on Samsung devices, used by both “untrusted” apps and apps running within the Samsung enterprise KNOX container. The Google Keyboard app (100-500M downloads) now comes pre-installed in newer Android devices.

Gesture typing is a mode whereby a user slides their finger from one letter to another without lifting the finger off the screen. Fig. 4.1 shows the “path” that a user’s finger would typically follow to enter the word “hello” in the keyboard (red trace). First, a user positions their finger on the letter “h” on the screen, then drags the finger to the letter “e”, ’l’ and then ’o’, at which point he lifts the finger off the screen. Each subsequence ($\vec{he}$, $\vec{el}$, $\vec{lo}$) can be represented as a vector. When the finger transitions from one subsection (e.g. $\vec{he}$) to the next (e.g. $\vec{el}$), the finger typically changes direction. Fig. 4.1 also shows an exception to this – the word “ask” (green trace).

When a user lifts their finger off the screen, the keyboard app interprets it as the end of a word and the “space” character is automatically added to the text. The process is repeated for each word in a sentence. The keyword app keeps track of the position of the finger on the screen and infers the most likely word entered by a user. This is fairly accurate in practice. The keyboard app is however limited by the dictionary of words it knows, that is, it never outputs misspelled words, unknown abbreviations or slang words. If a user really wants to enter words that are not recognized by the app (e.g. “lol”), he must add them to the “personal dictionary” section of the phone Settings.

4.2.2 Android & procfs

The Android OS is built on top of Linux. Its security model is based on the concept of *application sandboxes*. Prior to Android 4.3, application sandboxes were implemented on top of Linux discretionary access control (DAC). Upon installation, an Android app was given a unique user ID (UID) and ran with the privileges of that user every time it was started. The application-layer permission model relied on this application sandbox.

A permission (e.g. "Camera") was generally mapped to a dedicated Linux group (e.g. the "permission" group). Permissions had to be declared by app developers in the AndroidManifest.xml file, and were used to restrict access to system resources at run time. To these mechanisms, Android 4.3 adds the use of Mandatory Access Control (MAC) through SELinux. In practice, the SELinux policy is not as tight as one might expect, as we shall see shortly.

From Linux, Android inherits the *proc filesystem (procfs)*, a virtual filesystem that provides aggregated information about the system as well as detailed information about processes. Android also adds new entries within the procfs. The procfs information can help app developers during troubleshooting, and also provide useful information for which there is no Android API. Process-specific information is generally accessible under */proc/[PID]/** and */proc/pid_stat/[UID]/**, where *PID* is the process ID and UID the unique user ID. The security implications of process-specific information have been demonstrated in various papers [186, 187, 227] (Section 2.6 in Chapter 2). For example, Zhou *et al.* [186] show how traffic volume information gleaned through the file */proc/uid_stat/[UID]/tcp_snd* and */proc/uid_stat/[UID]/tcp_snd* can be used to fingerprint the Twitter app traffic and identify a Twitter user. Such attacks worked before Android M because the SELinux security policy was too loose, i.e. certain process-specific files remained readable by any app on a device.

In Android M however, the SELinux security policy was tightened up to fix this, in that an app can no longer access another process's specific files in procfs. This appeared to stop one app attacking another by relying on process-specific information. However, we decided to study the details more carefully. For example, what are the implications of exposing the file */proc/interrupts* which contains real-time interrupt counters received from peripheral? What are the security implications of exposing the file */proc/stat* that contains an aggregated software interrupt (a.k.a. context switch) counter? As we shall see, they open substantial side channels with very real security and privacy implications.

4.2.3 Attack Overview

The threat model we consider is a non-malicious but curious app running on the victim's device. This app does not require special permissions besides internet access (to send gleaned data to remote attackers) which, from Android M onwards, is automatically granted and non-revocable. This app does not actively attempt to break out of the sandbox; instead it observes and monitors publicly available "events" from the system while a user enters text in a victim app. Specifically, these "events" are the variations of (*i*) the system-wide screen interrupt counter and (*ii*) the system-wide context-switch counter, accessible through the files */proc/interrupts* and */proc/stat* respectively.

For each unique word entered by a user, the system undergoes a series of events that

```
# Samsung Galaxy S Plus with Swiftkey keyboard
$ cat /proc/interrupts
[...]
247:       7489     msmgpio  qt602240-ts

# Samsung Galaxy S3 with Google keyboard
$ cat /proc/interrupts
[...]
387:      31695        0          0  s5p_gpioint  melfas-ts

# Nexus 5 with Google keyboard
$ cat /proc/interrupts
[...]
362:       4016   msmgpio  s3350
```

Figure 4.2: Interrupt of interest in the file /proc/interrupts. The counter is highlighted in red and underlined.

can be used as a “fingerprint” to recognize that word. The challenge is that these events contain noisy data and have low entropy. Therefore, we use supervised machine learning to create a fingerprint. The fingerprint is constructed from training data and is used to infer sentences entered later in victim apps. In certain attack scenarios, training data are not even required (Section 4.3.4). In the general case however, we need a fingerprint, and we use both the screen interrupt counter and the system-wide context switch (a.k.a. software interrupt) counter as described in the following paragraphs.

Screen Interrupt Counter

This is available through the world-readable file */proc/interrupts*. Fig. 4.2 shows the relevant line containing the screen interrupt counter for different phones and keyboards. Fig. 4.3 (top) shows variations of the screen interrupt counter while a user types the word “hello” on a Nexus 5. The first section (denoted as (1)) corresponds to a user positioning their finger on the letter “h” and dragging it to the location of the letter “e”: the interrupt counter increases linearly with the number of CPU cycles. When the number of CPU cycles reaches 6.5, the interrupt counter ceases to increase and remains constant for a short period of time. This corresponds to the user’s finger transitioning from subsection $\vec{he}$ to $\vec{el}$. This transition generally involves the finger (*i*) slowing down, (*ii*) reaching a zero speed, and finally (*iii*) re-accelerating to reach the next letter. While the finger is idle (zero-speed), the screen need not report changes to the OS, so the OS no longer receives

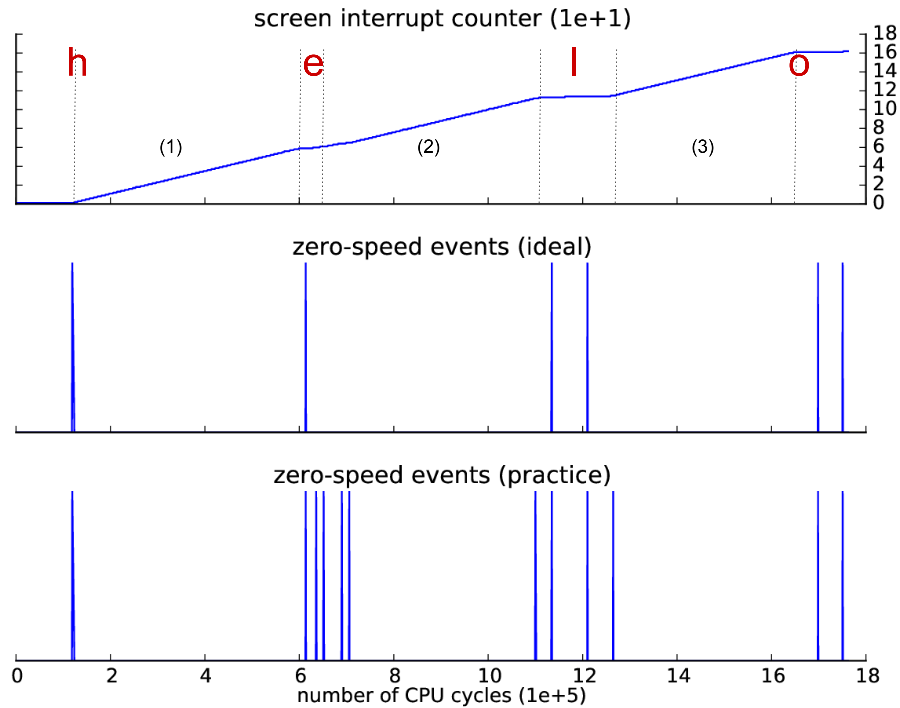


Figure 4.3: Screen interrupt counter (top) for word "hello" on Nexus 5; the "ideal" zero-speed events of the user's finger I would ideally want to infer (middle); the zero-speed events detected in practice (bottom). I assume that the user drags their finger towards the letter "l" once only, hence the single "l". The keyboard app automatically infers the second "l".

screen interrupts. This explains the plateau between each subsection (Fig. 4.3, top).

Fig. 4.3 (middle) depicts zero-speed positions of the finger as inferred by an "ideal" processing routine. These zero-speed events are indicative of the word entered so we use their positions as a feature for word fingerprinting. In practice, zero-speed events often correspond to a change of direction by the user's finger (e.g. to transition from subsequence $\vec{he}$ to $\vec{el}$). Sometimes however, no change of direction is needed to transition between subsequences. This is illustrated in Fig. 4.1 (green trace) when entering the word "ask" – $\langle\vec{as},\vec{sk}\rangle = \|\vec{as}\|\|\vec{sk}\|$ (i.e. $\cos(\theta) = 1$). Certain users still voluntarily slow down their finger around the letter "s", which also creates a zero-speed event we can observe. Note that the absence of zero-speed events does not reduce the effectiveness of the attack, since it is itself indicative of a specific word.

In practice, we may either miss zero-speed events or detect false positive ones. For example, the path of the finger may be a curve rather than a sequence of straight-line vectors. This is often the case if the angle θ between two consecutive subsequences is small, as illustrated in Fig.4.1 (light blue trace; $\vec{tre}$ to $\vec{es}$ correspond to French word "très" which means "very"). For this reason, some changes of direction (and their corresponding zero-speed events) may not be reliably observable. Therefore, in practice, we observe a

```
$ cat /proc/stat
[...]
ctxt 1781433
```

Figure 4.4: Software interrupt in the file /proc/stat.

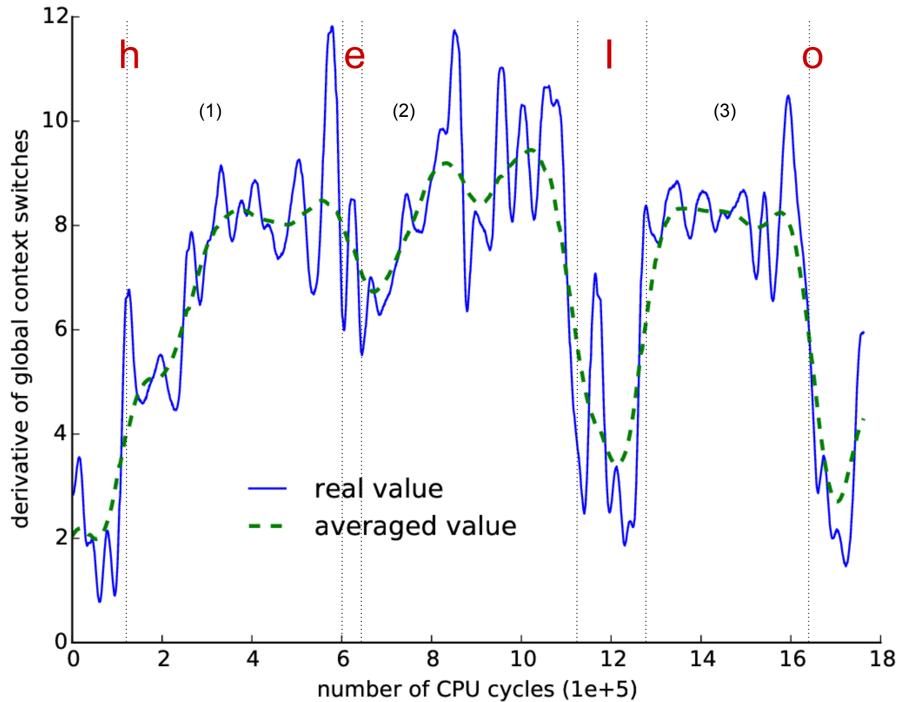


Figure 4.5: Speed of the software interrupt counter during input of word "hello" by a user. I assume that the user drags his finger towards the letter "l" once only, hence the single "l". The keyboard app automatically infers the second "l".

probability distribution of zero-speed events (Fig. 4.3, bottom), with different zero-speed events giving different amounts of information about words. Fig. 4.3 (bottom) illustrates the zero-speed events that the detection routine would typically detect in practice.

Global Context Switch Counter

From here on, we use the terms "context switch" and "software interrupt" interchangeably. The software interrupt counter is accessible through the file */proc/stat*. The relevant line is shown in Fig. 4.4. Unlike the screen's interrupt counter, the line is the same on all devices as it is hardware-independent. We found that its first derivative (i.e. its speed) provides information about text entered in the keyboard. Before computing the derivative of the counter, we first pass it through a Savitzky-Golay smoothing filter [228]. The context switch counter speed corresponding to Fig. 4.3 is shown in Fig. 4.5 (word "hello"). During subsection (1) ($\vec{he}$), the user's finger starts idle at letter "h" (x-axis around 1). Its average

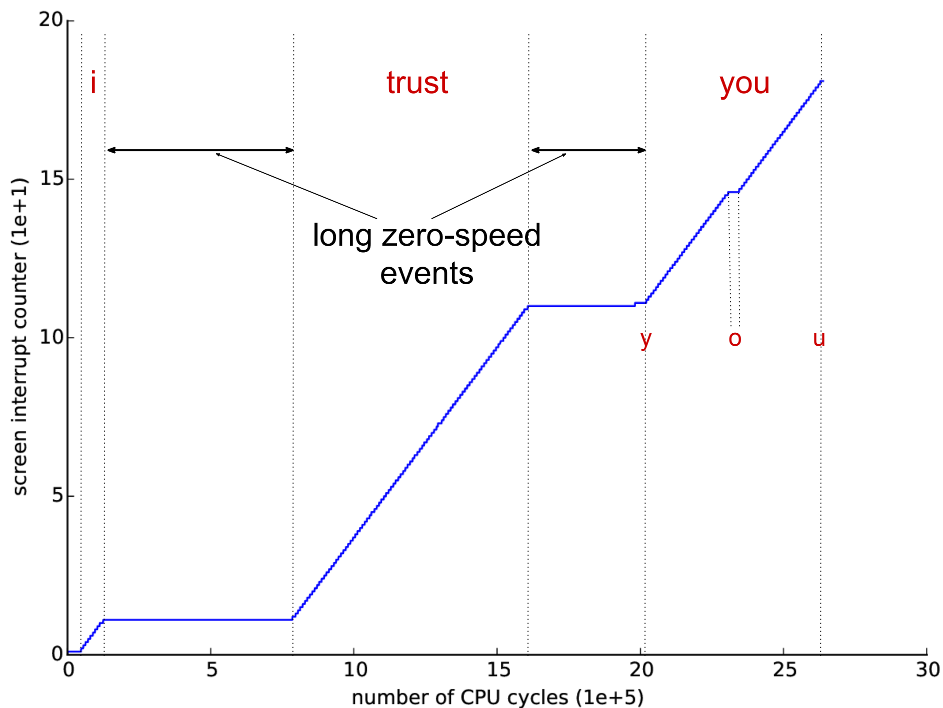


Figure 4.6: Screen's hardware interrupt counter during input of the sentence "I trust you" by a user. Long zero-speed events are used to detect words.

speed then increases until it reaches a local maxima which remains roughly constant on the interval $[4, 6]$. Finally, its speed decreases to a local minima when the finger reaches letter "e" at around 6.5. The same pattern repeats on intervals $[7, 12]$ and $[12, 17]$ corresponding to subsections $\vec{el}$ and $\vec{lo}$ respectively.

These patterns are rather intuitive. As it starts idle, the finger must first accelerate (i.e. its speed increases). Half-way through a subsection (say, $\vec{he}$), its speed starts to decrease until it reaches a local minimum. At this point, a new subsection starts and the pattern repeats itself. The variation of the global switch counter is caused by the keyboard app context-switching into kernel-land to retrieve the finger's current location. The speed of the finger is correlated with the speed at which the context switch counter varies as illustrated in Fig. 4.5. Therefore, this counter carries some information about the entered text, and we use it as an additional feature for word fingerprinting.

It is important to realize that monitoring (i.e. reading) the context switch counter may affect the measurements, since it involves invoking the syscall *read()* which requires a context switch. However, this turns out to have little effect on the attack for the following three reasons:

1. We read the file at almost constant intervals, so the number of context switches we generate is almost constant over time. Since we use the first derivative of the context switch counter, and the derivative of a constant function is zero, the measurement effect is to a first approximation removed;

2. Even if the interval between consecutive reads is not exactly constant, the smoothing filter we use further mitigates any artefacts generated by monitoring;

3. We use the same monitoring routine during the training and attack phase, so any residual artefacts we may add are taken into account when we create the fingerprints.

Sentence Decomposition into Words

A requirement of the attack is the ability for an attacker to chop a sentence (i.e. its corresponding series of "events") into its corresponding words. To this end, we re-use the global screen's interrupt counter. Recall from Section 4.2.1 that the keyboard app detects the end of a word when a user lifts his finger off the screen. While the finger does not touch the screen, there is no activity on the screen to be reported to the OS. Therefore the screen's interrupt counter remains constant. This corresponds to zero-speed events.

However, the zero-speed events induced by lifting off the finger last a lot longer than those caused by transitions between word sub-sequences. This is illustrated in Fig. 4.6 for the sentence "I trust you". We use this heuristics to detect the end and start of words. Through the evaluation (Section 4.3), we find this works more than 95.5% of the time in practice. This allows us to reliably chop a sentence signal into its constituent word signals. These are then passed through a fingerprinting routine as described next.

Supervised Training & Classification

Once we have word signals, we can use them to train a classifier. We first investigated using an SVM to classify words individually, but were not satisfied with the results: for half of users, word predictions were correct less than 10% of the time only. We wanted to find a solution that generalizes better across users. So we decided to explore a solution based on a Recurrent Neural Network (RNN). This can naturally model sequences of arbitrary length and consider contextual-information in a sentence beyond a local context-window. As detailed in the seminal paper by Mikolov *et al.* [229], it can theoretically propagate an *unbounded* history of previous words – we use a history of 5 words in our setup. In this regard, it is also more general than a Markov chain, which assumes that a word only depends on its immediate predecessor.

We use the RNN as a supervised classifier; that is, we train it using labelled examples to minimize classification errors on the training data. But unlike with an SVM, we train the RNN using lists of word signals representing sentences rather then individual word signals. At attack time, we use the trained RNN to classify decomposed word signals from intercepted keyboard swiping. The architecture of the RNN is shown in Fig.4.7(a); it is an Elman recurrent neural network [230] that consists of an input layer x_t, a hidden layer h_t with a recurrent connection to the previous hidden layer h_{t-1} and an output layer y_t.

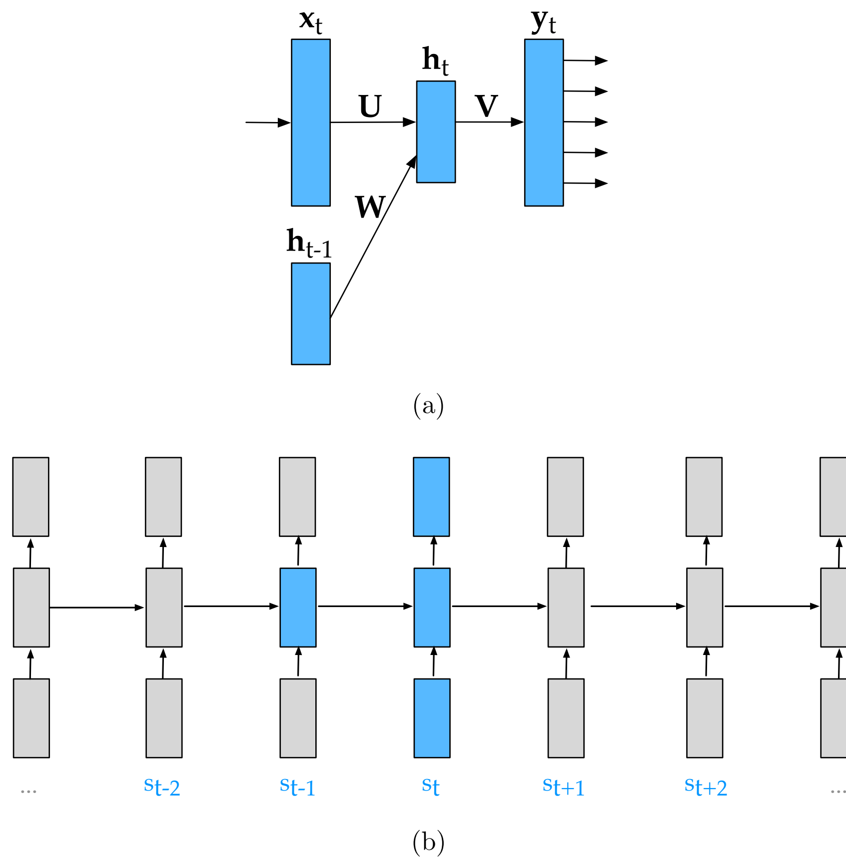


Figure 4.7: The architecture of the recurrent neural network. (a) shows the state of the RNN at any given time step. (b) shows the RNN unfolded across the entire input sequence. A context-window size $k = 5$ is used, and the middle of the context-window is s_t; the bottom layer is the input layer, the middle and top layers are the hidden and output layers, respectively.

The input layer is a real-valued vector representing a context-window of word signals, with the current signal at position t in the middle. The hidden layer h_{t-1} keeps a representation of all a sentence's context history up to the current word signal. The current hidden layer h_t is computed using the current input x_t and hidden layer h_{t-1} from the previous position. The output layer represents probability scores of all possible words, with the size of the output layer being equal to the size of the vocabulary set.

To train the RNN, we feed to it all sentence signals in the training data one at a time, where each sentence is represented as a list of decomposed word signals. Moreover, each word signal has a corresponding ground-truth word label from the vocabulary set. The goal of training is to make the RNN as accurate as possible at predicting ground-truth words according to some loss function, which measures the classification error of the RNN on the training data.

Concretely, let $S_i = s_0, s_1, \ldots, s_n$ be a list of word signals for sentence i in the training data, and $W_i = w_1, w_2, \ldots, w_n$ be the ground-truth words for S_i. To train the RNN on

(S_i, W_i), it reads all the signals in S_i in a left-to-right manner, and at each position t such that $0 \leq t \leq n$, the input x_t fed into the network is:

$$x_t = [s_{t-\lfloor k/2 \rfloor}; \ldots s_t; \ldots; s_{t+\lfloor k/2 \rfloor}], \tag{4.1}$$

where the right-hand side is the concatenation of all signals in a size k context window (we use $k = 5$ in all the evaluations). As the RNN moves across the input signals in S_i, it keeps a representation of all previously seen signals in its hidden layer up to the current step t, and it uses the values stored in h_{t-1} plus x_t to make a new prediction. Fig. 4.7(b) shows the RNN unfolded over an entire input sequence.

The RNN is trained with a cross-entropy objective, so does maximum-likelihood estimation over the training data. We use the backpropgation-through-time algorithm [231, 232] and stochastic gradient descent to minimize the cross-entropy error:

$$L(\Theta) = -\sum_i \log p_i + \frac{\lambda}{2} \|\Theta\|^2, \tag{4.2}$$

where the second part is an $\mathcal{L}_2$ regularization term to prevent over-fitting and λ is a regularization constant (I use $\lambda = 10^{-5}$); Θ is the parameterization of the network and consists of three matrices which are learned during supervised training[1]. Matrix $\mathbf{U}$ contains weights between the input and hidden layers, $\mathbf{V}$ contains weights between the hidden and output layers, and $\mathbf{W}$ contains weights between the previous hidden layer and the current hidden layer. Minimizing the loss in Eq. 4.2 maximizes the probabilities of desired output in the training data and minimizes the probabilities of incorrect output.

To make a prediction, the following recurrence[2] is used to compute the hidden layer activations at input position t:

$$h_t = f(x_t\mathbf{U} + h_{t-1}\mathbf{W}), \tag{4.3}$$

where f is a non-linear activation function; here we use the sigmoid function $f(z) = \frac{1}{1+e^{-z}}$. The output activations are calculated as:

$$y_t = g(h_t\mathbf{V}), \tag{4.4}$$

where g is the softmax activation function $g(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$ that squeezes raw output activations into a probability distribution. The probability scores at the output layer represent the probability of a word given all previous words, $p(w_t \mid w_{t-1}, w_{t-2}, \ldots, w_0)$, and are used in the attack scenarios that we describe in later sections.

[1]Note that the matrices are carried over across all predictions, and it is not the case that three new matrices are created for each new prediction. After each prediction, the values in these matrices are updated by backpropagation.

[2]I assume the input to any layer is a row vector unless otherwise stated.

Training Phase

In the general case, the attack requires a preliminary "training phase" during which we build a set of word fingerprints using the RNN. Concretely, as word fingerprint features, we use the following information for each word:

- The length (in CPU cycles) that it spawns. This is extracted from the system-wide screen's interrupt counter.
- The location of its zero-speed events. This is also extracted from the system-wide screen's interrupt counter.
- A stream of discretized values (sampled at regular time intervals) of the first derivative of the system-wide context switch counter.

In certain attack scenarios, the training phase is not even needed (Section 4.3.4). When it is needed, this phase requires users to enter lists of words in the keyboard app while another app collects the corresponding signals (i.e. the counters). These are used to train the RNN classifier and create the fingerprint set. Obviously, this phase requires knowledge of the words that are entered by users in order to map them to their corresponding signal. Currently, we build the fingerprint set (i.e. a training model) for each user in order to evaluate the efficiency of the attack (Section 4.3). In practice, the training phase could be performed by apps that receive enough genuine user input. For example, a repressive state might require citizens to use approved apps extensively for everything from tax returns to school homework, and seek to use the fingerprints generated from them to identify people who sent subversive messages via encrypted messaging apps provided by firms in other countries who were not prepared to respond to demands for monitoring or decryption. And perhaps eventually it could be possible to eliminate the need for per-user training by building the fingerprint set with enough users – such that the resulting fingerprint set works for most users. In this study, however, we focus on evaluating the feasibility of the attack rather than scaling it.

Attack Phase

In the attack phase, a malicious permissionless app records the counters from procfs while a user enters text in another victim app on the phone. Note that the attack could also be performed by a "normal" app against a "secure" app running in a KNOX container. Unlike in the training phase, the malicious app can only observe the signals (i.e. the counters) but not the words themselves. Using the fingerprint set at its disposal, it matches the observed signals against this set using the RNN – through the scores ouput by the output layer. More specifically, for a given sentence signal, the RNN outputs, for each word signal in a

sentence signal, a probability that the word signal corresponds to a particular dictionary word.

The malicious app must first determine when to start collecting the signals. When it detects that the user interacts with a "screen view" of interest (e.g. the conversation screen of WhatsApp where users enter chat messages), it starts signal collection. The means of detecting the current "screen view" (a.k.a. "Activity" in Android parlance) are not a contribution of this study. They have been demonstrated by previous work[1,2,3] [227]. User input is indicated by a signal similar to Fig. 4.6, i.e. with consecutive screen activity periods interleaved with short/long zero-speed events. The initial finger tap to pop the keyboard and the last finger tap corresponding to the "Send" button provide the attacker with further cues, which can be used to find the start and end of target input.

4.3 Evaluation

4.3.1 Methodology

Corpus

We use the NPS Internet Chatroom Conversations corpus (Release 1.0) [233] available through the NLTK framework [234]. It consists of around 10000 English sentences gathered from age-specific chat rooms of various online chat services in October and November 2006. Within the corpus, we restrict ourselves to the most common 200 words, to which we refer as the "dictionary" from here on. The choice of 200 words strikes a balance between testing our techniques and the burden we put on study participants; by entering dictionary words 15min every day, it took no less than three weeks to collect these samples for each participant.

Participants

We recruited participants through word-of-mouth among acquaintances. We went for this option so we could meet them regularly if need be. When improvements to the software were suggested by participants, we could patch our software and deploy it rapidly. We had 8 participants in our study, three females and five males. Two of them used the gesture feature on their own phone. Their age was between 25 and 40 years old. Three of them have a Computer Science degree, while the others have backgrounds in Psychology, Criminology, Biology, Electronics, and Telecommunications. All have at least a Bachelor's

[1] `https://gist.github.com/jaredrummler/07a3f723e96ec06fb761`

[2] `https://developer.android.com/reference/android/app/ActivityManager.html#getRunningTasks%28int%29`

[3] `http://www.modzero.ch/modlog/archives/2015/04/01/android_apps_in_sheeps_clothing/`

degree, and five have a PhD. Five are native English speakers. These demographics are not representative of the general population. However, our study is radically different from behavioural studies where demographics play a significant role. In this study, we are only interested in simple characteristics, such as finger speed, when people enter words in gesture-based keyboards. This is a pilot study and in any case we believe the results will generalize (as we will discuss later).

Before the experiment, we asked participants if they used gesture typing on their phone. Those who did were told they could start immediately. The rest were asked to familiarize themselves with it and only start once they felt at ease with it. That typically took a few hours or days. We did this because we wanted to assess our attack on people who actually know how to use the feature. If we tested beginners who drag their finger slowly from one letter to another, it could create a bias in our favour, and artificially improve the efficiency of our attack: recall from Section 4.2.3 that the slower a user's finger, the easier it is to detect "zero-speed" events reliably to build a word fingerprint. After this preliminary requirement, we assumed that participants' typing characteristics did not vary significantly over the experiment period. So we did not retrain users over the course of the experiment. If changes in typing characteristics were a concern, one could update the model with the most recent data. The task that participants completed is described next.

Data Collection

We built a proof-of-concept (PoC) victim app and malicious app to run side-by-side on the Android platform. These apps run in different processes under different UIDs. Therefore they belong to different sandboxes, as would be the case in practice (Section 4.2.2). We gave a Nexus 5 (OS ≥ 4.4) to the participants we recruited for the study.

Each participant enters lists of dictionary words in the victim app while the malicious app runs in the background and collects signals (i.e. the counters) from the files */proc/interrupts* and */proc/stat*. In the list of words entered by participants, each dictionary word appears 20 times, resulting in 4000 ($20 * 200$) word samples. On average, these samples represent 40MB per user when zip-compressed. We discuss how a curious app could upload this data stealthily to a remote server in Section 4.6. It takes three weeks for each participant to complete the data-collection task. The phones given to participants have about 60 apps installed on them, and an average of 200 processes running (as reported by the *ps* command). WiFi is enabled at all times. During the course of the experiment, participants witnessed the Android OS downloading updates; news apps regularly pushing articles; and games showing notifications. Our malicious app only monitors the PoC victim app, for ethical reasons.

Fingerprint Creation & Testing

These steps are run on a desktop in our evaluation (we discuss the feasibility of doing them on phones in Section 4.6). Signals collected from participants correspond to lists of words, so first, we chop each signal into its constituent word signals, using the heuristics presented in Section 4.2.3. This works over 99.5% of the time in practice. Once we have individual word signals, we randomly pick 85% of them as the training set, i.e. to create the corresponding word fingerprint. The other 15% are used as the testing set, that is, as unknown input we attempt to predict. For both the training and testing sets, we combine word signals to construct sentences in the corpus. These sentence signals are then used as input to the RNN (either for training or prediction). A trained model needs between 1.1 and 1.3MB worth of data when compressed, and up to 3MB without compression.

4.3.2 Word Prediction

We first want to understand how well words are inferred within a sentence. In this scenario, the RNN takes as input unknown sentence signals from users (i.e. from the testing set), and ranks each dictionary word in order of likelihood for each constituent word signals. The word that appears in the first position is the most likely word entered by the user given the signal, while the one that appears last is the least. For evaluation, we use all the word signals in our testing set. Fig. 4.8 shows the position of the correct word in the ranked list. About 34% of the time, the correct word appears in first position (first bin of the histogram); this is $\approx$ 68 times better than a random guess ($p_{random_guess} = \frac{1}{200} = 0.5\%$). The correct word appears in second position 9% of the time (second bin of histogram), etc. Fig. 4.9 shows the cumulative distribution of the position of the correct word in the ranked list. About 80% of the time, the correct word appears in the first 22 positions. Of course, there were some variations among participants, but the results did not indicate a correlation between users who had previously used swipe keyboards, and those who had not.

4.3.3 Detection of Sentences of Interest

Given a set of pre-defined sentences of interest (e.g. “I’m pregnant” or “I want to lose weight”), we ask if we can efficiently detect if a user enters them. A practical attack could be a school looking for pupils who sent messages bullying other students; parents trying to monitor the topic of discussion of their kids on social media; or just a curious app peeking at text entered by a user in the Google search bar. For the evaluation, we randomly select a set of sentences from our chat corpus. A sentence of interest (“SoI” from herein) need not span an entire sentence though; it may only be a subset of a longer sentence. For example, for the SoI “I’m pregnant”, we want to detect it within longer sentences such as

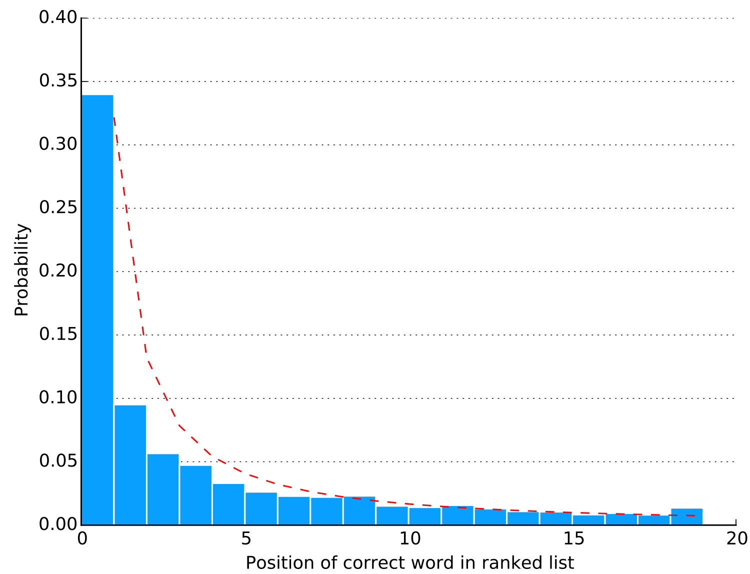


Figure 4.8: Distribution of the position of correct word guess.

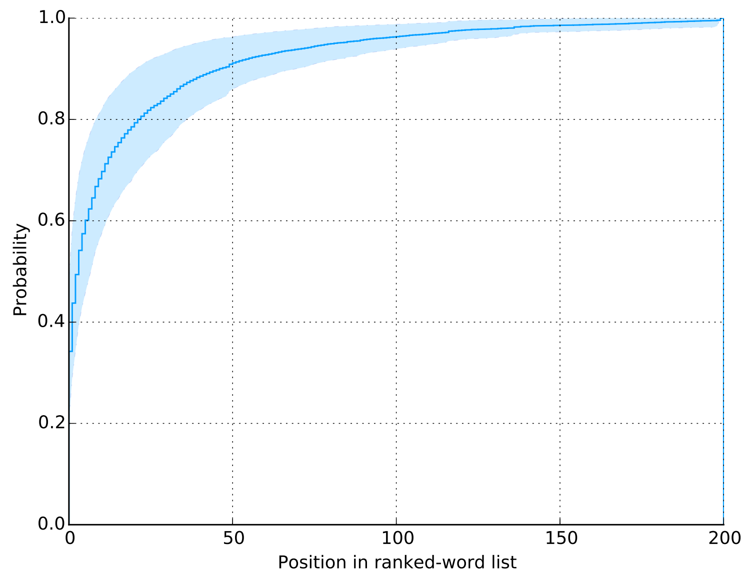


Figure 4.9: Cumulative distribution of the position of correct word guess.

"I think I'm pregnant too". Our detection routine outputs a match if, for each word in an SoI, the word appears in the first N positions in the ranked list output by the RNN. For the evaluation, we vary the parameter N.

Fig. 4.10 shows the True-Positive-Rate (TPR) vs. False-Positive-Rate (FPR), a.k.a. the ROC curve for SoIs. For SoIs containing at least 4 words, we correctly detect them 50% of the time with a False Positive Rate (FPR) close to 0. The results are similar for SoIs containing at least 5 and 6 words. An important metric missing from the ROC plot is the precision of our matching algorithm, that is, *when we output a match, how often are we correct?* Fig. 4.11 answers this question. For SoIs containing at least 4 words, we correctly

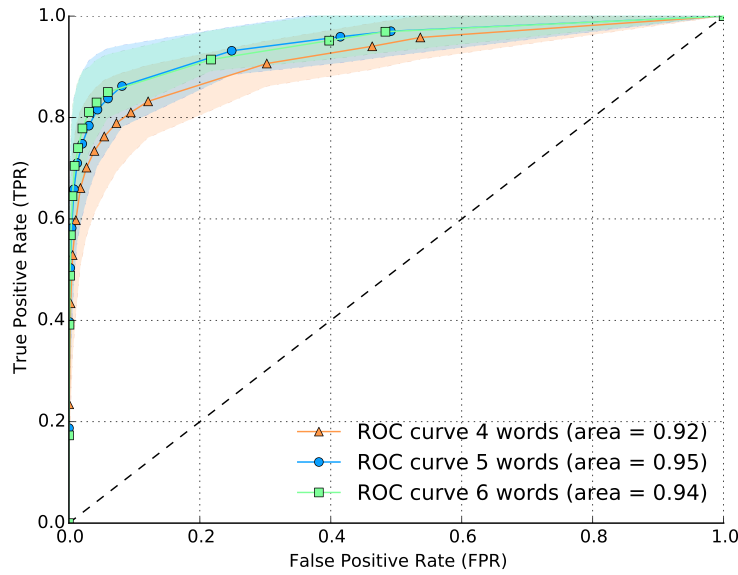


Figure 4.10: TPR-FPR curve (ROC) of known sentence detection. The shadow area represents the standard deviation for sentences containing at least 4 words.

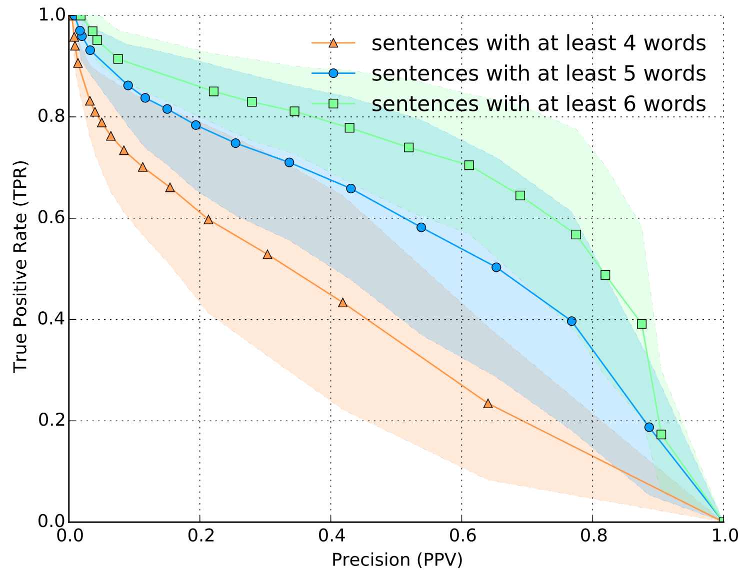


Figure 4.11: Precision-Recall curve of known sentence detection.

detect them 50% of the time ($TPR = 0.5$); and when we output a match we are correct 35% of the time ($PPV = 0.35$). This corresponds to a False Positive Rate (FPR) near 0 on the ROC curve of Fig. 4.10. For SoIs containing at least 5 words, the results improve: we correctly detect them 60% of the time ($TPR = 0.6$); when we output a match, we are correct 55% ($PPV = 0.55$). This corresponds to a False Positive Rate below 0.5% on the ROC curve. Intuitively, as the number of words in a sentence increases, we have more information to distinguish sentences. Therefore, for sentences with at least 6 words, results further improve to a $TPR = 0.7$ and $PPV = 0.6$ corresponding to a $FPR \leq 0.05$ on the ROC curve.

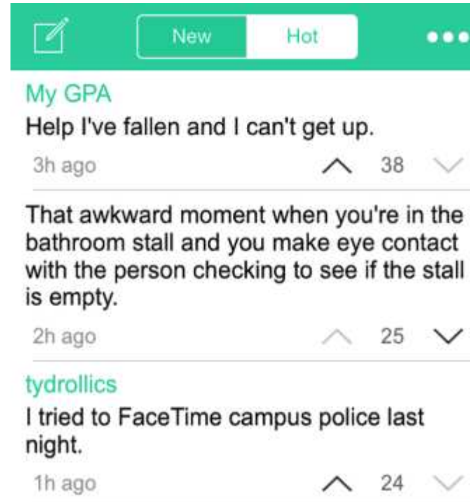


Figure 4.12: YikYak messaging board example.

4.3.4 De-Anonymization of Users

Given a list of known sentences entered by a set of users, we ask if we can efficiently map each sentence to the user that entered it. As per our threat model outlined in Section 4.2.3, we assume that each user has our curious app running on their device. A practical attack scenario could be to identify users of "anonymous" messaging board apps such as YikYak messenger[1], which has more than 1M downloads on Google Play. Such apps let users write "anonymous" posts on a messaging board. A school could try to find which pupil posted an inappropriate message. YikYak messaging boards are arranged by location: posts are visible to all users in the vicinity of the sender. Posts are anonymous in the sense that they do not contain a name, pseudonym or location data that would link them to their author (Fig. 4.12). Posts do contain a time, but this is not very precise. During a time period T, U users may post anonymously to the board. Assuming these U users are infected by a curious app, we ask if it can identify which user entered which post. We first used the YikYak app ourselves to see how many messages were posted on average over time. We found that within one minute, no more than a dozen messages were posted in our area.

Let N be the number of sentences posted during a time period T on an anonymous messaging board. We denote a sentence as $Sen_{i,1\leq i\leq N}$. On each user's device, the curious app observes a signal $Sig_{i,1\leq i\leq N}$ as described in Section 4.2.3. Since there are N posts, there are exactly N signals that correspond to them. Note that certain users could be the author of multiple posts, that is, the number of users $U \leq N$.

[1]http://www.yikyakapp.com/

Sentences of different lengths

If sentences on the messaging board each contain a different number of words, it becomes straightforward to map them to their corresponding signal Sig_i by simply counting the number of words in each Sig_i. As detailed in Section 4.2.3, we detect the number of words contained in a signal by counting "long" zero-speed events in the signal. Once we have the number of words entered by each user, we just map these to the length of sentences on the messaging board. No matter how many words each sentence contains, so long as each of them contains a different number of words, we can identify their author virtually all the time. The only condition is that we manage to properly count the number of words in a sentence, and our experiment reveals this works over 99.5% of the time in practice. Interestingly, in this attack, we neither need the user training phase nor the fingerprint. To make the task challenging, we study the case where all sentences have the same length.

Sentences with same length

This is the worst-case scenario for the attacker. Let L be the number of words in all sentences posted on the messaging board. The first step of our re-identification routine is to compute, for every signal Sig_i and every sentence Sen_i, a score that represents the likelihood that the signal Sig_i corresponds to Sen_i. Recall from Section 4.2.3 that our fingerprint routine outputs, for any of the L word signals $Sig_i[k], 1 \leq k \leq L$ in a sentence Sig_i and a dictionary word DW, the probability that $Sig_i[k]$ corresponds to DW. We define the score $score_{i,j}$ for sentence Sen_i and signal Sig_j as:

$$score_{i,j} \stackrel{\text{def}}{=} \sum_{k=1}^{L} log(proba(Sig_i[k] == Sen_j[k])), \tag{4.5}$$

where $Sen_i[k]$ is the kth word of Sen_i, $Sig_j[k]$ is the kth word-signal of Sig_j, and $proba(\mathsf{WS} == \mathsf{DW})$ is the probability that word-signal WS corresponds to dictionary word DW, as output by the RNN.

We then build a square "score matrix" where each row i represents a signal, each column j represents a sentence, and each element in the matrix is the score $score_{i,j}$ ($sc_{i,j}$) as illustrated in Fig. 4.13.

Sentence Prediction. We first evaluate how well we can infer the correct sentence given a sentence signal. Specifically, for each signal Sig_i (i.e. for each row i in the score matrix), we rank each sentence score ($sc_{i,j,1 \leq j \leq N}$) in increasing order. That is, the first sentence in the ranked list corresponds to the sentence with the highest score, and the last with the lowest score. Fig. 4.14 shows the position of the correct sentence in the ranked list for a set of $N = 35$ sentences. Recall that in practice, about a dozen messages are posted every minute. We nevertheless raise the bar to up to $N = 35$ messages for our

$$\begin{pmatrix} sc_{1,1} & sc_{1,2} & \cdots & sc_{1,N-1} & sc_{1,N} \\ & & \cdots & & \\ & & \cdots & & \\ \cdots & \cdots & sc_{i,j} & \cdots & \cdots \\ & & \cdots & & \\ & & \cdots & & \\ sc_{N,1} & sc_{N,2} & \cdots & sc_{N,N-1} & sc_{N,N} \end{pmatrix}$$

Figure 4.13: Score matrix.

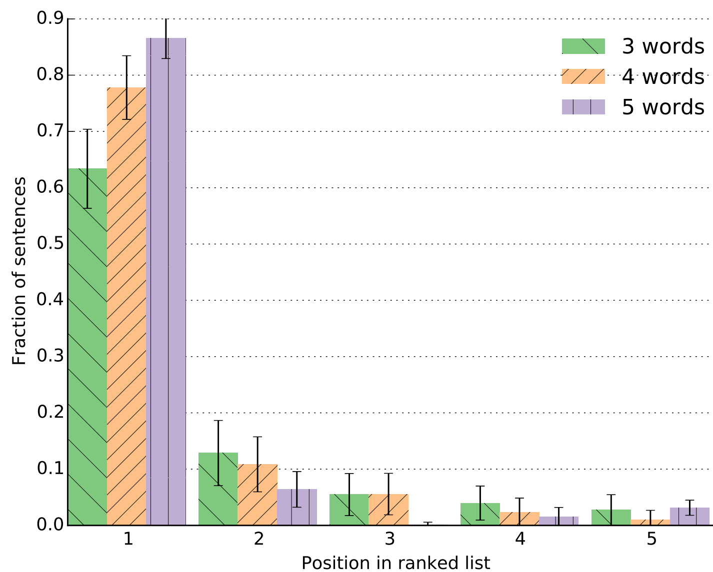


Figure 4.14: Position of correct sentence in ranked list (35 sentences).

evaluation. For sentences containing 3 words, the correct sentence appears in first position about 63% of the time. This increases to 77% and 86% for sentences containing 4 and 5 words respectively.

Naive Re-Identification Algorithm. Given a signal Sig_i and its corresponding list of scores $sc_{i,j,1\leq j\leq N}$, a simple solution to de-anonymize users is to select the top score in the list. We call this solution the naive solution. In our experimental setup, we randomly select N sentences each containing L words. Then we run the naive algorithm. We repeat this 200 times and average the results. These are presented in Fig. 4.15. For example, for sentences containing 5 words and for a set of 35 sentences, we correctly re-identify their author 86% of the time. This is consistent with the results of Fig. 4.14. In order to increase readability, and since the mean error between individual runs was always below 10%, we omit error bars. As the number of sentences in the set decreases, the results improve: for a set of 10 sentences each containing 5 words, we reach 92% de-anonymization. Intuitively, the more words a sentence contains, the more information we have about it. Therefore, as the number of words increases in a sentence, the re-identification improves (Fig. 4.15). We

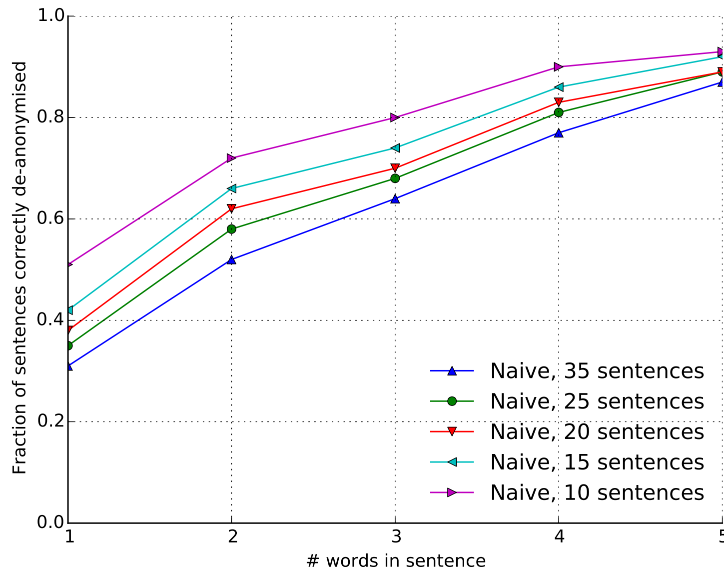


Figure 4.15: De-anonymization of sentences using the "naive" method.

next show how to improve these results significantly.

Optimal Re-Identification Algorithm. Naive re-identification is not optimal, so here we describe a better method. Our goal is to maximize the sum of the scores when selecting sentences corresponding to signals. Looking back at the score matrix (Fig. 4.13), this means our goal is to select a set of optimal $scores_{i,j}$. Since each sentence corresponds to a single signal, each row and column must have exactly one score selected; and the sum of the selected scores must be optimal.

Practically speaking, this means there are $N!$ possible assignments to test. For $N = 20$ sentences, this means more than $10^{18} \approx 2^{60}$ candidates; and for $N = 35$ sentences, this means $10^{40} \approx 2^{130}$ candidates. This is greater than the strength of a 1024-bit RSA key (2^{80}). Fortunately, our problem is equivalent to the so-called "assignment problem" [235] for which there exist solutions that run linearly in the size of the input, i.e. in $\mathcal{O}(N)$. More specifically, we use the *Munkres* algorithm [235] that runs in $\mathcal{O}(N)$ with $\mathcal{O}(N^2)$ space requirements.

As in the naive method, we randomly select N sentences each containing L words and run the *Munkres* algorithm. We repeat this 200 times and average the results. These are presented in Fig. 4.16. For a set of 35 sentences each containing 4 words, we correctly guess the author of more than 92% of the signals (it is around 77% with the naive method). Regardless of the set size, we correctly guess the author of more than 97% of the signals when sentences contain 5 words.

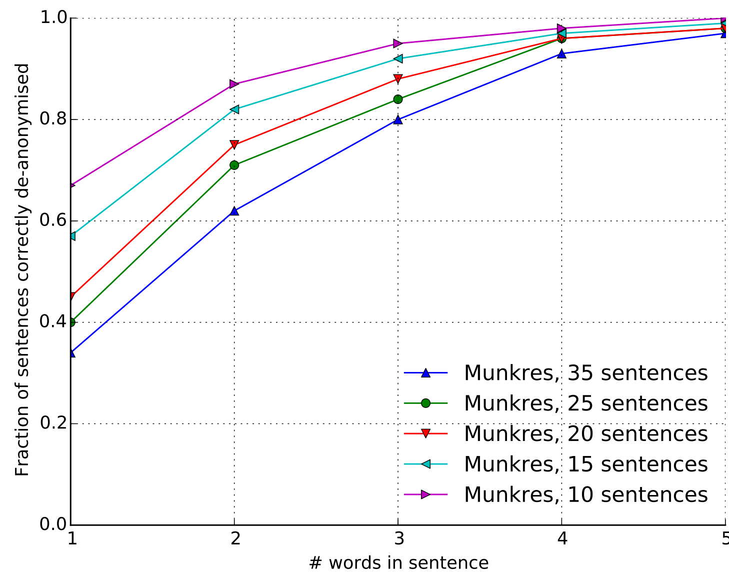


Figure 4.16: De-anonymization of sentences using "Munkres algorithm".

4.4 Countermeasures

Store/Cloud-based

Cloud-level mitigation techniques such as Google's Bouncer can be used to vet apps before they reach end users' devices. In addition to static and dynamic code analysis, Google can use the large amount of data it stores about developers, such as their activity across Google services, whether their IP has ever been associated with suspicious activity, and so on. This is a powerful tool. However, we have no visibility about what checks Google performs, and they are not infallible: from time to time some malicious apps manage to evade them.

App-level

At the app level, we are limited. An app cannot disable gesture typing from the default keyboard app. However Android allows arbitrary apps to include their own custom keyboard layout/code through the *KeyboardView* API. This way an app could provide its own keyboard without the gesture typing feature. However, re-implementing a keyboard can be tedious, and removing gesture typing could greatly inconvenience users who have grown accustomed to it.

Zhang *et al.* [101] suggest killing apps that may be collecting side-channel information in the background while the foreground app performs sensitive tasks. This provides some level of protection without changing the OS or the apps being protected. However it comes with several caveats. First, their approach relies on the assumption that a malicious app must monitor resources at high frequency to be successful. But Michalevsky *et al.* [181]

show that high sampling is not always necessary: by sampling the power consumption once a second, they can infer the route driven by a user (we discuss subsampling in the case of our attack later). Second, their techniques only protect foreground apps, not background processes. Third, they rely on monitoring app-specific procfs files; these are no longer accessible in Android M.

We conclude that app-level countermeasures are fragile and limited, so we investigate OS-level countermeasures next.

OS-level

OS-level countermeasures are more reliable since the OS can enforce a global policy that an app cannot. On Android, there have been inconsistencies between what resources are available through the framework APIs vs. those available through virtual files. The framework APIs enforce the permission model but the same is not always true for virtual files – certain permissions can be bypassed. For example, the virtual file */proc/net/arp* exposes the BSSID (i.e. the MAC address) of the WiFi Access Point a phone is currently connected to. This allows a curious app on the phone to find the location of a user's phone without requiring location permissions [186]. There are other pieces of information available through app-specific and global files in the virtual file system *procfs* (as well as */sys*). These represent the main source of leaks and inconsistencies that break the permission model. Therefore, we advocate restoring consistency, that is, we advocate prohibiting access to any virtual files (except those "owned" by the requesting app), perhaps through a stricter SELinux policy. Recall from Section 4.2.2 that the SELinux policy still allows access to certain global virtual files as of Android M. Of course, denying access to global virtual files could break apps that rely on them. In practice, we think this should affect only a very small number of apps, if any, as global virtual files exposed by *procfs* only provide admin-like information for troubleshooting, rather than relevant information for mobile apps. A trade-off could be to allow users to toggle this feature on and off for certain apps through an additional option within the "Developer" menu in phone Settings.

When we responsibly disclosed this work to Google, it became clear that they worried that protecting global procfs entries could break some utility apps. So might it be possible to have the OS rate-limit virtual file access, rather than prohibit it entirely? We study this for both attack scenarios in the following sections.

Rate-Limiting to Prevent the Detection of SoIs. Recall from Section 4.3.3 that in this scenario, an attacker has a pre-defined set of sentences of interest (SoIs), and wants to detect when these are entered by a user. Our current attack relies on the ability of an attacker to chop the sentence signal into its constituent word signals. For this, we used the zero-speed events extracted from the screen's interrupt counter (Section 4.2.3). If all

word signals look "enough" like the words of an SoI, we output a match. Therefore one way to defend against our current implementation is to make it infeasible for an attacker to correctly infer the number of words entered by a user. Note however that this may not thwart more advanced attacks that build a fingerprint based on the entire sentence signal rather that its constituent word signals.

Fig. 4.17 shows the effect that subsampling (i.e. rate limiting) has on our detection routine. With a reduction of the sampling rate by 2 ("2 subs"), we correctly detect the number of words in an 8-word sentence about 80% of the time only. This drops down to 10% for a 10-fold reduction, which corresponds to about 10ms on a Nexus 5. Therefore a 10-15ms rate-limiting policy appears to already provide good security.

However, looking back at our data, we found that we could improve our original word-splitting routine to use the software interrupt rather than the screen hardware interrupt. Fig. 4.18 illustrates the effect that subsampling has on the first derivative of the software interrupt counter. The top signal shows the original signal corresponding to a 4-word sentence. Even with a 500-fold reduction of the sampling rate, the number of words is till clearly visible. With a 1000-fold reduction, the detection becomes unreliable, and appears impossible with a 3000-fold reduction. Fig. 4.19 shows the number of words detected by our new routine subjected to subsampling (for all samples collected from users). Even with a 200-fold sampling rate reduction, more than 60% of the time we correctly detect the number of words (8). As we further reduce the sampling rate, the number of words detected moves towards zero. But even with a 3000-fold reduction of the sampling rate, we detect the presence of one word (50% of the time) rather than no word at all. A rate limit of 1.4-3s ("1400 subs"-"3000 subs" in Fig. 4.19) thwarts our attack on SoI detection, since with such sampling rates we never correctly detect the number of words in a sentence. Of course, this only defeats our current implementation, and it would be more prudent to prohibit access to virtual files entirely as suggested earlier.

Rate-Limiting to Prevent the Re-Identification of Users. In this scenario, we have a list of sentences posted on an "anonymous" messaging board by a set of users. We try to determine which user entered which sentence. We have extensively studied the re-identification of users when sentences contain the same number of words, as this is the worst-case scenario for an attacker (Section 4.3.4). If we apply the 1.4s rate limiting policy as for SoI detection, an attacker can no longer detect the number of words reliably, so this seems to thwart re-identification attacks on sentences with the same number of words. Recall that without the right number of words, we cannot extract where words start and end in the signal stream, as a result of which we cannot extract the features necessary for fingerprint. The 1.4s rate, however, is still not enough if sentences contain a different number of words. As we mentioned in Section 4.3, in this case an attacker need not train on users, but only detect the number of words. Let us consider 2 signals Sig_1 and Sig_2 corresponding to 2 sentences Sen_1 and Sen_2 containing $L_1 = 3$ and $L_2 = 8$

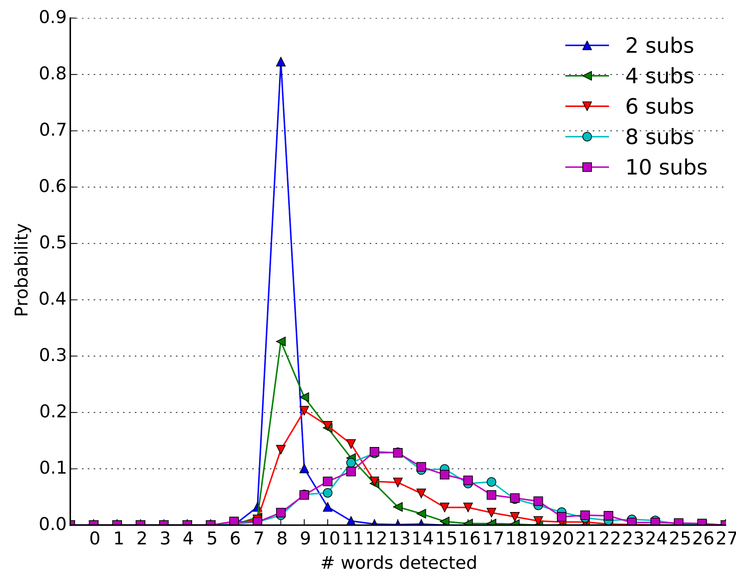


Figure 4.17: Distribution of the number of words detected using the screen interrupt counter, for sentences containing 8 words.

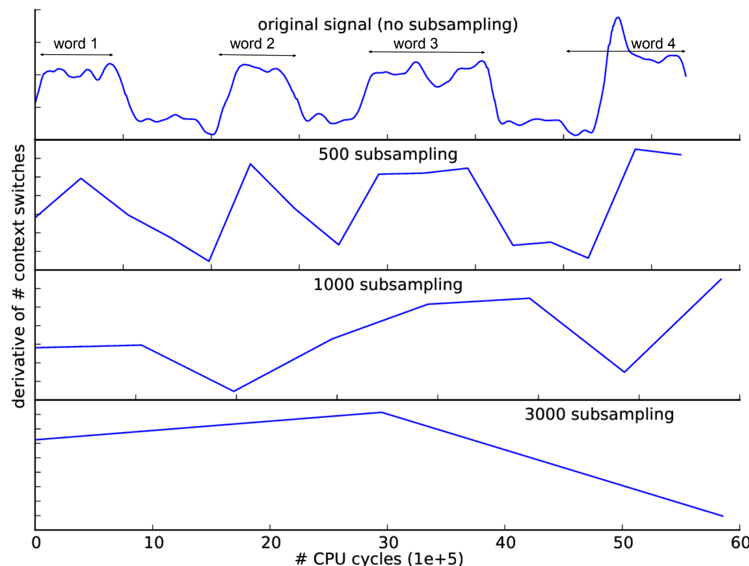


Figure 4.18: Effect of subsampling on the first derivative of the number of software interrupts measured by a malicious app.

words respectively. Even if an attacker cannot reliably infer the number of words in each $Sign_i$, she may be able to re-identify users solely based on the estimated number of words detected. Fig. 4.20 shows the number of words detected for sentences containing 3 and 8 words respectively, when subjected to subsampling. For example, for a 1000-fold reduction of sampling rate, if an attacker detects 5 words, she is sure the signal corresponds to Sen_2 containing 8 words, since our routine never detects more than 3 words for a signal corresponding to a 3-word sentence. So let us consider the following re-identification heuristics:

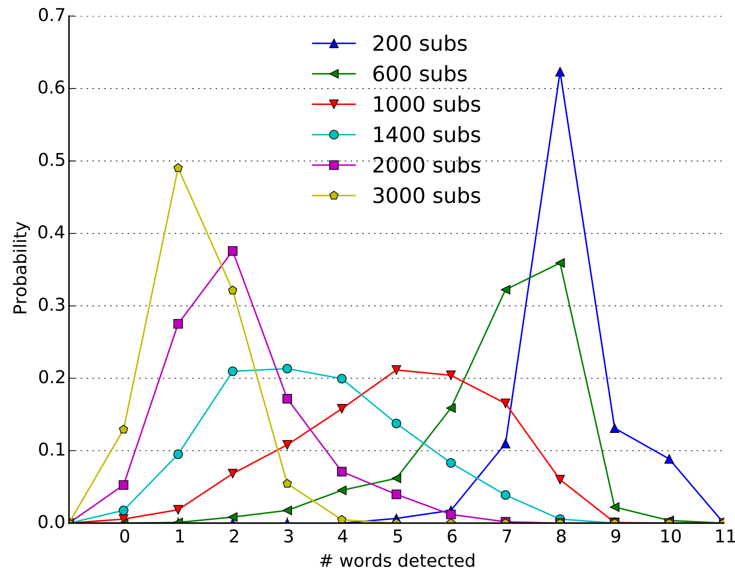


Figure 4.19: Distribution of the number of words detected using the software interrupt counter, for sentences containing 8 words.

$$Sig_i = \begin{cases} Sen_1, & \text{if } N_{detectedWord} \leq N_{cutt-off}, \\ Sen_2, & \text{otherwise.} \end{cases} \tag{4.6}$$

where $N_{cutt-off} = 3$ and $N_{cutt-off} = 0$ for 1000-fold reduction and 3000-fold reduction of sampling rate respectively. This allows an attacker to correctly re-identify users 84% and 43% of the time respectively. In other words, even a rate limit of 3s is not enough to thwart re-identification attacks that are based on the number of words and, in effect, on phrase length. We conclude that finding the right rate that thwarts all attacks – including those still unknown to us – is nontrivial. It is more prudent to simply prohibit access to virtual files as we first suggested.

4.5 To Patch or Not to Patch

In the previous section, we concluded that the OS should not expose the interrupt files to user-installed apps. More generally, in an attempt to prevent future side channels, we think it would be desirable to protect all procfs files. But is the countermeasure worth being implemented in practice? We discuss this now.

As for the attack presented in Chapter 3, the sucess of the interrupt-based side channel is probabilistic. However, an attacker need not physical access to a device to exploit it, and a malicious app suffices. This suggests this vulnerability should be closed before attacks become smarter and stealthier. One question that remains is the extent to which the mitigation may break utility apps. As far as hardware/software interrupts are concerned,

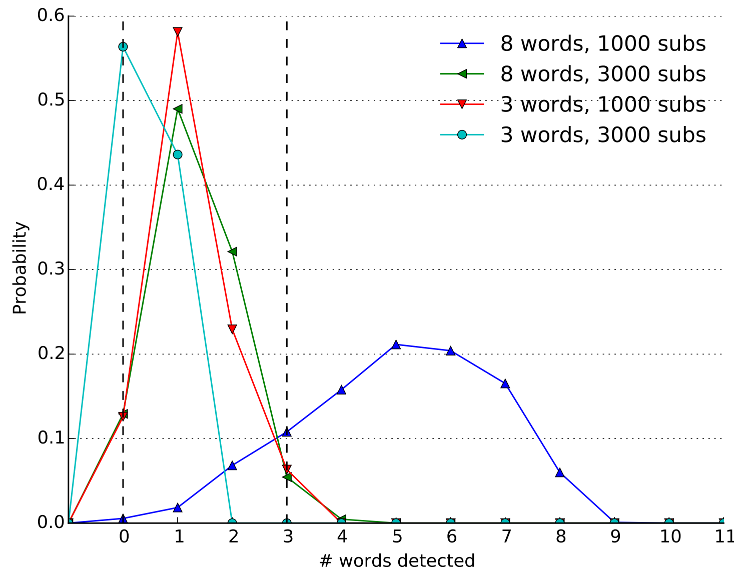


Figure 4.20: Distribution of the number of words detected using the software interrupt counter, for sentences containing 8 and 3 words.

we doubt they are useful to utility apps. So we suggest the countermeasure be implemented. As for other procfs files, there may be a higher chance that removing them breaks certain apps. So further investigation should assess which procfs files are indeed necessary. Once these files are identified, Google could create relevant APIs and permissions to provide the same functions. Permissions make triage of potentially malicious apps feasible. More expensive resources (e.g. dynamic or manual analysis) can then be dedicated to vet these apps.

4.6 Discussion

The dictionary set we used was limited by experimental constraints (Section 4.3). In terms of dictionary size, the practical limiting factor was the burden put on study participants. We were unable to test our techniques for larger set sizes. This is an area that will no doubt attract more work in the future.

We also attempted to recover arbitrary words. However, this turned out to be challenging. We investigated language models in combination with the RNN, but did not achieve satisfying results. One major issue was the lack of a large chat dataset to create a language model. We think our corpus was too small. We thought of using other sources of chat data such as Twitter, but were put off by the restrictions on the use of the data. In fact, Twitter now blocks large-scale downloads entirely to prevent data mining.

For now, we have demonstrated that aggregated interrupt counters consitute a threat. We are confident that our results could be improved in two ways. First, as noted above,

a larger chat corpus would help an attacker develop good language models. These are typically trained over millions of entries, while our corpus had only a few thousand. It would be very valuable to understand the full implications of such attacks under ideal conditions.

Second, we believe fine tuning the classifier parameters could also improve our results. More data from users could help to take advantage of deep learning capabilities of the RNN. Again, artifical neural networks excel at understanding complex data relationship through millions of samples, but we trained ours with less than 20 occurrences of each word for each user. The restricted number of participants made it difficult to assess the feasibility of creating a "master" model that works for most users to avoid the per-user training phase. Realisticallly, this may require hundreds or even thousands of users. It was out of scope of this paper, since we focused on piloting the attack, rather than scaling it. Consequently, with our current implementation, predictions made through a model trained on one user simply do not work for another. But the history of HCI suggests that, with enough users, we can move from user-dependent recognition to user-independent operation.

As well as scaling the attack across users, it may be worth while trying to scale the attack across devices. We demonstrated it on the Nexus 5, but we are confident it will generalize; we have looked at various phone models, and their counter streams all show similar properties during user input. However, different phones have different hardware and software characteristics. So could we build a model by training users on one phone, and predict text entered on a different one? This is another topic for further research.

As detailed in Section 4.3, our current implementation collects data on phones, but data pre-processing, fingerprint creation and prediction are performed offline on a desktop. But the processing power of current smartphone is not a barrier; we could do all the data processing locally. Model creation on a standard desktop takes a few minutes at most with code written in python. We believe the same computation would be feasible on a smartphone if we reimplemented it in C, albeit with a slight performance hit. The prediction phase is a lot faster and takes at most a few seconds on our desktop. Software that can collect data and build a model locally opens up the prospect that a malware writer could collect data at scale and use it for the methodological improvements described here, including building a better language model and developing user-independent recognition techniques.

Stealthy malware must blend in its environment in terms of energy consumption, network activity, and data storage. Energy consumption during data collection would be negligeable because the curious app monitors input only when needed. There is no busy loop constantly executing to drain the battery. Where energy consumption could become an issue is during model training if this was done on the phone. To be stealthy, malware could do this in steps, rather than all at once, e.g. by spreading the computation over

several days, or doing it only when the phone is plugged in to a power socket. In terms of storage, models take a bit over 1MB at rest (i.e. compressed) and about 3MB when used (uncompressed). Data collected for the training phase is about 40MB when compressed (Section 4.3). Although this is not small, neither is it big enough to make users suspicious. In the case where data processing is offloaded to a server, network activity should also be camouflaged, e.g. by uploading data over several days, or only when in wifi.

We use phones with around 60 apps for our evaluation. The apps generate noise for the counters we monitor. Our results show that under normal conditions, this noise is negligible. We also investigated our attack under heavy load while the browser was downloading a 100MB file. The download spanned the entire user input. Under this condition, our attack did not work. So our current implementation is very sensitive to ambient noise. But this does not mean that all such attacks will be. Even if the side channel turns out to be inherently sensitive to high loads, users generally only interact with one app at a time on smartphones. So maybe most of the time the system is under low load and the attack remains feasible, or maybe the curious app can just discard data when heavy load is detected.

Another interesting question is how we could combine sensor-based side channels with our attack. This is something we have not attempted yet. Another kind of attack that may become possible through monitoring interrupt counters is inference of what users type on normal keyboards. Attacks based on keystroke dynamics would benefit from the interrupt-based side channel described in this paper. The screen's hardware interrupt counter may also be used to infer other user activities on the screen. It could also have been used, for example, by PIN Skimmer [236] – where the authors used the microphone to detect user taps.

In short, the attack we have presented in this paper is really just an early prototype, and can probably be improved and extended in all sorts of interesting ways. Rather than waiting for this to happen, it may be prudent to tackle the problem now.

4.7 Summary

We presented a novel attack against Android soft keyboards that support gesture typing. By monitoring the number of screen's hardware interrupts and aggregated software interrupts during user input, we show that it is possible to breach users' privacy, both by identifying some sentences a target user typed and by identifying which user typed some incriminating sentence.

To the best of our knowledge, this work is the first to leverage global information exposed by procfs. This goes against the general belief that non-app-specific information exposed through virtual files is harmless. The attack also applies to the latest Android version where app-specific virtual files are inaccessible.

We investigated the efficiency of rate limiting as a countermeasure, but found that determining a proper rate limit is nontrivial and fails in subtle use cases. Therefore we advocate removing access to global virtual files in the next Android version.

Chapter 5

Security Analysis of Android Factory Resets

We study the implementation of Factory Reset on 21 Android smartphones from 5 vendors running Android versions v2.3.x to v4.3. We estimate that up to 500M devices may not properly sanitise their data partition where credentials and other sensitive data are stored, and up to 630M may not properly sanitise the internal SD card where multimedia files are generally saved. We found we could recover Google credentials on all devices presenting a flawed Factory Reset.

Full-disk encryption (FDE) has the potential to mitigate the problem, but we found that a flawed Factory Reset leaves behind enough data for the encryption key to be recovered. We discuss practical improvements for Google and vendors to mitigate these risks in the future. Recent Android versions try to take advantage of ARM's TrustZone technology to secure encryption keys. This was not the case at the time we did the work.

The work presented in this chapter was published in the 4th Workshop on Mobile Security Technologies (MoST) [237], and is in collaboration with Ross Anderson. I developed tools and scripts, and performed the evaluation across all devices. Ross helped with the writing and reviewing of the final paper. This work was also presented at the 2015's BlackHat Mobile Security Summit in London [238].

5.1 Introduction

The extraction of data from resold devices is a growing threat as more users buy second-hand devices[1] to amortise the purchase of a new device. Trade press reports[2] have already

[1] http://blogs.which.co.uk/technology/phones-3/mobile-phone-price-tracking/

[2] http://blog.avast.com/2014/07/08/tens-of-thousands-of-americans-sell-themselves-online-every-day

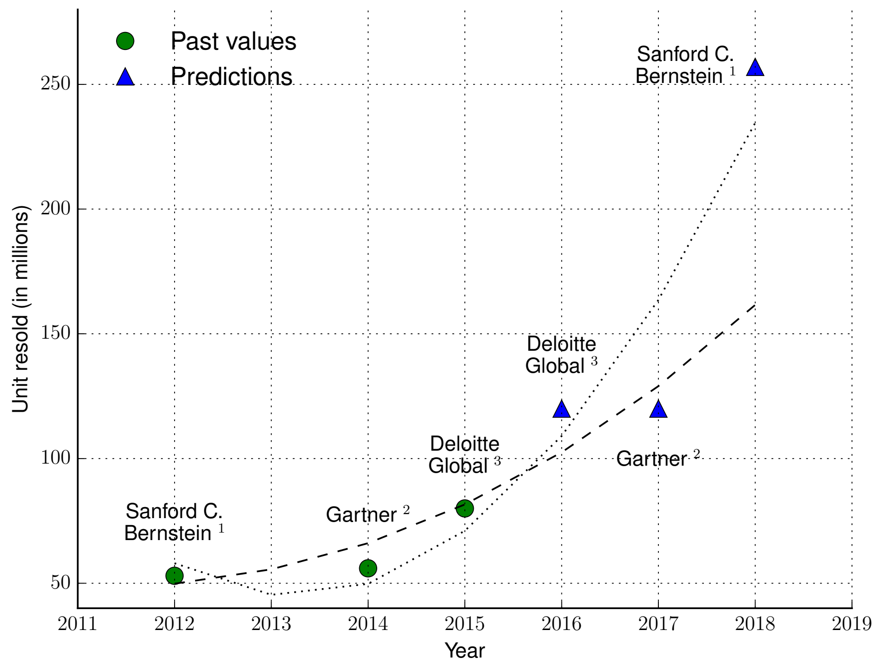


Figure 5.1: Market size of user smartphones.

raised doubts about the effectiveness of Android "Factory Reset", but this chapter presents the first comprehensive study of the issue.

According to a survey by Sanford C. Bernstein[1] in 2012, half of the US smartphone users said they would trade in their device the next time they upgrade. Overall, different surveys tend to agree on the current size of the used smartphone market (Fig 5.1). Sanford C. Bernstein estimate that 3 percent (53M) of phones were resold in 2012. Gartner[2] estimate that 56M smartphones were resold in 2014. Deloitte Global[3] estimate that 80M devices were resold worldwide in 2015. Nationwide, Deloitte India[4] expect 13 percent of Indian users to trade in their used phone in 2016. It is hard to validate these numbers as there is usually little information on how the studies were conducted. In fact, predictions for 2017-2018 vary greatly across studies (Fig 5.1). But it is not unreasonable to expect over 100M devices resold in 2018: this would represent below 5 percent of the wordwide shipment. So this does not represent a significant fraction of the overall market. However, in terms of numbers, if an ill-intentioned group of people were to get their hands on a fraction of these devices, then large-scale attacks may become feasible. What is a determining factor for this to happen? We discuss this next.

[1] http://www.forbes.com/sites/connieguglielmo/2013/08/07/used-smartphone-market-poised-to-explode-apple-iphone-holding-up-better-than-samsung-galaxy/

[2] https://www.gartner.com/newsroom/id/2986617

[3] http://www2.deloitte.com/au/en/pages/technology-media-and-telecommunications/articles/tmt-predictions-2016-telecomm-used-smartphones-17-billion-market.html

[4] http://dazeinfo.com/2016/02/01/used-smartphone-market-growth-india-usa-uk-apple-iphone/

Some surveys qualitatively assess what happens to used phones. For example, Deloitte India[1] estimate that two third of resold devices are resold to individuals, whereas the other third is traded to carriers and manufacturers. Deloite UK estimate that 20 percent of all phones are given/sold to a family member[2]. This cannot lead to large scale attacks since there is a limit to the number of relatives someone has. The same study estimates that 12 percent of devices are resold online. Groups of individuals could buy in bulk from online websites to scale attacks. But as we discuss in Section 5.4.3, we think this is difficult in practice. About eight percent of devices are traded in with carriers, manufacturers, or retail shops. There is little data about what happens to devices that these organisations collect. Usesd phones may be recycled, destroyed, or resold. Regardless of this, rogue employees in these organisations may get access to a sufficient quantity of devices to scale their attack. According to Sanford C. Bernstein[3], in 2012 the majority of used phones were collected in the US and resold to emerging markets, in particular China. Employees of the organisations in charge of the devices in the origin/destination countries could also mine devices in mass. So in effect, a determining factor for large scale attacks is the ability to gather a large number of devices, not what happens to the devices next.

Data sanitisation problems have the potential to disrupt market growth. If users fear for their data, they may stop trading their old devices, and buy fewer new ones; or they may continue to upgrade, but be reluctant to adopt sensitive services like banking or healthcare apps, thereby slowing down innovation. Last but not least, phone vendors may be held accountable under consumer protection or data protection laws.

To sanitise their devices, users are advised to use the built-in "Factory Reset" function on device disposal. Previous reports [239] have raised occasional doubts about the effectiveness of the implementations of this in Android, with claims that data can sometimes be recovered. This work provides the first comprehensive study of the problem. It (*i*) quantifies the amount of data left behind by flawed implementations, (*ii*) provides a detailed analysis of affected devices (versions, vendors), (*iii*) reveals the drivers behind the flaws, and (*iv*) discusses practical solutions to mitigate these problems (Section 5.3 and Section 5.6).

Concretely, we have found that a flawed Factory Reset lets an attacker access a user's Google account and its associated data backed up by Google services, such as contacts and WiFi credentials (Section 5.4). The study unveils five critical failures: (*i*) the lack of Android support for proper deletion of the data partition in v2.3.x devices; (*ii*) the incompleteness of upgrades pushed to flawed devices by vendors; (*iii*) the lack of driver support for proper deletion shipped by vendors in newer devices (e.g. on v4.[1,2,3]); (*iv*)

[1] `http://dazeinfo.com/2016/02/01/used-smartphone-market-growth-india-usa-uk-apple-iphone/`

[2] `https://www.deloitte.co.uk/mobileuk/assets/pdf/Deloitte-Mobile-Consumer-2015.pdf`

[3] `http://www.forbes.com/sites/connieguglielmo/2013/08/07/used-smartphone-market-poised-to-explode-apple-iphone-holding-up-better-than-samsung-galaxy/`

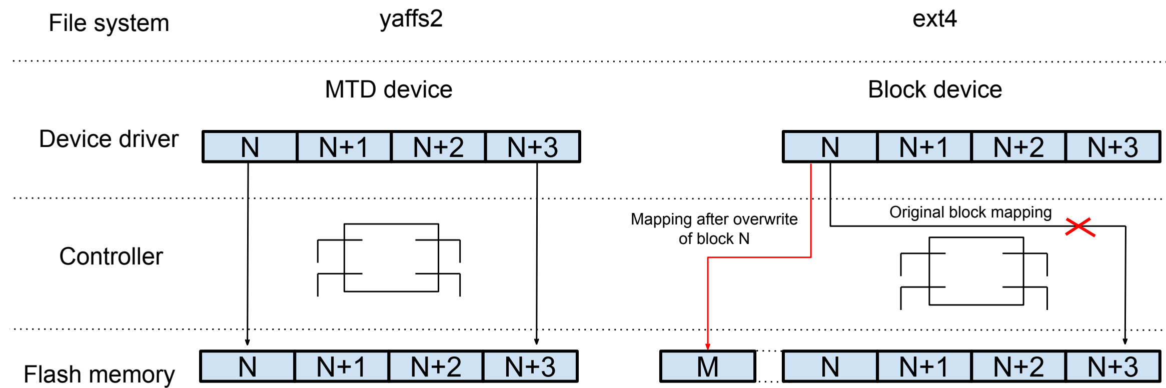


Figure 5.2: Yaffs2 with raw flash access vs. Ext4 with logical block access. On an eMMC (right), the logical block N is mapped to the physical block $N+3$ by the eMMC, and remapped to block M after an overwrite. MTD stands for Memory Technology Device.

the lack of Android support for proper deletion of the internal and external SD card in all OS versions; and (*v*) the fragility of full-disk encryption to mitigate those problems up to Android v4.4 (KitKat).

In summary, the contributions are as follows:

- This is the first comprehensive study of Android Factory Reset. We studied 21 Android smartphones from 5 vendors running Android versions from v2.3.x to v4.3.
- We highlight critical failures such as the lack of support by the Android OS and/or vendor-shipped drivers for secure deletion. These problems may affect devices even after upgrades are received.
- We discuss practical improvements for Google and vendors to mitigate these risks in the future.

5.2 Technical Background

5.2.1 Flash & File Systems

Smartphones use flash for their non-volatile memory storage because it is fast, cheap and small. Flash memory is usually arranged in pages and blocks. The CPU can read or write a page (of typically 512+16 to 4096+128 data+metadata bytes), but can only erase a block of from 32 to 128 pages. Each block contains both data, and "out-of-band" (OOB) data or metadata used for bad block management, error correcting codes (ECC) and file system bookkeeping. Blocks must be erased prior to being written to; yet flash chips support only a finite number of program-erase cycles, so wear-levelling algorithms are

used to spread the erase and write operations uniformly over all blocks. It is also worth mentioning that flash storage is usually over-provisioned, i.e. a chip has more internal space than it advertises to the OS, in anticipation of bad blocks and to further reduce wear.

Early Android devices (like Froyo, Android v2.2.x) used the flash-aware file system yaffs2 that handles wear-levelling and error correction. Since Gingerbread (v2.3.x), devices generally come with an embedded MultiMediaCard (eMMC) with proprietary wear-levelling algorithms implemented in hardware. An eMMC does not give the OS access to the raw flash, but exposes a block-like device, on top of which the OS lays a block file system like ext4. Blocks only give the OS a logical view of memory. Internally, each block is mapped to a corresponding physical block on the flash by the eMMC controller. When data in a logical block N is updated by the OS (Fig. 5.2, right), the corresponding physical block would typically be added to a "to-be-erased" list, then remapped to a "clean" physical block M, thereby leaving the original block content untouched. Therefore, to achieve secure deletion, eMMC standards define specific commands that must be used to physically remove data from memory.

5.2.2 Secure Deletion Levels

When removing a file, an OS typically only deletes its name from a table, rather than deleting its content. The situation is aggravated on flash memory because data update does not occur in place, i.e. data are copied to a new block to preserve performance, reduce the erasure block count, and slow down the wear. Therefore, there exist various recommendations, guidelines, and standards for sanitising data. The following levels of data sanitisation are relevant depending on the threat model considered [210].

The highest level of sanitisation is *analog sanitisation*, this degrades the analog signal that encodes information, so that its reconstruction is impossible even with the most advanced sensing equipment and expertise. For example, NIST's *Guidelines for Media Sanitization* (NIST 800-88) have a "purging" level that corresponds to analog sanitisation.

The second level is *digital sanitisation*. Data in digitally sanitised storage cannot be recovered via any digital means, including the bypass or compromise of the device's controller or firmware, or via undocumented drive commands. Unimpeded physical access to a flash chip and the manufacturer's data sheet may be required, but these are not available for typical smartphones.

The third level is *logical sanitisation*. Data in logically sanitised storage cannot be recovered via standard hardware interfaces like standard eMMC commands. For example, this corresponds to NIST 800-88's "clearing" level. For cellphones and PDAs, NIST 800-88 suggests "clearing" them by manually deleting data followed by a Factory Reset.

In this study, we consider cheap data recovery attacks that require neither expensive equipment to physically extract data from the chip nor specific per-chip knowledge. Only attacks that are oblivious to the underlying chip scale across devices and are likely to be profitable if exploited at scale. Therefore, in the rest of this document, by "proper" or "secure" sanitisation, we mean logical sanitisation.

5.2.3 Linux Kernel Deletion APIs

Privileged userspace programs can erase flash blocks through the *ioctl()* system call exposed by the Linux kernel. On raw flash, the *ioctl*'s *MEMERASE* option provides digital sanitisation. On an eMMC, there are two different options. The first is *BLKDISCARD* which provides no security guarantees. Internally, the kernel generally implements *BLKDISCARD* by passing the eMMC command "DISCARD" or "TRIM" to the chip. These do not request the eMMC to purge the blocks. They simply indicate that data is no longer required so that the eMMC can erase it if necessary during background erase events. They would typically be used when *unlink*ing a file. The second *ioctl* option is *BLK**SEC**DISCARD* and provides "secure" deletion. The kernel implements *BLKSECDISCARD* by passing to the chip one of the many "secure deletion" commands defined by eMMC standards. The actual eMMC command used depends on support by the chip. There is an "ERASE" command for logical sanitisation, and commands such as "SECURE TRIM", "SECURE ERASE", and "SANITIZE" for digital sanitisation.

5.2.4 Data Partitions

Android smartphones share three common partitions for data storage (Fig. 5.3). The first is the data partition, generally mounted on */data/*, that hosts apps' private directories. An app's private directory cannot be read or written to by other apps, so it is commonly used to store sensitive information such as login credentials. On older phones with a small data partition, one can also install apps on an external SD card; but this is usually not the default behaviour.

The second partition storing user data is the internal (primary) SD card. Despite its name, it is not an SD card per se, but a partition physically stored on the same chip. It is generally mounted on */sdcard/* or */mnt/sdcard/*, which is readable/writeable by applications. It is generally FAT-formatted or emulated with the Filesystem in Userspace (FUSE). In the latter case, files are physically stored on the data partition. The internal SD card is mainly used to store multimedia files made with the camera and microphone; it is generally exposed to a computer connected via USB – via Mass Storage, Media Transfer Protocol (MTP) or Picture Transfer Protocol (PTP).

The last partition containing user data is the external, removable SD card. It offers similar functionality to the internal SD card, but can be physically inserted and removed

by the user. If there is no internal SD card on the device, the external one becomes the "primary SD card"; otherwise it is called the "secondary SD card". The primary and secondary SD cards are sometimes referred to as "external storage".

Some devices also have hardware key storage. When supported, it is used principally by the default Account Manager app.

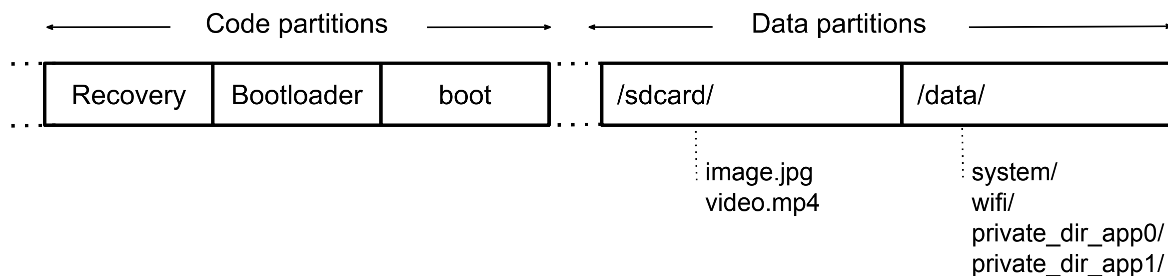


Figure 5.3: Common Android partitions. Each rectangle represents a partition on the same flash storage.

5.3 Analysis of Android Factory Reset

In the rest of the study, we refer to a "Factory Reset" or a "wipe" interchangeably.

5.3.1 Methodology

Between January and May 2014, we bought second-hand Android phones from eBay and from phone recycling companies in the UK, randomly selecting devices based on availability. As the project might possibly uncover personal information, it was first submitted to the university's ethics process for approval. In the rest of the paper, we refer to a "device" as the unique pair (phone name, OS version).

Devices Tested

We studied 26 different devices from 5 vendors[1], running Android versions ranging from v2.2 (Froyo) to v4.3 (Jelly Bean). The list of devices is as follows:

Froyo (v2.2.x): HTC Nexus One, Motorola Defy.

Gingerbread (GB, v2.3.x): Galaxy S Plus, HTC Wildfire S, HTC Desire S, Galaxy S, Galaxy S2, Galaxy ACE, LG Optimus L3, Nexus S.

Ice Cream Sandwich (ICS, v4.0.x): HTC Sensation, Galaxy S3, HTC Desire C, Galaxy S2, LG Optimus L5.

[1]Samsung, HTC, LG, Motorola and Google

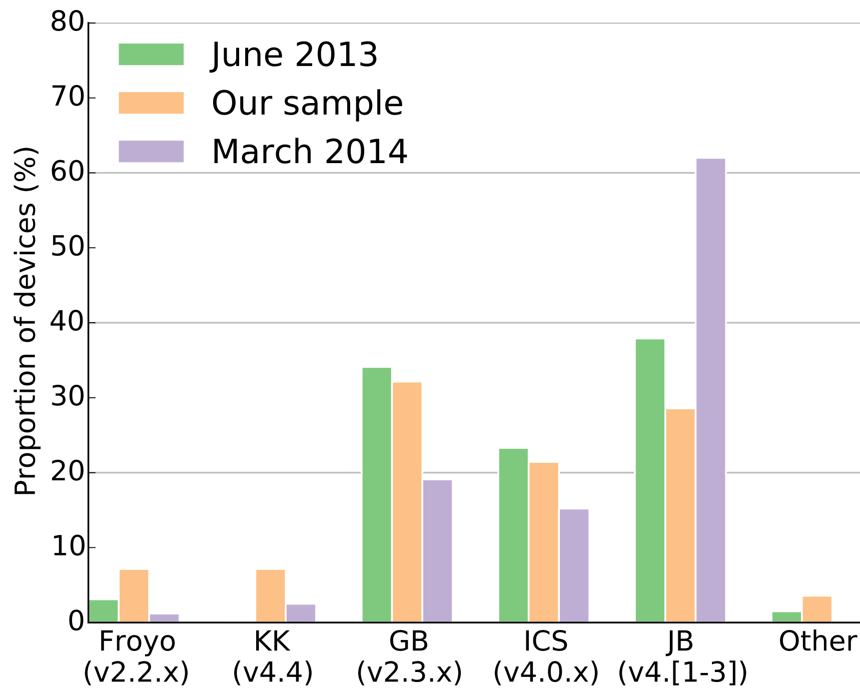


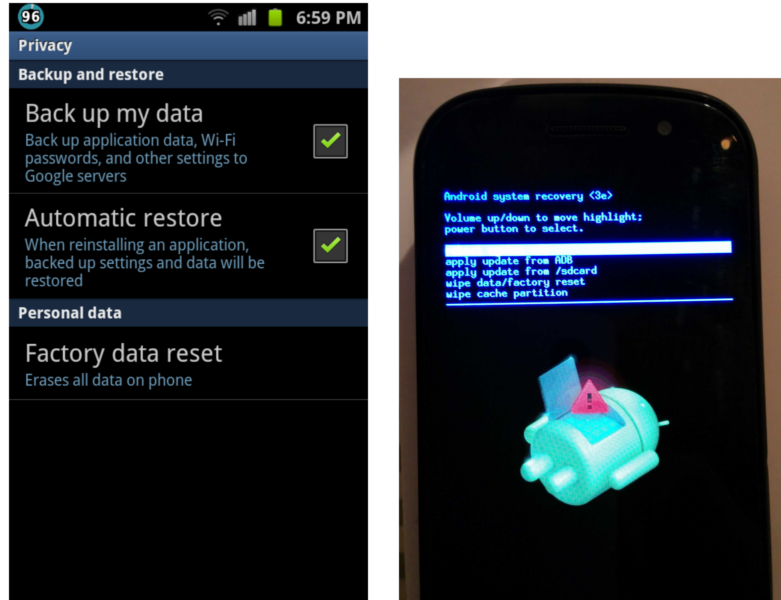
Figure 5.4: Android OS distribution for Froyo, KitKat (KK), Gingerbread (GB), Ice Cream Sandwich (ICS) and Jelly Bean (JB).

Jelly Bean (JB, v4.[1-3]): Nexus 4 (x2), Motorola Razr I, LG Optimus L7, Nexus S, Galaxy Note 1, HTC One S, HTC One X.

Other devices in our sample for which we do not present results (too few devices) include the Motorola Defy (Eclair, v2.1.x), Nexus 4 and Nexus 5 (KitKat, v4.4).

Android versions in the sample were the ones more frequently resold and traded at the time of the study. Fig. 5.4 shows the distribution of Android versions for our samples, compared to active devices in June 2013 and in March 2014, as reported by Google's Dashboard[1]. The samples are not representative of the OS version distribution at the time of acquisition, but are similar to the world-wide distribution 6 months earlier, in June 2013 (as one might expect from the time taken for new phones to enter the second-hand market). In September 2013, Google announced that one billion devices had been activated[2]. This represents 340M Gingerbread (GB, v2.3.x) devices, 230M Ice Cream Sandwich (ICS, v4.0.x) devices and 380M Jelly Bean (JB, v4.[1-3]) devices. The samples are representative of the second-hand market at the time of acquisition. We use the number of devices and their distribution across versions (in June 13) to approximate the number of devices affected by flawed Factory Resets throughout the following sections. Estimates are based on the assumption that each device accounts for the same percentage of the overall device population. This is not true in practice, and this is a limitation of the evaluation.

[1] `developer.android.com/about/dashboards/index.html`
[2] `plus.google.com/+SundarPichai/posts/NeBW7AjT1QM`



(a) Factory Reset in Settings. (b) Factory Reset in Recovery.

Figure 5.5: Factory Resets provided by most devices.

Pattern writing and validation

We tested each partition of interest (Section 5.2.4) by overwriting it with unique identifying patterns. Then we sanitised the device with the Factory Reset function and attempted to recover the written patterns. We investigated the Factory Reset suggested by vendors[1], that is, the one in Android Settings (Fig. 5.5(a) – recommended by 90% of vendors) and the one in Recovery/Bootloader mode (Fig. 5.5(b) – recommended by 70% of vendors if the phone cannot be booted). The Recovery and Bootloader modes are special modes a phone can be booted into via a combination of hardware keys. More specifically, we used the following steps to test each partition of interest:

Root Access. To be able to write to a partition bit-by-bit, we first needed low-level access to the flash storage. In the case of yaffs2, this meant access to the raw flash, while for an eMMC it meant access to the logical blocks. Android does not give such low-level access to apps. Rather, one needs to "root" the device. We achieved this with known root exploits or by booting a custom Recovery – in the latter we would backup the stock Recovery first. Previous work [240] suggested loading custom code via the Bootloader without requiring root access within the Android OS. However this only works on a handful of devices today.

Writing Patterns. We wrote identifying patterns on the entire partition. Each pattern was $1/4^{th}$ the device block size. For external storage, we wrote the patterns with the Android OS. For the data partition, devices would sometimes crash and reboot: in

[1] Alcatel, BlackBerry, Apple, HTC, Samsung, Huali, Nokia, Motorola, Lenovo, Sony, LG

this case we resorted to using a custom Recovery booted in the previous step. All patterns were delimited with common 10-byte HEADERs and FOOTERs (Fig. 5.7), and uniquely identified by a 4-byte counter (ID). We filled each pattern with random bytes (RANDOM) to avoid the underlying chip using compression, and added a 16-byte MD5 DIGEST over the RANDOM bytes for verification. Pattern generation was done on a laptop, and sent to the phone via USB with the Android Debug Bridge (adb) utility shipped with the Android SDK (Fig. 5.6).

```
# laptop: forward port 12345 to device
$ adb forward tcp:12345 tcp:12345

# device
$ /dev/busybox nc -l -p 12345 | /dev/busybox dd of=/dev/block/mmcblk0p2

# laptop: pipe pattern to local port
$ ./echo_pattern | nc localhost 12345
```

Figure 5.6: Pattern writing for Galaxy S's data partition (i9000). Reading a partition is achieved with similar commands.

Figure 5.7: Pattern written to a partition of interest.

Sanitisation. We performed a wipe with one of the recommended options given by vendors, i.e. (*i*) the Factory Reset in Settings, and (*ii*) the option in the Recovery/Bootloader mode. Because of the lack of formatting and file system introduced by the patterns, some devices would fail to perform the wipe on external storage. In this case, we first re-formatted them with the built-in Settings option. When we had installed a custom Recovery, we would also take care of re-installing the original one prior to the wipe (with a backup).

Imaging. We imaged the entire partition with similar commands as presented in Fig. 5.6.

Validation. The last step was the recovery and validation of patterns. We searched for non-deleted pattern candidates using their known header and footer, then validated each candidate by verifying the digest over the random data. We also investigated which percentage of the disk was properly sanitised (i.e. zeros for an eMMC or ones for a raw flash), in order to confirm that it was consistent with the number of patterns we had recovered.

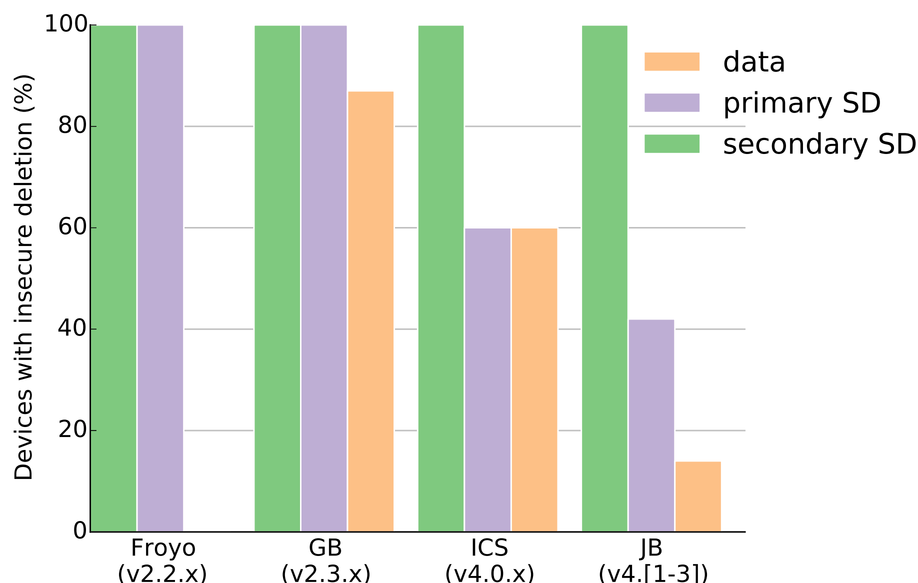


Figure 5.8: Percentage of devices with flawed logical sanitisation. Results for primary and secondary SD cards (i.e. external storage) implicitly assume the use of the Factory Reset in Settings since the Recover/Bootloader Factory Reset only sanitises the data and cache partitions.

5.3.2 Results and Discussion

Preliminary Results

We found that the sanitisation of external storage occurs only if a user selects the additional option "External Storage" in the Factory Reset in Settings. In this case, the Android OS first performs the sanitisation of external storage and then reboots into Recovery/Bootloader mode where the data partition and the cache partition (which contains mainly optimized .odex java classes) are sanitised. If a user Factory Resets his device with Recovery/Bootloader instead of Settings, external storage is not sanitised – subtle difference between devices are provided in the following paragraphs. Fig. 5.8 shows the results of built-in sanitisation for all devices studied.

Sanitisation of the data partition

Recall from Section 5.2.4 that the data partition stores sensitive information from Google and third-party apps, such as credentials.

On devices running the oldest version of Android we studied (Froyo, v2.2.x), the data partition was logically sanitised. These devices use a raw flash with the yaffs2 file system (Section 5.2.1), where it is infeasible to reformat (i.e. write) a partition without first properly sanitising it (i.e. erasing previous blocks). The Android Open Source Project

Table 5.1: Chronology of Factory Reset Implementations in AOSP.

		OS Versions				
Code	Partition	Froyo (2.2.x)	GB (2.3.x)	ICS (4.0.x)	JB (4.1.[1-3])	KK (4.4)
Android	primary SD	*format()* ✗	*format()* ✗	*format()* ✗	*ioctl(BLKDISCARD)* ✗	*ioctl(BLKDISCARD)* ✗
	secondary SD	none ✗	none ✗	none ✗	none ✗	none ✗
Recovery	data	*ioctl(MEMERASE)* ✓	*ioctl(BLKDISCARD)* ✗	*ioctl(BLKSECDISCARD)* ✓	*ioctl(BLKSECDISCARD)* ✓	*ioctl(BLKSECDISCARD)* ✓

(AOSP[1]) reveals the use of the *ioctl*'s *MEMERASE* command by the Recovery mode (Table 5.1), this provides digital sanitisation and confirms the results of Fig. 5.8.

From Gingerbread onwards ($\geq$ v2.3.x), all devices we encountered use eMMCs (except one), where it is possible to reformat a partition without properly sanitising it first – since the eMMC erases blocks on behalf of the OS. Fig. 5.8 shows that about 90% ($\approx$ 300M) of Gingerbread (v2.3.x) devices sanitise the data partition insecurely, in that at most a few hundred MB are deleted, representing between 60% and 99.9% of the data partition depending on its size. The Android Open Source Project (AOSP) reveals the use of the *ioctl*'s *BLKDISCARD* command by the Recovery mode (Table 5.1), this does not provides logical sanitisation and therefore confirms the results of Fig. 5.8. The only device in our sample that properly deletes the data partition is the HTC Wildfire S (Gingerbread); and it is because it uses yaffs2 rather than an eMMC.

As shown in Table 5.1, the following Android version (ICS, v4.0.x) marked the introduction of logical sanitisation support via *BLKSECDISCARD* in the AOSP code. This contradicts the results from Fig. 5.8 that show that 60% ($\approx$ 140M) of ICS (v4.0.x) devices incorrectly sanitise the data partition. Many of these ICS devices in our sample were initially released with GB (v2.3.x) and received upgrades to ICS (v4.0.x). We verified that the phone binaries indeed contained the newer code from AOSP, i.e. with logical sanitisation support. We then turned our attention to lower-level code, and found that vendor upgrades likely omitted device drivers necessary to expose the logical sanitisation functionality from the underlying eMMC. In practice, this means that the secure command *BLKSECDISCARD* is not supported by *ioctl*, and it returns errno 95 (*EOPNOTSUPP*). It could be the case that the eMMC itself does not support secure deletion, but we think this is unlikely since the 2007's 4.2 eMMC standard[2] already provided the compulsory "ERASE" command for logical sanitisation. We found evidence corroborating this claim on certain phones at least: when unlocking the Bootloader[3] of the HTC Sensation XE, the data partition was properly sanitised whereas it was not during a Factory Reset. Devices affected include the Samsung Galaxy S Plus, S (25M units sold[4]) and S2 (40M units sold [4]), and the HTC Sensation XE. Only the Google Nexus S in our sample properly sanitised its data partition after receiving upgrades to ICS. The problem is likely to persist after further upgrades to Jelly Bean (JB, v4.[1-3]), although we could not ascertain this as our sample did not contain such devices.

Besides upgrade issues, devices shipped with newer Android versions such as ICS (v4.0.x) and JB (v4.[1-3]) are not free of problems either (Fig. 5.8). They too are not always shipped with proper device drivers for secure deletion. For example, the LG

[1]`android.googlesource.com`

[2]`www.jedec.org/standards-documents/docs/jesd-84-b42`

[3]This procedure lets users install custom software, but wipes data to prevent a thief from recovering user's data via forensic software

[4]`www.tomshardware.com/news/Samsung-Galaxy-S-S2-S3,20438.html`

Optimus L5 shipped with ICS did return errno *EOPNOTSUPP* when we attempted a secure deletion. More intriguing, the Motorola Razr I, shipped with JB (v4.[1-3]), did not return any errors, but the secure deletion resulted in no block being deleted at all. Due to these driver issues, Fig. 5.8 shows that 15% (≈ 55M) of JB (v4.[1-3]) devices improperly sanitise the data partition.

Sanitisation of the Primary SD Card

Recall from Section 5.2.4 that the primary SD card corresponds either to the internal one or to a physically removable one in the absence of the former. It mainly hosts multimedia files made with the camera and possibly third-party apps.

We found that no Froyo (v2.2.x) and Gingerbread (GB, v2.3.x) devices we examined logically sanitised their primary SD card. This represents more than 340M devices. The AOSP code reveals that in these versions, Android only formats the primary SD card with a call to *Fat::format()* (Table 5.1), this confirms the results from Fig. 5.8. In practice, a few dozen MB at most are logically sanitised.

As depicted in Table 5.1, the AOSP reveals no code changes in the sanitisation of the primary SD card in the following Android version (ICS, v4.0.x). Yet, Fig. 5.8 shows about 40% (≈ 90M) of ICS devices properly sanitise their primary SD card (i.e. ≈ 140M of devices do not). One may conclude that vendors have customised the AOSP code and added secure deletion support for the primary SD card, but this is incorrect. This logical sanitisation is due to the following reasons: (*i*) the primary SD card on these phones corresponds to the internal one, (*ii*) these devices use an emulated SD card physically stored on the data partition (Section 5.2.4) and (*iii*) proper sanitisation of the data partition is implemented as per AOSP, and so gets "inherited" by the primary SD card. Only when these three fortunate conditions are met can one be confident that the primary SD card is logical sanitised.

The following Android version (JB, v4.[1-3]) marked the addition of insecure deletion via *ioctl*'s *BLKDISCARD* command. This confirms why 40% (≈ 150M) of devices still fail to logically sanitise their primary (internal or external) SD card. At the time of writing, we are not aware of any changes to the handling of the primary SD card, therefore we expect these results to hold on all other Android versions.

Sanitisation of the Secondary SD Card

In our sample, no devices properly sanitised the secondary (external) SD card. We found that Android generally does not attempt to sanitise it at all (Table 5.1), which explains the results from Fig. 5.8.

Vendor Customisation Inconsistencies

Besides the various differences of sanitisation between versions and models already highlighted, we discovered other vendor issues. For example, we mentioned that only the Factory Reset in Settings provides an option to sanitise the primary SD card (Section 5.3.2). Therefore one might advise users to use Settings rather than Recovery to sanitise devices. Unfortunately, vendor customisations sometimes make Recovery more reliable. For example, the two HTC One-series phones in our sample properly sanitised their primary (internal) SD card in Recovery (contrary to AOSP), but not in Settings (as one would expect from the AOSP source code). It is likely that this result holds for many of the other HTC One-series devices. This also violates HTC guidelines: on its website[1], it suggests users first try Settings, and resort to Recovery only "if you can't turn HTC One [X] on or access settings". On its web page, HTC has put up a note to discharge itself of any responsibility: "A factory reset may not permanently erase all data from your phone, including personal information".

eMMC implementation of logical sanitisation

In general, We found that devices in our sample logically sanitised all bytes requested through the *ioctl* command, except for one phone: the Google Nexus 4. This has an 6189744128B data partition, fully used by the file system. The last 16KB were not sanitised and fully recoverable about 20% of the time after a Factory Reset. Our hypothesis is that this might be a bug in the eMMC itself (or its corresponding drivers), since we have not seen similar problems in other devices.

Number of logical blocks to sanitise

If issued with the non-secure sanitisation command "DISCARD", an eMMC applies a "don't care" policy to the block. According to the standard, "the original data may be remained partially or fully accessible to the host dependent on device". This further means that data originally exposed at a logical block located at logical offset LO_{org}, could be re-mapped at a different logical offset, say $LO_{remapped}$ as shown in Fig. 5.9. If the file system of size M does not fill the entire partition, there is a risk that the deleted block's data "crosses" the filesystem boundary. Therefore, if the sanitisation only attempts to securely sanitise the blocks used by the file system ($[0, M]$ in Fig. 5.9), it is plausible that some remapped blocks (within $]M, S]$) would not be purged. We stress that we have not found evidence of this happening in our device sample. However, in order to reduce the possibility of this happening, we suggest vendors erase the entire partition, rather than just the part used by the file system. In our device sample, the HTC One-series phones

[1] www.htc.com/us/support/htc-one-s-t-mobile/howto/315367.html

left a few MB of space at the end of the data partition. It would be more cautious to sanitise the entire partition. Similarly, the AOSP code currently truncates the partition size to a multiple of 4096 when creating the file system and when computing the size to wipe during Factory Reset. Sanitising the entire partition would be more prudent.

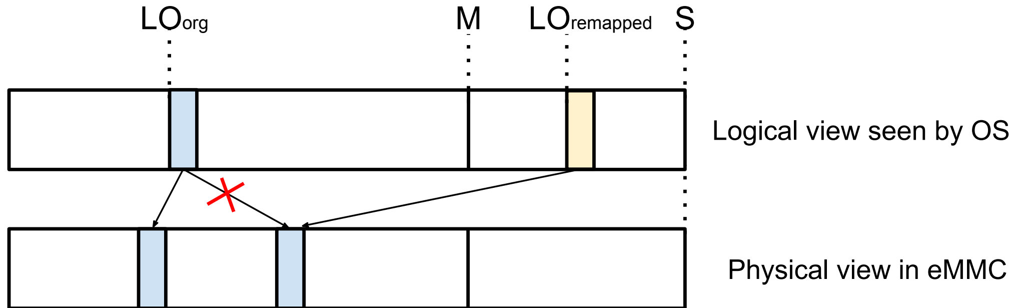


Figure 5.9: Cross file system boundary example for the data partition.

5.4 Data Recovery in Practice

Our objective here was not to implement new tools and algorithms, but to evaluate the feasibility and scalability of attacks. Data on Android smartphones is generally stored in SQLite databases and text-like files.

To extract SQLite files, we initially investigated the use of file carving with *Scalpel* [241]. File carving is the practice of searching for files by leveraging the knowledge of their content and structure, rather than relying on file system metadata[1]. In practice, we found that the database file header and its content were not always contiguously allocated, probably because of repetitive updates. Furthermore, each application's database has a different layout, and records are not always located contiguously on the partition. Therefore, parsing the data from such fragments is not fully reliable with simple techniques. So we used SQLite file carving only as a preliminary step.

We quickly realised that most of the data (including database files), exhibits specific and distinct formats. For instance, the list of installed apps is stored in the file */data/system/packages.xml* with a well-defined structure of the form <*package name="a.b.c" codePath="/system/app/myapp.apk" ...*>. Therefore we primarily used pattern matching to recover data on all second-hand phones from Section 5.3.1, and file carving solely to extract multimedia files.

[1] A simple file carver would search for files' headers and footers. More advanced ones would also look for file fragment candidates and piece them together using certain properties of the underlying data.

Table 5.2: Practical Recovery of Data

	Storage	Extraction	Percentage Devices	Attack usage	Comments
Phone Owner	data partition	automated pattern matching on third-party apps like Facebook or Google accounts app	100%	contact user to blackmail (assuming compromising data is recovered)	extraction through default Phonebook app's data generally requires some human intervention
Installed apps	data partition	automated pattern matching on */data/system/packages.xml*	100%	identify high-value targets / adjust forensic strategy based on easy-to-retrieve app-formatted data	
Contacts	data partition	automated pattern matching on third-party apps like Facebook or WhatsApp (Fig. 5.10(a))	100%	sell in underground markets	associating names to contact details in Phonebook app data generally requires some human intervention
Browsing	data partition	automated pattern matching	100%	blackmail user	
Credentials	data partition	automated pattern matching for browser cookies, WiFi (Fig. 5.10(b)), Google (Fig. 5.10(c)) and other apps.	100%	sell in underground markets	Google master token recovered 80% of the time
Multimedia	data partition, primary SD	automated file carving (*Photorec*[a]) for camera-made images and video and Ib thumbnails	100%	blackmail user	
Conversations	data partition	automated pattern matching for third party messaging apps and emails	100%	blackmail or sell in underground markets	identifying SMS required some human intervention, emails recovered in 80% of devices but only a few

[a] http://www.cgsecurity.org/wiki/PhotoRec

5.4.1 General Results

We hunted for the information shown in Table 5.2 in devices with a flawed Factory Reset. For example, we recovered some "Conversations" (SMSes, emails, and/or chats from messaging apps) in all devices (Column "Percentage Devices") using pattern matching (column "Extraction"). Compromising conversations could be used to blackmail victims (column "Attack usage"). Gmail app emails were stored compressed. By searching for relevant headers, we were able to locate candidates and then decompress them. We found emails in 80% of the flawed devices, but generally only a few per device (column "Comments").

5.4.2 Case Study: Hijacking Google Accounts

To improve usability and user engagement, most smartphone apps replace passwords with authentication tokens the first time a user enters his password. After the first password-based authentication, users are automatically logged-in with the authentication token; emails can be retrieved, calendar notifications downloaded, etc. without user intervention. These tokens are often stored on non-volatile flash storage on the data partition. Some Google tokens for the account *username@gmail.com*, are shown in Fig. 5.10(c). The first one, which we call the "master token", is the long random string starting with "AFc". It gives access to most Google user data. As a test, we Factory Reset a test phone, then recovered the master token. We then created the relevant files and rebooted the phone. After the reboot, the phone successfully re-synchronised contacts, emails, and so on. We recovered Google tokens in all devices with flawed Factory Reset, and the master token 80% of the time. Tokens for other apps such as Facebook can be recovered similarly. We stress that we have never attempted to use those tokens to access anyone's account.

5.4.3 Possible Attackers

Individuals buying devices on auction websites such as eBay are possible attackers. They need to spend a non-negligible time to bid and follow up on auctions. Furthermore, they have to pay a few dollars for commission and shipping fees for each device. So low-value data like contacts and email addresses do not seem profitable. Recovery and analysis of conversations and images (to blackmail victims) would generally require human intervention or more advanced tools, with the possible exception of browser history where simple keyword search can be effective. Blackmailing users requires enough devices to hit compromising data and enough users to hit a gullible mark. But this requires (*i*) a significant time investment to bet on/follow items and (*ii*) great logistics to buy, process, and re-sell devices. Therefore we think that only people with enough time on their hands

```
447123456789@s.whatsapp.netHey there! I am using WhatsApp.07123456789
Bob
BobBob
447709876543@s.whatsapp.net*** no status ***+447709876543
Alice
Alice Alice
```

(a) Whatsapp contacts with name and phone number.

```
network={
        ssid="SSID1"
        key_mgmt=NONE
network={
        ssid="SSID3"
        psk="mypassword"
        key_mgmt=WPA-PSK
```

(b) WiFi passwords.

```
username@gmail.comcom.googleAFcb4KRs88NZlzN-r6qHrSHGF1TWyh...TKw==
clDQAAAJ4AAABQPfQhNXLTDYDLgHoIFDdDIEojBokYr_6ad0WeSr2kVpK4...B-0pd
androidmarketDQAAAJ8AAAD1NNQaeO_yxfgNMtSvnQVangE3DAatlKtTo...INkZV
```

(c) Android tokens.

Figure 5.10: Example of pattern matching results.

could make extra cash on top of an existing income this way. In general, high-value data like banking credentials appear likely to be the most profitable criminal option.

Smartphone salesmen in brick-and-mortar shops can process devices cheaply, since it may be part of their job to receive second-hand devices and they can scan these devices without paying auction and shipping costs. So low-value data may be profitable for them. As it is common for merchants to talk to customers, they could also identify higher-than-average-value targets suitable for blackmail. Attackers who can add forensic software to the recycling chain may further increase the number of devices processed, and the amount of low-value data recovered. Attackers with access to corporate devices could also gain access to high-value data. On the other hand, shop staff will be easier for the police to identify and arrest once a complaint of blackmail is made. Thieves are yet another category of attackers: data are less likely to be deleted from a stolen device, so they are out of scope of this chapter – we study such attacks in Chapter 6.

Although a lot of data can be recovered using pattern matching, this does not necessarily translate into actual profitability.

5.5 Alternative Sanitisation Methods

We have considered the following methods to mitigate flawed Factory Resets.

Filling up the partition of interest with random-byte files, in the hope of overwriting all unallocated space, could be achieved via third-party non-privileged apps after the

built-in Factory Reset. This would require the app to be installed manually by users after a Factory Reset is performed. Otherwise, Google credentials stored on the file system (necessary to install an app from Google Play) will not be erased by the procedure. This sanitisation procedure also adds an additional layer of uncertainty because it uses the file system rather than direct flash access. File systems also vary across devices and may be proprietary (such as Samsung's RFS). We therefore felt this option would not scale reliably across devices, and we discarded this method in our tests.

Overwriting the entire partition "bit-by-bit" once did provide logical sanitisation for all devices and all partitions we studied; it is therefore a reliable alternative. The drawback of this method is that it requires privileged (i.e. root) access to devices in practice. Therefore, it is likely to put off ordinary users. This method does not provide thorough digital sanitisation, since the flash is over-provisioned – but an attacker is unlikely to recover the data through public APIs exposed by the Linux kernel. Note that physical extraction of the eMMC is not necessarily required to recover data, for example if the eMMC exposes hidden features accessible with a patched kernel or a custom driver. Yet this is phone-dependent and out of scope of our threat model. To have high certainty that all blocks are erased, one would need to overwrite the partition multiple times to purge all additional blocks. But this would require knowledge of the wear-levelling algorithms and over-provisioning for every eMMC, which are generally proprietary. Furthermore, the over-provisioning could differ even for instances of the same device, for example if different grades of flash were used. Since we were concerned only with massively scalable attacks, we did not consider this issue further, but firms with high assurance requirements might have to unless they can use encryption, which we consider next.

Enabling Full Disk Encryption (FDE) on first use of the device would be more appropriate for ordinary users if devices support it. Enabling FDE only before performing a Factory Reset (as suggested by Google[1]) may only provide logical sanitisation, not thorough digital sanitisation (plain-text data could still be present on the flash as it is over-provisioned). FDE was introduced in ICS (v4.0.x) so it cannot help the large number of affected GB (v2.3.x) devices. On one HTC phone running GB (v2.3.x), we found an encryption option, but it left all the data behind. We assume this was a vendor customisation and may only encrypt allocated space. FDE for the internal SD card is not supported on all phones, and not all v4.x devices support FDE on the data partition despite AOSP's support. As a rule of thumb, only devices with an emulated internal SD card inherit the "encryption support" from the data partition when supported. On supported Android versions, the encryption key is stored encrypted with a key derived from a salt and a user-provided PIN (or password). This encrypted blob is referred to as the "crypto footer" in the AOSP source code. An attacker who gains access to the crypto footer has enough information to brute-force the user's PIN offline. The footer is stored

[1] `www.bbc.com/news/technology-28264446`

in a dedicated partition or in the last 16KB of the data partition – the exact location is configured by vendors through the "encryptable=" option of the Android fstab file. In either case, we found that a flawed Factory Reset failed to erase this footer. Consequently, to logically sanitise a device with encryption, it is essential to select a strong password to thwart offline brute-force attacks. As most people just use a 4-6 digit PIN, it would usually be trivial to brute-force.

Mobile Anti-Virus (MAV) apps have a "remote wipe" feature to sanitise data on lost or stolen smartphones. We study their effectiveness in Chapter 6 and show that remote wipe functions are not an alternative to a flawed built-in Factory Reset.

5.6 Recommendations

For vendors, we recommend using a recent eMMC with support for digital sanitisation, and to properly expose it in the Bootloader, Recovery and Android kernels. More generally, previous research has shown that vendors' customisations are a source of security problems [66, 191, 242]. Therefore, we provide the following guidelines to the AOSP developers, hoping they can reduce the chance of slip-ups in the future:

1. Use an emulated primary SD card: this ensures that only one partition needs to be properly sanitised on the phone, reducing the space for mistakes.

2. Erase the entire partition, not only the part explicitly used by the file system. This reduces the chance of unfortunate surprises due to eMMC wear-levelling block management and deletion implementation problems.

3. Implement sanitisation of all partitions in one place only; for example in Recovery mode or Bootloader mode; and have Settings simply have the phone reboot into the appropriate mode with the right parameters. Having the relevant code in one place eases code review, testing, and reduces possible mistakes.

4. Expose an option to have the Recovery mode perform a sanitisation validation, by reading back the entire partition and checking it.

5. Provide test units for vendors to test sanitisation in the Android Compliance Suite Test (CST). Have the tests fail if secure sanitisation fails, e.g. if not supported or if the verification step 4 fails.

6. Do not resort to an insecure sanitisation if the secure one fails - as it is currently the case[1].

[1] source.android.com/devices/tech/security/best-practices.html

7. Before a Factory Reset takes place, a broadcast Intent could be sent to apps, so that they could take necessary steps to invalidate their credentials – assuming that Internet connection is available.

8. Store the encryption metadata at the start of the data partition in a crypto header, rather than at the end in a crypto footer. This reduces the risk of dictionary attacks in the event of flawed sanitisation, since the first blocks are generally overwritten during partition formatting. Storing the metadata on the data partition also ensures that there is only one partition to take care of, as above. We discuss improvements to the current encryption implementation next.

5.7 Encryption Requirements

A typical FDE implementation must protect the keys from an attacker with physical access to a non-wiped device, e.g. JTAG attacks, cold boot attacks, etc. The FDE key should also be protected at rest: this typically involves encrypting the FDE key with a combination of a user-supplied password/PIN and/or keys stored on CPU's protected non-volatile storage [243]. It also involves compartmentalising the software, determining which parts should be allowed to see the key in clear, the access control for users, etc. This is non-trivial in practice: as shown by the recent FBI vs. Apple case, a single mistake can be fatal[1].

If the sole purpose of FDE is to mitigate a flawed Factory Reset (FR), the implementation can be simpler. In fact, if the threat model is only an attacker who attempts to recover data from a wipe phone, it is no longer necessary to implement access controls and software compartmentalisation: we only need the FDE key and metadata to be properly erased during Factory Reset. The current Android implementation keeps the FDE key in a footer. The key is encrypted under a user-supplied PIN with low entropy. If the FR is flawed and does not erase the footer, the FDE key may be recovered by brute forcing the PIN offline. Can we improve on this simple design to mitigate a flawed FR?

The first solution is to increase the entropy of the key that encrypts the FDE key. For this, we need hardware support because users cannot remember long keys. For example, the CPU may come with dedicated non-volatile storage to store additional long keys. This storage should be digitally sanitised during Factory Reset. The deletion of the key material implies personal data stored on the larger chip is irrecoverable. It is hard for this solution to scale across multiple Android phone vendors, but possible for iOS devices that are vertically integrated – in fact iOS uses a similar approach [243].

The second solution does not require hardware support. A device may store the encrypted footer on a different chip which Android can digitally sanitise. This is simple to

[1] `http://www.bbc.co.uk/news/technology-35601035`

implement with "raw flash" as presented in Section 5.3. During FR, the Android kernel only needs to sanitise the chip that stores the footer, whereas the FR of the larger eMMC containing personal data may fail safe. As presented in Section 5.3, the code that erases the "raw flash" is less prone to coding mistakes and vendor customisation problems. This solution is simple to deploy, and the code can be released in the Android source tree to make auditing feasible.

The third solution is to have a dedicated chip that does FDE in hardware. This is a viable solution for a platform such as iOS which is vertically integrated. On the contrary, for Android, there are problems with this approach. First, this would put security in the hands of additional vendors/suppliers: but researchers have repeatedly shown how this introduces vulnerabilities (Section 2.3 in Chapter 2). In fact, there is already evidence that SSD manufacturers sometimes implement sanitisation incorrectly [210]. Second, we fear this solution would hamper design/code security audits by third-party security researchers.

To conclude, for the sole purpose of mitigating a flawed Factory Reset, the second option is the simplest to implement reliably and deploy at scale for the Android platform. This, however, does not account for the access controls, trusted boot, software compartmentalization, etc. that would be required to thwart a more capable attacker with physical access to a non-wiped device.

5.8 Summary

We conducted the first thorough analysis of Android factory reset functions by studying 21 Android smartphones from 5 vendors running Android versions v2.3.x to v4.3. We presented a detailed and chronological analysis of flaws across Android versions. We tracked these issues to (*i*) Android failures, (*ii*) inadequate vendor upgrade practices, and (*iii*) improper vendor integration and testing practices.

Chapter 6

Security Analysis of Android Anti-Theft Apps

In this chapter, we study the "anti-theft" mechanisms available to consumers to thwart unauthorised access to personal data on stolen Android smartphones. With millions of devices stolen in the USA in 2013 alone, such attacks are a serious and growing problem. The main mitigation against unauthorised data access on stolen devices is provided by "anti-theft" apps; that is, with "remote wipe" and "remote lock" functions. We study the 10 most downloaded Mobile Anti-Virus (MAV) apps that implement these functions. They have been downloaded hundreds of millions of times.

We investigate the general security practices of MAVs, as well as the implementation of their "remote wipe" and "remote lock" functions. Through the analysis, we uncover flaws that undermine MAV security claims and highlight the fragility of third-party security apps. We find that MAV remote locks can be unreliable due to poor implementation practices, Android API limitations, and vendor customisations. We find that mobile OS architectures leave third-party security apps little leeway to improve built-in Factory Resets, therefore MAV remote wipe functions are not an alternative to a flawed built-in Factory Reset. We conclude the only viable solutions are those driven by vendors themselves.

The work presented in this chapter was published in the 4th Workshop on Mobile Security Technologies (MoST) [244], and is in collaboration with Ross Anderson. I conducted the analysis for all apps. Ross helped with the writing and reviewing of the final paper.

6.1 Introduction

The extraction of personal data on stolen devices is a growing concern. In 2012, smartphone robberies represented almost 50% of all robberies in San Francisco, 40% in New York

City and were up 27% in Los Angeles[1]. In 2013, 3.1M devices were stolen in the USA[2], and 120,000 in London[3]. For the half of all users who do not lock their screen[4], the main anti-theft protections in use today are "remote wipe" and "remote lock" functions. Products that offer these remote anti-theft data protections include a range of enterprise and consumer-grade offerings.

In this study, we focus on the latter and study the 10 most downloaded Mobile Anti-Virus (MAV) apps. When we conducted this study, the Google Play store showed that these apps were prevalent with the top 2 MAVs downloaded between 100M and 500M times each, the third between 50M and 100M times, and the following 4 between 10M and 50M. In comparison, the top enterprise app (for mobile device management) was downloaded less than 5M times.

Our analysis reveals eight issues: (*i*) misinformation given to users following a remote wipe and lock; (*ii*) questionable MAV authentication practices; (*iii*) the limitation of, and the restriction imposed by, Android's APIs and architecture; (*iv*) the misuse of Android security APIs by MAVs; (*v*) inconsistency of Android's API across versions; (*vi*) incorrect Android API documentation; (*vii*) MAV reliance on carrier network (in)security; and (*viii*) unfortunate customisations by vendors.

For example, we found that because of Android API differences across versions, 9 MAVs can be un-installed by a thief before a Gingerbread (v2.3.x) phone is remotely locked by its user; because of API misuse, 4 MAV locks can be bypassed (Section 6.6); and because of vendor customisations, all MAV locks can be circumvented.

By comparing MAV remote wipe functions with the findings on Android built-in Factory Resets (Chapter 5), we find that mobile OS architectural decisions that were aimed at enhancing security (e.g. the permission system and lack of root access) get in the way of MAVs that attempt to improve the reliability of flawed factory resets. Therefore, MAV remote wipe functions are not an alternative to a flawed built-in factory reset (Section 6.7).

In summary, the contributions are as follows:

- We present the first comprehensive study of Mobile AV (MAV) implementations in the context of device theft, including their general security practices and anti-theft functions.
- We uncover major failures that may affect millions of users. These flaws are not only caused by questionable practices by MAV developers, but also by vendor

[1] gizmodo.com/5953494/hold-on-tight-smartphone-mugging-is-more-popular-than-ever

[2] www.consumerreports.org/cro/news/2014/04/smart-phone-thefts-rose-to-3-1-million-last-year/index.htm

[3] www.london.gov.uk/media/mayor-press-releases/2013/07/mayor-challenges-phone-manufacturers-to-help-tackle-smartphone

[4] www.consumerreports.org/cro/news/2014/04/smart-phone-thefts-rose-to-3-1-million-last-year/index.htm

customisations and by the limitation of the Android OS. Remote lock functions can therefore be bypassed, and remote wipe functions are not an alternative to flawed Factory Resets.

- We discuss possible countermeasures, but conclude that only vendor-provided software has the potential to raise the reliability of anti-theft mechanisms.

6.2 Background

A prerequisite to understanding this chapter is Section 5.2 of Chapter 5 that introduces (*i*) flash memory and file systems (Section 5.2.1); (*ii*) the different data partitions present on a typical Android phone (Section 5.2.4); and (*iii*) the different levels of sanitisation (Section 5.2.2).

For a fair comparison with built-in Factory Resets, we take the same approach as in the previous chapter, in that we consider a remote wipe to be "secure" or "proper" if it provides logical sanitisation.

Throughout this chapter, we refer to the "device sample" or just the "sample" as the pool of devices studied in Chapter 5.

6.2.1 Bootloader, Recovery and Safe Modes

Besides partitions storing personal user data, phones also store (binary) executable files in dedicated partitions (Fig. 5.3 in previous Chapter 5). These contain binaries to boot in normal mode (i.e. Android) and other less-known special modes of operation. Android smartphones generally have three extra modes of operation that are useful to our discussion: the Bootloader mode, the Recovery mode and the Safe mode. A user can boot into them by pressing a combination of hardware keys on the device. There exist subtle variations between vendors, so we try to keep the description general.

The Recovery mode is generally a headless Android OS used for performing updates and backups to the current installation: updates may be stored in external storage or sent in-band from a computer connected via USB. The Bootloader is not based on Android, and it allows flashing new software and partitions to a device, generally via USB. To achieve their functionalities, both the Recovery and Bootloader mode must run with high privileges. Typically, this means unrestricted access to both the Android OS binaries and partitions storing user data. There are three kinds of Bootloader and Recovery protections we have found on devices: open, protected, and locked.

Open Bootloaders/Recoveries let anyone with physical access to a device install custom updates. We found this to be true for most Samsung and LG devices in the devices we used in Chapter 5.

Locked Bootloaders attempt to lock devices to a certain carrier or vendor by enforcing signature verification on software updates. This is true of most HTC devices we encountered. To disable the signature verification, a locked bootloader needs to be "unlocked". This may be possible via OS or bootloader exploits. HTC also lets users "unlock" their Bootloader through their website, but voids the warranty of the device thereafter. Upon unlock, the Bootloader is supposed to wipe all data on the device, so as to prevent thieves from recovering users' data after installing forensic software. We note that a locked Bootloader/Recovery is not a panacea: as the key used to sign a software update is owned by the vendor, an insider – or a server or CA compromise – could leak it to attackers. We stress that this is not a hypothetical scenario: for one phone in our sample, we found an implementation of Recovery that passed the signature verification and let us root the device. In practice, a locked Bootloader/Recovery may provide enough security for average users, but not for firms with high assurance requirements.

Protected Bootloaders/Recoveries genuinely try to protect users: the lock does not serve any business purpose. Users are empowered to unlock their Bootloader to install custom software without voiding the warranty. This is mostly true of Google phones. Unlike open and locked Bootloaders, a protected one can be "re-locked". If a thief wants to install forensic software on the device, he can unlock the Bootloader but this will also wipe the device's data.

The Safe Mode boots the main Android OS, but disables all user-installed apps. This is primarily used for users to un-install misbehaving apps, for example malware that may lock the screen and render the phone unusable. Even though apps are disabled in the user interface, they can still be launched via a shell. Obtaining a shell can be achieved by first enabling the Android Debug Bridge (adb) developer option in the default phone Settings; and then plugging the device into a computer via USB. By design, this gives a shell prompt on the computer to interact with the device.

6.2.2 Mobile Anti-Virus (MAV) Apps and Device Admin API

At the time of the study, Mobile Anti-Virus (MAV) apps had already been downloaded hundreds of millions of times from the Google Play store. They generally achieve their remote anti-theft protections with an app installed on the device in combination with an online web interface accessible from a standard web browser. A user who loses his phone can log in the web interface and remotely instruct the phone app to wipe or lock the device. An exception amongst the apps we studied was *Dr.web*: instead of using a web interface, it required users to define trusted phone numbers, from which a user could send remote commands to his lost phone via SMS.

A simple attack against anti-theft solutions is the use of "Faraday bags" to block all radio-frequency communications between a stolen device and its cloud service, thereby

preventing any remote action from a device's owner. We leave this problem aside for the moment and discuss possible countermeasures in Section 6.8. In this study, we highlight other important issues which we believe are relevant to improve the reliability of current anti-theft solutions in general.

All MAVs make use of a special set of functions accessible via Android's "Device Administration API", that provides administration features at the system level. Once an application is granted access to this API, it becomes a "device admin" and gains access to security "policies" like the password policy (e.g. to enforce password strength), or the encryption policy. Each policy within the admin set must be explicitly requested in an app's manifest. Fig. 6.1 shows the relevant code for requesting access to the `force-lock`, `wipe-data`, `reset-password` and `disable-camera` policies. Unlike traditional Android permissions, the admin permission and policies are not granted at installation time: they must be approved all at once by a user in the Android default Settings. When not granted, an app can still run, but without admin privileges. The two admin policies relevant to this study are the wipe and screen lock policies that can be used to protect users' data when devices are lost. At runtime, an admin app with the `wipe-data` policy can invoke the *wipeData(int flag)* function to perform a wipe. It currently supports wiping the data partition only ($flag = 0$) or with the additional wiping of the primary SD card ($flag = WIPE_EXTERNAL_STORAGE$). The API does not support wiping the secondary (external) SD card. Internally, *wipeData()* uses the device's built-in Factory Reset – so its reliability varies across devices as presented in Chapter 5. An admin app with the `force-lock` policy can also use the built-in PIN screen to lock the phone screen (e.g. by invoking the *lockNow()* function). One additional security protection is that an admin MAV cannot be un-installed unless its admin privileges are first removed in the default Android Settings. Nevertheless, even an admin MAV has limitations: it cannot access other apps' private directories in the data partition, nor can it bypass the file system to read/write arbitrary content from/to storage.

If a user forgets to enable the admin permission for a MAV, the app can neither use built-in wipe and lock features, nor overwrite partitions reliably bit-by-bit to sanitise data storage. Therefore, it must resort to less reliable ad-hoc mechanisms. For example, it may use public Android APIs with the traditional permissions granted at installation time (like contact APIs to remove contacts from the Phonebook app). However, this generally results in the deletion of records in the associated SQLite file, which does not provide logical sanitisation. For external storage (which is accessible by all apps), a non-admin app may fill existing files with random bytes, *unlink* them, create new ones (in the hope of overwriting unallocated file system space), or format the partition. User-installed apps generally do not expose sanitisation APIs on the phone, so their data would typically remain intact. As for the screen lock, a MAV could detect when it loses screen focus, and subsequently launch one of its "views" (a.k.a. Android Activities) to foreground to "lock" the screen.

```
# =========== AndroidManifest.xml ==========
<receiver android:permission="BIND_DEVICE_ADMIN">
  <intent-filter>
   <action android:name="DEVICE_ADMIN_ENABLED" />
  </intent-filter>
  <intent-filter>
   <action android:name="DEVICE_DISABLE_REQUESTED"/>
  </intent-filter>
</receiver>

# =========== device_admin.xml ==========
<device-admin [...] >
  <uses-policies>
    <reset-password />
    <force-lock />
    <wipe-data />
    <disable-camera />
  </uses-policies>
</device-admin>
```

Figure 6.1: Device admin request example. Permission and action names are purposely shortened for readability. Two broadcast receivers are declared: one to receive a notification when the user has accepted the admin permission, another when the user is trying to disable it in the default Android Settings. Four policies are requested.

6.3 Methodology

We restricted ourselves to the 10 most-downloaded MAVs on Google Play – the Google Device Admin app was not in the top 10. We downloaded them using a Samsung Galaxy S Plus phone between Nov. 2013 and Apr. 2014. The Samsung Galaxy S Plus runs Gingerbread (v2.3.5), has a primary SD card formatted in FAT, and we inserted a secondary 2GB removable SD card in its slot. We conducted a review of each app's code using *apktool*[1] and simple runtime analysis to validate our findings. For the runtime analysis, we added logging code to apps and repackaged them. Many apps already shipped with some code to log runtime information; so in this case, we also toggled relevant variables in the code to enable the logging.

In the following sections, we report our findings on the general security of MAV solutions, and specifically focus on anti-theft functions in Section 6.6 and Section 6.7. We also report on the discussions we had with MAV developers after responsible disclosure of

[1] code.google.com/p/android-apktool/

the findings.

6.4 Account Authentication

MAVs are sensitive-permission hungry, and their web interface is a proxy to the rich functionalities they offer. Through the web interface, MAVs offer sensitive functions such as access to personal data backups, remotely taking pictures, remotely enabling the microphone, etc. By gaining access to a victim's account, an attacker could therefore remotely access personal information or lock the device to demand a ransom. To protect user accounts, all MAVs in our sample ask users to select a password at first run of the app.

Findings

We found *questionable authentication practices* (Table 6.1, column "Account Password"). For example, all MAVs accept short passwords – ranging from 4 (*Dr.web*) to 8 minimum characters (*Kaspersky*). Four of them do not accept special characters (*McAfee*, *Avira*, *TrendMicro* and *TrustGo*). *Avast* does accept special characters, but processes them somehow: when entering the password *hello"'@#%&*/-+()*, we could then log in with the truncated version *hello"'@#%*. More generally, all apps accept weak passwords, except *Kaspersky* which enforces the use a combination of uppercase, lowercase and numerals.

We hypothesise that in practice, it is difficult for MAVs to enforce strong passwords: as users rarely interact with MAVs or their corresponding web interface, they would inevitably forget their password if not easily memorable. Furthermore, if a password is set up on a phone, the keyboard limitations make it inconvenient to mix upper and lower case, let alone adding non-alpha characters.

We also found indications that certain web services may not store user credentials properly. When websites enforce a maximum password length, it is often indicative of bad storage practices – when stored hashed, passwords can be arbitrarily long [245]. Three MAVs (*McAfee*, *TrustGo*, and *Norton*) fall in this category (column "length" in Table 6.1). We were not able to verify improper credential storage for these apps though. We tried the "recover lost password" but this only provided a reset link, not a clear-text password. Nevertheless, there is no valid reason for MAVs to limit the length of passwords for more paranoid users.

Online rate limiting is a natural defensive measure against online guessing attacks. At the time of the study, we found that three products implemented it (*McAfee*, *Norton* and *Lookout*, column "online rate limiting" in Table 6.1). For *Norton*, the lockout period did not work when we tested it. By the time this work was first published as a conference paper, more MAV solutions had implemented rate-limiting in their web interface, but

some still failed to enforce it in within the app – so an attacker could reverse-engineer the protocol between the app and its server to brute-force the password. It is important to realise that while account locking might thwart an all-out targeted online guessing, a slower, distributed, throttled attack could still succeed [246]. Rate limiting and account locking also interact poorly with targeted smartphone theft: if, prior to stealing a device, an attacker can lock her victim's account (or render its access slower), she can prevent him from remotely locking or wiping the stolen device.

Response from MAVs

MAVs that responded acknowledged these findings. They generally pointed out that usable authentication is challenging. Therefore we think this is an area worth investigation in future research.

6.5 App Configuration & User Interface

Without admin privileges, MAVs cannot take advantage of built-in lock and wipe features; yet admin privileges must be granted explicitly by users. It is commonly accepted that it is hard for users to configure and use security software safely [247–250], and this can be even harder for small-screen devices. Therefore, it is important for MAVs to warn, guide, and inform users accordingly.

Findings

As shown in Table 6.1, only four MAVs warn users if they do not run as admin (column "in-app warning"). For *Avira*, if a user clicks the warning but does not subsequently grant admin privileges, the warning disappears for ever. *Norton*, *Avast*, and *TrendMicro* go further in the wrong direction: they display "Anti-Theft is on", "You are protected" and "Device now protected" respectively, either in the app or the online interface, even when apps do not run as admin. The column "in-app flow" (Table 6.1) refers to a MAV that automatically launches the Android Settings view for granting admin privileges, when a user visits the relevant "Anti-Theft feature" menu of the app. This reduces the chance of misconfiguration, since a user need not struggle with finding the relevant Settings option. Three apps take this approach. However, we found that if a user has granted admin privileges, but later decides to remove them, apps generally fail to notify "out-of-band" (i.e. outside the app – this is important as users do not interact with AV apps regularly and would miss in-app notifications).

Similarly, none of the MAVs warn users about app misconfiguration in the web interface either (column "web warning"). Furthermore, most of them *misinform users about the*

Table 6.1: MAV Implementations: General Findings

	User interface			Account Password				SSL	
	web warning	in-app warning	in-app flow	online rate limiting	length	special characters	only strong	certificate validation	pinning
AVG	✗	✗	✗	✗	$6 \leq l < \infty$	✓	✗	✓[2]	✗
Lookout	✗	✗	✗	1*hour* wait if 13 incorrect attempts[0]	$5 \leq l < \infty$	✓	✗	✓[2]	✗
Avast	✗	"Issues"[1]	✗	✗	$7 \leq l < \infty$	✓[0]	✗	✓[2]	✗
Dr.web	n/a	✗	✓	n/a	$4 \leq l < \infty$	n/a	n/a	n/a	n/a
Norton	✗	✗	✗	CAPTCHA and account locked[0] 1*hour*	$6 \leq l \leq 50$	✓	✗	✓[2]	✗
McAfee	✓[1]	orange flag	✓	reset password after 5 attempts	$8 \leq l \leq 32$	✗	✗	✓[3]	✗
Kaspersky	✗	"issues"	✓	✗	$8 \leq l < \infty$	✓	✓	✓[2]	✓
TrustGo	✗	✗	✗	✗	$6 \leq l \leq 15$	✗	✗	✓	✓
TrendMicro	✗	✗	✗	✗	$8 \leq l < \infty$	✗	✗	✗[2]	✗
Avira	✗	"Action Required!"	✗	✗	$5 \leq l < \infty$	✗	✗	✓	✗

✗means absence of protection, whereas ✓is the opposite. [0]implementation problems are reported in the text of the paper. [1]warning provided but in a specific tab. [2]use of http at registration time or startup.
[3] certificate validation during registration is broken.

results of a remote wipe and lock: they merely claim it is "successful" even if the phone app does not run as admin. We provide further details in Section 6.6 and Section 6.7 why the lack of admin privileges significantly reduces the reliability of anti-theft functions. A few solutions do give information in the web interface but only after a wipe command is issued, which is generally too late because this happens when the device is already lost.

Given the lack of guidance and warning provided by MAVs to users, it is likely that many users will run an insecure setup. In fact, during the course of this study, we realised the need to enable the "admin" privileges through reading Android API documentation. Throughout the following sections, we therefore take account of this observation and highlight several consequences.

Response from MAVs

MAVs that responded acknowledged these findings. They all say that they are taking actions to improve the UI of their app.

6.6 Lock Implementations and Effectiveness

One anti-theft option for users to protect their data on stolen devices is to remotely lock the phone screen, generally through a web page provided by MAVs. In the following sections, we assume that the Android Debug Bridge (adb) is disabled or protected in the device Settings (if not, then a thief can get an interactive shell and access user data as described in Section 6.2.1). As users may have improperly configured their MAV (Section 6.5), we study outcomes whether it runs as admin or not. The following sections present the many attack vectors we discovered during the study.

6.6.1 Removal of MAVs & API Misuse

Finding 1

When a MAV does not run as admin, it must resort to ad-hoc solutions to implement the screen lock, and must additionally ensure the app starts at device boot time. At boot time, there is a race condition, in that the custom lock screen should appear fast enough to prevent a thief from un-installing the MAV. We found that for four MAVs, the custom screen lock does not show up quickly enough and can therefore be un-installed (column "race protection" in Table 6.2). By rebooting a stolen device and winning the race, a thief can remove the app and prevents the owner from remotely protecting the device.

Table 6.2: MAV remote Wipe and Lock Functions

	Wipe			Lock						
	data partition	primary SD	secondary SD	lock	on-device rate limiting	un-install protection	admin lock protection	non-admin lock protection	counter reset protection	race protection
AVG	admin	admin,format, unlink	unlink[1]	user-selected 4-6-digit PIN or “alpha” password	✗	✗	bp:brute-force	bp:Safe mode	n/a	✓
Lookout	admin	file overwrite, unlink	file overwrite, unlink	random 4-digit PIN	$T = \frac{N}{3} \times 1min$[0]	✗	✓[0]	bp:Safe mode	bp:remove battery	✗
Avast	admin	unlink,format, admin	unlink,format	user-selected 4-6-digit PIN	✗	bp:GSM BTS	bp:GSM BTS	bp:Safe mode	n/a	✗
Dr.web	admin	admin,unlink	unlink	4-char password	✗	✗	bp:Safe mode	bp:Safe mode	n/a	✓
Norton	admin	admin	format or unlink[2]	random 4-digit PIN	optional wipe after 10 attempts	✗	✓	bp:Safe mode	✓	✓
McAfee	admin	admin,unlink, format[3]	unlink,format[3]	user-selected 6-digit PIN	$1hour$ wait after 10 attempts	bp:Safe mode	bp:Safe mode	bp:Safe mode	bp:remove battery	✗
Kaspersky	admin	unlink	unlink	user-selected 4-16-digit PIN	✗	n/a	built-in lock	✓	n/a	✓
TrustGo	admin	unlink	unlink	web password	$30min$ wait after 5 attempts	✗	bp:Safe mode	bp:Safe mode	✓	✓
TrendMicro	admin	format,unlink	format,unlink	web password	✗	✗	bp:Safe mode	bp:Safe mode	n/a	✗
Avira	admin	unlink	unlink,format	user-selected 4-digit PIN	unlock with web only after 3 attempts	✗	✓	bp:Safe mode	✓	✓

✗means absence of protection, whereas ✓is the opposite. “bp:method” means that the protection is present but can either be bypassed using “method” or its effectiveness reduced using “method”. Note that vendors’ customisations may allow bypass of properly implemented protections. [0]implementation problems are reported in the text of the paper.
[1]only if the app is not an admin and formatting of the internal (primary) SD card fails. [2]only if the app is not an admin. Format if $SDK >= 17$, *unlink* files otherwise. [3]formats if a tablet.

Finding 2

By design (Section 6.2.1), the Safe mode lets a thief un-install a MAV so long as the app does not have admin privileges. A race condition is no longer needed in this case, and all MAVs incorrectly configured by users (i.e. non-admin) are therefore vulnerable. Again, a user who remotely locks his screen is generally left clueless about the problem since MAV web interfaces do not provide details (Section 6.5).

Finding 3

An admin MAV should invoke the built-in screen lock, which eliminates race conditions and safe mode bypass on the condition that it requests the `force-lock` policy and calls the relevant APIs (e.g. *lockNow()*). We found that four MAVs *misuse the security API*, in that they do not enable the default screen lock even though they request the `force-lock` policy (*Dr.web*, *McAfee*, *TrustGo*, and *TrendMicro*). They can therefore be bypassed via Safe mode (Table 6.2, column "admin lock protection"). Although *Avast* properly enables the built-in lock, it can also be bypassed – we defer this discussion to Section 6.6.4.

Finding 4: Even if an admin MAV properly enables the built-in lock screen, a thief could un-install the app before a user has remotely locked his device – on the condition that he first manages to remove the admin privileges. We found that seven MAVs leave the removal of admin privileges unprotected (Table 6.2, column "un-install protection"). Only three apps take account of this attack vector. *Kaspersky* enables the built-in screen lock on first run of the app; the device remains locked at all time without the need for remote activation, and so is immune to this attack. *McAfee* and *Avast* prompt a thief with a PIN if he attempts to remove admin privileges. For *Avast*, the lock (including the anti-removal lock) can be bypassed, but we defer the discussion to Section 6.6.4. For *McAfee*, it *misuses the Android API* so its anti-removal lock can be circumvented by re-booting into Safe mode, and the app subsequently removed. *McAfee*'s code for handling the removal of admin privileges is illustrated in Fig. 6.2. It uses the callback function *onDisabled()* to be notified of admin changes: when this function is called by the Android framework, *McAfee* locks the screen. Unfortunately, on Gingerbread devices (v2.3.x), the *onDisabled()* function is called only *after* the admin privileges are disabled. As a result, the subsequent call to lock the screen cannot make use of the built-in lock screen and must be implemented by the app itself. Therefore, when rebooting into Safe mode, the lock screen does not show up since apps are disabled in this mode (Section 6.2.1). At this point, the app can also be removed as it no longer has admin privileges.

Further investigation reveals that the *onDisabled()* function is called *before* admin privileges are dropped on subsequent Android versions we tested (ICS (v4.0.x), Jelly Bean (v4.[1-3]). This problem is aggravated because *the Android documentation is incorrect.*

```
public class McAfeeReceiver extends DeviceAdminReceiver {

 public void onDisabled(Context paramContext,
                           Intent paramIntent){
  [...] // removed
  displayLockScreen();
 }
}
```

Figure 6.2: Code of McAfee's anti-removal screen.

More specifically, it is oblivious to differences between versions[1]. It only states that the *onDisabled()* function is "called prior to the administrator being disabled" without specifying the relevant Android versions. We used the Internet Archive[2] to trace changes to the API description. We found that the API documentation was also incorrect at the time when Gingerbread was the most recent Android version (Dec 2010 – Oct 2011). Another function, *onDisabledRequested()*, is called *before* admin privileges are dropped for all Android versions we tested. So it is more reliable, but MAV apps currently do not use it.

There are also usability problems associated with the protection of app removal. As users tend to forget passwords/PINs, a user who genuinely attempts to un-install a MAV may be presented with a (PIN) screen lock he cannot remove. Therefore MAVs must provide additional information in their lock screen, e.g. to guide users how to reset the PIN. But the *default Android lock screen has certain limitations*, in that it cannot be customised. As a result, additional information cannot be provided with the default lock screen, and MAVs resort to implementing their own lock screen; which we explained can be bypassed through race conditions or Safe mode. Android Lollipop (v5.0) provides a new function (*startLockTask()*[3]) for "screen pinning", that is, to "temporarily restrict users from leaving your task". However this would still be by-passable in Safe mode. Therefore, we think Android would benefit from a customisable lock screen.

Response from MAVs

Problems were generally acknowledged. One company argued that the Safe mode bypass of their lock screen was "low risk" because "third-party apps do not run in Safe mode". This is incorrect. First, a thief could read a user's emails manually and gain access to

[1] developer.android.com/reference/android/app/admin/DeviceAdminReceiver.html

[2] web.archive.org/web/20100501000000*/https://developer.android.com/reference/android/app/admin/DeviceAdminReceiver.html

[3] https://developer.android.com/about/versions/android-5.0.html

credentials for other websites – e.g. contained in emails or through a reset link sent to their Inbox. Second (Section 6.2.1), even though disabled in the user interface, apps can be installed and launched via a shell in Safe mode to automate the process.

One company argued that they do not enable the built-in lock because it may violate the Google policy. This states that “an app downloaded from Google Play [...] must not make changes to the user’s device outside of the app without the user’s knowledge and consent”. We told them we disagreed because when remotely locking their device, a user is implicitly giving consent.

Regarding the lack of un-installation protection, we have received feedback from few developers. In fact, many tested apps did not have a proper contact/email for bug reports. We had to use the “customer support” portal instead. Customer support assured us they would forward the bug report to the engineers but we have not heard from them in most cases.

6.6.2 Rate Limiting

Finding 1

Admin MAVs that properly enable the built-in screen lock generally also overlay their own lock on top of the default lock screen. This is used to customise the look-and-feel or offer additional information. For example, MAVs sometimes want to provide an email address to contact the phone owner if the device is lost. This overlay is also a source of problems. The built-in screen lock, on most devices, enforces a 30*sec* wait period after 5 failed PIN attempts. Unfortunately, half the MAVs do not have rate limiting protection on their custom screen-lock (column “on-device rate limiting”), thereby annihilating the protection of the built-in one. This makes brute force practical since MAVs accept common weak PINs like “1111” or “1234”. Only *Kaspersky* warns users to use at least 7 digits, yet it still accepted weak PINs. This issue again highlights *a limitation of the default lock screen*, in that it cannot be customised enough, therefore MAVs have tried to find ways around this. This suggests that the default lock screen would benefit from an additional admin “lock rate limiting” policy for apps to express requirements such as “enforce an X sec wait after Y successive incorrect PINs”.

Finding 2

Some MAV apps do enforce stronger rate-limiting policies than the built-in default (column “on-device rate limiting”). For *Lookout* however, we found that it could be circumvented on our Galaxy S Plus even when the app runs as admin: by clicking the Home button, a thief can navigate away from the *Lookout*’s custom screen lock, and benefit from the lower rate limiting of the built-in one. In Safe mode, a non-admin MAV’s custom screen lock

does not show up, so if the built-in lock has less protection, a thief can use that instead – on certain devices such as the Samsung Galaxy S Plus, the built-in screen has no rate limiting.

Finding 3

In order to enforce rate-limiting after too many incorrect PIN attempts, MAVs must store a retry counter. For *Lookout*, we found that removing the battery resets the retry counter, and for *McAfee*, it decreases it by one. This attack is not always practical, as a thief must reboot the phone but the boot time may exceed the wait period enforced by a MAV. In *Lookout*'s case, it significantly decreases the wait period – which increases linearly with the number of incorrect PIN attempts (Table 6.2, column "on-device rate-limiting").

Finding 4

If an attacker knows the username of his victim, he could also leverage the lack of rate limiting in the online interface to unlock the screen remotely (Section 6.4). This is mostly true in targeted attacks. In non-targeted ones, it may still be possible because some MAVs, by default, display the owner's email address in their custom lock (e.g. *Avira* and *TrendMicro*). This unintentionally leaks the username necessary to perform an online dictionary attack.

6.6.3 MAV Response

The findings were acknowledged. One company responded to "Finding 4" by masking some of the characters in the email address.

6.6.4 Network-level Attacks

Finding 1

Avast, downloaded more than 100M times on Google Play, has an option (including in its "anti-removal" lock) that lets thieves send temporary unlock PINs via SMS to a (compulsory) contact pre-configured by the genuine device owner. This function is accessible in the custom screen lock. When used, the stolen device sends a temporary PIN via SMS – in clear – to a trusted contact. It is known that GSM only authenticates a phone to the network, but not the network to the phone. Therefore it is possible to have the phone connect to a rogue GSM station – a.k.a. Base Transceiver Station (BTS) – to intercept the SMS. Because phones usually fall back to GSM in the absence of 3G, a thief

could set up a rogue BTS to intercept the temporary PIN and bypass the lock screen to remove the app. Other interception techniques are presented in Section 2.3 of Chapter 2.

Similarly, *Dr.web* locks/unlocks devices through commands sent via SMS from a trusted contact. If the SMS sender is a pre-configured trusted "buddy", a password is not required (this is enabled by default). In a targeted scenario where a trusted buddy's phone number is known, a thief could spoof the SMS's sender ID to bypass the password. One could use online services, a rogue BTS station, or a femtocell.

Finding 2

Recall from Section 6.2.2 that MAVs generally keep a TCP connection to a server in order to receive remote commands to be executed on the device. An attacker who can impersonate as a MAV server could therefore send an unlock command to the phone app. MAVs generally use TLS to secure their connection and properly verify their server's certificate but do not implement pinning (Table 6.1, column "SSL"). *TrendMicro*, however, did not properly validate the CN of the certificate. It accepted any domain (i.e. CN) so long as it was signed by a trusted CA. *McAfee* improperly authenticated its server during account registration: the app accepted selfsigned certificates as well as certificates signed by a trusted CA but for a different domain (i.e. CN). Even though they used TLS, *Avast*, *AVG*, *Lookout*, *Norton*, *Kaspersky*, and *TrendMicro* additionally made some non-encrypted requests. Instead of maintaining a TCP connection to their server, certain MAVs such as *Avira* re-use the Google connection and push commands to the phone through Google push APIs. We found that the Google service did not implement certificate pinning. On a side note, *TrustGo*'s user web interface used a combination of http and https, making cookie recovery trivial for a network attacker.

MAV Response

The latest version of their apps fixes certificate validation issues – not necessarily pinning. We have not received feedback regarding unprotected PINs and commands sent via carriers' networks (Finding 1).

6.6.5 Vendor and Customised Android Failures

Even if anti-theft functions are properly implemented by MAVs, customisations by vendors may allow an attacker to bypass their protection.

Finding 1

We refer to "charging mode" as the mode in which a device is, when it is switched off and plugged into a power supply. We found that certain phones boot up a headless Android and expose a debugging interface when booted in charging mode. For example, the LG L7 running Jelly Bean (v4.1.2) exposes adb: the data partition is mounted and one folder is writeable, so an attacker could push code and exploits. The Samsung Galaxy S Plus also exposes adb but we did not have enough permissions to push files; yet it remains a concern.

Finding 2

On phones with open Bootloaders, a thief can install custom upgrade: this bypasses admin MAV screen locks and the built-in one. Thieves could therefore install forensic software to recover personal data. Protected Bootloaders may also contain logic errors or vulnerabilities[1,2]. We found that on the Google Nexus running Android v4.2, unlocking the Bootloader did not erase the data partition; on the HTC Desire C, a Bootloader unlock wiped only the data partition; and on the HTC Desire S, there was a Bootloader option to read arbitrary files. When booted in Recovery mode, the Samsung Galaxy S2 also exposes adb, and the LG L7 further has a writeable folder. The latter even warns "Using this mode, service center can back up and recover your data".

Finding 3

Users who want more control over their device may install a custom Recovery from which they gain additional functionalities, such as root access, the ability to install custom Android builds, custom UI themes, etc. As the Recovery has unrestricted access to Android binaries and data partitions (Section 6.2.1), it is important to prevent unauthorised users (i.e. thieves) from accessing it. We found this is not always the case. For example, prevalent Recoveries such as CWM[3] or TWRP[4] often expose unprotected options that allow flashing arbitrary software via USB even when the screen is locked in the Android OS. In our tests, we could push arbitrary Android updates via the following options. The "zip updates" option accepts arbitrary updates from files present on the primary SD card, so an attack is feasible on phones with a removable primary SD card. The "adb sideload" accepts updates sent in-band via USB. Newer Android versions protect adb-via-USB using a

[1] `www.codeaurora.org/projects/security-advisories/fastboot-boot-command-bypasses-signature-verification-cve-2014-4325`

[2] `www.codeaurora.org/projects/security-advisories/incomplete-signature-parsing-during-boot-image-authentication-leads`

[3] `www.clockworkmod.com`

[4] `www.teamw.in/project/twrp2`

public-key authentication scheme tied to the owner's computer (though bypassable on certain devices[1,2]). This security feature was not enforced in the custom recoveries we tested: adb generally remained available and unprotected even when not enabled in the Android Settings, and the signature verification on updates could be toggled off manually.

Finding 4

Vendors also have desktop software that one can use to backup or transfer files from a smartphone via USB (like Samsung Kies or HTC Sync Manager). We found that the activation of the HTC Sync Manager automatically enables the Android Debug Bridge (adb) on the device – probably because it piggy-backs on its protocol. Some HTC devices keep adb on even when the screen is locked, allowing a thief to get an interactive shell on the device via USB. We leave a comprehensive security analysis of vendors' software for future research.

6.6.6 Misc

Android has in the past been victim of PIN bypass vulnerabilities that let a thief access partially, or fully, a device's user interface and data. It may be because of third-party apps[3][4] or vendor customisations[5]. These were out of scope of this study.

Even when a lock cannot be bypassed, current Android phones allow a thief to Factory Reset the device from the Recovery or Bootloader mode. This deletes the data, but not always securely (Chapter 5). Therefore, a thief could alternatively Factory Reset a device and use forensic tools to recover insecurely-deleted data on devices with a flawed Factory Reset function. This may change in newest Android devices with the "kill-switch" function introduced to curb phone theft in certain US states. This was out of scope of our study.

Some HTC devices have a DIAGnostic that can be accessed via specially-formatted SIMs or SD cards known as "gold cards". It gives access to vendors' "repair menus". On our devices, data was wiped when entering the menu, but this may not be the case on other models.

We discuss hardware attacks and baseband attacks in Section 6.8.

[1] `labs.mwrinfosecurity.com/advisories/2014/07/03/android-4-4-2-secure-usb-debugging-bypass`

[2] `github.com/secmobi/BackupDroid`

[3] `seclists.org/fulldisclosure/2013/Jul/6`

[4] `www.bkav.com/top-news/-/view_content/content/46264/critical-flaw-in-viber-allows-full-access-to-android-smartphones-bypassing-lock-screen`

[5] `shkspr.mobi/blog/2013/03/new-bypass-samsung-lockscreen-total-control`

6.6.7 Encryption to the Rescue

The use of Full Disk Encryption (FDE) has the potential to mitigate many of the attacks when the phone is powered down, so long as the encryption key cannot be recovered. On Android devices that support FDE, the encryption key is stored encrypted with a key derived from a salt and a user-provided PIN (or password)[1]. This encrypted blob is referred to as the "crypto footer" in the AOSP source code. We found that if the PIN entered by a user is invalid, Android reads a retry counter from the crypto footer, increments it and writes it back. By monitoring power consumption, an attacker could detect the write back, and cut the power supply to circumvent the write[2]. This attack was well known in the smartcard community in the 1990s; by fifteen years ago, it was standard practice to increment the retry counter before each PIN prompt and reset it only if a correct PIN is entered [251].

Another problem stems from the "Fast boot" option on HTC devices. It is enabled by default, and when powered down, a device transitions into a hibernation/sleeping state so as to speed up boot time. When powered up, the PIN is not prompted so it may be stored in RAM or on disk. Failure to disable the fast boot option may undermine the benefit of FDE. We leave this for future research.

6.7 Wipe Implementations

6.7.1 General Results

The second option provided by anti-theft functions is the "remote wipe". We study MAV wipe implementations with admin privileges granted or not.

Admin MAVs

Table 6.2 (column "data partition") shows that all MAVs use the admin API to sanitise the data partition. For external storage though, implementations differ.

Half of them use the admin API to sanitise the primary SD card (the internal SD card on the test Galaxy S Plus). If the sanitisation fails, they implement "backup" mechanisms. For instance, *AVG*, *Avast*, and *McAfee* format the partition and *unlink* all files; this does not provide logical sanitisation.

The other half of MAVs *misuses the Android security API* in the sense that they do not pass the flag *WIPE_EXTERNAL_STORAGE* to sanitise external storage, and

[1] Newer Android versions make use of TrustZone to improve security

[2] Limitations: (*i*) a full kernel is running with certain userspace processes, (*ii*) the OS may cache (and therefore delay), the write-back to non-volatile storage, (*iii*) rebooting the device may be too slow.

Table 6.3: Wipe Implementations without admin privileges

	result timing	disable data sync	media	SMS/MMS	contacts	accounts	account password	browser history	browser credentials	call logs	WiFi	cache	VPN	own data
AVG	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗
Lookout	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✗	✓	✗	✗
Avast	✓	✗	✓	✓	✓	✗	✗	✓	✗	✓	✗	✗	✗	✗
Dr.web	n/a	✗	✗*	✓	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗
Norton	✓	✗	✗*	✓	✓	✗	✗	✓	✗	✓	✗	✗	✗	✗
McAfee	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗
Kaspersky	✗	✓	✗*	✓	✓	✗	✓	✗	✗	✓	✗	✗	✗	✗
TrustGo	✗	✗	✗*	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗
TrendMicro	✗	✗	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✓
Avira	✓	✗	✗*	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

*data is nevertheless removed by deleting the corresponding files on the SD card

this even when apps have been granted admin privileges. Instead, they resort to ad-hoc mechanisms: they generally format the partition and/or *unlink* files; this does not provide logical sanitisation. We think the misuse of the Android API may be caused by *misleading/outdated Android documentation*: it states that the flag parameter to the *wipeData(int flag)* function "must be 0"[1].

The Android API currently *does not expose an API to securely sanitise the secondary SD card*. Therefore, all MAVs implement their own mechanisms to sanitise it, as a result of which data is always recoverable. We note that even when used properly, the admin API does not ensure that logical sanitisation is available on the device (Chapter 5).

Non-Admin MAVs

When MAVs do not run as admin, they must resort to other mechanisms to sanitise partitions, e.g. by using Android APIs (Section 6.2.2). Recall that this situation is likely to arise because of UI issues highlighted in Section 6.5. In this case, MAVs attempt to delete data as shown in Table 6.3; but this does not provide logical sanitisation. Furthermore, we found that none of the MAVs manage to remove the primary account using Android APIs. Some of them try to, but fail because the OS does not seem to allow it: this failure is never reported to a user in the online interface. Android *does not appear to expose an API to delete browser, WiFi and VPN credentials*, and no MAVs manage to delete them.

A thief could also attempt to recover MAV credentials on the phone in order to access

[1] `https://developer.android.com/guide/topics/admin/device-admin.html`

online backups made by a user. We found that only *TrendMicro* attempts to delete its "own data" (Table 6.3).

We also found that three MAVs display success for a remote wipe in their web interface, even if the wipe has yet to be performed (Table 6.3, column "result timing"). This is because on reception of the wipe command from their server, MAV apps acknowledge it: some web interfaces interpret this as wipe success. All apps, to some extent, *misinform users about wipe results*, merely claiming that a remote wipe succeeds when errors occur, like when an exception is thrown, when formatting fails, or when a file cannot be *unlink*ed. They all fail to give accurate details about the wipe results in general. *McAfee*, *Avast*, and *Avira* fare slightly better because they indicate that the wipe cannot use the built-in Factory Reset if admin privileges are not granted. But this is too late when the device is already lost.

6.7.2 Case Studies of Lookout and Avast

Lookout and *Avast* both have unique implementations for sanitising external storage. *Lookout* sanitises external storage by overwriting files before *unlink*ing them. Its developers implicitly assume that the file system overwrites files "in-place". We tested this hypothesis by creating 1000 files on the primary SD card of the Galaxy S Plus (FAT-formatted). We overwrote the files, and found that more than 90% of them were not updated "in-place" by the file system – and thus were recoverable. We ran the same experiment with *Lookout*'s remote wipe, and obtained similar results.

Avast has a special option called "thorough wipe" to sanitise external storage. This creates a 1MB file and overwrites it 1000 times with 0*s* (all values are hardcoded in the app). Its developers implicitly assume that the file system does not update files "in-place", which would ensure that 1GB of space is overwritten. We showed this to be mostly true on the Galaxy S Plus's primary SD card formatted with FAT. However, on an ext4-formatted partition, we found that more than 99% of files were updated "in-place" on an HTC One S. Hence, mostly the same blocks, spanning 1MB, are overwritten. Furthermore, all devices we have encountered, have at least 2GB of primary SD card, while some have up to 25GB (HTC One X). As a result, even if 1GB of data was indeed overwritten, it would not represent the entire partition. Note that the ext4 filesystem may also use compression which will further reduce the amount of data overwritten.

These two examples highlight how *the security permission model* can sometimes backfire and have a negative impact on security: because third-party apps cannot legitimately gain high privileges (i.e. root), they cannot overwrite an entire partition bit-by-bit, and therefore they resort to unreliable methods to mitigate a flawed Factory Reset (Chapter 5).

MAV Response

We have received feedback only from one of the two companies. The dev team said they are trying to improve the reliability of remote wipe without impacting usability.

6.7.3 Inherent Problem of Remote Wipe

We found an inherent timing problem between the time of wipe and the time of success displayed to users in the web interface: We call this problem the "Time-Of-Wipe-Time-Of-Success (TOWTOS)" attack. Concretely, this occurs when a wipe completion is reported to the user in the web interface before the wipe effectively takes place on the phone, and the wipe subsequently fails. The failure could be caused by the removal of the battery by a thief, by the phone being shutdown, or by a TLS truncation attack [252] by a thief attempting to desynchronise the phone app and the server. This is currently inevitable for a MAV that performs an admin wipe because it must report to the user before it is itself wiped. *Lookout* aggravates this problem by launching a 10s timer before launching an admin wipe. The TOWTOS highlights *another limitation of the current Android admin API*, in that it is not possible for an app to report results to its web service after a wipe completes.

6.8 Discussion

We have highlighted three main failures in remote data protection functions: (*i*) the misuse of Android security APIs by MAVs, (*ii*) the limitation of Android APIs and permission model, (*iii*) vendor customisation issues. One attack we discarded till now is the use of Faraday bags. There are two lines of thought to mitigate it. The first involves an online server that is contacted during user authentication to a device, the approach is taken by CleanOS [253], KeyPad [254] and other theft-resilient solutions [255]. However, these can severely impact usability by (*i*) degrading responsiveness of the device during authentication when there is poor network coverage, and (*ii*) by locking a genuine user out if a server is DoSed, unavailable, or if there is no network coverage.

Therefore, solutions that solely rely on the user and device seem the only viable options. Authentication based on user behaviour [256], swipe characteristics [257], context [258], typing characteristics [259] and app usage [260] are appealing but usually lack accuracy and speed as well as increasing power consumption. Solutions based on an extra device users carry [23] often impose extra cognitive load and are seen as a burden. Biometrics such as Apple's fingerprint are relatively fast and effortless, and therefore appealing to average users on high-end devices.

Overall, we think that device-based solutions are more likely to offer usable protections against the attacks described in this paper. More importantly, given the limitations imposed by the Android API and the permission model, we think the only viable solutions are those driven by vendors themselves. They, only, can integrate their solutions seamlessly whilst taking full advantage of the platform and hardware features. The rise of wearable computing may also enable better authentication mechanisms if battery life improves; this is another avenue for future research to consider.

6.9 Summary

We studied consumer-grade protections against unauthorised access to personal data on stolen Android smartphones. We further highlighted the market and technical fragmentation which means that there is no consistent security guarantee across devices.

We unveiled critical failures on mobile antivirus remote lock and remote wipe functions. In addition to the limitations imposed by the Android security model and APIs, these are caused by questionable MAV designs and vendor customisations.

Chapter 7

Conclusions

Through this work, we explored how intrinsic characteristics of the smartphone ecosystem enable new avenues for attack. We focused our work around the four following axes.

The first axis revolves around built-in sensors and peripherals. In Chapter 3, we continued an already existing line of research that explores the impact of built-in sensors on security. While previous work focused on accelerometer-based side channels to infer PINs entered by users, we showed that the built-in camera and microphone are also a source of leakage during PIN input; and they can be abused to infer PINs too. This chapter showed that all shared resources need to be carefully considered because they open up the possibility of side-channel attacks on both one-OS and multiple-OS smartphones. We hope this study raises awareness of the difficulty of designing a sound trusted path in general. Designers must be aware of side-channel risks and engineer the overall system accordingly. On smart OSes, reasoning about the security of a trusted path becomes even more complex as new features and services are added over time. We provided OS-level guidelines to prevent side channels based on sensors and peripherals being exploited during user input. The guidelines roughly say that when a user performs a sensitive action, any incoming data should be disabled, and the same policy should be enforced on connected wearable devices. To improve our results, future research could better incorporate the a-priori probability distribution of PINs, inter-key measurements, and other sensors' data. One could also investigate different supervised algorithms, monitor PIN input several times, or try to detect touch events without the microphone (i.e. from the video alone). Another approach could be to adapt this work for pattern lock inference.

The second axis revolves around attacks caused by smartphones' particular form factors. These are small touch-screen devices where usability can be an issue when entering text. In Chapter 4, we focused our attention on a recent input method known as "gesture typing" that was designed to ease user input on smartphones. We studied how an attacker could abuse the system-wide screen hardware counter and software counter to infer what users type on their phone through the gesture typing feature. To the best of our knowledge, this work is the first to leverage global information exposed by procfs to do this. This

goes against the general belief that non-app-specific information exposed through virtual files is harmless. The attack also applies to the latest Android version where app-specific virtual files, repeatedly shown harmful by previous research, are inaccessible. Unlike the attack studied in Chapter 3, we think this one could have been prevented. There have been numerous papers showcasing the danger of unprotected procfs. This line of research started more than ten years ago when people realised that procfs exposed each application's current instruction pointer. In 2009, Zhang and Wang [187] exploited the exposure of the stack pointer to conduct a keystroke-dynamics attack on password input. Such attacks have attracted interest again recently in the context of Android. But by ignoring previous warnings and by not adhering to the principle of least privileges, Android designers are enabling potential attacks that could be easily prevented. This demonstrates a lack of a principled security analysis during the OS design.

The third axis revolves around the personal aspect of smartphones. They have become ubiquitous and we carry them everywhere we go. Hence, we are likely to lose them. In Chapter 6, we studied anti-theft software for Android. Among the vulnerabilities we discovered, we highlighted the misuse of Android security APIs by mobile antivirus apps, certain limitations of the current Android APIs, vendor-introduced issues, and the lack of proper API documentation. We think most of these issues could have been prevented if testing had been done more thoroughly. This, again, shows a lack of a principled approach to security in industry today. Functionality testing is usually performed intensively, but not adversarial testing. In the case of anti-theft software, we think that functional and adversarial testing are near identical since the core function of these apps is to enhance security. Yet, little testing seems to have been done by developers. Future research on anti-theft solutions could investigate in more detail the level of security provided by smartphones when capable attackers have physical access to them. This could encompass new attack vectors introduced by HTC's "Fast boot" option, by device-management vendor software, and by the new "kill-switch" introduced in the USA to curb smartphone theft.

The fourth axis we studied revolves around the fast pace at which people change their device (≈ 2 years). Already 60% of users trade in their old device to amortise the purchase of the new one. In Chapter 5, we studied the effectiveness of Android Factory Reset used by people to sanitise their device before selling it. We showed that one version of Android was particularly affected as it did not erase sensitive data. We also reported several issues in other versions, generally caused by deficiencies in upgrades pushed by vendors. Any vulnerability introduced by Google has long-lasting repercussions because vendors still fail to push upgrades promptly, and sometimes they even release new products running older Android versions (e.g. smartTVs). We think the problems in Factory Reset highlight two issues. First, the maintenance of smart devices in a fast-changing technology environment is prone to errors. In fact, early Android versions that used raw flash properly erased non volatile memory storage. But when the market moved to the use of eMMC, a vulnerability was introduced during the port. Second, Factory Reset problems again demonstrate a

lack of thorough adversarial testing before product release. We consider that the Factory Reset problems could have been prevented. Future research on device sanitisation should continue to investigate the level of security provided by smartphones' built-in sanitisation functions, to see whether the situation improves following the disclosures we reported. It could also investigate the level of security provided by these, i.e. whether they provide digital sanitisation or not. Another avenue to investigate is how to provide a Factory Reset that also re-initialises the code partition to thwart malware-infected phones. Restoring an infected device to a pristine state could become important in the future. There have been cases of malware that gains root access being found on both Google Play[1] and third-party app stores[2] recently. Backdooring resold devices or infiltrating the supply chain is also likely to become a threat in the future. In January 2014, it was hypothetised that a newly-discovered android bootkit was inserted into devices by retailers in an IT mall in Zhongguancun, Beijing[3]. Later in March, the Marble security firm claimed having discovered a pre-installed malicious NetFlix app in phones from various vendors[4]. One month later, Kaspersky also detected malware supposedly pre-installed with a kit sold by Chinese company Goohi[5]. Other companies like mSpy[6] also sell smartphones from various vendors that come pre-loaded with spyware. Apple's solution to this is to provision a phone's software through iTunes on an Apple computer (the device must be plugged into the computer via USB). It is not clear if this would work for Android devices. First, the procedure assumes a trusted computer, but many machines affected by malware run Windows. So if the computer is infected, would a user believe the computer or the phone if they display inconsistent messages? Would this lead to social-engineering attacks? Second, if the phone malware has taken over the device, the phone might attempt to install malware on the computer, via (*i*) social engineering, (*ii*) by exploiting Windows autorun/autoplay features [261], by exploiting USB driver vulnerabilities, or simply by masquerading as a Human Interface Driver (HID, e.g. a keyboard) [261] to gain code execution on the computer. All these scenarios must be carefully considered in a non-"walled garden" environment such as Android. Another avenue to address the code sanitisation problem could be a special hardware button on phones to reflash the OS without the need for a computer. But how would one educate people to push this button? More research is needed if we are to do this properly.

To summarise, we have explored those inherent characteristics of smartphones that can be abused to attack them. Out of the four attacks we studied, we think three could have been prevented. All the vulnerabilities we exploited results from not following best security

[1]https://blog.lookout.com/blog/2016/01/06/brain-test-re-emerges/

[2]https://blog.lookout.com/blog/2015/11/04/trojanized-adware/

[3]http://blogs.360.cn/360mobile/2014/01/17/oldboot-the-first-bootkit-on-android/

[4]http://www.pcadvisor.co.uk/news/security/3505208/pre-installed-malware-turns-up-on-new-phones/

[5]https://www.securelist.com/en/blog/208213028/Caution_Malware_pre_installed

[6]http://www.mspy.com/spy-phone/

practice, and demonstrate the lack of a systematic and principled approach to security design and testing in the industry today. The mobile space provides an increasingly rich list of failures at every stage of the development process: design, conception, testing, maintenance, documentation, etc. This can only lead to ever more attacks.

How can we curb this trend towards complacency in security? Better tools should be developed to help design and testing. More importantly, we must research how to incentivise people and corporations to do the right thing. More research is needed to help organisations better structure their development cycle to minimize mistakes whilst remaining competitive and reactive. For example, should we advise companies to have a dedicated security group to test products (such as Google's Project Zero)? Or do we advise companies to have a security-savvy person affiliated to each product to assist other developers through the development cycle? Would this scale? Shall we try to sensitise all developers to security as Microsoft tried in 2003? And what would be the best approach? Learning takes time and effort: so how could we make it easier for developers to stay up-to-date with security theory and practice? Given the pace at which companies must develop products to remain competitive and innovative, a better understanding of how to best structure the security development life cycle appears necessary.

Bibliography

[1] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, “Execute this! analyzing unsafe and malicious dynamic code loading in android applications,” in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[2] “Google report - android security 2014, year in review.” `https://source.android.com/devices/tech/security/reports/Google_Android_Security_2014_Report_Final.pdf`.

[3] Alcatel-Lucent, “Mobile malware: A network view,” in *BlackHat Mobile Security Summit*, 2015.

[4] W. A. Jansen, S. Gavrila, V. Korolev, R. Ayers, and R. Swanstrom, *Picture password: a visual login technique for mobile devices.* US Department of Commerce, National Institute of Standards and Technology, 2003.

[5] S. Uellenbeck, M. Dürmuth, C. Wolf, and T. Holz, “Quantifying the security of graphical passwords: The case of android unlock patterns,” in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[6] Z. Zhao, G.-J. Ahn, J.-J. Seo, and H. Hu, “On the security of picture gesture authentication,” in *USENIX Security Symposium*, 2013.

[7] S. Wiedenbeck, J. Waters, L. Sobrado, and J.-C. Birget, “Design and evaluation of a shoulder-surfing resistant graphical password scheme,” in *International Working Conference on Advanced Visual Interfaces (AVI)*, 2006.

[8] A. J. Aviv, K. Gibson, E. Mossop, M. Blaze, and J. M. Smith, “Smudge attacks on smartphone touch screens,” in *USENIX Workshop on Offensive Technologies (WOOT)*, 2010.

[9] S. Schneegass, F. Steimle, A. Bulling, F. Alt, and A. Schmidt, “Smudgesafe: Geometric image transformations for smudge-resistant user authentication,” in *ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, 2014.

[10] A. Studer, T. Passaro, and L. Bauer, "Don't bump, shake on it: The exploitation of a popular accelerometer-based smart phone exchange and its secure replacement," in *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[11] N. Karapanos, C. Marforio, C. Soriente, and S. Capkun, "Sound-proof: Usable two-factor authentication based on ambient sound," *arXiv preprint arXiv:1503.03790*, 2015.

[12] Y. Lee, "Real-time mobile gesture-based authentication." `http://www-ipv4.cl.cam.ac.uk/~lmrs2/student_projects/2014_Jason_SigVerify_partII.pdf`, 2014.

[13] P. Saravanan, S. Clarke, D. H. P. Chau, and H. Zha, "Latentgesture: active user authentication through background touch analysis," in *International Symposium of Chinese CHI (Chinese CHI)*, 2014.

[14] C. Bo, L. Zhang, X.-Y. Li, Q. Huang, and Y. Wang, "Silentsense: silent user identification via touch and movement behavioral biometrics," in *Annual ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2013.

[15] H. Xu, Y. Zhou, and M. R. Lyu, "Towards continuous and passive authentication via touch biometrics: An experimental study on smartphones," in *Symposium on Usable Privacy and Security (SOUPS)*, 2014.

[16] A. De Luca, A. Hang, F. Brudy, C. Lindner, and H. Hussmann, "Touch me once and i know it's you!: implicit authentication based on touch screen patterns," in *ACM Conference on Human Factors in Computing Systems (CHI)*, 2012.

[17] M. Frank, R. Biedert, E.-D. Ma, I. Martinovic, and D. Song, "Touchalytics: On the applicability of touchscreen input as a behavioral biometric for continuous authentication," *IEEE Transactions on Information Forensics and Security*, 2013.

[18] T. Feng, Z. Liu, K.-A. Kwon, W. Shi, B. Carbunar, Y. Jiang, and N. K. Nguyen, "Continuous mobile authentication using touchscreen gestures," in *IEEE International Symposium on Technologies for Homeland Security (HST)*, 2012.

[19] N. Sae-Bae, K. Ahmed, K. Isbister, and N. Memon, "Biometric-rich gestures: a novel approach to authentication on multi-touch devices," in *ACM Conference on Human Factors in Computing Systems (CHI)*, 2012.

[20] Z. Sitova, J. Sedenka, Q. Yang, G. Peng, G. Zhou, P. Gasti, and K. S. Balagani, "HMOG: A new biometric modality for continuous authentication of smartphone users," *IEEE Transactions on Information Forensics and Security*, 2015.

[21] M. Jakobsson, E. Shi, P. Golle, and R. Chow, "Implicit authentication for mobile devices," in *USENIX Summit on Hot Topics in Security (HotSec)*, 2009.

[22] O. Riva, C. Qin, K. Strauss, and D. Lymberopoulos, "Progressive authentication: Deciding when to authenticate on mobile phones," in *USENIX Security Symposium*, 2012.

[23] F. Stajano, "Pico: no more passwords!," in *International Workshop on Security Protocols (SPW)*, 2011.

[24] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[25] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[26] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Symposium on Usable Privacy and Security (SOUPS)*, 2012.

[27] P. G. Kelley, S. Consolvo, L. F. Cranor, J. Jung, N. Sadeh, and D. Wetherall, "A conundrum of permissions: installing applications on an android smartphone," in *International Conference on Financial Cryptography and Data Security (FC)*, 2012.

[28] H. Fu and J. Lindqvist, "General area or approximate location?: How people understand location permissions," in *Workshop on Privacy in the Electronic Society (WPES)*, 2014.

[29] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications," in *USENIX Security Symposium*, 2013.

[30] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[31] Z. Xie and S. Zhu, "Appwatcher: Unveiling the underground market of trading mobile app reviews," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2015.

[32] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang, "Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing," in *ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, 2012.

[33] C. Amrutkar and P. Traynor, "Short paper: rethinking permissions for mobile web apps: barriers and the road ahead," in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012.

[34] L. Yang, N. Boushehrinejadmoradi, P. Roy, V. Ganapathy, and L. Iftode, "Short paper: enhancing users' comprehension of android permissions," in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012.

[35] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, D. Wagner, *et al.*, "How to ask for permission," in *USENIX Summit on Hot Topics in Security (HotSec)*, 2012.

[36] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. android and mr. hide: fine-grained permissions in android applications," in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012.

[37] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.

[38] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: trading privacy for application functionality on smartphones," in *ACM Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2011.

[39] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on android)," in *International Conference on Trust & Trustworthy Computing (TRUST)*, 2011.

[40] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, "User-driven access control: Rethinking permission granting in modern operating systems," in *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[41] J. Sellwood and J. Crampton, "Sleeping android: The danger of dormant permissions," in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2013.

[42] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang, "Upgrading your android, elevating my malware: Privilege escalation through mobile os updating," in *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[43] Z. M. Defense, "Stagefright: Scary code in the heart of android," in *BlackHat Convention (BH)*, 2015.

[44] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2011.

[45] Y. Z. X. Jiang, "Detecting passive content leaks and pollution in android applications," in *Network and Distributed System Security Symposium (NDSS)*, 2013.

[46] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace, "Hare hunting in the wild android: A study on the threat of hanging attribute references," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[47] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *USENIX Security Symposium*, 2011.

[48] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on android security," in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[49] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *USENIX Security Symposium*, 2013.

[50] S. H. Kim, D. Han, and D. H. Lee, "Predictability of android openssl's pseudo random number generator," in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[51] J. S. Marcus Niemietz, "Ui redressing attacks on android devices," in *BlackHat Convention (BH)*, 2012.

[52] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu, "Towards discovering and understanding task hijacking in android," in *USENIX Security Symposium*, 2015.

[53] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, "What the App is That? Deception and Countermeasures in the Android User Interface," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[54] F. Roesner and T. Kohno, "Securing embedded user interfaces: Android and beyond," in *USENIX Security Symposium*, 2013.

[55] C. Mulliner, "Vulnerability analysis and attacks on nfc-enabled mobile phones," in *International Conference on Availability, Reliability and Security (ARES)*, 2009.

[56] R. Verdult and F. Kooman, "Practical attacks on nfc enabled cell phones," in *International Workshop on Near Field Communication (NFC)*, 2011.

[57] C. Miller, "Exploring the nfc attack surface," in *BlackHat Convention (BH)*, 2012.

[58] M. Naveed, X. Zhou, S. Demetriou, X. Wang, and C. A. Gunter, "Inside job: Understanding and mitigating the threat of external device mis-bonding on android," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[59] S. Demetriou, X. Zhou, M. Naveed, Y. Lee, K. Yuan, X. Wang, and C. A. Gunter, "What's in your dongle and bank account? mandatory and discretionary protection of android external resources," in *Network and Distributed System Security Symposium (NDSS)*, 2015.

[60] J. L. E. . C. Kasmi, "You don't hear me but your phone's voice interface does," in *Hack In Paris (HIP)*, 2015.

[61] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee, "Jekyll on ios: When benign apps become evil," in *USENIX Security Symposium*, 2013.

[62] T. Wang, Y. Jang, Y. Chen, S. Chung, B. Lau, and W. Lee, "On the feasibility of large-scale infections of ios devices," in *USENIX Security Symposium*, 2014.

[63] L. Xing, X. Bai, T. Li, X. Wang, K. Chen, X. Liao, S.-M. Hu, and X. Han, "Unauthorized cross-app resource access on mac os x and ios," *arXiv preprint arXiv:1505.06836*, 2015.

[64] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *Network and Distributed System Security Symposium (NDSS)*, 2012.

[65] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on android security," in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[66] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[67] E. Reshetova, F. Bonazzi, T. Nyman, R. Borgaonkar, and N. Asokan, "Characterizing seandroid policies in the wild," *arXiv preprint arXiv:1510.05497*, 2015.

[68] J. Danisevskis, M. Piekarska, and J.-P. Seifert, "Dark side of the shader: Mobile gpu-aided malware delivery," in *International Conference on Information Security and Cryptology (ICISC)*, 2014.

[69] R. Welton, "Remotely abusing android," in *BlackHat Mobile Security Summit*, 2015.

[70] H. Zhang, D. She, and Z. Qian, "Android root and its providers: A double-edged sword," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[71] D. Shen, "Attacking your trusted core: Exploiting trustzone on android," in *BlackHat Convention (BH)*, 2015.

[72] D. R. Thomas, A. R. Beresford, and A. Rice, "Security metrics for the android ecosystem," in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2015.

[73] K. Nohl, "Rooting sim cards," in *BlackHat Convention (BH)*, 2013.

[74] J. Liu, Y. Yu, F.-X. Standaert, Z. Guo, D. Gu, W. Sun, Y. Ge, and X. Xie, "Small tweaks do not help: Differential power analysis of milenage implementations in 3g/4g usim cards," in *European Symposium on Research in Computer Security (ESORICS)*, 2015.

[75] R.-P. Weinmann, "Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[76] C. Mulliner, N. Golde, and J.-P. Seifert, "Sms of death: From analyzing to attacking mobile phones on a large scale," in *USENIX Security Symposium*, 2011.

[77] M. Solnik and M. Blanchou, "Cellular exploitation on a global scale: The rise and fall of the control protocol," in *BlackHat Convention (BH)*, 2014.

[78] K. Nohl and C. Paget, "Gsm: Srsly," in *Chaos Communication Congress (26C3)*, 2009.

[79] K. Nohl, "Breaking gsm phone privacy," in *BlackHat Convention (BH)*, 2010.

[80] S. Meyer, "Breaking gsm with rainbow tables," *arXiv preprint arXiv:1107.1086*, 2011.

[81] O. Dunkelman, N. Keller, and A. Shamir, "A practical-time attack on the a5/3 cryptosystem used in third generation gsm telephony," in *IACR Cryptology ePrint Archive*, 2010.

[82] R. Borgaonkar, A. Shaik, N. Asokan, V. Niemi, and J.-P. Seifert, "Lte and imsi catcher myths," in *BlackHat Convention (BH)*, 2015.

[83] F. van den Broek, R. Verdult, and J. de Ruiter, "Defeating imsi catchers," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[84] N. Golde, K. Redon, and J.-P. Seifert, "Let me answer that for you: Exploiting broadcast information in cellular networks," in *USENIX Security Symposium*, 2013.

[85] N. Golde, K. Redon, and R. Borgaonkar, "Weaponizing femtocells: The effect of rogue devices on mobile telecommunications," in *Network and Distributed System Security Symposium (NDSS)*, 2012.

[86] P. Langlois, "Sctpscan-finding entry points to ss7 networks & telecommunication backbones," in *BlackHat Convention (BH)*, 2007.

[87] M. Arapinis, L. I. Mancini, E. Ritter, and M. Ryan, "Privacy through pseudonymity in mobile telephony systems," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[88] M. Arapinis, L. Mancini, E. Ritter, M. Ryan, N. Golde, K. Redon, and R. Borgaonkar, "New privacy issues in mobile telephony: fix and verification," in *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[89] A. Shaik, R. Borgaonkar, N. Asokan, V. Niemi, and J.-P. Seifert, "Practical attacks against privacy and availability in 4g/lte mobile communication systems," *arXiv preprint arXiv:1510.07563*, 2015.

[90] V. C. Perta, M. V. Barbera, and A. Mei, "Exploiting delay patterns for user ips identification in cellular networks," in *Privacy Enhancing Technologies Symposium (PETS)*, 2014.

[91] H. Kim, D. Kim, M. Kwon, H. Han, Y. Jang, D. Han, T. Kim, and Y. Kim, "Breaking and fixing volte: Exploiting hidden data channels and mis-implementations," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[92] C.-Y. Li, G.-H. Tu, C. Peng, Z. Yuan, Y. Li, S. Lu, and X. Wang, "Insecurity of voice solution volte in lte mobile networks," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[93] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang, "An untold story of middleboxes in cellular networks," in *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2011.

[94] D. Wu and R. K. Chang, "Indirect file leaks in mobile applications," in *Workshop on Mobile Security Technologies (MoST)*, 2015.

[95] S. Adappa, V. Agarwal, S. Goyal, P. Kumaraguru, and S. Mittal, "User controllable security and privacy for mobile mashups," in *ACM Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2011.

[96] R. Wang, L. Xing, X. Wang, and S. Chen, "Unauthorized origin crossing on mobile platforms: Threats and mitigation," in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[97] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and fixing origin-based access control in hybrid web/mobile application frameworks," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[98] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love android: An analysis of android ssl (in) security," in *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[99] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[100] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde, "Can't you hear me knocking: Identification of user actions on android apps via traffic analysis," in *ACM Conference on Data and Applications Security and Privacy (CODASPY)*, 2015.

[101] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang, "Leave me alone: App-level protection against runtime information gathering on android," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[102] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[103] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[104] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "Oauth demystified for mobile application developers," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[105] F. Cai, H. Chen, Y. Wu, and Y. Zhang, "Appcracker: Widespread vulnerabilities in user and session authentication in mobile apps," in *Workshop on Mobile Security Technologies (MoST)*, 2015.

[106] B. Reaves, N. Scaife, A. Bates, P. Traynor, and K. R. Butler, "Mo(bile) money, mo(bile) problems: analysis of branchless banking applications in the developing world," in *USENIX Security Symposium*, 2015.

[107] L. Simon and R. Anderson, "Security analysis of consumer-grade anti-theft solutions provided by android mobile anti-virus apps," in *Workshop on Mobile Security Technologies (MoST)*, 2015.

[108] "Android security state of the union 2014." `https://googleonlinesecurity.blogspot.co.uk/2015/04/android-security-state-of-union-2014.html`.

[109] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2012.

[110] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: Separating smartphone advertising from applications," in *USENIX Security Symposium*, 2012.

[111] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in android," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2012.

[112] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo, "Don't kill my ads!: balancing privacy in an ad-supported mobile application market," in *ACM Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2012.

[113] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient privilege de-escalation for ad libraries in mobile apps," in *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2015.

[114] J. Seo, D. Kim, D. Cho, T. Kim, and I. Shin, "Flexdroid: Enforcing in-app privilege separation in android," in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[115] H. Fu, Y. Yang, N. Shingte, J. Lindqvist, and M. Gruteser, "A field study of run-time location access disclosures on android smartphones," in *NDSS Workshop on Usable Security (USEC)*, 2014.

[116] H. Almuhimedi, F. Schaub, N. Sadeh, I. Adjerid, A. Acquisti, J. Gluck, L. F. Cranor, and Y. Agarwal, "Your location has been shared 5,398 times!: A field study on mobile app privacy nudging," in *ACM Conference on Human Factors in Computing Systems (CHI)*, 2015.

[117] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale," in *International Conference on Trust & Trustworthy Computing (TRUST)*, 2012.

[118] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[119] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *USENIX Conference on Operating systems design and implementation (OSDI)*, 2014.

[120] C. Qian, X. Luo, Y. Shao, and A. T. Chan, "On tracking information flows through jni in android applications," in *International Conference on Dependable Systems and Networks (DSN)*, 2014.

[121] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, “Riskranker: scalable and accurate zero-day android malware detection,” in *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[122] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, “Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets,” in *Network and Distributed System Security Symposium (NDSS)*, 2012.

[123] S. Liang, A. W. Keep, M. Might, S. Lyde, T. Gilray, P. Aldous, and D. Van Horn, “Sound and precise malware analysis for android via pushdown reachability and entry-point saturation,” in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2013.

[124] F. Maggi, A. Valdi, and S. Zanero, “Andrototal: a flexible, scalable toolbox and service for testing mobile malware detectors,” in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2013.

[125] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, “Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale,” in *USENIX Security Symposium*, 2015.

[126] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, “Crowdroid: behavior-based malware detection system for android,” in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[127] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, “Vetting undesirable behaviors in android apps with permission use analysis,” in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[128] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, “Andrubis-1,000,000 apps later: A view on current android malware behaviors,” in *International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[129] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, “Effective real-time android application auditing,” in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[130] L.-K. Yan and H. Yin, “Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis,” in *USENIX Security Symposium*, 2012.

[131] A. Reina, A. Fattori, and L. Cavallaro, “A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors,” in *European Workshop on System Security (EUROSEC)*, 2013.

[132] C. Lever, M. Antonakakis, B. Reaves, P. Traynor, and W. Lee, "The core of the matter: Analyzing malicious traffic in cellular carriers," in *Network and Distributed System Security Symposium (NDSS)*, 2013.

[133] F. Wei, S. Roy, X. Ou, *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[134] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *ACM Conference on Data and Applications Security and Privacy (CODASPY)*, 2012.

[135] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of piggybacked mobile applications," in *ACM Conference on Data and Applications Security and Privacy (CODASPY)*, 2013.

[136] J. Crussell, C. Gibler, and H. Chen, "Andarwin: Scalable detection of semantically similar android applications," in *European Symposium on Research in Computer Security (ESORICS)*, 2013.

[137] W. Zhou, X. Zhang, and X. Jiang, "Appink: watermarking android apps for repackaging deterrence," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2013.

[138] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *USENIX Security Symposium*, 2012.

[139] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, "I-arm-droid: A rewriting framework for in-app reference monitors for android applications," in *Workshop on Mobile Security Technologies (MoST)*, 2012.

[140] Y. Zhou, K. Patel, L. Wu, Z. Wang, and X. Jiang, "Hybrid user-level sandboxing of third-party android apps," in *ASIACCS*, 2015.

[141] A. Nadkarni, V. Tendulkar, and W. Enck, "Nativewrap: ad hoc smartphone application creation for end users," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2014.

[142] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock android," in *USENIX Security Symposium*, 2015.

[143] Y. Song and U. Hengartner, "Privacyguard: A vpn-based platform to detect information leakage on android devices," in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2015.

[144] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android," in *Network and Distributed System Security Symposium (NDSS)*, 2013.

[145] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, "Practical and lightweight domain isolation on android," in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[146] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh, "Cells: a virtual mobile smartphone architecture," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[147] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang, "Airbag: Boosting smartphone resistance to malware infection," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[148] W. Chen, L. Xu, G. Li, and Y. Xiang, "A lightweight virtualization solution for android devices," *IEEE Transactions on Computers (TOC)*, vol. 64, 2015.

[149] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter, "L4android: a generic operating system framework for secure smartphones," in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[150] J. Danisevskis, M. Peter, J. Nordholz, M. Petschick, and J. Vetter, "Graphical user interface for virtualized mobile handsets," in *Workshop on Mobile Security Technologies (MoST)*, 2015.

[151] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "Armlock: Hardware-based fault isolation for arm," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[152] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *International Cryptology Conference (CRYPTO)*, 1999.

[153] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *International Cryptology Conference (CRYPTO)*, 1996.

[154] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Cryptographer's Track at the RSA Conference*, 2006.

[155] D. J. Bernstein, "Cache-timing attacks on aes," tech. rep., 2005.

[156] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-vm attack on aes," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2014.

[157] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S $ a: A shared cache attack that works across cores and defies vm sandboxing - and its application to aes," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[158] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *USENIX Security Symposium*, 2015.

[159] Y. Yarom and K. E. Falkner, "Flush+ reload: a high resolution, low noise, l3 cache side-channel attack," in *IACR Cryptology ePrint Archive*, 2013.

[160] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom, ""ooh aah... just a little bit": A small amount of side channel can go a long way," in *Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2014.

[161] J. van de Pol, N. P. Smart, and Y. Yarom, "Just a little bit more," in *Cryptographer's Track at the RSA Conference*, 2015.

[162] D. Gruss, C. Maurice, and K. Wagner, "Flush+ flush: A stealthier last-level cache attack," *arXiv preprint arXiv:1511.04594*, 2015.

[163] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographer's Track at the RSA Conference*, 2006.

[164] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[165] A. Bogdanov, T. Eisenbarth, C. Paar, and M. Wienecke, "Differential cache-collision timing attacks on aes with applications to embedded cpus," in *Cryptographer's Track at the RSA Conference*, 2010.

[166] M. Weiss, B. Heinz, and F. Stumpf, "A cache timing attack on aes in virtualization environments," in *International Conference on Financial Cryptography and Data Security (FC)*, 2012.

[167] R. Spreitzer and T. Plos, "On the applicability of time-driven cache attacks on mobile devices," in *International Conference on Network and System Security (NSS)*, 2013.

[168] R. Spreitzer and B. Gérard, "Towards more practical time-driven cache attacks," in *International Workshop on Information Security Theory and Practices (WISTP)*, 2014.

[169] M. Lipp, D. Gruss, R. Spreitzer, and S. Mangard, "Armageddon: Last-level cache attacks on mobile devices," *arXiv preprint arXiv:1511.04897*, 2015.

[170] J. Cache, "Fingerprinting 802.11 Implementations via Statistical Analysis of the Duration Field," tech. rep., 2006.

[171] V. C. Perta, M. V. Barbera, and A. Mei, "Exploiting delay patterns for user ips identification in cellular networks," in *Privacy Enhancing Technologies Symposium (PETS)*, 2014.

[172] V. Brik, S. Banerjee, M. Gruteser, and S. Oh, "Wireless device identification with radiometric signatures," in *Annual ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2008.

[173] T. Stöber, M. Frank, J. Schmitt, and I. Martinovic, "Who do you sync you are?: smartphone fingerprinting via application behaviour," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2013.

[174] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[175] S. Khattak, L. Simon, and S. J. Murdoch, "Systemization of pluggable transports for censorship resistance," *arXiv preprint arXiv:1412.7448*, 2014.

[176] C. V. Wright, L. Ballard, S. E. Coull, F. Monrose, and G. M. Masson, "Spot me if you can: Uncovering spoken phrases in encrypted voip conversations," in *IEEE Symposium on Security and Privacy (S&P)*, 2008.

[177] A. M. White, A. R. Matthews, K. Z. Snow, and F. Monrose, "Phonotactic reconstruction of encrypted voip conversations: Hookt on fon-iks," in *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[178] J. Mäntyjärvi, M. Lindholm, E. Vildjiounaite, S. marja Mäkelä, and H. Ailisto, "Identifying users of portable devices from gait pattern with accelerometers," in *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2005.

[179] Y. Michalevsky, D. Boneh, and G. Nakibly, "Gyrophone: Recognizing speech from gyroscope signals," in *USENIX Security Symposium*, 2014.

[180] S. Nawaz and C. Mascolo, "Mining users' significant driving routes with low-power sensors," in *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2014.

[181] Y. Michalevsky, G. Nakibly, A. Schulman, and D. Boneh, "Powerspy: Location tracking using mobile device power analysis," *arXiv preprint arXiv:1502.03182*, 2015.

[182] Z. Xu, K. Bai, and S. Zhu, "Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2012.

[183] L. Cai and H. Chen, "Touchlogger: Inferring keystrokes on touch screen from smartphone motion," in *USENIX Summit on Hot Topics in Security (HotSec)*, 2011.

[184] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury, "Tapprints: your finger taps have fingerprints," in *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[185] S. Dey, N. Roy, W. Xu, R. R. Choudhury, and S. Nelakuditi, "Accelprint: Imperfections of accelerometers make smartphones trackable," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[186] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, location, disease and more: Inferring your secrets from android public resources," in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[187] K. Zhang and X. Wang, "Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems," in *USENIX Security Symposium*, 2009.

[188] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[189] "Into the droid: Gaining access to android user data," in *DEFCON Hacking Conference*.

[190] M. Ossmann and K. Osborn, "Multiplexed wired attack surfaces," in *BlackHat Convention (BH)*, 2013.

[191] A. Pereira, M. Correia, and P. Brandão, "Usb connection vulnerabilities on android smartphones: Default and vendors customizations," in *Conference on Communications and Multimedia Security (CMS)*, 2014.

[192] A. Mahajan, M. Dahiya, and H. Sanghvi, "Forensic analysis of instant messenger applications on android devices," *arXiv preprint arXiv:1304.4915*, 2013.

[193] T. Müller and M. Spreitzenbarth, "Frost: Forensic recovery of scrambled telephones," in *International Conference on Applied Cryptography and Network Security (ACNS)*, 2013.

[194] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman, "Protecting data on smartphones and tablets from memory attacks," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[195] T. Müller, F. C. Freiling, and A. Dewald, "Tresor runs encryption securely outside ram," in *USENIX Security Symposium*, 2011.

[196] T. Müller, A. Dewald, and F. C. Freiling, "Aesse: a cold-boot resistant implementation of aes," in *European Workshop on System Security (EUROSEC)*, 2010.

[197] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "Cleanos: Limiting mobile data exposure with idle eviction," in *USENIX Conference on Operating systems design and implementation (OSDI)*, 2012.

[198] C. N. K. Michael Becher, Maximillian Dornseif, "Firewire - all your memory are belong to us," in *CanSecWest Applied Security Conference (CanSecWest)*, 2005.

[199] A. Boileau, "Hit by a bus: Physical access attacks with firewire," in *Ruxcon Security Conference (Ruxcon)*, 2006.

[200] D. R. Piegdon, "Hacking in physically addressable memory - a proof of concept," in *Seminar of Advanced Exploitation Techniques*, 2006.

[201] M. Breeuwsma, M. De Jongh, C. Klaver, R. Van Der Knijff, and M. Roeloffs, "Forensic data recovery from flash memory," *Small Scale Digital Device Forensics Journal (SSDDFJ)*, vol. 1, no. 1, pp. 1–17, 2007.

[202] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "Guitar: Piecing together android app guis from memory images," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[203] J. P. van Zandwijk, "A mathematical approach to nand flash-memory descrambling and decoding," *Digital Investigation*, vol. 12, pp. 41–52, 2015.

[204] J. Luck and M. Stokes, "An integrated approach to recovering deleted files from nand flash data," *Small Scale Digital Device Forensics Journal (SSDDFJ)*, vol. 2, no. 1, pp. 1941–6164, 2008.

[205] D. Billard and R. Hauri, "Making sense of unstructured flash-memory dumps," in *ACM Symposium on Applied Computing (SAC)*, 2010.

[206] A. B. Lewis, "Reconstructing compressed photo and video data," Tech. Rep. UCAM-CL-TR-813, University of Cambridge, Computer Laboratory, Feb 2012.

[207] R. J. Walls, E. Learned-Miller, and B. N. Levine, "Forensic Triage for Mobile Phones with DEC0DE," in *USENIX Security Symposium*, 2011.

[208] S. Varma, R. J. Walls, B. Lynn, and B. N. Levine, "Efficient Smart Phone Forensics Based on Relevance Feedback," in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2014.

[209] S. L. Garfinkel and A. Shelat, "Remembrance of data passed: A study of disk sanitization practices," *IEEE Security and Privacy*, vol. 1, no. 1, pp. 17–27, 2003.

[210] M. Y. C. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, "Reliably erasing data from flash-based solid state drives," in *USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[211] M. Freeman and A. Woodward, "Secure state deletion: Testing the efficacy and integrity of secure deletion tools on solid state drives," in *Australian Digital Forensics Conference*, 2009.

[212] P. Gutmann, "Secure deletion of data from magnetic and solid-state memory," in *USENIX Security Symposium*, 1996.

[213] J. Lee, J. Heo, Y. Cho, J. Hong, and S. Y. Shin, "Secure deletion for nand flash file system," in *ACM Symposium on Applied Computing (SAC)*, 2008.

[214] B. Lee, K. Son, D. Won, and S. Kim, "Secure data deletion for usb flash memory," *Journal of Information Science and Engineering (JISE)*, vol. 27, no. 3, pp. 933–952, 2011.

[215] J. Reardon, S. Capkun, and D. A. Basin, "Data node encrypted file system: Efficient secure deletion for flash memory," in *USENIX Security Symposium*, 2012.

[216] L. Simon and R. Anderson, "PIN Skimmer: Inferring PINs Through The Camera and Microphone," in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2013.

[217] S. Das, L. Green, B. Perez, and M. Murphy, "Detecting user activities using the accelerometer on android smartphones," in *Summer TRUST REU*, 2010.

[218] R. Templeman, Z. Rahman, D. Crandall, and A. Kapadia, "PlaceRaider: Virtual theft in physical spaces with smartphones," in *Network and Distributed System Security Symposium (NDSS)*, 2013.

[219] Z. Yaniv, "Random Sample Consensus (RANSAC) Algorithm, A Generic Implementation." `http://isiswiki.georgetown.edu/zivy/writtenMaterial/RANSAC.pdf`, Oct 2010.

[220] G. Roth, "Homography." `http://people.scs.carleton.ca/~c_shu/Courses/comp4900d/notes/homography.pdf`, 2013.

[221] J. Bonneau, S. Preibusch, and R. Anderson, "A birthday present every eleven wallets? The security of customer-chosen banking PINs," in *International Conference on Financial Cryptography and Data Security (FC)*, 2012.

[222] J. Koetsier, "Pin analysis." `http://www.datagenetics.com/blog/september32012/`, 2013.

[223] C. Cachin, *Entropy measures and unconditional security in cryptography.* PhD thesis, ETH Zurich, 1997.

[224] S. Brostoff and M. A. Sasse, "“ten strikes and you're out”: Increasing the number of login attempts can improve password usability," in *ACM Conference on Human Factors in Computing Systems (CHI)*, 2003.

[225] O. Riva, C. Qin, K. Strauss, and D. Lymberopoulos, "Progressive authentication: deciding when to authenticate on mobile phones," in *USENIX Security Symposium*, 2012.

[226] A. T. Ozcan, C. Gemicioglu, K. Onarlioglu, M. Weissbacher, C. Mulliner, W. Robertson, and E. Kirda, "BabelCrypt: The Universal Encryption Layer for Mobile Messaging Applications," in *International Conference on Financial Cryptography and Data Security (FC)*, 2015.

[227] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it: UI state inference and novel android attacks," in *USENIX Security Symposium*, 2014.

[228] A. Savitzky and M. J. E. Golay, "Smoothing and Differentiation of Data by Simplified Least Squares Procedures," *Anal. Chem.*, vol. 36, pp. 1627–1639, July 1964.

[229] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *International Speech Communication Association (ISCA)*, 2010.

[230] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[231] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, 1988.

[232] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[233] E. N. Forsythand and C. H. Martell, "Lexical and discourse analysis of online chat dialog," in *International Conference on Semantic Computing (ICSC)*, 2007.

[234] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python.* O'Reilly Media, Inc., 1st ed., 2009.

[235] J. Munkres, "Algorithms for the assignment and transportation problems," *Journal of the Society of Industrial and Applied Mathematics.*

[236] L. Simon and R. Anderson, “Pin skimmer: Inferring pins through the camera and microphone,” in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2013.

[237] L. Simon and R. Anderson, “Security Analysis of Android Factory Resets,” in *Workshop on Mobile Security Technologies (MoST)*, 2015.

[238] L. Simon and R. Anderson, “Security Analysis of Android Factory Resets,” in *BlackHat Mobile Security Summit*, 2015.

[239] R. Schwamm and N. C. Rowe, “Effects of the factory reset on mobile devices,” in *Journal of Digital Forensics, Security and Law (JDFSL)*, 2014.

[240] T. Vidas, C. Zhang, and N. Christin, “Toward a general collection methodology for android devices,” *Digital Investigation: The International Journal of Digital Forensics & Incident Response*, vol. 8, pp. S14–S24, Aug. 2011.

[241] G. G. Richard III and V. Roussev, “Scalpel: A frugal, high performance file carver,” in *Digital Forensics Research Conference (DFRWS)*, 2005.

[242] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, “The impact of vendor customizations on android security,” in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[243] D. S. Andrey Belenko, “Evolution of ios data protection and iphone forensics: from iphone os to ios 5,” in *BlackHat Convention (BH)*, 2011.

[244] L. Simon and R. Anderson, “Security Analysis of Consumer-Grade Anti-Theft Solutions Provided by Android Mobile Anti-Virus Apps,” in *Workshop on Mobile Security Technologies (MoST)*, 2015.

[245] J. Bonneau and S. Preibusch, “The password thicket: Technical and market failures in human authentication on the web,” in *Workshop on the Economics of Information Security (WEIS)*, 2010.

[246] M. Javed and V. Paxson, “Detecting stealthy, distributed ssh brute-forcing,” in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[247] S. Clark, T. Goodspeed, P. Metzger, Z. Wasserman, K. Xu, and M. Blaze, “Why (special agent) johnny (still) can’t encrypt: A security analysis of the apco project 25 two-way radio system,” in *USENIX Security Symposium*, 2011.

[248] S. Fahl, M. Harbach, T. Muders, M. Smith, and U. Sander, “Helping johnny 2.0 to encrypt his facebook conversations,” in *Symposium on Usable Privacy and Security (SOUPS)*, 2012.

[249] A. Whitten and J. D. Tygar, "Why johnny can't encrypt: A usability evaluation of pgp 5.0," in *USENIX Security Symposium*, 1999.

[250] S. Ruoti, N. Kim, B. Burgon, T. van der Horst, and K. Seamons, "Confused johnny: When automatic encryption leads to confusion and mistakes," in *Symposium on Usable Privacy and Security (SOUPS)*, 2013.

[251] R. Anderson and M. Kuhn, "Tamper resistance: A cautionary note," in *USENIX Workshop on Electronic Commerce (WOEC)*, 1996.

[252] B. Smyth and A. Pironti, "Truncating tls connections to violate beliefs in web applications," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2013.

[253] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "Cleanos: Limiting mobile data exposure with idle eviction," in *USENIX Conference on Operating systems design and implementation (OSDI)*, 2012.

[254] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy, "Keypad: An auditing file system for theft-prone devices," in *ACM European Conference on Computer Systems (EuroSys)*, 2011.

[255] P. D. MacKenzie and M. K. Reiter, "Networked cryptographic devices resilient to capture," in *IEEE Symposium on Security and Privacy (S&P)*, 2001.

[256] E. Shi, Y. Niu, M. Jakobsson, and R. Chow, "Implicit authentication through learning user behavior," in *International Conference on Information Security and Cryptology (ICISC)*, 2011.

[257] S. M. Kolly, R. Wattenhofer, and S. Welten, "A personal touch: Recognizing users based on touch screen behavior," in *International Workshop on Sensing Applications on Mobile Phones (PhoneSense)*, 2012.

[258] E. Hayashi, S. Das, S. Amini, J. Hong, and I. Oakley, "Casa: Context-aware scalable authentication," in *Symposium on Usable Privacy and Security (SOUPS)*, 2013.

[259] E. Maiorana, P. Campisi, N. González-Carballo, and A. Neri, "Keystroke dynamics authentication for mobile phones," in *ACM Symposium on Applied Computing (SAC)*, 2011.

[260] H. Khan and U. Hengartner, "Towards application-centric implicit authentication on smartphones," in *ACM Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2014.

[261] Z. Wang and A. Stavrou, "Exploiting smart-phone usb connectivity for fun and profit," in *Annual Computer Security Applications Conference (ACSAC)*, 2010.