UNIVERSITÀ DEGLI STUDI DI PADOVA

Sede Amministrativa: Università degli Studi di Padova

Dipartimento di Matematica

CORSO DI DOTTORATO DI RICERCA IN: Scienze Matematiche

CURRICOLO: Informatica

CICLO: XXIX

# SECURITY ISSUES OF MOBILE AND SMART WEARABLE DEVICES

**Coordinatore:** Ch.mo Prof. Bruno Chiarellotto

**Supervisore:** Ch.mo Prof. Mauro Conti

**Dottorando:** Hossein Fereidooni

University of Padua
Department of Mathematics
Doctorate Degree in Mathematics
Curriculum in Computer Science

# Security Issues of Mobile and Smart Wearable Devices

**Candidate**
Hossein Fereidooni

**Supervisor**
Prof. Mauro Conti
*University of Padua, Italy*

# Acknowledgments

On completion of my Ph.D. thesis I would like to sincerely thank all those who supported me in realizing and finishing my work. I would never have been able to finish this dissertation without the guidance of my Ph.D advisor Prof. Mauro Conti and my committee members, support from my family and wife, and help from my friends and lab mates.
I would also like to thank my parents. They have brought me tremendous love since I was born. Finally, I would like to thank my wife Shokoufeh and my daughter Raha. They were always there cheering me up and accompanied me through good and bad times. This dissertation is dedicated to them.

# Abstract

Mobile and smart devices (ranging from popular smartphones and tablets to wearable fitness trackers equipped with sensing, computing and networking capabilities) have proliferated lately and redefined the way users carry out their day-to-day activities. These devices bring immense benefits to society and boast improved quality of life for users. As mobile and smart technologies become increasingly ubiquitous, the security of these devices becomes more urgent, and users should take precautions to keep their personal information secure. Privacy has also been called into question as so many of mobile and smart devices collect, process huge quantities of data, and store them on the cloud as a matter of fact. Ensuring confidentiality, integrity, and authenticity of the information is a cybersecurity challenge with no easy solution.

Unfortunately, current security controls have not kept pace with the risks posed by mobile and smart devices, and have proven patently insufficient so far. Thwarting attacks is also a thriving research area with a substantial amount of still unsolved problems. The pervasiveness of smart devices, the growing attack vectors, and the current lack of security call for an effective and efficient way of protecting mobile and smart devices.

This thesis deals with the security problems of mobile and smart devices, providing specific methods for improving current security solutions. Our contributions are grouped into two related areas which present natural intersections and corresponds to the two central parts of this document: (1) *Tackling Mobile Malware*, and (2) *Security Analysis on Wearable and Smart Devices*.

In the first part of this thesis, we study methods and techniques to assist security analysts to tackle mobile malware and automate the identification of malicious applications. We provide threefold contributions in tackling mobile malware: First, we introduce a Secure Message Delivery (SMD) protocol for Device-to-Device (D2D) networks, with primary objective of choosing the most secure path to deliver a message from a sender to a destination in a multi-hop D2D network. Second, we illustrate a survey to investigate concrete and relevant questions concerning Android code obfuscation and protection techniques, where the purpose is to review code obfuscation and code protection practices. We evaluate efficacy of existing code de-obfuscation tools to tackle obfuscated Android malware (which provide attackers with the ability to evade detection mechanisms). Finally, we propose a Machine

Learning-based detection framework to hunt malicious Android apps by introducing a system to detect and classify newly-discovered malware through analyzing applications. The proposed system classifies different types of malware from each other and helps to better understanding how malware can infect devices, the threat level they pose and how to protect against them. Our designed system leverages more complete coverage of apps' behavioral characteristics than the state-of-the-art, integrates the most performant classifier, and utilizes the robustness of extracted features.

The second part of this dissertation conducts an in-depth security analysis of the most popular wearable fitness trackers on the market. Our contributions are grouped into four central parts in this domain: First, we analyze the primitives governing the communication between fitness tracker and cloud-based services. In addition, we investigate communication requirements in this setting such as: (i) Data Confidentiality, (ii) Data Integrity, and (iii) Data Authenticity. Second, we show real-world demos on how modern wearable devices are vulnerable to false data injection attacks. Also, we document successful injection of falsified data to cloud-based services that appears legitimate to the cloud to obtain personal benefits. Third, we circumvent End-to-End protocol encryption implemented in the most advanced and secure fitness trackers (e.g., Fitbit, as the market leader) through Hardware-based reverse engineering. Last but not least, we provide guidelines for avoiding similar vulnerabilities in future system designs.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Mobile and smart wearable devices, as shown in Figure 1.1, are rapidly emerging as popular appliances with increasingly powerful computing, networking, and sensing capabilities. Perhaps the most successful examples of such devices so far are high-end smartphones and tablets with powerful processors, 4G network and high-bandwidth connectivity. In addition, the rise of cloud-computing paradigms, complementary storage and computing services have led to the ever-increasing popularity of such devices.

New mobile and smart devices are not only playing a significant role in bringing paradigms such as wearable computing or the Internet of Things (IoT) to reality, but also finding innovative and attractive applications in several domains. As an example, in health-care domain both medical staff and patients are increasingly reaping the benefits of such devices; from regular smartphones and tablets to the new generation of smart wearable systems for health monitoring.

Figure 1.1: Example of Mobile and Smart Wearable Devices.

Mobile and smart devices often lack stringent security measures, and some attacks are able to exploit existing vulnerabilities in these devices. Many security issues originate from how securely vendors implemented mechanisms for authentication and encryption. Recently, there have been several reported attacks in different domains such as:

- *Cars.* Fiat Chrysler recalled 1.4 million vehicles after researchers demonstrated an attack where they could take control of the vehicle remotely.[1]
- *Smart home devices.* Researchers at Symantec discovered multiple vulnerabilities in several commercially IoT devices, including a smart door lock that could be opened remotely online without a password.[2]
- *Medical devices.* Researchers have found potentially deadly vulnerabilities in dozens of devices. For instance, insulin pumps, x-ray systems, CT-scanners, medical refrigerators, and implantable defibrillators.[3]
- *Smart TVs.* Hundreds of millions of Internet-connected TVs are potentially vulnerable to click fraud, botnets, data theft, and even ransomware.[4]

In this chapter, Section 1.1 explains our research motivations and contributions and Section 1.2 lists the scientific publications resulted form our research work.

## 1.1 Research Motivations and Contributions

Often, users are unaware of the security and privacy risks that mobile and smart devices pose, until after an event creates media waves. End-users expect the developers to build software and hardware securely. They assume that proper security controls exist to maintain the safety and privacy of their information. Addressing the security and privacy concerns associated with the usage of these devices is the key to unleashing the benefits that wearable devices offer to society. Motivation is the key to successful attacks and for good security against such attacks. We need to escalate our security motivation to approach the motivation of adversaries. Ubiquitous mobile and smart devices, the growing attack vectors, and the current lack of an effective and efficient way of protecting these devices motivated us to do research in this line of study. This thesis just scratches the surface of the current problems and more efforts need to be done in this research line.

The research work presented in this dissertation concentrates on security and privacy issues on mobile and smart devices with a focus on Android-powered smartphones and wearable fitness tracking devices. In particular, in this dissertation we present our research in two main logical parts:

- *Tackling Mobile Malware.*
- *Security Analysis on Wearable Fitness Devices.*

[1] http://securityaffairs.co/wordpress/38844/hacking/jeep-cherokee-hack-fiat-recall.html
[2] {http://link.springer.com/article/10.1007%2Fs10586-016-0617-2}
[3] http://iotsecurityconnection.com/posts/symantecs-take-on-the-risk-of-things
[4] https://securelist.com/blog/incidents/73229/malware-on-the-smart-tv/

In the following, we briefly introduce each of the above-mentioned parts, and sum up our contributions.

### 1.1.1 Tackling Mobile Malware

Since their first release in late 2008[5], Android smartphones have been replacing traditional mobile phones. The advent of such high-powered and affordable smart devices has redefined the way that mobile phone users carry out their day-to-day activities. The Android operating system is the most widely used on mobile devices and hence is a popular target of attack for cyber criminals. Gartner reported that worldwide sales of Android smartphones in 2016 has reached more than 296 million devices, which accounted for 86.2% of the market share.[6] Due to Android's popularity and widespread user-acceptance, the amount of malware targeting the Android platform has increased significantly in recent years. As such, malicious applications pose a significant threat to smartphones' platform security. One major source of such a threat is the ability to incorporate third-party applications, from online markets but also by other means.

People use those mobile devices for several types of applications, often involving personal information (contacts, emails, agenda, pictures, banking, etc.). According to the Bring Your Own Device (BYOD) policy, adopted by many companies [8], the very same personal device is also used to access the IT infrastructure of the company where the smartphone owner is employed. In this scenario, the security of these devices, as well as the assets that they allow access to, are at stake.

Mobile Malware refers to software programs designed to damage or take any kind of unwanted actions on a mobile Operating System (OS) such as disrupting OS operations, gather sensitive information, bypass access controls, gain access to private information and display unwanted advertising. Malware applications, based on their harmful function, can be divided into the following categories:[7]

- Adware: it is a type of malware that automatically delivers advertisements. Adware has been criticized because it usually includes code that tracks a user's personal information and passes it on to third parties.
- Spyware: it is a type of malware that spies and tracks user activity without their knowledge, also collects information about user's surfing habits, browsing history, or personal information (such as credit card numbers). The capabilities of spyware can include keystrokes collection, financial data harvesting or activity monitoring.
- Virus: a virus is a type of malicious software capable of copying itself and spreading to other mobile devices and requires user intervention to infect a device. The user must actually run the software which contains the virus' code. Macro virus is another type of virus which is spread via Macro code and is launched when a file or document is opened. The code can be embedded inside a Microsoft Office document (e.g. Microsoft Word document, Microsoft Excel).

[5] `http://www.cnet.com/news/a-brief-history-of-android-phones/`
[6] `http://www.gartner.com/newsroom/id/3415117`
[7] `http://www.malwaretruth.com/the-list-of-malware-types/`

- Worm: It is a type of malware that spreads through to an entire network by exploiting OS vulnerabilities without user intervention, this is because worms are self-replicating.
- Trojan: Torjan is a type of malware requiring user intervention to infect a device. Like virus, user must run the software that contains the Trojan's code. However, Trojan is a different from of virus in the sense that it often appears to be a legitimate software that users might have been searching for. There exist several types of Trojans:

  (i) Trojan Downloader: when launched, downloads additional file(s) that actually contain the final payload;

  (ii) Trojan Injector: when launched, injects malicious code into another process, often a legitimate process, to evade detection;

  (iii) Trojan Dropper: when launched, drops executable files containing the malware's payload.

- Backdoors: a backdoor is a software designed to bypass normal authentication procedures and compromise the OS.
- Command & Control Bot: Bots are software programs created to automatically perform specific operations. Bots are commonly used for DDoS attacks,
- Ransomware: it is a type of malware spreads by attackers with the goal of demanding a ransom from their victims, most often for financial benefits. Crypto ransomware will usually go through all of the directories, files and sometimes network shares of the user's device. It will open files and then encrypt the contents of those files. Users are forced to pay the attacker to remove the restrictions and gain access to their files. This type of payment is usually done with Bitcoins.
- Rootkit: it is a malware that can evade all anti-malware software and can affect device's OS itself. Rootkits often function as keyloggers, and their removal often requires the user to format their devices.
- Ghostware: it is a malware that deletes itself off the system once identified. This malware is programmed to remove itself from an infected device, leaving no tracks for further investigation.

The rapid growth of Android-powered smartphones and their widespread usage has come hand-in-hand with a similar increase in the number and sophistication of virulent software targeting Android platforms. All of these make Android OS a very attractive target to criminals. Security experts recorded of over 3.2 million new Android malware applications in 2016 (an increase of almost 40 percent compared to 2015). They also counted over 750,000 new malware applications in the first quarter of 2017 and expect around 3.5 million new Android malware apps for 2017.[8]

[8] https://www.gdatasoftware.co.uk/news/2017/02/threat-situation-for-mobile-devices-worsens

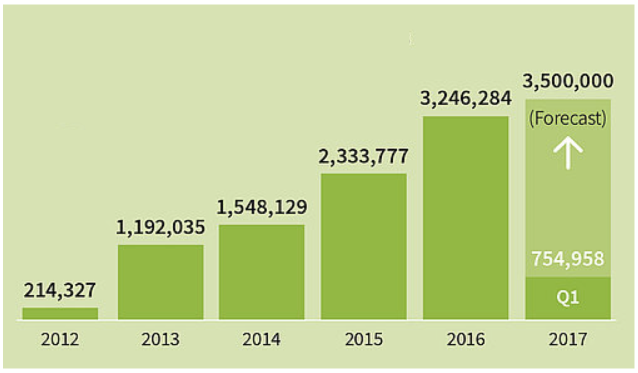


Figure 1.2: New Android Malware Samples per Year.

Nowadays, malicious software detection is mainly performed with heuristic and signature-based methods struggling to keep up with malware evolution. In fact, security for smartphones still needs a thorough understanding (as proven by several attacks, e.g., the ones in [9–12]).

## Secure Message Delivery Games for D2D Communications

Device-to-Device (D2D) communication is expected to be a key feature supported by next generation cellular networks. D2D can extend the cellular coverage allowing users to communicate when telecommunications infrastructure are highly congested or absent. In D2D networks, any *message delivery* from a *source* to a *destination* relies exclusively on intermediate devices. Each device can run different kinds of *mobile security software*, which offer protection against viruses and other harmful programs by using real-time scanning in every file entering the device. We investigate the best D2D network path to deliver a potentially malicious message from a source to a destination. Although our primary objective is to increase security, we also investigate the contribution of energy costs and quality-of-service to the path selection. We propose the *Secure Message Delivery* (SMD) protocol, whose main functionality is determined by the solution of the *Secure Message Delivery Game* (SMDG). This game is played between the *defender* (i.e., the D2D network) which abstracts all legitimate network devices and the *attacker* which abstracts any adversary that can inject different malicious messages into the D2D network in order, for instance, to infect a device with malware.

**Contribution.** We propose the *Secure Message Delivery* protocol [13]. The primary objective of this protocol is to choose the most secure path to deliver a message from a sender to a destination in a multi-hop D2D network. SMD can work on top of underlying physical and MAC layer protocols [14, 15]. Apart from security, SMD respects the energy costs and Quality-of-Service (QoS) of each route. This happens by giving certain weights to each of the involved parameters (security, energy, QoS) with more emphasis to be put on security. We formulate *Secure Message Delivery Games* (SMDGs) in order to derive an optimal behavior for the SMD. In these games, one or more adversaries, abstracted by the *attacker*, aim at increasing the security damage, incurred to the defender (i.e., network), by injecting malicious messages into the D2D network. On the other hand, the defender chooses the “best route” for message delivery. In SMDGs, the utility of the defender is influenced by: (i) the probability of the delivered message to be correctly classified as malicious or benign before it is delivered to the intended destination; (ii) the *energy cost* associated with *message forwarding*, and *message inspection* on relay devices during message delivery; and (iii) the QoS of the message communications on the chosen D2D path.

### Android Code Obfuscation Techniques

Mobile devices have become ubiquitous due to the centralization of private user information, contacts, messages and multiple sensors. Google Android, an open-source mobile Operating System (OS), is currently the market leader. Android popularity has motivated the malware authors to employ a set of cyberattacks leveraging code obfuscation techniques. Obfuscation is an action that modifies an application code, preserving semantics and intended functionality. Code obfuscation is a contentious issue. Theoretical code analysis techniques indicate that attaining a verifiable and secure obfuscation is impossible. However, obfuscation tools and techniques are popular both among malware developers (to evade anti-malware) and commercial software developers (protect intellectual rights).

**Contribution.** We conduct a survey to investigate concrete and relevant questions concerning Android code obfuscation and protection techniques [16]. The purpose is to review code obfuscation and code protection practices, and evaluate efficacy of existing code de-obfuscation tools. In particular, we discuss Android code obfuscation methods, custom app protection techniques, and various de-obfuscation methods. Furthermore, we review and analyze code protection techniques popular among malware authors which evade the de-obfuscation efforts. We believe that there is a need to investigate efficiency of the defense techniques used for code protection and this would be beneficial to the researchers and practitioners, to understand obfuscation and de-obfuscation techniques and propose novel solutions on Android.

### Android Malware Detection

The ubiquitous use of Android smartphones continues to threaten the security and privacy of users' personal information. Its fast adoption rate makes the smartphone an interesting target for malware authors to deploy new attacks and infect millions of devices. Moreover, the growing number and diversity of malicious applications render conventional defenses ineffective. Thus, there is a need to not only better understand the characteristics of malware families but also, to generate features that are robust and efficient for classification over an extended period of time. We propose ANASTASIA; a system to detect malicious Android applications through statically analyzing applications' behaviours. ANASTASIA provides a more complete coverage of security behaviors when compared to other state-of-the-art solutions. We utilize a large number of statically extracted features from various security behavioral characteristics of an application. We build a Machine Learning-based detection framework with high performance detection and an acceptable false positive rate. The significance of our work is to develop a lightweight malware detection system for Android-powered smartphones that leverages robust, effective, and efficient features. Besides this, in order to assess our solution, we use a reliable and updated malware data-set in terms of diversity and number of malware applications.

**Contribution.** We present an Android malware detection method that uses several informative features with good discriminative power to discern benign from malware apps [17]. To extract these features, we design and build a tool named `uniPDroid`, written in Python programming language. Our tool can be used to extract a plethora of informative features from our extensive dataset. We conduct an extensive static analysis on a well-labelled data-set of 29,864 Android applications. We use several Machine Learning classification algorithms including ensemble, eXtreme Gradient Boosting and Deep Learning to discover the most performant one in terms of accuracy and speed. Our experimental evaluations show that our proposed detection method is very effective and efficient and obtains a true positive rate in detecting malware applications as high as **97.3%** and false negative rate as low as **2.7%**.

### Android Malware Classification

Although it is critical to distinguish malicious applications from clean ones, it is also important to efficiently classify malware into their correct families. Distinguishing and classifying different kinds of malware from each other is fundamental to gaining a better understanding of how malware can infect devices, the threat level they pose and how to protect against them. Malware authors often redistribute a repackaged version of existing malware and therefore, by correctly classifying the original malware, it becomes easier for anti-virus engines to detect repackaged versions. To address the aforementioned issues, we focus solely on malicious applications to *firstly* investigate how to efficiently and accurately classify malware samples into their correct families, and *secondly* generate robust feature sets that will stand the test of time and still be relevant over a period of years; this is tested through the experimental work referred to as cumulative classification.

**Contribution.** We present an Android malware classification method that uses several informative features with good discriminative power to categorize malicious apps under their respective family names [18]. We extract the features such as intents, permissions used by an app, critical API calls, Linux system commands, and some other features that might indicate capability of performing malicious activities by an app. We carry out family-by-family malware classification. To find the class label associated with each malware sample in our dataset, we have written several scripts in `Bash` and `Python` programming languages. Then, we group $15,884$ Android malware in our repository into 78 different malware families. We accumulate Android malware apps and then carry out cumulative classification where the classification results are continuously updated as new malware samples are discovered. We leverage boosting techniques to obtain as much detection and classification performance as possible for Android malware detection in the wild. Our experimental evaluations show that our proposed method is effective and efficient with a true positive rate of 92% in family-by-family classification of malware applications.

### 1.1.2 Security Analysis on Wearable Fitness Devices

Wearable devices for fitness tracking and health monitoring have gained considerable popularity and become one of the fastest growing items in the smart devices market. Tens of millions of these devices are shipped yearly to consumers who routinely collect information about their exercising patterns. According to the International Data Corporation (IDC)[9], the market for wearable devices will experience an annual growth rate of 20.3%, culminating in 213.6 million units being shipped in 2020, as shown in Figure 1.3.

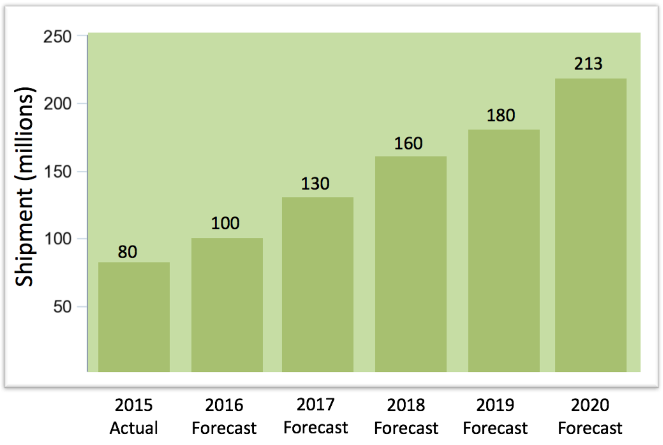


Figure 1.3: Worldwide Wearable Device Forecast.

Wearable devices (e.g., fitness trackers) are equipped with smart sensors, and make use of a web connection, usually using Bluetooth to connect wirelessly to smartphones. They use these sensors to connect to users, and they help users to achieve goals such as staying fit, active, losing weight, burning calories, capturing and monitoring vital signs and other health information.

Smartphones push this health-related data to vendors' cloud platforms, enabling users to analyze summary statistics on-line and adjust their habits. Third-parties including health insurance providers now offer discounts and financial rewards in exchange for such private information and evidence of healthy lifestyles. Given the associated monetary value, the authenticity and correctness of the activity data collected becomes imperative.

As fitness wearables grow more sophisticated, they collect more and more information about user health and movements which pose a threat towards users' security and privacy. Wearable devices raise unique security and privacy vulnerabilities, as shown by some recent studies [19–22]. However, security and privacy on wearable fitness trackers will be of paramount importance. But security and privacy remain a challenge. In particular, many of the small, lightweight and mass-produced devices that comprise IoT are not conducive to robust security protections. Given the current poor state of security on connected devices, they will demonstrate an increasingly attractive target to criminals who look for easy targets to attack.

[9] `http://www.idc.com/getdoc.jsp?containerId=prUS41530816`

### Security Analysis, Reverse Engineering and Spoofing Popular Fitness Devices

Data collected by fitness trackers have been used as evidence in court trials in the US, as reported by Forbes Magazine [23] in 2014. Police and attorneys have started to recognize wearable devices as the human body's "black box", the NY Daily News [24] wrote in April 2016. Some health insurance companies recently started to offer discounts if the insured persons provide personal data from their fitness trackers. This could attract scammers who manipulate the tracked data to fraudulently gain financial benefits or even influence a court trial. We analyze a representative sample's different wearable fitness tracking devices with diverse security and protection mechanisms including devices from top manufacturers and less well-known brands.

**Contribution.** We provide researchers and practitioners with an overview of the current technologies employed in fitness tracking products and highlight the needs for massive improvement in order to ensure customer privacy and secure operation of fitness tracking products and services [25]. We analyze communications between the tracker's associated fitness app (installed on the smartphone) and the cloud service, as fitness trackers typically utilize the user's smartphone to upload data to the cloud service. We investigate the requirements that must be taken into account while handling critical data, such as i) Data Confidentiality, 2) Data Integrity, and 3) Data Authenticity. We also consider several criteria to analyze the safety and robustness of fitness trackers' communication protocols (between fitness app and cloud service) such as *Use of End-to-End data encryption*, *Data In-transit Encryption (e.g., HTTPS protocol)*, *Data at Rest Encryption (e.g., Encrypted Data-base)*, *Existence of Proprietary Encoding*, *Presence of Data Integrity Checking Mechanisms*, and *Use of SSL Certificate Pinning.*

We reveal that several products have significant deficiencies with regard to above-mentioned requirements. We show that designing and deploying efficient and effective security controls in a robust and concrete way are overlooked by wearable manufactures. We document successful injection of fabricated data (along with Proof-of-Concept attack) and demonstrate malicious users to obtain financial benefits can forge activity records that appear valid to cloud services, which are not backed by genuine activity history.

### Security Analysis, Reverse Engineering and Spoofing Advanced Fitness Devices

As we explain in Chapter 6, we selected a representative subset of different fitness trackers to conduct security analysis. Preliminary results suggest all devices suffer from serious security flaws and are subject to MITM attacks. Fitbit trackers are the only exception, as they use end-to-end encryption. Hence they seem to be the most secure fitness trackers on the market, motivating us to choose Fitbit as the target of our security study.

Given the value fitness data has towards litigation and income, researchers analyzed potential security and privacy vulnerabilities specific to activity trackers [26–29]. Rahman et al. investigated the communication protocol used between early Fitbit wearables and web servers, as well as possible attacks [26]. Cyr et al. studied the different layers of the Fitbit Flex ecosystem and argued correlation and MITM attacks are feasible [27]. Recent work documents firmware vulnerabilities found in Fitbit trackers [28], and the reverse engineering of cryptographic primitives and authentication protocols [29]. The identified weaknesses have been meanwhile patched by the vendor. However, as rapid innovation is the primary business objective, security considerations remain an afterthought rather than embedded into product design. Therefore, wider adoption of wearable technology is hindered by distrust [30, 31].

**Contribution.** We undertake an in-depth security analysis of some of the most popular Fitbit trackers [32]. We reveal serious security and privacy vulnerabilities present in a representative sample of Fitbit devices which, although difficult to uncover, can be exploited at scale once identified. Specifically, we analyze the primitives governing the communication between trackers and cloud-based services, implement an open-source tool to extract sensitive personal information in human-readable format, and demonstrate malicious users can inject spoofed activity records to gain personal benefits. To circumvent end-to-end protocol encryption implemented in the latest firmware, we perform hardware-based RE and document successful injection of falsified data that appears legitimate to the Fitbit cloud.

## 1.2 Publications

Part of the research presented in this dissertation during my PhD program culminated in peer-reviewed conference, workshop, book chapter and journal publications. In the following, sections 1.2.1, 1.2.2, and 1.2.3 list the publications, in chronological order, including published and currently submitted works that have resulted from this PhD thesis.

### 1.2.1 Conference and Workshop Publication

[C1] Hossein Fereidooni, Jiska Classen, Tom Spink, Paul Patras, Markus Miettinen, Ahmad-Reza Sadeghi, Matthias Hollick, Mauro Conti. Breaking Fitness Records without Moving: Reverse Engineering and Spoofing Fitbit. In Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2017), in press, Atlanta, Georgia, USA, September 18-20, 2017.

[C2] Hossein Fereidooni, Tommaso Frassetto, Markus Miettinen, Ahmad-Reza Sadeghi, and Mauro Conti. Fitness Trackers: Fit for Health but Unfit for Security and Privacy. In Proceedings of the 2nd IEEE International Workshop on Safe, Energy-Aware, & Reliable Connected Health (CHASE 2017 workshop: SEARCH 2017), Philadelphia, Pennsylvania, USA, July 17-19, 2017.

[C3] Hossein Fereidooni, Mauro Conti, Alessandro Sperduti, Danfeng Yao,. ANASTASIA: ANdroid mAlware detection using STAtic analySIs of Applications. In Proceedings of 8th IFIP International Conference on New Technologies, Mobility & Security (NTMS 2016), Cyprus, November 21-23, 2016.

[C4] Emmanouil Panaousis, Tansu Alpcan, Hossein Fereidooni, Mauro Conti,. Secure Message Delivery Games for Device-to-Device Communications. In Proceedings of 5th International Conference, on Decision and Game Theory for Security (GameSec 2014), Los Angeles, CA, USA, November 6-7, 2014.

### 1.2.2 Book Chapter Publication

[B1] Hossein Fereidooni, Veelasha Moonsamy, Mauro Conti, Lejla Batina. Efficient Classification of Android Malware in the wild using Robust Static Features. In Protecting Mobile Networks and Devices: Challenges and Solutions, CRC Press - Taylor & Francis, 2016 (Editors: Weizhi Meng, Xiapu Luo, Jianying Zhou, Steven Furnell).

### 1.2.3 Magazine and Journal Publication

[J1] Parvez Faruki, Hossein Fereidooni, Vijay Laxmi, Mauro Conti, Android Code Protection via Obfuscation Techniques: Past, Present and Future Directions. Under Submission at: “Computer Science Review” Journal (Elsevier), 2016.

# Part I

# Tackling Mobile Malware

# Chapter 2

# Secure Message Delivery Games for D2D Communications

Nowadays, the vast demand for anytime-anywhere wireless broadband connectivity has posed new research challenges. As mobile devices are capable of communicating in both cellular (e.g., LTE) and unlicensed (e.g., IEEE 802.11) spectrum, the Device-to-Device (D2D) networking paradigm has the potential to bring several immediate gains. Networking based on D2D communication [33–36] not only facilitates wireless and mobile peer-to-peer services but also provides energy efficient communications, locally offloading computation, offloading connectivity and high throughput.

Another emerging feature of D2D is the establishment and use of multi-hop paths to enable communications among non-neighboring devices. In multi-hop D2D communications, messages are delivered from a source to a destination via intermediate devices, independently of operators' networks. Relay by device has been proposed by the Telecommunication Standardization Advisory Group (TSAG) in the International Telecommunication Union Telecommunication Sector (ITU-T).

A key question in *multi-hop D2D networks* is, which route should the originator of a message choose to send it to an intended destination? To motivate the application of our model, we emphasize in the need for *localized applications*. In particular, these applications run in a collaborative manner by groups of devices at a location where telecommunications infrastructures:

- are not presented at all, e.g., underground stations, airplanes, cruise ships, parts of a motorway, and mountains;
- have collapsed due to physical damage to the base stations or insufficient available power, e.g., areas affected by a disaster such as earthquake;
- are over congested due to an extremely crowded network, e.g., for events in stadiums, and public celebrations.

Furthermore, relay by device can be leveraged for commercial purposes such as advertisements and voucher distributions for instance in large shopping centers. This is considered a more efficient way of promoting businesses than other traditional methods such as email broadcasting and SMS messaging due to the immediate identification of the clients in a surrounding area. Home automation and building security are another two areas that multi-hop message delivery using D2D communications is likely to overtake our daily life in the near future. Lastly, multi-hop D2D could be leveraged towards the provision of anonymity against cellular operators as proposed in [37].

Due to the large number of areas D2D communications are applicable to, devices are likely to be an ideal target for attackers. Malware for mobile devices evolves in the same trend as malware for PCs. It can spread for instance through a Multimedia Messaging System (MMS) with infected attachments, or an infected message received via Bluetooth aiming at stealing users' personal data or credit stored in the device. An example of a well-known worm that propagates through Bluetooth was Cabir [38], which consists of a message containing an application file called `caribe.sis`. Mabir, a variant of Cabir, was spread also via MMS by sending out copies of itself as a .sis file. Van Ruitenbeek et al. [39] investigated the effects of MMS viruses that spread by sending infected messages to other devices. In addition, Bose and Shin [40] examined the propagation of malware that spread via SMS or MMS messages and short-range radio interfaces while Polla et al. [41] have made a thorough survey on mobile malware.

**Contribution.** In this chapter, we assume that each device has some host-based intrusion detection capabilities (e.g., antivirus). Therefore, a device would be able to detect malicious application-level events as in [42]. We assume that each device has its own detection rate which contributes towards the overall detection rate of the routes that this device is on. To increase the level of security of a message delivery, the route with the highest detection capabilities must be selected to relay the message to the destination. Apart from security, energy consumption is of crucial importance because devices (e.g., smartphones) usually impose strict energy constraints. This becomes more important due to the limited CPU and memory capabilities that devices have, which entail higher energy cost as opposed to cases where no message inspection takes place.

In this chapter, we propose the *Secure Message Delivery* (SMD) protocol. The primary objective of this protocol is to choose the most secure path to deliver a message from a sender to a destination in a multi-hop D2D network. SMD can work on top of underlying physical and MAC layer protocols [14, 15]. Apart from security, SMD respects the energy costs and Quality-of-Service (QoS) of each route. This happens by giving certain weights to each of the involved parameters (security, energy, QoS) with more emphasis to be put on security.

We formulate *Secure Message Delivery Games* (SMDGs) in order to derive an optimal behavior for the SMD. In these games, one or more adversaries, abstracted by the *attacker*, aim at increasing the security damage, incurred to the defender (i.e., network), by injecting malicious messages into the D2D network. On the other hand, the defender chooses the "best route" for message delivery.

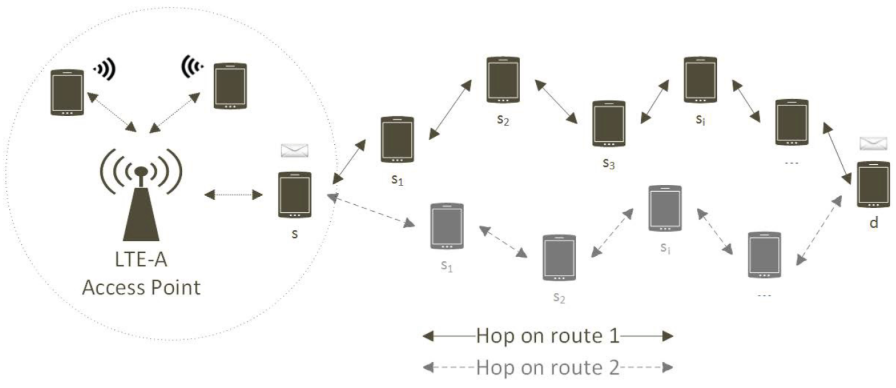


Figure 2.1: Example of a D2D network.

In SMDGs, the utility of the defender is influenced by: (i) the probability of the delivered message to be correctly classified as malicious or benign before it is delivered to the intended destination; (ii) the *energy cost* associated with *message forwarding*, and *message inspection* on relay devices during message delivery; and (iii) the QoS of the message communications on the chosen D2D path.

The remainder of this chapter is organized as follows. In Section 2.1 we present the system model whilst Section 2.2 formulates the SMDGs and it provides their solutions. In Section 2.3 the SMD routing protocol for D2D networks is described. We present some preliminary simulation results in Section 2.4 for different number and types of malicious messages distributions, and different D2D network profiles. Section 2.5 summarizes the most relevant related work within the intersection of game theory, security and mobile distributed networking. Section 2.6 concludes this chapter by summarizing its main contributions, limitations and highlighting our plans for future work.

## 2.1 System Model

This section presents our system model and its different components. We assume a multi-hop Device-to-Device (D2D) communication network that extends a cellular network (e.g., LTE Advanced) as illustrated in Figure 2.1.

Data transmission takes place in the application layer in the form of data units called *messages*. Any device can be the source (`s`) of a message and each message has a final destination (`d`). When `d` is not within the transmission range of `s`, a route must be established to allow message delivery. Therefore, there is an apparent need for the devices to collaborate to relay messages towards `d`.

We refer to the $i$-th mobile device by $s_i$, and define the set of all legitimate mobile devices in a mobile network as $S \triangleq \{s_i\}$. When the $l$-th type of message, denoted by $m_l$, has to be delivered to a destination device (`d`), a route must be chosen by `s` to serve that purpose. Formally, we denote route $j$ by $r_j$. The devices on $r_j$ must forward $m_l$ towards `d`. We define the set of all routes from `s` to `d` as $R \triangleq \{r_j\}$, and the set of all devices that constitute $r_j$ is expressed by $S_j$.

We denote the set of all different types of messages[1] by $\mathcal{M}$. This equals the union of the set of all malicious undetected messages ($\mathcal{M}_m$), and the set of all benign messages ($\mathcal{M}_b$). Therefore, $\mathcal{M} \triangleq \mathcal{M}_m \cup \mathcal{M}_b$. An *attack* is defined as the attempt of the attacker to harm d through the delivery of a malicious message. When $m_l$ stays undetected prior to be delivered to d, we say that it causes harm $\mathcal{H}_l$, which is associated with the damage caused to an asset that the device holds (e.g., data loss). We also assume that any false alarm has loss equivalent to $\mathcal{F}$. The security effectiveness of a device against a malicious message is denoted by $\delta(s_i, m_l)$, and it is equivalent to the detection rate of an attack. The vector $\Delta(s_i) \triangleq \langle \delta(s_i, m_1), \ldots, \delta(s_i, m_\psi) \rangle$ defines all the different values of security effectiveness of $s_i$ with regard to the different messages. For more convenience, Table 1 summarizes the notation used in this work.

| | | | |
|---|---|---|---|
| $S$ | Set of devices | $s_i$ | device $i$ |
| $m_l$ | message $l$ | $h^\star$ | Maximum possible route length in hops |
| $\mathtt{s}$ | Message source | $\mathtt{d}$ | Message destination |
| $P^A$ | Attacker | $P^D$ | Defender |
| $R$ | Set of routes from $\mathtt{s}$ to $\mathtt{d}$ | $r_j$ | $j$-th route from $\mathtt{s}$ to $\mathtt{d}$ |
| $S_j$ | Set of devices on $r_j$ | $\mathcal{M}$ | Set of messages |
| $\mathcal{M}_m$ | Set of malicious messages | $\mathcal{M}_b$ | Set of benign messages |
| $\delta(s_i, m_l)$ | Security effectiveness of $s_i$ against $m_l$ | $\Delta(s_i)$ | Security effectiveness vector of $s_i$ |
| $\sigma_i$ | Security energy cost of $s_i$ | $f_i$ | Forwarding energy cost of $s_i$ |
| $\epsilon_i$ | Total message delivery energy cost of $s_i$ | $\mathtt{e_j}$ | Total energy cost on $r_j$ |
| $T$ | Lifetime of a Nash message delivery plan | $\mathtt{E}$ | Vector of energy costs, $\forall\ r_j$ from $\mathtt{s}$ to $\mathtt{d}$ |
| $\mathtt{h_j}$ | Number of hops on $r_j$ | $\mathtt{H}$ | Vector of hops, $\forall\ r_j$ from $\mathtt{s}$ to $\mathtt{d}$ |
| $C^{(s_i)}$ | Confusion matrix of $s_i$ | $C^{(r_j)}$ | Confusion matrix of $r_j$ |
| $\mathcal{F}$ | False alarm loss | $\mathcal{H}_l$ | Security damage if $m_l$ undetected |
| $w_s$ | Security cost weight | $w_{fa}$ | False alarm cost weight |
| $w_e$ | Energy cost weight | $w_q$ | QoS cost weight |
| $D$ | Payoff matrix of $P^D$ | $A$ | Payoff matrix of $P^A$ |
| $d_{jl}$ | Utility of $P^D$ for $(r_j, m_l)$ | $a_{jl}$ | Utility of $P^A$ for $(r_j, m_l)$ |
| $\mathbf{D}^*$ | Nash message delivery plan | $r^*$ | Nash route |

Table 2.1: The SMD Protocol Notation.

### 2.1.1 Collaborative Detection

In our model, the aim of the devices is to detect malicious messages injected through an *entry point* into the D2D network. We assume that each device that

[1]Very often, we use the terms *types of messages*, and *messages* interchangeably according to the context.

receives a message is responsible for inspecting it by using its detection capabilities to the best level possible. Based on the results of the detection, the device updates the *confusion matrix* of the route. This is a right stochastic matrix, which holds the probability of the different messages being detected correctly, being confused with other messages or being identified as benign. This matrix type was initially proposed in [43] (p. 100).

Each device that receives a message, follows exactly the same procedure until the message arrives at d. At this point, the confusion matrix should have taken the most accurate detection values (ideally is the identity matrix) due to all inspections undertaken by the devices on this route. Collaborative detection of a malicious message along a path requires forwarding state information, which includes results of the inspections previously conducted on the message. This prevents unnecessary duplication of inspections, thus saving energy.

### 2.1.2 Device Confusion Matrix

Given the set of messages $\mathcal{M}$, the linear mapping $C^{(s_i)}\colon \mathcal{M} \to \mathcal{M}$ describes the *detection capability* of $s_i$ for a message received. This capability is modeled using a stochastic *device confusion matrix* as follows:

$$C^{(s_i)} \triangleq [C^{(s_i)}_{uv}]_{\psi\times\psi}, \text{ where } 0 \leq C^{(s_i)}_{uv} \leq 1, \ \forall u, v \in \{1, \dots, \psi\}. \tag{2.1}$$

A confusion matrix value $C^{(s_i)}_{uv}$ denotes the probability of a message $u$ being reported as message $v$. If $m_u \neq m_v$, then the device confuses one message for another. Such misinterpretation is beneficial for the attacker because the attack associated with the message is not mitigated. If $m_u \in M_m$, and $m_v \in \mathcal{M}_b$, $C^{(s_i)}_{uv}$ is the probability of the D2D network failing to report an attack. If $m_u \in \mathcal{M}_b$, and $m_v \in \mathcal{M}_m$, then $C^{(s_i)}_{uv}$ is the probability of a *false alarm*. One of the objectives of the D2D network must be the confusion matrix to become the identify matrix (*no confusion*) by the time a message is delivered to d. In another sense, if the confusion matrix is the identity matrix, every single malicious message can be detected before it infects d. However this case is not likely to be achieved in practice due to, for instance, 0-day vulnerabilities, and other misclassification errors. To motivate the computation of confusion matrices we present the following example.

Example 1. Assume $S = \{s_1, s_2\}$, and $\mathcal{M} = \{m_1, m_2, m_3\}$. Also, $m_1, m_2 \in \mathcal{M}_m$, and $m_3 \in \mathcal{M}_b$. We also set the false alarm rate equal to 0.05 for both devices. The security effectiveness vectors are $\Delta(s_1) = \langle 0.5, 0.8\rangle$ and $\Delta(s_2) = \langle 0.75, 0.6\rangle$. We also assume that none device confuses a malicious message for another malicious message and therefore $C^{(r_j)}_{uv} = 0, \forall\, u \neq v,\, m_u, m_v \in \mathcal{M}_m$. Then the devices confusion matrices are the following:

$$C^{(s_1)} = \begin{pmatrix} 0.5 & 0 & 0.5 \\ 0 & 0.8 & 0.2 \\ 0.05 & 0.05 & 0.9 \end{pmatrix}, \ C^{(s_2)} = \begin{pmatrix} 0.75 & 0 & 0.25 \\ 0 & 0.6 & 0.4 \\ 0.05 & 0.05 & 0.9 \end{pmatrix}. \tag{2.2}$$

### 2.1.3 Route Confusion Matrix

Similarly, given the set of messages $\mathcal{M}$, the linear mapping $C^{(r_j)}\colon \mathcal{M} \to \mathcal{M}$ describes the final detection capability of the D2D network on $r_j$. This is the *route* *confusion matrix* for $r_j$ derived from the confusion matrices of the devices that constitute this route. In the problem we examine, the order of detectors does not matter. Therefore, the confusion matrix for each combination can be computed prior to the message delivery.

An advanced way of deriving the route confusion matrix values is to use a boosting meta-algorithm such as *Adaboost* [44]. If we consider that each device detector is a weak classifier then boosting makes classifiers focusing on data that was previously misclassified. The underlying concept of Adaboost is that several weak classifiers can yield a strong classifier. The confusion matrix of a route is a representation of the weighted classifiers on the devices. It is worth mentioning here that boosting is effective only when all devices trust each other. For the boosting scheme to work there is a need for a broadcasting system which updates the classifiers and pre-sets confusion matrices for the combination of detectors. Nevertheless, such a system has to be implemented anyway for updating virus signatures and anomaly detector parameters. Thus, the update of the classifiers can be piggybacked on top of them.

A"naive" alternative to boosting can be a *linear combination algorithm* where each device contributes linearly to the final route detection capability by some weight determined by characteristics of the route (e.g., #hops).

### 2.1.4 Energy Costs and QoS

Each time a device receives a message it spends energy: $(i)$ to detect any sign of malice (security energy cost, $\sigma_i$) and $(ii)$ to forward a message towards $\mathtt{d}$ (forwarding energy cost, $f_i$). The former is determined by all required intrusion detection tasks undertaken during message inspection. The second is related to the energy spent for relaying the message towards the next-hop on the route from $\mathtt{s}$ to $\mathtt{d}$. We denote by $\epsilon_i$ the *secure message delivery cost* incurred to a device during message delivery. Formally, we have that $\forall s_i \in S: \ \epsilon_i \triangleq \sigma_i + f_i$.

The total *route energy cost* on $r_j$, when a message is delivered over $r_j$, is denoted by $\mathtt{e_j}$, and it is derived by $\mathtt{e_j} = \sum_{s_i \in S_j} \epsilon_i$. The energy costs of all routes between $\mathtt{s}$ and $\mathtt{d}$ are given by the vector $\mathtt{E} \triangleq \langle \mathtt{e_1}, \dots, \mathtt{e_\xi} \rangle$.

Apart from security and energy efficiency, QoS is an important consideration when deciding upon message delivery. We denote by $\mathtt{h_j}$ the number of hops on $r_j$. In this research work, we measure the QoS of a route as $\mathtt{h_j}/\mathtt{h}^\star$, where $h^\star \triangleq N_S - 1$, and $N_S$ is the total number of devices in the D2D network. The number of hops of all routes $r_1, \dots, r_\xi$ from $\mathtt{s}$ to $\mathtt{d}$ are given by $\mathtt{H} \triangleq \langle \mathtt{h_1}, \dots, \mathtt{h_\xi} \rangle$.

In this work, we assume a best effort message delivery service without acknowledgments. Along with having higher end-to-end delay due to this assumption, as the number of hops increases the probability of a message to be lost is higher. This is due to mobility, which is meant to be common in D2D networks. It is worth noting here that our model does not consider real-time multimedia communications because they require higher bandwidth than what a typical multi-hop D2D network provides.

### 2.1.5 Network Profiles

To allow the expression of different *network profiles*, we have defined an importance costs vector $[w_s, w_{fa}, w_e, w_q]$. By $w_s$, we denote the security importance weight which accounts for the level of importance the defender gives to some expected security damage (e.g., data theft); $w_{fa}$ is the importance of the false alarm cost (i.e., cost for dropping an innocent message); $w_e$ is the importance that the defender places into the energy cost which can influence the network lifetime and speed up network fragmentation; and $w_q$ is the importance of the QoS for the defender which accounts for the message success delivery rate and end-to-end delay. This vector allows the network designer to define their *network profile* based on their requirements, measured in terms of security, energy preservation, and QoS.

## 2.2 Secure Message Delivery Games

In this section, we use game theory to model the interactions between a D2D network (the *defender*) and any adversarial entity (the *attacker*). The latter aims at launching an attack against a device by sending a malicious message to it through the network's entry point as depicted in Fig. 2.1. Formally, we define the set of players as $\mathcal{P} \triangleq \{P^D, P^A\}$.

The objective of $P^D$ is to securely deliver a message to the intended destination $\mathtt{d}$. By secure delivery we refer to the message being relayed through the network and collaboratively inspected by the devices on its way to $\mathtt{d}$, in order to mitigate any security risk inflicted by $P^A$. Therefore the security objective of $P^D$ is to correctly detect and filter out malicious messages before they reach their destination. Every request for message delivery to $\mathtt{d}$ defines a *Secure Message Delivery Game* (SMDG).

### 2.2.1 Game Characterization

The SMDG is a non-cooperative two-person zero-sum game. The explanation to the zero-sum nature of SMDG is that we have assumed that the attacker aims at inflicting the highest possible damage to the defender. We could model a game where the benefit of the attacker is smaller than the loss of the defender. However, we have left this for future work along with the investigation of different attacker profiles that are associated with different payoffs. The defender primarily aims at delivering the message securely to $\mathtt{d}$ while the attacker aims at infecting $\mathtt{d}$ with some malware attached to a malicious message as we mentioned previously. The SMDG is a repeated game since players make their decisions once for a pair of $\langle \mathtt{d}, T \rangle$, where $T$ is a predefined timeout, and $\mathtt{d}$ is the destination device for which the game is played. Afterwards, they repeat the game for either every other destination or when $T$ expires. The value of $T$ may depend on the devices' mobility. For instance, high mobility dictates small $T$ in order valid routes to be discovered.

In SMDG, the players make their decisions concurrently without any *order of play*. However, an order of play can be imposed as an alternative where the attacker becomes the *leader* and the defender the *follower* of a Stackelberg game. Nevertheless, this consideration is out of the scope of this research work

### 2.2.2 Strategies and Payoffs

The pure strategies of $P^D$ consists of all routes from $\mathtt{s}$ to $\mathtt{d}$. Therefore, the action set of $P^D$ is defined as $\mathcal{A}^D \triangleq R = \{r_1, r_2, \ldots, r_\xi\}$. On the other hand, the pure strategies of $P^A$ are the different messages that $P^A$ can choose to send to `d`. A message can be one of the following:

$$\{\texttt{malicious}_1, \ldots,\ \texttt{malicious}_\texttt{n}, \texttt{harmless},\ \texttt{surveillance}\} \tag{2.3}$$

Then, the finite action set of the attacker is defined as:

$$\mathcal{A}^A \triangleq \mathcal{M} = \{m_1, \ldots, m_\psi\} = \{m_1, \ldots, m_n\} \cup \{\texttt{harmless}, \texttt{surveillance}\}.$$

We denote by $G_\texttt{d} \triangleq \langle D, A\rangle$ an $\xi \times \psi$ bi-matrix game where the $P^D$ (i.e., row player) has a payoff matrix $D \in \mathbb{R}^{\xi\times\psi}$ and the payoff matrix of $P^A$ (i.e. the column player) is denoted by $A \in \mathbb{R}^{\xi\times\psi}$.

$P^D$ chooses as one of their pure strategies one of the rows of the payoff bi-matrix $(D, A) \triangleq (d_{j,l}, a_{j,l})_{(r_j,m_l)\in[\xi]\times[\psi]}$. For any pair of strategies, $(r_j, m_l) \in [\xi] \times [\psi]$, $P^D$, $P^A$ have payoff values equivalent to $d_{j,l}$ and $a_{j,l}$, respectively. The payoff of the defender for a given pair of players' pure strategies $(r_j, m_l)$ follows:

$$U_D(r_j, m_l) \triangleq d_{j,l} \triangleq -w_s(1 - C_{ll}^{(r_j)})\mathcal{H}_l\ -\ w_{f_a}(1 - C_{ll}^{(r_j)})\mathcal{F} - w_e e_j - w_q h_j. \tag{2.4}$$

Generally, the first term is the expected security damage (e.g., data theft) inflicted by the attacker due to malicious messages being undetected while the second term expresses the expected cost of the defender due to false alarms. This accounts for benign messages that are dropped due to being detected as malicious. The next to last term is the energy cost of the defender when message delivery takes place over $r_j$ while the last term expresses the expected QoS experienced on this route. Since players act independently, we can enlarge the strategy spaces, so as to allow the players to base their decisions on the outcome of random events. Therefore we consider the mixed strategies of both $P^D$ and $P^A$. The mixed strategy $\mathbf{D} \triangleq [q_1, \ldots, q_\xi]$ of the defender is a probability distribution over the different routes from `s` to `d`, where $q_j$ is the probability of delivering a message via $r_j$. We refer to a mixed strategy of $P^D$ as the *message delivery plan.* On the other hand, the attacker's mixed strategy $\mathbf{A} \triangleq [p_1, \ldots, p_\psi]$ is a probability distribution over the different messages, where $p_l$ is the probability of choosing $m_l$.

When considering mixed strategies, the defender's objective is quantified by the utility function:

$$\begin{aligned} U_D(\mathbf{D}, \mathbf{A}) \ &= \sum_{j=1}^{\xi}\sum_{l=1}^{\psi} q_j d_{j,l}\, p_l = -w_s[\sum_{m_l\in\mathcal{M}_m}\ \sum_{r_j\in R} q_j\,(1 - C_{ll}^{(r_j)})\, p_l\, \mathcal{H}_l]\ - \\ &w_{f_a}[\sum_{m_l\in\mathcal{M}_b}\ \sum_{r_j\in R} q_j\,(1 - C_{ll}^{(r_j)})\, p_l\, \mathcal{F}]\ - w_e\mathbf{D}\mathtt{E}^T\ - w_q\mathbf{D}\mathtt{H}^T, \\ &\text{where } j \in \{1, \ldots, \xi\},\ l \in \{1, \ldots, \psi\}. \end{aligned} \tag{2.5}$$

Because SMDG is a zero-sum game, the attacker's utility is given by $U_A(\mathbf{D}, \mathbf{A}) = -U_D(\mathbf{D}, \mathbf{A})$. This can be interpreted as, the attacker can cause the maximum damage to the defender.

### 2.2.3 Nash Equilibrium

SMDG is a two-person zero-sum game with finite number of actions for both players, and according to Nash [45] it admits at least a Nash Equilibrium (NE) in mixed strategies. Saddle-points correspond to Nash equilibria as discussed in [43] (p. 42).

The following result, from [46], establishes the existence of a saddle (equilibrium) solution in the games we examine and summarizes their properties.

**Theorem 1** (Saddle point of the SMDG)**.** *The Secure Message Delivery Game defined admits a saddle point in mixed strategies,* $(\mathbf{D}^*, \mathbf{A}^*)$*, with the property that*

$$\mathbf{D}^* = \arg\max_{\mathbf{D}} \min_{\mathbf{A}} U_D(\mathbf{D}, \mathbf{A}),\ \forall \mathbf{A} \ \ \textit{and} \ \ \mathbf{A}^* = \arg\max_{\mathbf{A}} \min_{\mathbf{D}} U_A(\mathbf{D}, \mathbf{A}),\ \forall D.$$

*Then, due to the zero-sum nature of the game the following holds:*

$$\max_{\mathbf{D}} \min_{\mathbf{A}} U_D(\mathbf{D}, \mathbf{A}) = \min_{\mathbf{A}} \max_{\mathbf{D}} U_D(\mathbf{D}, \mathbf{A}).$$

*The pair of saddle point strategies* $(\mathbf{D}^*, \mathbf{A}^*)$ *are at the same time security strategies for the players, i.e., they ensure a minimum performance regardless of the actions of the other. Furthermore, if the game admits multiple saddle points (and strategies), they have the ordered interchangeability property, i.e., the player achieves the same performance level independent from the other player's choice of saddle point strategy.*

Our results can be extended to non-zero sum, bi-matrix games. In the latter case, the existence of a NE is also guaranteed, but the additional properties hold only in the case where the attacker's utility is a positive affine transformation (PAT) of the defender's utility.

**Definition 1.** *The Nash message delivery plan, denoted by* $\mathbf{D}^*$*, is the probability distribution over the different routes, as determined by the NE of the SMDG.*

The minimax theorem states that for zero sum games NE and minimax solutions coincide. Therefore, $\mathbf{D}^* = \arg\min_{\mathbf{D}} \max_{\mathbf{A}} U_A(\mathbf{D}, \mathbf{A})$. This means that regardless of the strategy the attacker chooses, the *Nash message delivery plan* is the defender's security strategy that guarantees a minimum performance.

We can convert the original matrix game into a linear programming (LP) problem and make use of some of the powerful algorithms available for LP to derive the equilibrium. For a given mixed strategy $\mathbf{D}$ of $P^D$, $P^A$ can cause a maximum damage to $P^D$ by injecting a message $\widehat{m}$ into the D2D network. In that case, the utility of $P^D$ is minimized and it is denoted by $U_D(\mathbf{D}, \widehat{m})$ (i.e., $U_D^{\min} = U_D(\mathbf{D}, \widehat{m})$). Formally, $P^D$ seeks to solve the following LP:

$$\max_{\mathbf{D}} U_D(\mathbf{D}, \widehat{m}\,)$$

$$\text{subject to} \begin{cases} U_D(\mathbf{D}, m_1) - U_D(\mathbf{D}, \widehat{m})e \geq 0 \\ \qquad\vdots \\ U_D(\mathbf{D}, m_\psi) - U_{\mathcal{D}}(\mathbf{D}, \widehat{m})e \geq 0 \\ \mathbf{D}e = 1 \\ \mathbf{D} \geq 0 \end{cases} \Rightarrow \begin{cases} \sum_{j=1}^{\xi} q_j d_{j,1} - U_D(\mathbf{D}, \widehat{m})e \geq 0 \\ \qquad\vdots \\ \sum_{j=1}^{\xi} q_j d_{j,\psi} - U_D(\mathbf{D}, \widehat{m})e \geq 0 \\ \mathbf{D}e = 1 \\ \mathbf{D} \geq 0 \end{cases}$$

In this problem, $e$ is a vector of ones of size $\xi$.

## 2.3 The Secure Message Delivery Protocol

In this section, we present the *Secure Message Delivery* (SMD) routing protocol whose routing decisions are taken according to the *Nash message delivery plan.* SMD increases security in a D2D network by mitigating the risk of adversaries harming legitimate devices via, for instance, malware attached to messages. SMD has been designed based on the mathematical findings of the SMDG and its main goal is to maximize $U_D(\mathbf{D}, \mathbf{A})$.

According to SMD, each time a request for message delivery to `d` is issued, `s` has to compute the *Nash message delivery plan* by solving an SMDG for this destination. To this end, the device uses its latest information about confusion matrices, QoS and energy costs. Then, the message is relayed and collaboratively inspected by the devices on its way to `d`. The objective of the network (i.e., $P^D$) is to correctly detect and filter out malicious messages before they infect `d`.

### 2.3.1 SMD Considerations

The SMD protocol takes routing decisions that increase the probability of detecting malicious messages. Apart from security, SMD utilizes standard approaches to take into account (i) the energy costs resulting from message forwarding and inspection, and (ii) the QoS of the chosen route. According to SMD, the devices maintain routing tables with at least three metrics per route:

- the route confusion matrix,
- the total expected energy cost on this route and,
- the shortest path in terms of number of hops (i.e., QoS).

If the only factor affecting the routing decision was security, then the route with the highest detection capability would be always chosen. This would result to a faster depletion of this route's energy as opposed to when a combination of different routes is chosen. Consequently, the D2D network would suffer fragmentation across the entire topology and consequently security would be reduced. This is the motivation behind considering energy costs upon path selection. Nevertheless, while the shortage of a device's battery can be solved by, for example, by using mobile solar cells as discussed in [35], and QoS might not be so much of a concern for message communications, secure message delivery remains a critical issue.

The formulation of the defender's utility function allows a device to decide how important the expected QoS and energy costs are compared to the expected security damage. For instance, the defender can decide to set the energy costs equal to 0 when a constant source of energy supply is available or to give a higher importance to security losses than QoS.

Due to the best effort nature of the communications (as a result of the multi-hop environment) the higher the number of hops (i.e., QoS) of a route the more likely a message is to be lost during its delivery via that route. QoS accounts for a successful message delivery rate and therefore the defender might never really want to ignore it. In general, SMD allows network designers to customize the protocol based on the *network profile* of the D2D network. In any case, all defender's preferences are reflected to the *Nash message delivery plan.*

### 2.3.2 Routing

Getting inspired by the functionalities of the well-known *Dynamic Source Routing* (DSR) [47] routing protocol, SMD consists of two main stages. [2]

**SMD - Stage I.**
In the first stage, `s` broadcasts a Route REQuest ($\texttt{RREQ}_\texttt{d}$) to discover routes towards `d`. Each device that receives a $\texttt{RREQ}_\texttt{d}$ acts similarly by broadcasting it towards `d` and caches relevant information (i.e., originator of the request, ID of the $\texttt{RREQ}_\texttt{d}$). When `d` receives a $\texttt{RREQ}_\texttt{d}$, it prepares the $\texttt{RREP}_\texttt{d}$ and sends it back towards `s` by using the reverse route which is built during the delivery of $\texttt{RREQ}_\texttt{d}$ to `d`. Each $\texttt{RREP}_\texttt{d}$ carries information about the route. This information includes the route confusion matrix ($E_1$), the total energy costs due to inspection and forwarding on this route ($E_2$), and the total number of hops ($E_3$). All three fields are updated while the $\texttt{RREP}_\texttt{d}$ is traveling back to `s`.

Each device, involved in route discovery, that receives $\texttt{RREP}_\texttt{d}$, it updates $E_1$ by using boosting (e.g., Adaboost) or simply a linear combination algorithm without learning features. The same device (e.g., $s_i$) updates $E_2$ by adding its total energy cost $\epsilon_i$ to the route energy cost. Lastly, $E_3$ is increased by 1 in every hop from `s` to `d`. According to SMD, after `s` sends a $\texttt{RREQ}_\texttt{d}$ it has to await for some timeout $T_{req}$. [3] Within this period `s` aggregates $\texttt{RREP}_\texttt{d}$ messages and updates its routing table with information from those messages.

**SMD - Stage II.**
In the second stage, `s` uses its routing table to solve the SMDG by computing the *Nash message delivery plan* $\mathbf{D}^*$. The latter has a lifetime equivalent to $T$, as defined earlier. Then, `s` probabilistically selects a route according to $\mathbf{D}^*$ to deliver the message to `d`. The chosen route is called the *Nash route* and it is denoted by $r^*$. Note that for the same `d` and before $T$ expires, `s` uses the same $\mathbf{D}^*$ to derive $r^*$, upon a message delivery request. Algorithm 1 summarizes the main SMD functionalities.

It is worth noting here that the complexity of the SMD protocol measured in terms of the number of messages exchanged in performing route discovery is $\mathcal{O}(2N_S)$, where $N_S$ is the total number of devices in the D2D network.

## 2.4 Performance Evaluation

### 2.4.1 Simulation Parameters

In this section, we evaluate the performance of SMD by simulating 30 devices and 6 routes between `s` and `d`. [4] The number of devices per route is selected randomly and the maximum number of devices per route has been set to 10. The number of malicious messages vary from 2 to 20 with an incremental step of 2.

We consider different network profiles to assess the performance of the SMD protocol. Note here that the network profile refers to the preference of the D2D network

[2]In the DSR protocol a node rebroadcasts a RREQ message only the first time that it is received. In addition, the route is accumulated in the message and reversed by the receiver.

[3]Setting the time-out T is out of scope in this work.

[4]Numeric analysis: `https://github.com/manpan/game_theoretic_routing`

```
Data: $\mathtt{s}, \mathtt{d}, m_l$
Result: $m_l$ delivered
STAGE 1:
 $\mathtt{s}$ seeks for a route to $\mathtt{d}$ by broadcasting $\mathtt{RREQ_d}$
 if device $s_i$ receives $\mathtt{RREQ_d}$ then
     if $s_i \neq \mathtt{d}$ then
         $\mathtt{s} \leftarrow s_i$
         Execute Stage 1
     else
         Send an $\mathtt{RREP_d}$ back towards $\mathtt{s}$ using the reverse route $r_j$
     end
end
STAGE 2:
 if device $s_i$ receives $\mathtt{RREP_d}$ then
     if $s_i \neq \mathtt{s}$ then
         Update $C^{(r_j)}$, $\mathtt{e_j}$, $\mathtt{h_j}$
         Attach $\langle C^{(r_j)}, \mathtt{e_j}, \mathtt{h_j} \rangle$ to the $\mathtt{RREP_d}$
         Relay $\mathtt{RREP_d}$ back towards $\mathtt{s}$
     else
         Cache $\langle C^{(r_j)}, \mathtt{e_j}, \mathtt{h_j} \rangle$ to the routing table
         break;
     end
end
$\mathtt{s}$: Derive the Nash message delivery plan $\mathbf{D}^*$
 $\mathtt{s}$: Choose $r^*$ probabilistically as dictated by $\mathbf{D}^*$
 $\mathtt{s}$: Deliver $m_l$ to $\mathtt{d}$ over $r^\star$
```

**ALGORITHM 1:** SMD Stages.

in terms of security (i.e., risk appetite), QoS (i.e., delay in message delivery), energy cost (i.e., spent for message inspection and message forwarding), and false alarm (probability of dropping benign messages) as determined by the *cost importance vector*.

We have used a uniform random generator to create the security effectiveness values for all devices. From these values the simulator creates all devices' confusion matrices. Then, we derive the route confusion matrices by using the Algorithm 2. Note that Algorithm 2 is executed by each device at the step of Algorithm 1 where $C^{(r_j)}$ is updated. This is a linear algorithm (less efficient than boosting due to lack of learning features) which allows us to get some preliminary results about the performance of SMD. This algorithm implements a weighted method according to which each device contributes to the route security effectiveness by

*(Device security effectiveness)* $\times$ *(1/Maximum number of hops in the network).*

| **Network Profile** | $w_s$ | $w_{fa}$ | $w_e$ | $w_q$ | **Network Profile** | $w_s$ | $w_{fa}$ | $w_e$ | $w_q$ |
|---|---|---|---|---|---|---|---|---|---|
| `Security` | 10 | 0.5 | 0 | 0 | `Security & Energy Efficiency` | 5 | 0.5 | 5 | 0 |
| `Security & QoS` | 5 | 0.5 | 0 | 5 | `Security & QoS & Energy Efficiency` | 4 | 0.5 | 3 | 2.5 |

Table 2.2: The importance cost vectors used in our simulations.

```
Data: $C^{(s_i)}, C^{(r_j)}$
Result: Updated $C^{(r_j)}_{uv}$
for $u \in \mathcal{M}$ do
    for $v \in \mathcal{M}$ do
        if $u \in \mathcal{M}_m$ then
            if $v == u$ then
                $C^{(r_j)}_{uv} \leftarrow C^{(s_i)}_{uv}/h^\star + C^{(r_j)}_{uv}$
            end
            if $v \in \mathcal{M}_b$ then
                $C^{(r_j)}_{uv} \leftarrow 1 - C^{(r_j)}_{uu}$
            else
                // probability a malicious message $u$ to be confused
                   with another malicious message
                $C^{(r_j)}_{uv} \leftarrow 0$
            end
        end
        if $u \in \mathcal{M}_b$ then
            if $v \notin \mathcal{M}_b$ then
                // $f_a$: device false alarm rate
                $C^{(r_j)}_{uv} \leftarrow f_a/h^\star + C^{(r_j)}_{uv}$
                $f^{route}_a \leftarrow C^{(r_j)}_{uv}$
            else
                // $f^{route}_a$: route false alarm rate
                $C^{(r_j)}_{uv} \leftarrow 1 - f^{route}_a$
            end
        end
    end
end
```

**ALGORITHM 2:** How a device $s_i$ updates the route confusion matrix.

The final route detection capability not only depends on the detection capability of each device on the route but also on the number of devices. As a result of this, the longer a route is the better its final security effectiveness. After the route confusion matrices have been derived, the simulator computes the *Nash message delivery plan* for each of the network profiles presented in Table 2.2.
We evaluate the performance of SMD by measuring the defender's expected cost when `s` uses SMD instead of a shortest path routing protocol. According to the latter, `s` chooses the path with the minimum number of hops to `d`. For each message delivery and protocol used we compute the defender's total expected cost which includes security, false alarm, energy and QoS costs.

We have considered 10 Cases each representing a different attacker's action set akin to different number of available malicious messages namely; $2, 4, \ldots, 20$. For each Case we have simulated 1,000 message deliveries for a fixed network topology and we refer to the run of the code for the pair $\langle$Case,#message deliveries$\rangle$ by the term Experiment. We have repeated each Experiment for 25 independent network topologies to compute the standard deviation. We do that for all 10 Cases and each type of *attacker profile*.

In this work we consider 2 different attacker profiles; *Uniform* and *Nash*. A *Uniform* attacker chooses any of the available messages with the same probability whilst a *Nash* attacker plays the attack mixed strategy given by the NE of the SMDG. Therefore, we have totally simulated

*10 (Cases) × 1,000 (Message deliveries) × 25 (Runs of each experiment) × 2 (Attacker profiles) = 500,000 Message deliveries.*

Per message delivery, the simulator chooses an attack sample from the attack probability distribution which is determined by the attacker profile. The simulator aggregates the cost values of each Experiment for both SMD and the shortest path routing protocol.

## 2.4.2 Simulation Results

We have plotted the improvement on the total expected defender's cost when SMD is chosen as opposed to the shortest path routing protocol. The plots illustrate different number of available malicious messages, attacker profiles and importance cost vectors, in Figures 2.2 and 2.3.

From both figures we notice that SMD outperforms the shortest path routing protocol with the highest improvement to be achieved under the "`Security`" network profile. From Fig. 2.2 we notice that the average values of this improvement fluctuate approximately within the range $[30\%, 43\%]$. The second best performance is achieved under the "`Security & QoS`" network profile and it is only slightly better than the improvement we get under the "`Security and Energy Efficiency`" profile. The lowest improvement is noticed under the "`Security & QoS & Energy Efficiency`" network profile with the mean values to be within the range $[10\%, 18\%]$.

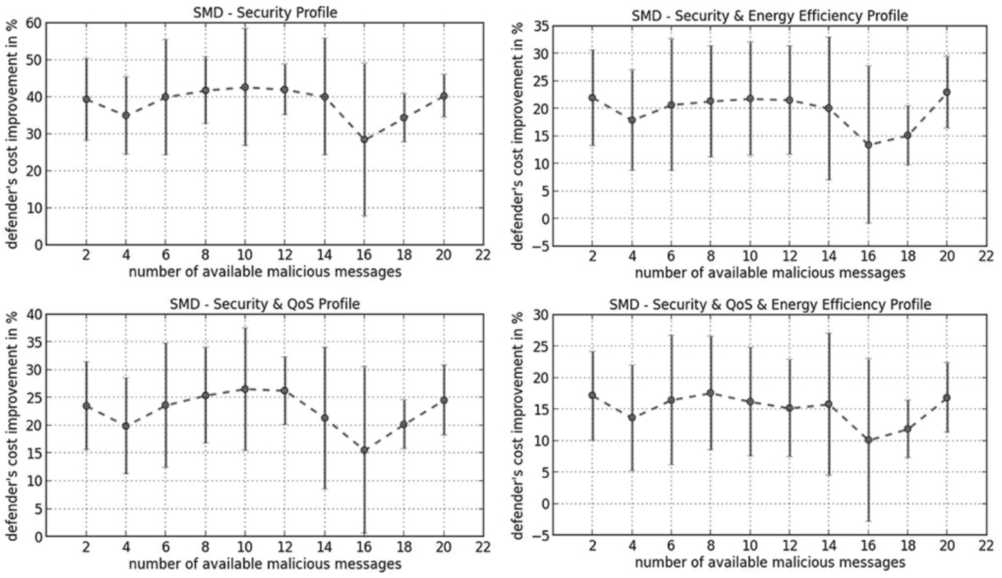


Figure 2.2: Simulation results in presence of a uniform attacker.

We notice the same trends for a Nash attacker as illustrated in Fig. 2.3. One difference in the results is that under the network profile `Security & QoS` the difference in improvement compared to the `Security & Energy Efficiency` is more pronounced as opposed to the scenarios with a Nash attacker. We also notice that for all network profiles SMD improves the defender's expected cost in a greater

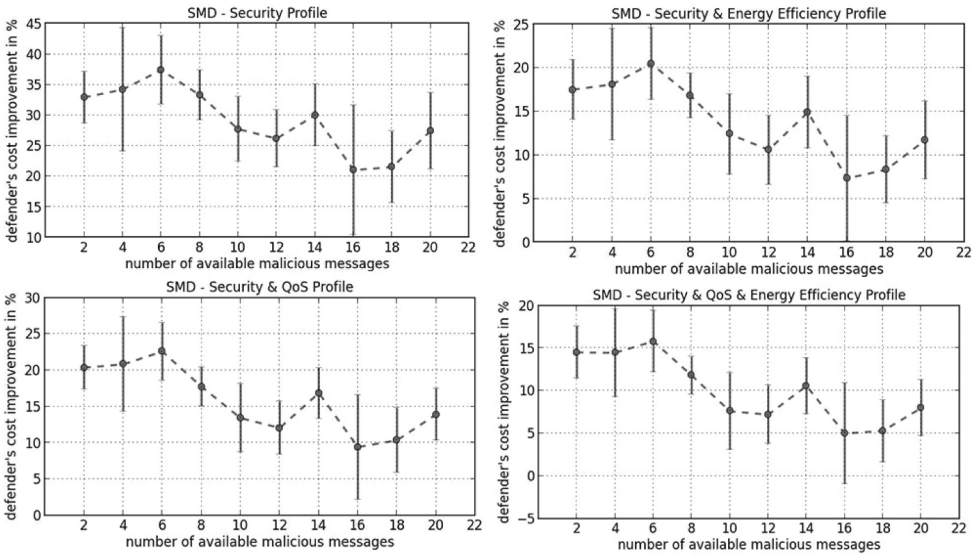


Figure 2.3: Simulation results in presence of a Nash attacker.

degree in the presence of a Uniform Attacker rather than a Nash attacker although the defender chooses the Nash routing plan in either cases (since it minimizes the maximum potential cost inflicted by the attacker). This is due to the attacker maximizing the minimum defender's expected cost at the NE as stated in Theorem 1. On the other hand, the uniform attacker follows a naive distribution to inject different messages into the D2D network and therefore achieving a worse performance than the Nash attacker.

As a generic comment, the more focused objectives SMD has the higher the improvement of the defender's expected cost is, compared to a shortest path protocol. We also notice that the standard deviation is large in all Experiments. This can be explained by looking at the results from the different Experiments in more detail. By doing so, we noticed that occasionally the same routes are chosen by both SMD and the shortest path routing protocol. This can be explained by the number of available routes being only 6 in our simulations here. The generic trends demonstrate the improvement that SMD introduces even without the use of a boosting algorithm. These preliminary results are promising and we have plans for further investigations when a boosting algorithm (e.g., Adaboost) is used and a larger number of devices and routes are given. In addition, we are planning to examine different mobility levels and see how these affect the expected defender's cost under different network profiles with SMD. [5]

## 2.5 Related Work

The papers we discuss in this section have used game theory in favor of security in mobile distributed networks. These address different challenges including

[5]Since we have not used a network simulator in this work and our Python simulations did not consider a specific mobility, mobility parameters (e.g., average speed) have not been investigated. The routes were shuffled every a while.

*secure routing* and *packet forwarding* [48–52], *trust establishment* [48, 53], *intrusion detection* [53–57], and *optimization of energy costs* [58–61].

In [48], Sun et al. presented an information theoretic framework to evaluate trustworthiness in ad hoc networks and to assist malicious detection and route selection. According to their mechanism, a source node chooses a route to send a message to a destination by looking up the packet-forwarding nodes' trustworthiness, and selecting the most trustworthy route. Yu et al. examined, in [49], the dynamic interactions between "good" nodes and adversaries in mobile ad hoc networks (MANETs) as secure routing and packet forwarding games. They have derived optimal defense strategies and studied the maximum potential damage, which incurs when attackers find a route with maximum number of hops and they inject malicious traffic into it. Extension of the previous work is presented in [51], where Yu and Liu examined the issues of cooperation stimulation by modeling the interactions among nodes as multi-stage secure routing and packet forwarding games. In [50], the same authors focused on a two-player packet forwarding game stating that nodes must not help their opponents more than their opponents has helped them back. Felegyhazi et al. have studied in [52] the Nash equilibria of packet forwarding strategies with TFT (Tit-For-Tat) punishment strategy in a repeated game.

In [58], the authors presented a Bayesian hybrid detection approach to preserve the energy spent for intrusion detection. In the proposed static game, the defender fixes the prior probabilities about the types of his opponent. The dynamic game allows the defender to update his belief about his opponent's type based on new observed actions and the game history. The authors formulated the attacker/defender game model in both static and dynamic Bayesian game contexts, and investigated the equilibrium strategies of the two players. Lui et al. in [59] put forwarded a more comprehensive game framework and they used cross-feature analysis on feature vectors constructed from the training data to determine the actions of a potential attacker in each stage game. They proposed to use the equilibrium monitoring strategies to operate between a lightweight IDS and a heavyweight IDS. In [61], Marchang et al. proposed a game-theoretic model of IDS for MANETs. They have used game theory to model the interactions between the IDS and the attacker to determine whether it is essential to always keep the IDS running without impacting its effectiveness in a negative manner.

In [56], Patcha et al. provided a mathematical framework to analyze intrusion detection in MANETs. They model the interaction between an attacker and an individual node as a two player non-cooperative signaling game. The sender could be a *regular* or a *malicious* node. A receiving node equipped with an intrusion detection system (IDS) detects a "message/attack" with a probability depending on his belief, and the IDS updates the beliefs according to this message. However, it is not explained how the IDS updates the beliefs according to this message. The same authors have also reinforced the suitability of using game theory for modeling intrusion detection by giving a theoretically consistent model in [57]. They used the concept of multi-stage dynamic non-cooperative game with incomplete information to model intrusion detection in a network that uses host-based IDSs. A cooperative approach is proposed in [54] by Otrok et al. to detect and analyze intrusions in MANETs. The authors used the Shapley value to analyze the contribution of each node to the network security and proposed pre-defined security classes to decrease false positives. They also considered cache poisoning and malicious flooding attacks. Santosh et al. in [55], employed game theoretic approaches to detect intrusions and identify anomaly behaviors of nodes in MANETs. The authors aimed at building an IDS based on a cooperative scheme to detect intrusions in MANETs using game theoretic concepts.

In [53], Cho et al. developed a mathematical model to analyze and reveal the optimal rate to perform intrusion detection related tasks. They enhanced the *system reliability of group communication systems* in MANETs given information regarding operational conditions, system failures, and attacker behaviors. They have also discussed to prolong the system lifetime and cope with inside attacks. They proposed that intrusion detection should be executed at an optimal rate to maximize the mean time to failure of the system.

Finally in [60], Panaousis and Politis present a routing protocol that respects the energy spent by intrusion detectors on each route and therefore prolonging network lifetime. However, this protocol does not investigate the effect of different malicious messages. It rather takes a simplistic approach according to which the attacker either attacks or not a route.

As we have seen in this section, a substantial amount of game theoretic models for security in distributed mobile networks (e.g., mobile ad hoc networks) have been proposed in the literature. However, none of them addresses all aspects of security, QoS and energy efficiency at the same time. Motivated by this observation, our work contributes to the existing literature by bringing together these three aspects, under a generic but also customizable model provided by the SMDGs. Furthermore, our work defines the adversary's pure strategies to be a set of different malicious messages. And this is not an aspect of investigation of papers identified by our literature review. It is worth noting that we consider the work undertaken, in this chapter here, as the first step towards a more complex and advanced game theoretic secure message delivery protocol for D2D networks.

## 2.6 Summary

In this chapter we have investigated secure message delivery for device-to-device networks in a hostile environment with possible malicious behavior. We have formulated *Secure Message Delivery Games* (SMDGs) to study the interactions between the *defender* (i.e., device-to-device network), and different adversaries, which are abstracted by the player called *attacker*. The defender seeks the "best route" to deliver a message from a source device to a destination device whilst the latter aims to harm the destination with mobile malware attached to a message. The defender solves an SMDG to derive the *Nash message delivery plan* (i.e., Nash mixed strategy). Then, the defender probabilistically chooses a route according to this plan and delivers the message to the destination. Due to the multi-hop nature of the network, intermediate devices relay the message towards the destination. Apart from forwarding, the relaying devices are responsible for the inspection of the message to identify malicious signs and therefore providing security for the D2D message communications.

We have proposed the *Secure Message Delivery* (SMD) routing protocol which takes routing decisions according to the *Nash message delivery plan*. Apart from security, the protocol respects energy costs and end-to-end delay with the ability to be customized to consider each objective at a different degree. We have undertaken simulations to show how much SMD improves the defender's expected utility compared to a shortest path routing protocol. We believe this improvement will be more pronounced when we implement boosting techniques for the computation of the final intrusion detection capabilities (i.e., confusion matrices) of the routes. We have also plans to take into account the remaining energy of each route in the utility function of the defender, and investigate the impact of mobility to the results. Lastly, future work will consider a network-wide extension of the per-message game where the attacker aims to spread a mobile malware while the defender is attempting to stop it.

# Chapter 3

# Android Code Obfuscation Techniques: Past, Present and Future Directions

Android mobile device OS is currently the market leader [62] [63]. The availability of Internet, Global Positioning System (GPS) and custom apps have increased the popularity of the mobile devices. The official Android market, Google Play is the dominant app distribution platform accessible to all Android devices [64]. Google Play also allows installation of third-party developers and app stores [65]. The elevated Android popularity has allured the malware authors already employing obfuscation and protection techniques [66]. The malware authors are propagating encrypted and obfuscated premium-rate SMS malware, evading Google Play security [67]. On the other hand, app developers are concerned about code misuse; hence they employ code obfuscation, encryption and custom protection techniques. The weak code protection techniques lower the code integrity and escalate the risk of plagiarism and malware attacks. For instance, a rooted device facilitates identification of app internals and evades device security.

In software engineering lexicon, reverse engineering is defined as a set of methods for obtaining `APK` source from the executable code. The code obfuscation is employed by malware authors to evade anti-malware. In particular, the architecture-neutral compiled Java code is amenable to reverse engineering. The app developers are concerned about safety and protection of the developed intellectual algorithms and data. Malware authors use obfuscation, code encryption, dynamic code loading, and native code execution evading the Google Play protection [68].

Code obfuscation is reported as a reasonable and easy alternative compared to the other protection techniques [69] [70]. Code obfuscation is a set of purposeful techniques to render the code unreadable. Code obfuscation transforms the code by changing its physical appearance, preserving the intended program logic and behavior. Furthermore, code obfuscation can also protect the software from being reverse engineered [71]. Hence, malware developers have already leveraged the obfuscation by developing recent malware apps evading the Play stores and commercial anti-malware [72,73]. Android permits app distribution from third-party developers and other third party app stores. Application developers employ obfuscators to protect the proprietary logic and sensitive algorithms to avoid the misuse. The code obfuscation techniques can be used to: (i) protect the intellectual property; (ii) prevent piracy; and (iii) prevent app misuse. The obfuscation techniques employed by malware authors evade the existing commercial anti-malware solutions. The Android development environment has an in-built obfuscator Proguard [74] for app code protection.

A De-obfuscator is very important if the app source code is misplaced, or unavailable. De-obfuscation can also be used to verify correct execution of obfuscated app. A popular app may be a `Trojan` with hidden malicious payload. Zhou et al. [75] studied 49 Android malware families, and reported more than 86% repackaged malware among 1260 `APK`. However, manual analysis is time consuming task against 2 million Google Play apps reported in June 2016 [76]. Several tools and techniques like code obfuscation, data encryption, encoding are available for source code protection.

Android is the dominant mobile platform among users and developers [62]. The open nature of Android allows apps from third-party developers and other third-party markets. The popularity is an opportunity for plagiarists to clone, obfuscate and hijack the popular apps with their Trojanized versions. Code obfuscation and protection techniques can protect the intellectual property. However, they are equally popular among malware developers.

The remainder of this chapter is organized as follows. In Section 3.1, we discuss basics of Android platform, execution mechanism, and its difference with Java development model. Section 3.2 covers types and purpose of code obfuscation with an in-depth description of app protection techniques. Section 3.3 elaborates obfuscation and optimization tools used by malware authors. In Section 3.4, we discuss the code protection techniques used by code packers. In Section 3.5, we explore the reverse engineering tools used for de-obfuscating the protected code, while Section 3.6 explores related work. Section 3.7 elaborates the recent trends in obfuscation and possible future research directions. Finally, in Section 3.9, we conclude this chapter.

## 3.1 Android Overview

Android OS is developed, maintained by Google, and propagated by the Open Handset Alliance (OHA) [77,78]. In the following, we discuss Android OS architecture and app compilation procedure.

### 3.1.1 Android Architecture

The Android software stack is formed of four different layers as illustrated in Figure 3.1: (i) linux kernel; (ii) native user-space; (iii) application framework; and (iv) application layer. The base of Android is Linux kernel adapted for limited processing capability, restricted memory and constrained battery availability. The Android platform customized "vanilla" kernel for resource constrained mobile devices. The Binder driver for inter-process communication (IPC), Android shared memory (ashmem), and wakelocks are important modifications to suit the Android devices.

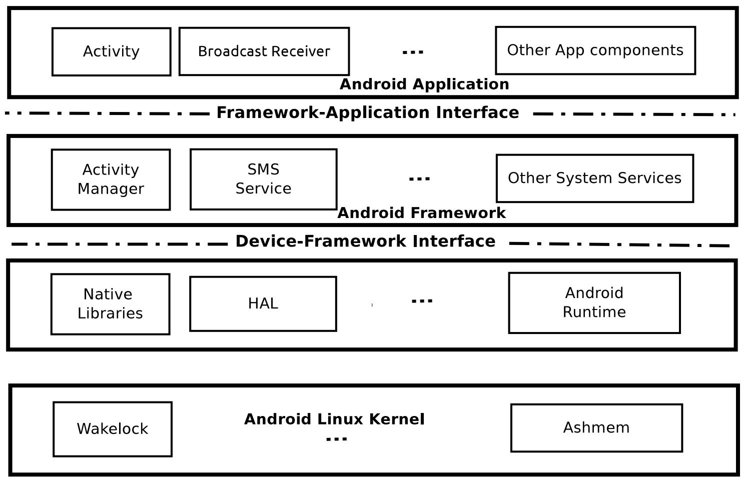


Figure 3.1: Android Architecture.

Native Space and Application Framework layers form the Android middleware. The bottom layer blocks are the components developed in C/C++. However, the top two layers are implemented in Java. Native components are directly executed on the processor, bypassing the DVM. Figure 3.1 illustrates Hardware Abstraction Layer (HAL), which facilitates low level device driver implementation [79]. Application Framework provides interaction to the developer. The source code is converted into a single Dalvik executable `.dex` which is interpreted by DVM. User apps are at the top most layer.

There are many known open-source and commercial tools to reverse engineer the Android apps. Thus, unprotected apps may unknowingly give away the source code to the attackers for misuse. Android runs the APK files on Dalvik Virtual Machine (DVM), a register-based virtual machine to suit the mobile devices [78]. The JVM is stack-based whereas, Dalvik is register-based VM [80]. JVM employs Last In First Out (LIFO) stack with `PUSH` and `POP` operations. The DVM stores register-based operands in the CPU registers and requires explicit addressing. Figure 3.2 illustrates the procedure of converting `Java` source code to an `APK` archive.

### 3.1.2 Android Compilation

`dx` is Android SDK tool that converts Java source code to Dalvik bytecode [81]. It merges multiple class files into a single `.dex` file. Android manifest stores name and version of the app, libraries, declared permissions, assets, and other uncompiled resources. The content is merged into a single archive, an Android application PacKage (APK) [78]. Many open-source and proprietary tools are available for reverse engineering the application. The unprotected apps may unknowingly give away their source code to the attacker, permitting visibility to the internals of the APK. The easy availability of source may lead to loss of revenue, reputation issues, access to intellectual property, and legal liabilities.

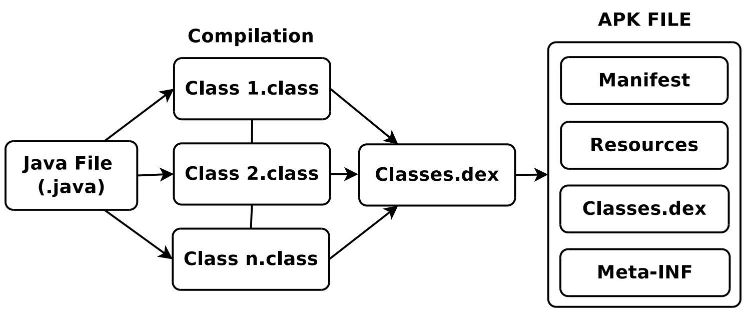


Figure 3.2: Compilation of Java code to Android `APK`.

The Android KitKat 4.4 introduced Android RunTime (ART) to replace the DVM. The new runtime is proposed to improve the Android OS performance. We briefly compare the Dalvik and ART to underline their importance.

- *Dalvik:* Dalvik VM is the core of Android Dalvik bytecode execution. The Dalvik runtime is based on the Just-In-Time (JIT) compilation that remains independent of the machine code. When a user runs an Android app, the `.dex` code is compiled to the machine code. Dalvik VM performs JIT compilation and optimization during the app runtime to improve performance. However, the presence of JIT adds latency and memory pressure. Though mobile devices are improving their resources, the new runtime is more efficient in comparison to Dalvik VM.

- *Android RunTime (ART):* The Android KitKat version 4.4 introduced an optional runtime Android RunTime (ART) to experimentally replace the DVM to improve performance. In ART, the `APK` the bytecode is converted to machine code at install time. Ahead-of-time compilation (AOT), a pre-compilation technique, saves the machine code in persistent storage. It loads the machine code at runtime, saving the CPU and memory as compared to DVM. In particular, the `.dex` file is compiled as `.oat` file in ELF format. The ART reads the `.dex` file using `.dexFile`, `openDexFileNative` from `libart.so` library. In case the `oat` file is not available, the runtime invokes `dex2oat` tool and compiles `.dex` to `.oat` [82]. Otherwise, ART loads the `.oat` file in memory cache map. Once the `.oat` file is loaded, ART creates `OatFile` data structure to store the information. The `ART` reduces app startup time as code is converted to native at install time, which improves battery life. However, the installation takes more time and space.

## 3.2 Code Obfuscation Techniques

Code obfuscation or mutation techniques alter the code appearance in the existing binary from one generation to another, to evade the anti-malware. Malware authors employ obfuscation techniques to protect malicious logic to evade the anti-malware [83]. The app developers also use obfuscation and code protection methods to protect the code against reverse engineering. Plagiarist and malware authors employ obfuscation to evade the security tools. Malware authors also employ obfuscation to plagiarize the popular and paid apps. Obfuscation provides significant protection and obscures explicit details. Some obfuscation techniques operate directly on the source code; some obfuscate the bytecode. Collberg et al. classify the obfuscation techniques as: (i) Control-flow; (ii) Data; (iii) Layout; and (iv) Preventive obfuscation [84, 85]. Figure 3.3 illustrates detailed outline of existing code obfuscation and protection techniques discussed subsequently. An obfuscator protects the proprietary software and prevents its reverse engineering. The obfuscated program maintains the semantics of the original app. The original and obfuscated version produce the same output when executed. The malware authors use code protection and obfuscation techniques to protect malware logic, algorithms and hide the suspicious information such as strings, domain names and server address to delay malware being detected.

A de-obfuscator restores the original app from the obfuscated code with reverse engineering tools. The reconstruction of the source may not be possible. For example, identifier renaming obfuscated variables cannot be restored to their original names once changed by the obfuscator. According to [86], code obfuscation or transformation is the most suitable technical protection for type-safe language like Java. According to Udupa et al. [87], surface obfuscation affects the syntax of the target program. However, it is not possible to hide the code semantics. If an identifier is renamed, the program output remains the same. The deep transformation obfuscates the control flow of target program [88].

***Definition:*** Collberg et al. define the obfuscating transformation as the conversion $\tau$ of a source app $A$ into the target app $A'$ [84] :

$$A \xrightarrow{\tau} A'.$$

Transformed app $A'$ is an obfuscated version of the original program $A$ if:

- App $A$ "does not terminate, or ends with an error condition, then $A'$ may or may not terminate" [84].
- Otherwise, $A'$ must terminate and generate the output similar to $A$ [89].

Android apps are distributed as archive APK files available from: (i) Google Play; (ii) Third-party app marketplaces; and (iii) Android debug bridge. Once installed, the apps from the devices can be also accessed with Android Developer Tools (ADT). These APK files can be reverse engineered to the Dalvik bytecode from `classes.dex`, the app executable file. In the following subsection, existing code obfuscation approaches are detailed according to the outline made in Figure 3.3.

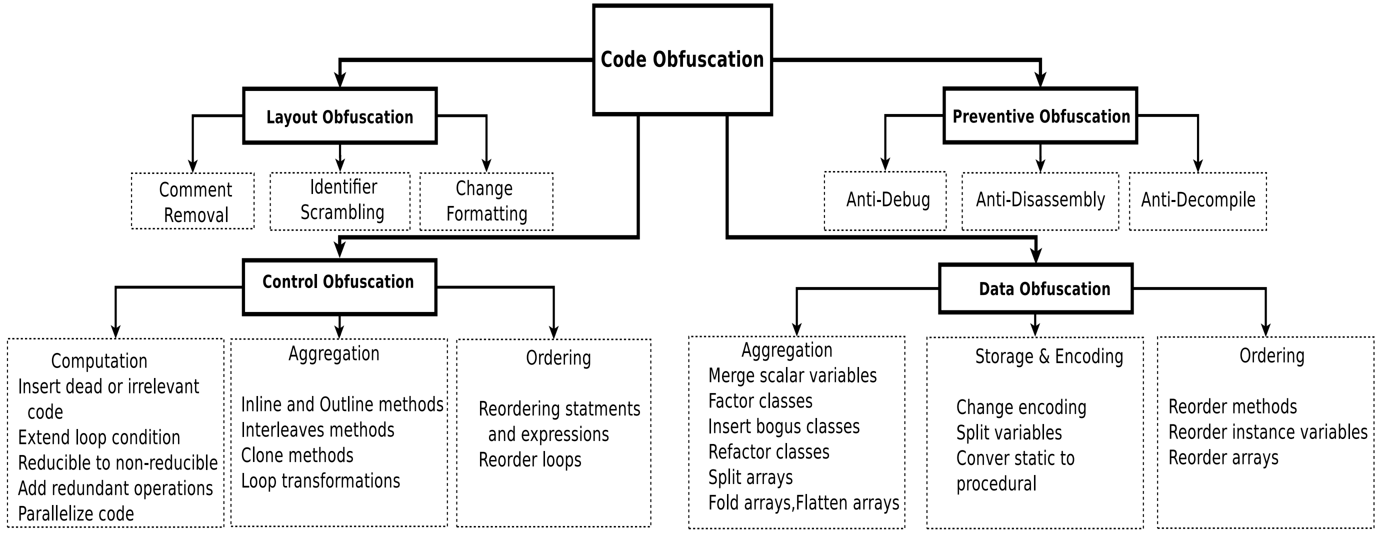


Figure 3.3: Obfuscation Classification.

### 3.2.1 Control Flow Obfuscation

The main idea behind Control flow obfuscation is to break the program flow control. The obfuscation mangles the functional blocks and breaks the recursive disassemblers. The functional blocks that do not belong together are intermingled to confuse the reverse engineering. The control flow transformation changes the execution paths of a program, while maintains intended functionality. The control flow obfuscation can be categorized as Aggregation, Computation and Ordering obfuscation techniques. The techniques can be further classified into following sub-categories as illustrated in Figure 3.3.

**Control flow computation:** The control flow obfuscation techniques hide the control flow and append additional code and complicate `APK` disassembly. The code insertion can be an additional method or irrelevant code [90]. However, the recent compilers remove unused code for execution efficiency during the code optimization phase. To counter this, one can insert irrelevant bytecode such as `PUSH` or `POP` within the high-level code. Hence, it is not removed. Computation control obfuscation can be broken down as described in the following:

1. *Inserting Dead or Irrelevant Code:* The dead code block can never be reached; hence, is never executed. Inserting dead-code statements increases size of code and analysis time. A programmer can insert code block that is never executed [91]. For example, one can include extra methods or irrelevant statement blocks [92]. For instance, the code snippet before the Add Dead-code Switch Statements (ADSS) [78] obfuscation is illustrated in listing 3.1. The Java bytecode switch construct can be used to insert control switch that is never executed [93]. However, the switch increases the connectedness and complexity of the method. Thus, the obfuscation evades the decompiler that cannot remove the dead switch. Listing 3.2 illustrates the ADSS obfuscation [93].

```java
//before ADSS obfuscation
if (writeImage != null) {
  try {

    File file = new File("out");
    ImageIO.write(writeImage, "png", file);
  }
  catch (Exception e) {
  System.exit(1);
  }
}
 System.exit(0);
```

Listing 3.1: before obfuscation, from [94, 95]

```java
// ADSS obfuscated code
if(obj != null) {
 try {
  ImageIO.write((RenderedImage)obj,png,
new File(out));
    }
  catch(Exception exception2)  {
   i += 2;
   System.exit(1);
    }
}
label_167:
{ while(lI1.booleanValue() == ___)
  { switch (i) {
    default: break;
    case 3: break label_167;
    System.exit(1);
    continue;
       }
    }
    System.exit(0);
}
```

Listing 3.2: ADSS obfuscated, from [94, 95]

2. *Extend Loop condition:* Rewriting the test condition as a complex loop function introduces obfuscation in the code. It can be accomplished by extending the loop condition with the addition of more test cases having no effect on the result. The code in Listing 3.4 illustrates the extension of a simple `if condition`.

```
// Before loop extension

int x = 1;

if (x > 200)
{
  ...
  x ++;
  Foo(x)
}
```

Listing 3.3: code before Loop extension

```
// After loop extension obfuscation

int x = 1;
while (x> 200 || x%200==0)
{
  ...
  x ++;

// calling function
Foo(x)

}
```

Listing 3.4: Loop extension obfuscation

3. *Reducible to Non-Reducible flow-graph obfuscation (RNR)* [89]: A reducible flow graph can be made complex by turning it reducible to non-reducible. The Java bytecode has a `goto` instruction. However, the Java language does not have a corresponding `goto` statement [96]. Hence, a plagiarist can misuse the `goto` bytecode and obfuscate with an arbitrary control-flow transformation. Java language can only express a structured control flow. Hence, "the control flow graphs produced by the Java programs is always reducible. However, Java bytecode can express this as non-reducible flow graphs, thus obfuscating reducible flow graph to non-reducible" [96]. According to [84], this is achieved by converting a structured loop into a loop with multiple headers. Listing 3.5 and 3.6 illustrate the effect of RNR obfuscation.

```
// Before:

Statement 1;
while (condition1)
{

  Statement2;
}
```

Listing 3.5: before RNR Obfuscation.

```
// After:
Statement 1;
if(condition2) {
  Statement2;
  while(condition1){
    Statement2;
  }
else {
  while(condition1){
    Statement2;
  }
}
```

Listing 3.6: RNR Obfuscated code.

4. *Add Redundant Operands:* Appending insignificant terms within the code during basic calculations hinders reverse engineering. For example, let us assume an integer variable $p$ which stores the product of two integer variables $a$ and $b$. The code listings 3.7 and 3.8 illustrates the code snippets before and after

redundant operators obfuscation. The transformed code generates exactly the same output. However, the obfuscated snippet appears complex during the analysis.

```
//original code

public int sum{

int a,b = 5, 7;
int p;
p = a * b;
System.out.println(``Product ='' + p);

}
```

Listing 3.7: before redundant operators

```
//Redundant Operators Obfuscation

public int sum{
int a,b = 5,7;
double i,j = 0.0005, 0.0007;
double p;
p = (a * b) + (i*j);
System.out.println(``Product =''+ (int) p);
}
```

Listing 3.8: redundant operators

5. *Parallelize Code:* The introduction of threads can affect the readability due to increased code complexity. The parallelization improves the performance. However, the motivation in this case is to hide the correct code flow. Collberg et al. [84] suggested the following techniques: (i) Creating dummy processes; or (ii) Splitting sequential sections of a program into multiple concurrent and parallel processes.

**Control-flow Aggregation (CFA):** The CFA alters the program statements grouping [97]. CFA can be further classified into:

1. *Inline and Outline methods:* In Java, replacing a method call by its actual body (inlining) make the code complicated and difficult to understand. Code optimizers use these techniques. It also is a useful obfuscation transformation. Code inlining removes procedural abstraction from the program. Conversely, outlining selects a group of statements in a procedure and re-use them to generate a sub-procedure. For instance, inlining two procedures `A` and `B` necessitates the calling one after the other and outlining a portion of the combined code inside a new procedure.

2. *Method Interleaving (MI):* Identifying an interleaved method is a difficult reverse engineering task. MI merges the body and parameter list of different methods. Furthermore, method interleaving inserts additional parameters to differentiate calls from individual methods [84, 89, 96].

3. *Method Cloning (MC):* A reverse engineer examines the body and signature of a subroutine to determine code functionality, a necessary step for reverse engineering. One can create function clones and further introduce repetitive calls to such functions.

4. *Loop Transformations (LT):* In [84], authors observed that some transformations increase the code complexity. Such obfuscation techniques break down the iteration space to fit the inner loop cache [89]. The availability of compile-time loop bounds turns the loop to a compound body with several loops known as loop fission [89].

**Control-flow ordering (CFO):** Control flow ordering obfuscation changes the execution order of the code statements. For instance, loops can be iterated backward rather than forward. Control ordering obfuscations can be categorized into:

1. *Reorder Statements and Expressions (RSE):* Changing the order of statements and expressions has a significant effect on the Dalvik bytecode. It can disrupt the link between Java source and corresponding Dalvik bytecode [91].
2. *Reorder Loop (RL):* RL transformation can be run backward to confuse the analysis. Code Listing 3.9 and 3.10 illustrate the usage of loop reversal obfuscation.

```
// Original code
x = 0;
while (x < maxNum){
  i[x] += j[x];
  x++;
}
```

Listing 3.9: before loop reversal

```
// Loop reversal to change the control flow
x = maxNum;
while (x > 0)
{
  x--;
  i[x] += j[x];
}
```

Listing 3.10: loop reversal obfuscation

**Control-Flow Flattening (CFF):** CFF transforms the source code such that static analysis cannot determine the targets of branches. In this technique, the basic blocks of a program have same predecessor and successor. During execution, the actual control flow is controlled by a *dispatch variable*. A switch block has dispatch variable and a jump table to switch indirectly towards intended successor. At runtime, the value of dispatch variable decides which code block to be executed next.

Figure 3.4(a) illustrates the control-flow graph of Listing 5.3 and Figure 3.4(b) shows the flattened control flow of the mentioned program. Compilers primarily use such methods during code optimization. However, malware authors leverage this technique to hide the semantic as well as syntax structure of a malware APK. This obfuscation technique prevents or at least delays manual analysis.

```
public int fill( int a , int b )
{
  int diff = 0;
  if ( a > b ) { diff = a-b ;
    do
    { b ++;
    } while ( a ! = b ) ;
  }
  else { diff = b-a ;
    do { a ++;
  } while ( a ! = b ) }
  return diff ;
}
```

Listing 3.11: control flow flattening

### 3.2.2 Data Obfuscation

Data obfuscation techniques can modify the structure of an APK. They can be classified into following sub-categories as illustrated in Figure 3.3.

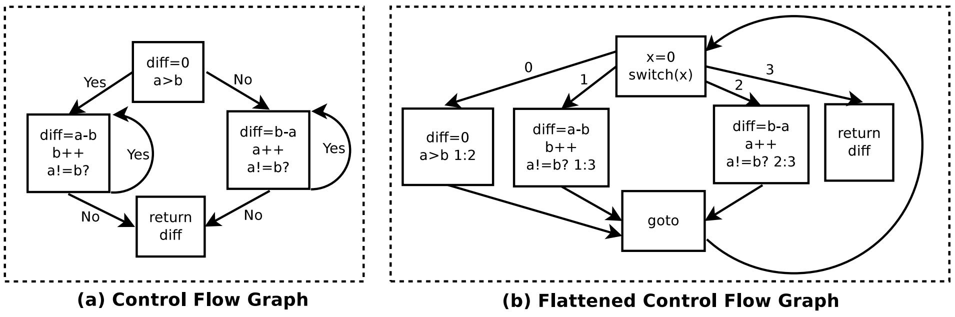


Figure 3.4: CFG of listing 5.3

**Data aggregation:** This obfuscation changes the data grouping. In the following, we list some sub-categories of this technique:

1. *Merging Scalar Variables:* Scalar variable merging technique combines multiple scalar variables in a single one. For example, $m$ scalar variables $Var_1$, $Var_2$, $Var_3$,.., $Var_k$ can be merged into a single array variable $Var_m$. Variables, in a way similar to arrays or integers, can be merged or even promoted as objects [84, 89, 96]. For example, we can merge two 32-bit integers `X`, `Y` in `Z`, a 64-bit variable.

2. *Class Transformations (CT):* Class transformations can be leveraged to make the program analysis difficult. One good way of achieving this is to use inheritance and interfaces to create deep class hierarchies to build a complex distributed application. Further, one or more dummy classes/methods can confuse the reverse engineer.

3. *Array Transformations (AT):* Array transformation is an effective obfuscation technique to convert the readable string information as unreadable [89]. The AT: (i) splits an array; (ii) merge two or more arrays; (iii) flattens an array dimensions; or (iv) folds or increases the array dimensions.

**Data Storage and Encoding (DSE):** The DSE affects how the data is stored and interpreted. Such methods obscure the data structures within the programs. Data storage obfuscation converts a local variable into a global. Furthermore, data encoding obfuscations replaces an integer variable `i` with an expression `x*i+y`. In the following, we discuss such techniques.

1. *Change Encoding (CE):* Programmers follow some standard conventions to write code. The encoding transformation techniques exploit this fact. The more transformations we employ, the less likelihood to understand the code. In fact, changing encoding reshapes the data into less natural forms. For example, we can replace all the references initializing an index variable `i`, with the expression `i= x*i+y`, where `x=6` and `y=5`. When the code needs to use the index value, the obfuscator inserts the expression `(i-5)/6`. Finally, instead of incrementing the variable by one, add six to the value. The obfuscation scales and offsets the index from the desired value to compute the

real index. Listings 3.12 and 3.13 illustrates encoding.

```
//Before:
int i = 1;
while (i <= 100)
{
  result = arr[i-1];
  i++;
}
```

Listing 3.12: before encoding obfuscation

```
//After:
int i = 11;
while (i <= 605)
{ result = arr[(i-5)/6];
  i+=6;
}
```

Listing 3.13: After encoding obfuscation

The data modification techniques can be used to evade the string based malware detectors. A simple example illustrated in Listing 3.14 and 3.15 demonstrates data modification obfuscation.

```
// original declaration
  int a = 30
```

Listing 3.14: before integer obfuscation

```
// obfuscating integer declaration
class bar {
public int getValue () {
return 30; }
}
Foo f = new bar () ;
int a = f.getValue () ;
```

Listing 3.15: after integer obfuscation

2. *Split Variables (SV):* Boolean variables can be replaced by a boolean expression. The relevant example is illustrated in Table 3.1. Variable $b$ in the original code is expressed as$f(b_1, b_2)$. In this example, function `F` is `XOR`. However, it can be generalized to any function with any number of variables. This also adds another layer of obfuscation due to the fact that some assignments of a value have different results. Let us say $b$ = True. We can further split the variable into $b_1 = 0$ and $b_2 = 1$ in the lookup Table 3.1 and convert it to original Boolean value. To split the variable $b$ into $b_1$ and $b_2$, we have to

| $b_1$ | $b_2$ | $b$=$F(b_1 \oplus b_2)$ |
|---|---|---|
| 0 | 0 | *Incorrect* |
| 0 | 1 | *Correct* |
| 1 | 0 | *Correct* |
| 1 | 1 | *Incorrect* |

Table 3.1: Lookup table to split variables [1]

define: (i) a function, $F(b_1, b_2)$ that maps $b_1$ and $b_2$ to variable $b$, (ii) to the inverse function, $F^{-1}(b)$ that maps $b$ to $b_1$ and $b_2$, and (iii) new operations defined on $b_1$ and $b_2$. $b$ has been split into two shorter integer variables $b_1$ and $b_2$. If $b_1$=$b_2$=0 or $b_1$=$b_2$=1 then $b$ is false, otherwise, $b$ is True. Boolean $b$ are masked as arithmetic operations on the integers $b_1$ and $b_2$ with the split variable technique [1].

3. *Convert Static to Procedural Data:* Strings store critical information such as copyright information, license key, and software expiry date. If the static string information is converted to procedural data, reverse engineering becomes complicated. A simple method is to convert the string to a program which computes the string [98].

**Data Ordering:** This obfuscation alters the data ordering. Ordering transformation randomizes data declaration order within a program. "An array stores a list of integer numbers. The array has the *i-th* element in the list at position *i*. A function `f(i)` can be used to determine the position of the *i-th* element in the list" [83, 86]. Randomizing the declaration order impedes reverse engineering process [86, 92].

1. *Reorder Methods:* randomize the declarations of methods within the code to harden the reverse engineering.
2. *Reorder Arrays:* randomize the order of parameters to methods and use a mapping function to reorder data within arrays.
3. *Reorder Instance Variables:* randomize the declarations of instance variables within the class.

```
// original code
int a = 7 + 70/2 + 1 ;
int b = "testing".length();
```

Listing 3.16: before aggregation

```
// obfuscated code
int a = 43;
int b = 7;
```

Listing 3.17: aggregation obfuscation

### 3.2.3 Layout Obfuscation

Layout transformation is used to modify the source and binary structure of a program. Different layout transformation techniques are illustrated in Figure 3.3. The Layout Obfuscation can be classified as: (i) identifier scrambling; (ii) output format changes; (iii) comments, or debug information.

Replacing the identifier names with mandarin non-alphabetical characters increases the code complexity. For example, in code Listing 3.18 the SmsManager Class encrypts the string before sending it over the network. However, the original identifiers names are amenable to reverse engineering. To avoid code inference, oBad Trojan employs identifier mangling obfuscation in Listing 3.19.

```
package iAmDriving{
  public class LetsNavigate{...}
}
package iAmConnecting {
  public class MyBluetoothHandle{..}
}
package userIsActive{
  public class MainActivity{...}
}
```

Listing 3.18: before obfuscation

```
package d {
  public class d { ... }
}
package e {
  public class e { ... }
}
package f {
  public class f { ...}
}
```

Listing 3.19: identifier renaming [88]

Listing 3.19 replaces the strings by a single character. Identifier mangling reduces the meta-information by replacing them with random alphabets. Random names do not carry any information about the object or the behavior. Hence, the interpretation becomes difficult. Proguard obfuscator within the Android SDK implements a similar approach. oBad Trojan leverages the identifier mangling with character permutations of “o, O”, “c, C”, “i, I” and “l, L” in lower and upper case [73]. Listing 3.20 illustrates the identifier renaming obfuscation.

```java
public final class CcoCIcIf {
private static final byte [ ] COcocOlo;
private static boolean CcoCIcI;
private static long OoCOocll;
private static long OoCOocll;
private static String cOIcOOo;
private static final OoCOocll lOIlloc;
private static ArrayList occcclc;
private static final occccl coclClII;
private static Thread ooCclcC;
}
```

Listing 3.20: oBad Trojan Identifier scrambling Obfuscation [88].

### 3.2.4 Preventive Transformations

Preventive transformation techniques are employed to evade the commercial debugging and de-compilation tools. Preventive obfuscation takes advantage weak or non-existent mapping between the high-level language and its corresponding bytecode to inject illegal, unused or rarely used bytecode. Changes in Java bytecode crashes the decompiler due to the presence of new instructions. Figure 3.3 illustrates the preventive obfuscation techniques.

**Anti Debugging:** This technique is a preventive obfuscation that inserts some validation code to identify the presence of a debugger. Once it identifies being executed in the debugger, the app behaves benign without revealing the malicious behavior.

**Anti De-compilation:** The transformation prevents reverse engineering of Dalvik or Java bytecode to high-level programming constructs. The obfuscator employs `goto` constructs valid with the bytecode which are not a part of Java language.

**Bytecode Encryption:** Bytecode Encryption changes the structure of data. Low et al. [86] suggest that encryption methods as an alternative to defeat decompilation. Code transformers encrypt the data with innovative methods to decrypt the encrypted content.

**Android Manifest Obfuscation:** Another technique to obfuscate data in Android application is Manifest Obfuscation. However, Android middleware verifies the manifest meta-data; some obfuscation techniques have been observed aiming at producing errors during the binary decoding [88]. In the process of `APK` file creation, the manifest is compiled into a binary XML file. DexGuard [99], the commercial extension of ProGuard [74] can obfuscate and optimize the binary XML files to avoid detection.

**String Encryption:** String Obfuscation hides the plain text strings that reveal sensitive information. The sensitive plaintext information can be misused. The strings must be available in plaintext form at runtime. Hence, the developer prefer to encrypt the plaintext string. String Obfuscation can be employed with an invertible encryption function employing *AES*, *DES* or *XOR* encryption. Resultant output replaces the original plaintext. The conversion creates byte sequence equivalent to a string as data array stored inside private class fields [91]. The strings are brought back to the original state at runtime. Hence, the strings are retrieved as plaintext when required. Dynamic analysis is useful as it extracts runtime information. Code listings 3.21 and 3.22 illustrates code snippet before and after string encryption. The encrypted code converts readable string information into an array of strings with `ASCII` characters and obscures the code readability.

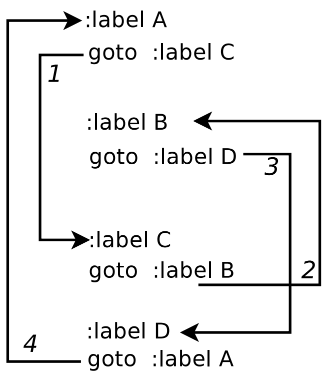
```java
//original code
public void onClick(DialogInterface arg1,
int arg2) {
 try { Class.forName("java.lang.System")
.getMethod("exit", Integer.TYPE)
.invoke(null, Integer.valueOf(0));
  return;
  } catch:(Throwable throwable) {
  throw throwable.getCause();
   }
}
```

Listing 3.21: before encryption [100]

```java
//String encryption
public void onClick(DialogInterface arg1,
int arg2)
 { try {
  Class.forName(COn.ËŁ(GCOn.ËŃ[0xA],
COn.ËŃ[0x09], GCOn.ËŃ[0xB]))
.getMethod(COn.ËŁ(i1, i2, i2 | 6),
Integer.TYPE)
  .invoke(null, Integer.valueOf(0));
  return;
     } catch:(Throwable throwable) {
  throw throwable.getCause();
       }
}
```

Listing 3.22: string encrypted [100]

**Bad Code Injection:** Figure 3.5 illustrates recursive `goto` sequences with an indirect recursion. This transformation thwarts the existing static analysis tools (disassemblers, decompilers) by inserting bytecode statements not available with high-level language. Some instructions may be valid in Java. However, their corresponding Dalvik instructions may not be available. For example, Java language does not have `goto` construct. However, when a loop or switch constructs are converted to Dalvik, the `goto` statement is generated. Hence, `goto` can be injected in the Dalvik bytecode. In particular, a dummy method with recursive `goto` statements can be used.


```
:label A
goto  :label C

:label B
goto  :label D

:label C
goto  :label B

:label D
goto  :label A
```

Figure 3.5: Bad code injection.

**Reflection:** Reflection is a powerful Java programming technique to extend additional functionality such as verifying backward compatibility or dynamically load methods. It is used in debugging and testing tools [101]. Java Reflection application programming interface (API) allows a program to access the class information during execution to [101, 102]: (1) create new objects; (2) invoke a method; or (3) modify the code control-flow. Reflection can be alternatively employed as data obfuscation technique. Code listings 3.23 and 3.24 illustrate the importance of reflection obfuscation.

Reflections is used in Android for a variety of purposes such as:

1. **Invoking hidden API methods:** Certain Android framework features are intentionally hidden from the developers during compilation. The hidden API may not support all Android devices, or the stable version is yet to be made public.

2. **Providing backward compatibility:** New Android versions incorporate additional features and get released in the incremental higher versions. A developer may use reflection to verify if a particular feature present in older version exists with the new version. If available, then only calls that feature/method.

3. **Interacting with JSON data:** Data from server is loaded in JavaScript Object Notation format (*JSON*) from the web. Hence, it is parsed using reflections in Android.

4. **Libraries:** Native libraries can be loaded in an APK using reflection API. Some applications employ custom native libraries for improved performance.

```java
// A Standard Call in Java
Crypto cryptModule = new Crypto();
privateKey = cryptModule.getPrivateKey();
```

Listing 3.23: before obfuscation

```java
// Invoke using reflection
Object reflectedClassInstance =
Class.forName (pe.mnit.secureApp.Crypto).
newInstance();

Method methodToReflect =
reflectedClassInstance.getClass().
getMethod(getPrivateKey);

Object invokeResult = methodToReflect.invoke
(reflectedClassInstance);
```

Listing 3.24: code reflection obfuscation

### 3.2.5 Repackaging Popular Apps

Repackaging Android APK is a popular practice among the malware authors. In particular, malware authors reverse engineer the popular apps. The app is reverse engineering, malware payload is inserted and released at less monitored Android third-party app markets. In [103], authors discuss a set of methods to replace the app developer library by the plagiarist ad-libraries to divert the advertisement revenues. Thus, a developer is robbed off the advertisement revenue.

Repackaging is the one of the widely used Obfuscation technique employed by malware authors on Android platform [104]. Zhou et al. [75, 105] reported 86%

repackaged malware among the 1260 Android Malware Genome dataset. Another study reported 5-13% repackaged and malicious applications among the well-known six third-party app stores [75]. The other third-party app stores do not have robust app verification and vetting. Hence, there is a higher chance of plagiarized and repackaged apps. Furthermore, authors in [104] reported around 30% plagiarized, and cloned apps even at the Google Play. The repackaging process typically employs following steps:

- Identify the most popular, free and paid apps at the Google Play, Download the `APK`.
- Employ disassembly tools like *apktool* [106] and reverse engineer the app.
- Insert malicious component as bytecode or Java source with help of `dxtool` [81], from Android IDE.
- Integrate malicious component in the `APK`.
- Re-organize `AndroidManifest.xml` and *resources* with additional component(s).
- Re-assemble the `APK` using *apktool*.
- Disperse repackaged malware `APK` at unofficial or regional app markets.

Repackaging is used by malware authors as a technique to evade commercial anti-malware. Repackaged apps create an imbalance in-app distribution markets, hurt the developer reputation, and inflict monetary loss to the developers [66, 107]. Malware authors also employ repacking to divert the advertisement revenues by replacing the original advertisements with their own. AndroRAT APK Binder [108] is a remote Trojan embedded inside popular with a motive to control the device remotely. Malware authors coerce affected device to send premium SMS or call premium rate numbers, record call, or acquire a device location. Furthermore the remote Trojan copies files without user consent.

### 3.2.6 Custom Obfuscation Techniques

Malware authors have been very active in developing customized obfuscation to defeat the anti-malware. In this section, we discuss following custom techniques primarily used by malware writers to protect the malicious code. Apvrille et al. [3] list some interesting techniques used by malware authors to defeat application analysis and reverse engineering:

**Using very long class names:** Decompilers tend to crash when the class names are too long or written in `non-ASCII` format [109]. Few malicious apps have demonstrated use of very long class names to defeat reverse engineering tools `Android/Mseg.A!tr.spy` [110] reported in the wild has successfully evaded commercial anti-malware with this technique.

**Hide Packages, JAR inside raw resources:** Malware developers hide the malicious executable package inside the resource files to avoid code inspection. For example, `Android/SmsZombie.A!tr` hides malicious package within a jpeg file `a33.jpg`, in the assets directory [2]. `Android/Gamex.A!tr` conceal an encrypted malicious package within asset `logos.png`, again an image file [2]. Table 3.2 enlists interesting malicious apps using the discussed techniques [3].

| **Android malware name** | **Purpose of Obfuscation** |
|---|---|
| `Gamex.A!tr` | The asset `log-os.png` is a ZIP. However, it has the capability of being a valid ZIP file (for instance, when XOR'ed with the right key). |
| `SmsZombie.A!tr` | Hides malicious package in `a33.jpg`. |
| `DroidCoupon.A!tr` | The malicious payload is stored in a png file in resources. The rage in cage exploit, escalates device privileges. |

Table 3.2: hidden malware payload inside APK resource.

**NOP to modify bytecode control flow:** No Operation (NOP) is an assembly language instruction, that does nothing at all. A sequence of NOP instructions wastes the CPU cycles and adds to the code complexity. Malware authors can leverage NOP instruction to modify the bytecode flow without revealing the program control-flow. NOP insertion modifies code syntax to evade anti-malware. This approach is easy and commonly used.

**Path Obfuscation:** Path obfuscation is used to achieve the cloning transformation [111]. The idea is to change the path such that there are different methods, but the same meaning. This technique is used within URLs to obfuscate the HTTP-based attacks [111].

**Hiding bytecode:** Hiding bytecode obfuscates APK with a variable length fill-array-data-payload instruction to hide the original bytecode [112]. This technique can be detected by looking for Dalvik bytecode employing `goto` obfuscation followed by fill-array-data opcode, illustrated in Figure 3.6. The bytecode is hidden in the fill-array-data which remains invisible to the disassemblers.

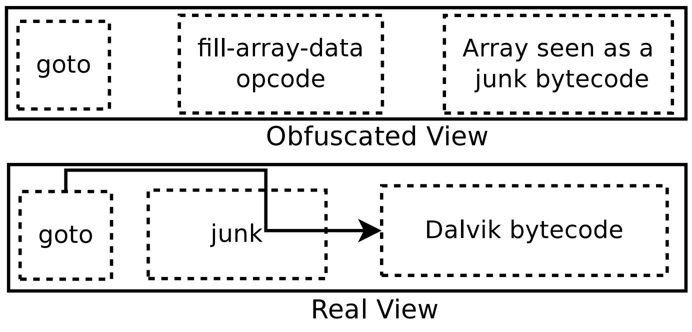


Figure 3.6: Hiding bytecode in the array of fill-array-data.

**String table:** A string table can be used to hide strings. In this technique, a malicious app builds a string table as an array of characters. The table hides suspicious strings. The reverse engineering tools fails to identify strings. Listing 3.25 illustrate use of string table as an obfuscation technique.

## 3.3 Custom Code Obfuscation Tools

Obfuscation techniques, though not invincible, are very popular among malware writers. Figure 3.7 illustrates the steps employed by obfuscation methods to transform and optimize the code.

```
package Eg9Vk5Jan;
class x18nAzukp {
   final private static char[][] OGqHAYqtswt8g;
   static x18nAzukp()
   { v0 = new char[][48];
     v1 = new char[49];
     v1 = {97, 0, 110, 0, 100, 0, 114, 0, 111, ...
     v0[0] = v1;}
protected static String rLGAEh9gGn73A(int p2) {
     return new String(Eg9Vk5Jan.x18nAzukp.
OGqHAYqtswt8g[p2]);
} ...
new StringBuilder(x18nAzukp.rLGAEgGn73A(43))
```

Listing 3.25: String Table

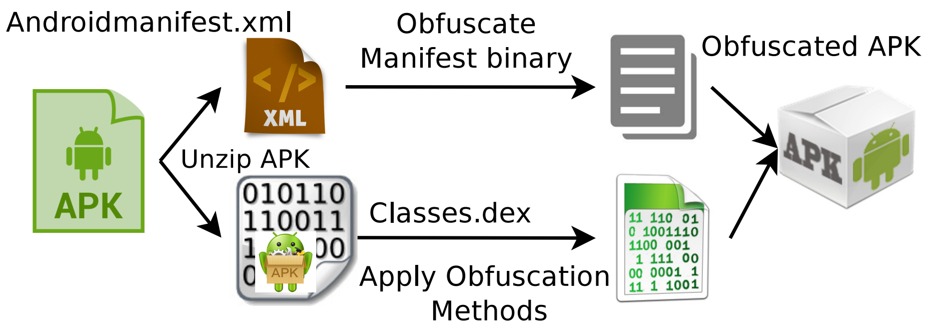


Figure 3.7: APK Obfuscation and optimization methodology.

### 3.3.1 Proguard

Proguard obfuscator is a part of Android software development kit (SDK) [74]. Proguard is a Java source code transformer. Google recommends Proguard to protect Android APK. Proguard has an in-built optimizer, shrinker and a weak obfuscator. The Obfuscator "tool removes unused or unnecessary code, merges the identical code blocks, employs peep hole optimization, removes debug information, renames objects and restructures the original code" [100].

```
//original code listing
public static String exec(String cmd,
Boolean root) {
        BufferReader mybufferReader;
        DataStream
testdataOutputStream;
        Process process;
        String string = "sh";
        if(root.booleanValue()) {
            string = "su";
    }
     StringBuild teststringBuilder =
new StringBuilder();
        try {
        process = Runtime.getRuntime()
.exec(string);
 dataOutputStream = new DataOutputStream
(process.getOutputStream());
dataOutputStream.writeBytes(cmd + "\n");
mybufferReader = new BufferedReader(
new InputStreamReader(process
.getInputStream()));
        }
```

Listing 3.26: before obfuscation [100]

```
// Proguarded output

public static String a(String arg6,
Boolean arg7) {
        Process process;
        String string = "mksh";
        if(arg7.booleanValue()) {
            string = "su";
        }

  testStringBuilder stringBuilder =
new StringBuilder();
        try {

            process = Runtime.getRuntime()
.exec(string);

       DataOutputStream dataOutputStream =
new DataOutputStream(process
.getOutputStream());
            dataOutputStream

.writeBytes(String.valueOf(arg6) + "\n");

       BufferedReader bufferedReader =
new BufferedReader(
  }
```

Listing 3.27: Proguarded code [100]

### 3.3.2 Allatori

Allatori [113] is a commercial product from Smardec. Besides identifier renaming, Allatori tool can also modify the source code. Allatori is a code optimizer, shrinker, obfuscator and a watermarking tool. The tool obscures the loops within the program such that reverse engineering tools are easily evaded. Such an approach increases the code size and makes the program logic less readable. Moreover, Allatori also encrypts the strings and decrypts them at runtime. Allatori has the following notable features: (i) reduced `.dex` file size; (ii) improves `APK` execution speed; (iii) decreases memory usage; (iv) removes debug code; and (v) employs simple obfuscation.

### 3.3.3 Dalvik-obfuscator

dalvik-obfuscator is an open-source bytecode transformation tool [114]. The analyst must provide an `APK` file as input to obtain the obfuscated app version. Dalvik-obfuscator employs the popular junk byte injection approach on the x86 platform. Dalvik-obfuscator is composed of a set of tools/scripts to obfuscate and manipulate `.dex` files. The obfuscator iterates through all the methods, insert junk bytes and unconditional branch in the code block, to ensure it is never executed.

### 3.3.4 DexGuard

DexGuard [99] is a professional code optimizer and obfuscator developed by Eric Lafortune. It performs code optimization, code shrinking, and encryption. Dexguard converts the class and methods names into `non-ASCII` values and strings are encrypted with the encryption algorithms. DexGuard has following features in addition to Proguard: (i) Reflection obfuscation at runtime; (ii) Encrypt strings

```
// original code

public void onClick(
DialogInterface arg3, int arg4)
{
  System.exit(0);

}
```

Listing 3.28: before obfuscation

```
// Dexguard obfuscated code
public void onClick(
DialogInterface arg7, int arg8) {
   try {

  Class.forName
("java.lang.System").getMethod("exit",Integer.
TYPE).invoke(null, Integer.valueOf(0));
  return;
   } catch:(Throwable throwable) {
  throw throwable.getCause();
      }
}
```

Listing 3.29: Dexguarded [100]

within an array; (iii) assets, resource and library encryption; (iv) encrypts Java class names; and (v) identifies APK tampering.

### 3.3.5 APKfuscator

APKfuscator is a dead code injection obfuscator [115]. Available as an open-source, APKfuscator employs quite a few variations of dead code injection. APKfuscator functions on bytecode level and leverages the Unix filesystem restriction that does not allow a class name to exceed 255 characters. APKfuscator employs three variations of dead code injection: (i) insert illegal opcodes; (ii) use legitimate opcodes into "bad" objects; and (iii) inject code inside the .dex header by exploiting a discrepancy between the claims of the official .dex documentation and DEX verifier.

## 3.4 Code Packers and Protectors

Android code Packers insert new malicious DEX file and encrypt the classes.dex in the existing .dex file within an APK. The new .dex is decrypted in memory during runtime using DexClassLoader, a Java class loader [3, 7, 116]. Packers were developed for Android platform to protect the legitimate app from unwanted tampering and modifications. However, malware developers use packers to obfuscate the dalvik bytecode and evade anti-malware scanners.

### 3.4.1 Code Packers

Packing encrypts the executable code to prevent static analysis. The unpacker routine brings the code in readable form. Malware developers employ executable code packers to evade reverse engineering of malicious DEX. The runtime unpacking routine brings the code into its original form. The code protectors can also be used in conjunction with existing obfuscation techniques to harden static analysis. Figure 3.8 illustrates working of code packers.

**APK Protect[1]:** is a professional code packing tool with anti-debug, anti-decompile and anti-disassembly support [117]. The packer performs code obfuscation with string encryption with Base64 encoding. It employs Java reflections to load

[1]https://sourceforge.net/projects/apkprotect/ (accessed August, 2016.)

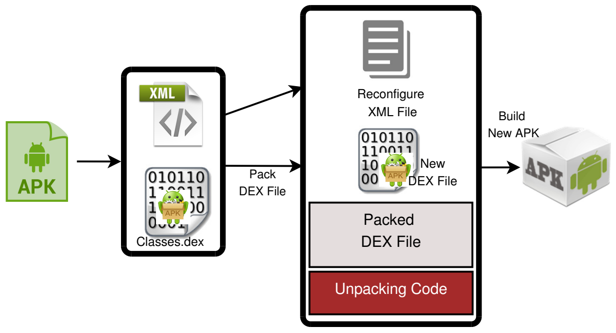


Figure 3.8: Code Packing steps.

the code dynamically. APK Protect has following features: (i) debugger detection; (ii) app encryption; (iii) code reflections; (iv) anti-debugging; (v) anti-disassembly; and (vi) anti-decompilation.

**HoseDex2Jar**[2]**:** is an executable packer to encrypt `.dex`, repackage encrypted file, and store inside 112 header bytes of the target `.dex`. Figure 3.9 illustrate code packing procedure with HoseDex2Jar. The code packer performs: (i) `.dex` repackaging; and (ii) code encryption.

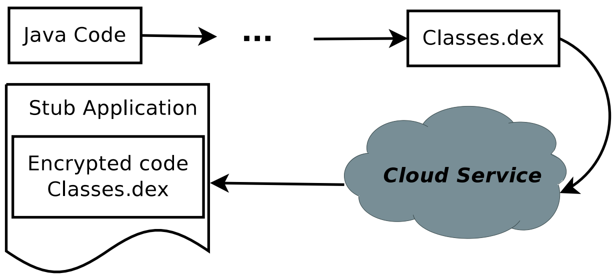


Figure 3.9: HoseDex2Jar Packer.

**Bangcle:**[3] is an online `APK` packing tool [7]. The developer must register at the Bangcle and use Bangcle Assistant tool to upload the package. The app must be uploaded with the Keystore to retrieve the protected `APK`. The packing process changes the `APK` name, inserts new assets, native libraries and modifies the Android manifest. Figure 3.10 illustrates the code packing procedure. Bangcle packer provides (i) online `APK` wrapping; (ii) resists reverse engineering; (iii) online anti-debugging, anti-tamper and anti-decompilation service.

**PANGXIE**[4]**:** is a Proof-of-Concept (PoC) packer armed with anti-debugging and anti-tampering techniques [100]. The packer encrypts the `.dex` bundled inside the assets of an `APK`. Though, PANGXIE evades static analysis, the obfuscator increases the APK size. Figure 3.11 illustrates the code unpacking procedure based on runtime execution.

[2] https://github.com/strazzere/dehoser (accessed August, 2016.)
[3] https://www.crunchbase.com/organization/bangbang-security/entity (accessed August, 2016.):
[4] http://www.packers.com/ (accessed August, 2016.)

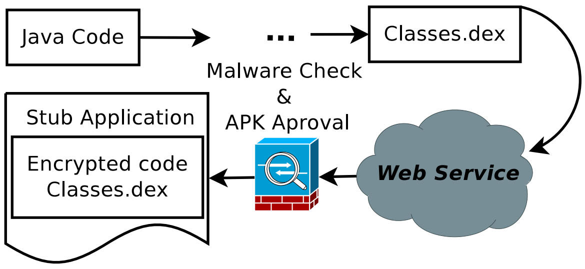


Figure 3.10: Functioning of Bangcle Packer.

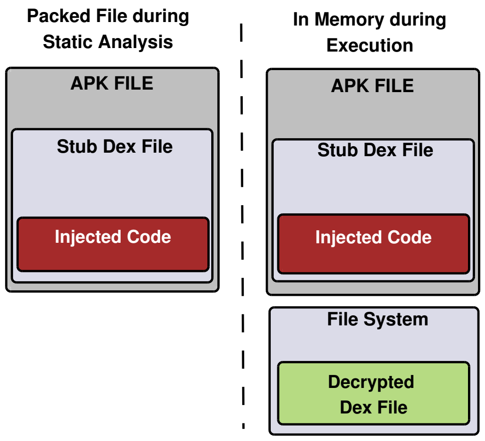


Figure 3.11: Unpacking procedure during Dynamic analysis.

### 3.4.2 Comparison of Obfuscation and Protection Techniques

In the following, we evaluate the effectiveness of Packers based on following attributes illustrated in Table 3.3.

**Code Obfuscation:** This technique prevents the analysis of the code either at source code or bytecode level [118]. The Android Integrated Development Environment (IDE) has Proguard, an in-built obfuscator to transform the Java class names, fields, and method names [74]. Additionally, the persistent methods like control-flow obfuscation, reorder program flow, readable string encryption, and dynamic code loading has been employed by recent malware [119,120]. Furthermore, the app developers also use Java Reflection methods and invoke the native code functionality using Java Native Interface (JNI) to hinder static analysis.

**Dynamic Code Modification:** Android user apps, developed in Java are converted to Dalvik bytecode using `Dx` tool [116], part of Android SDK. Dalvik Virtual Machine (DVM) verifies the bytecode, and executes it in the VM. It is difficult for a programmer to modify bytecode from VM during runtime. Malware developers use Java Native Interface (JNI) methods to modify and load the bytecode at runtime DVM [121]. ART pre-compiles the `.dex` file as `oat` in the ELF format. To avoid precompilation, the native code modifies the instructions of `.dex` and `.oat` data structures.

| | **Packer Protection Techniques** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Packer | Code Obfuscation | Dynamic Code Loading | Dynamic Code Modification | Debugger Detection | Append shared Libraries | Additional Class insertion | DVM Support | ART Support |
| APKProtect | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✗ |
| Ali | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✗ |
| Baidu | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| Bangle | ✔ | ✔ | ✗ | ✔ | ✔ | ✔ | ✔ | ✔ |
| Ijiami | ✔ | ✔ | ✗ | ✔ | ✔ | ✔ | ✔ | ✔ |
| HoseDex2jar | ✔ | ✔ | ✗ | ✔ | ✗ | ✗ | ✔ | ✗ |
| Pangxie | ✔ | ✗ | ✗ | ✗ | ✗ | ✔ | ✔ | ✗ |

Table 3.3: Comparing packer protection methods

**Dynamic Code Loading:** Android permits loading of external `.jar` or a `.dex` at runtime. The executable code appears different compared to its static visuals. Malware authors leverage this facility by encrypting the malicious executable, then decrypting and loading them in the VM at runtime.

**Anti-debugging:** Android kernel has in-built GNU debugger `(gdb)` to attach process for debugging. Packers get attached to `Ptrace` [122] tool to evade gdb under the assumption that only a single process can be attached to the target process for monitoring. Hence, the gdb cannot attach itself to the `APK`; preventing the `APK` debugging. Advanced Packers identify Java Debug Wire Protocol (JDWP) threads being attached to an `APK`. Furthermore, the Packers can identify themselves being monitored within emulated environment. Table 3.3 illustrates a comparison of the crucial methods prevalent among the leading Packers based upon their properties. The Android Packers are a new phenomenon and evolving quickly. We examined known malicious apps on portals and packed with stand alone Packers during October and November 2015. Table 3.3 shows that all of the Packers use one or more code obfuscation techniques and append shared library. The comparison shows that few Packers support ART. Except Bangcle and Ijiami, other packers are not equipped to perform runtime modifications.

## 3.5 Android Bytecode Analysis

In this section, we discuss various Dalvik bytecode de-obfuscation tools and techniques. To de-obfuscate an `APK`, the reverse engineered bytecode must be available. Hence, we discuss the Dalvik bytecode, as it is the nearest readable intermediate code for disassembly and analysis [123]. Figure 3.12 illustrates the process of bytecode extraction and analysis of a normal Android `APK` file. This process is followed by analysts and malware authors for their respective purpose.

### 3.5.1 Bytecode and De-Obfuscation Tools

In the following, we briefly discuss open-source and commercial de-obfuscation tools.

**Dexdump:** Dexdump is a part of the Android software development kit (SDK) [124]. Dexdump is a Dalvik executable (`dex`) file dissection tool that can be used to disassemble Dalvik bytecode. Dexdump is a linear sweep disassembler that finds a valid instruction at the last byte of the instruction being analyzed. Linear sweep disassembler can be circumvented easily by inserting the junk bytes to prevent the disassembly.

**Smali:** Smali is a Dalvik bytecode assembler [125]. The package contains baksmali to disassemble the assembled code. Hence, both smali and baksmali can be used to disassemble, modify and reassemble the Android apps [126]. Baksmali performs recursive disassembly by following the address of jump towards the current instruction.

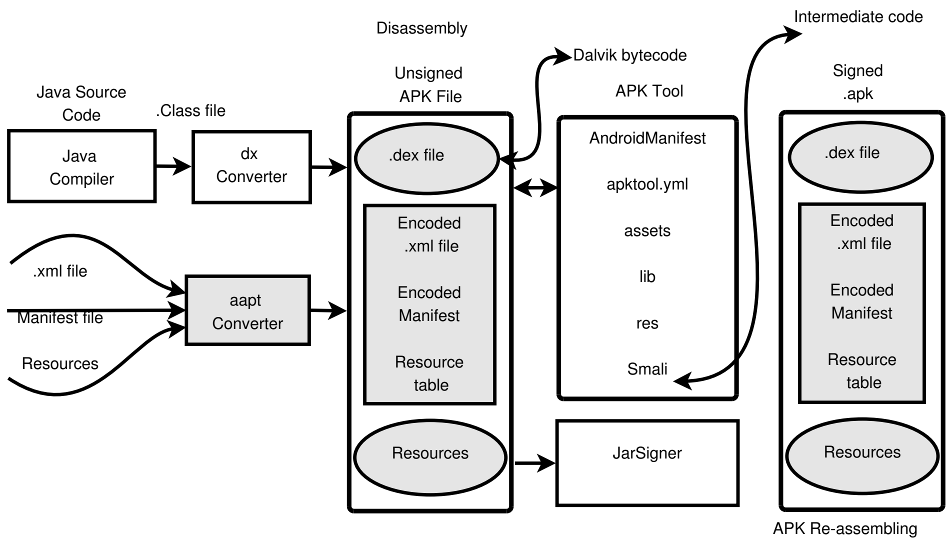


Figure 3.12: Disassembly of an `APK` file

**Androguard:** Androguard is a stable, open-source tool for static analysis of `APK` files [127]. Androguard has a recursive disassembler and semantic analysis methods to identify similarity, dissimilarity, call graph analysis and signature of malicious apps. It provides a graphical preview of call graphs to assist the human analyst to detect cloned and repackaged APK files.

**APKInspector:** APKInspector is static, bytecode analysis tool. APKInspector leverages Ded [128], Smali/Baksmali [125], APKtool [106] and Androguard [127] to reverse engineer APK bytecode. APKInspector performs: (i) meta-data analysis; (ii) sensitive permission usage; (iii) generates bytecode control-flow graph; (iii) generates call-graph with call-in and call-out structures.

**dex2jar:** dex2jar disassembler reverse engineers dalvik bytecode and creates corresponding `.jar` files. The tool is capable of handling `dex` and `odex` file types. dex2jar provides various API to extend the functionality [129]. dex2jar facilitates retargeting and conversion of `dex` to `jar` from the `APK` bytecode. The tool facilitates conversion back to`dex` bytecode, once java bytecode is modified.

**Dare:** Dare tool retargets the `.dex` bytecode from `classes.dex` to java `.class` employing type inference algorithm [130]. The java bytecode can be modified with existing techniques used by Java developers and decompilation tools. Octeau et al. [131] evaluated Dare tool and reported its more accurate than dex2jar.

**Dedexer:** Dedexer [132] converts `.dex` to Jasmin, an intermediate code similar to Java bytecode. Dedexer facilitates conversion of single file for each class; hence generates easy to read directory format [101, 102]. However, unlike the apktool, Dedexer cannot regenerate disassembled class files.

**JEB:** JEB [133] is a professional `APK` reverse-engineering tool. The decompiler creates a graphical visualization of the target `APK` file. The app elements such as components, androidmanifest, images, developer certificate, plain string are visualized from the GUI with good cross references. The professional version is capable of converting complex `.dex` bytecode to source by exploiting the rich dalvik semantics. JEB can also deobfuscate simple code transformation methods [106, 129]. Additional functionality is possible due to availability of API. Analysts can extend the decompiler for custom features.

**IDA Pro:** IDA Pro [134] is popular `x86` disassembler. IDA 6.0 supports Dalvik disassembly. IDA has a complete GUI with options to extend the functionality with supported API plugins to extend analysis functionality. IDA Pro has an additional capability to disassemble specific parts of the code in a file selected by the user. Dalvik bytecode can be represented as a graph making it easy to follow the control flow within a program.

**Dexter:** Dexter is a free online analysis service [135] and static analysis tool to process an input `APK` file. It provides rich information about: (i) `APK` permission; (ii) obfuscated code and packages; and (iv) mapping between broadcast receivers and the data-store.

**Dexguard:** Dexguard is a set of scripts to perform automated strings deobfuscation and recovery of the `dex` file [136]. This tool is a mix of static and dynamic analysis consisting of (i) `dex` File reader; (ii) Dalvik disassembler; (iii) Dalvik emulator; and (iv) `dex` File parser.

**Radare2:** Radare2 [137] is an interactive bytecode disassembly and analysis tool with precise control during reverse engineering procedure. Radare2 decompiles the code using open-source decompiler boomerang [138]. This tool is a recursive disassembler that allows a user to specify starting address for decompilation. The hybrid approach makes Radare2 more efficient against smart obfuscation techniques in comparison to other approaches.

### 3.5.2 Stealth Obfuscation

Malware writers employ different code protection techniques to delay the malware reverse engineering. Android malware evolution is following the trends similar to Windows, reported in various studies. The Packers can be employed to protect the existing code appear genuine and evade blacklisting signature approach. In the following, we discuss popular techniques employed to hide malware code. Bremere et al. [139] demonstrated the possibility of bytecode injection within a class with virtual function. Such techniques can be used by attackers to:

1. develop a benign app capable of bytecode interpretation and loading.

2. read the bytecode from `APK` or a remote host.

3. inject the malicious bytecode inside a benign app.

At present, such techniques are limited to returning integer values. However, malware authors can misuse the extended version to inject malicious bytecode at runtime. In [4], authors demonstrated a Proof-of-Concept code to hide Dalvik methods from existing reverse engineering tools. Furthermore, they demonstrated methods to identify such hidden methods. Malware authors can place malicious logic inside hidden methods and evade anti-malware. Furthermore, authors developed `Hidex`, a tool [140] to detect invisible methods.

In [2], the authors demonstrated various methods to manipulate AES or DES algorithm output and represented the malware payload as custom PNG, JPG or an audio file. Furthermore, the authors demonstrated that, such payloads remain undetected from existing anti-malware. Cyber criminals may be interested in hiding malware `APK` inside the assets or resource folders. Besides, malware author can develop a genuine `APK`, and include malicious JPG image inside app assets. The malicious app loads the asset at runtime to execute malicious behavior. Following protection, obfuscation, and optimization techniques can be interesting to malware researchers:

1. employ Proguard to protect app logic: Proguard renames variables, without disturbing the class names. Hence, it is easy to identify the obfuscated classes, since the methods are modified without changing class names.

2. Strings encoded with Base64: malware authors exploit various forms of string transformations such as string encryption within arrays, non-ASCII character replacement or, hidden resource files inside Base64 encrypted strings with Base64 encoding methods. Thus, binary data can be hidden with Base64 encoded strings.

3. Dynamic loading: Dynamic code loading can be used to download malicious code at runtime. Malware authors upload genuine `APK` at market place. The malware code is dynamically loaded at runtime as discussed in Table 3.4. The anti-malware have the same privileges like any other app inside Android Sandbox. Hence, dynamic code loading is extensively used for malware loading.

4. Native Code: identify native code presence with class definition filtering. Additionally, identify the app code which access system related information and resources interacting with Android framework.

5. Reflection: identify reflection methods, fields and class names to load them at runtime.

6. Header size: bytecode injection inside the `classes.dex` header can be exploited by taking advantage of discrepancy between the dalvik bytecode documentation and the file.

7. Encoding: it verifies the presence of mixed endianness with a flag check.

8. Cryptographic code: `javax.crypto` and `java.security.spec` packages provide facilities to implement encryption/decryption in classes and interface application to study misuse of cryptographic functionality.

Malware authors have misused the existing code protection and obfuscation methods to evade the commercial anti-malware. Table 3.4 summarizes malware obfuscation chronology and illustrates the methods and tools used by to obfuscate `APK` files. Malware writers leveraged custom encryption, string encryption, and URL encryption. They encrypted network communications and encoded URLs. The recent malware apps exploit sophisticated techniques like steganography and code obfuscation tools to harden the malware reverse engineering.

In 2013, malware developer used Dexguard to evade the anti-malware. The malware writers pushed Javascript payloads inside the resource folder and encrypted non-Dalvik code in the obfuscated apps. Some malware samples developed in the year 2014 employed online Packing tool services apart from string encryption and obfuscation. Additionally, malware writer encrypted `data.xml` files inside APK archive to evade anti-malware and harden reverse engineering. Dendroid [72] is a stealth remote administration toolkit employing hidden behavior sending premium-rate SMS, voice calls, recording audio video without user consent. The Trojan evades the existing commercial anti-malware techniques. Elish et al. [141] proposed user-intention based anomaly detector to detect such stealth malicious apps.

## 3.6 Existing Surveys and Related Work

Shabtai et al. [142] proposed an Android threat taxonomy on Android platform. In [143], the authors survey different attack vectors, and discuss the attack taxonomy on Android. In [75], authors, carried out Android malware characterization of 49 Android malware families. The authors reported simple obfuscation techniques employed by malware such as Anserver, and Bgserv. Enk et al. [144] discussed the existing research, primarily targeting Android platform.

The authors reviewed the Android platform security and app analysis methods. Furthermore, the authors discussed limitations of analysis techniques like rule-based detection, ex-filtration of sensitive information and inter-application privilege escalation attacks. However, the code obfuscation and protection techniques are not covered in the proposed analysis techniques. The proposed review gives an extensive insight into the obfuscation techniques and code protection methods. Furthermore, the review compares various code obfuscation and de-obfuscation techniques employed by malware authors and anti-malware industry.

Tangil et al. [145] discuss evolving mobile malware, their infection and distribution techniques and related case studies. The authors perform a comprehensive study of greyware and malware app detects between 2010 and 2013. Furthermore, the authors discuss various research problems, briefly discussing the impact of malware detection. In [66], authors discuss the Android security issues, malware penetration and various defense methods for app analysis and mobile platform security. Furthermore, the authors briefly review obfuscation techniques. However, they concentrate more on malicious repackaging, a common problem with Android apps.

In [91], authors focus on developing efficient Dalvik bytecode obfuscation techniques. They study the Google Play apps to identify the feasibility of availing the source code with reverse engineering tools. Authors propose efficient obfuscator design to defeat the existing de-obfuscation tools (i.e., smali, dedexer, ded). Schulz

| Malware | Year | Obfuscation/Encryption/ Protection/Optimization Method | Tool/Technique used |
|---|---|---|---|
| SlemBunk | 2015 | Code Obfuscation | Class, Method and Field Obfuscation |
| Trojan.Dropper.RealShell | 2015 | Custom APK Obfuscation | Stores files in Assets Folder |
| Dendroid.A!tr | 2014 | Obfuscated with Dexguard | Dexguard Tool |
| SmsSend.ND | 2014 | Packed with APKProtect Packer | Code Packing Tool |
| Freejar.B | 2014 | Packed with Bangcle Packing service | Online code packing Service |
| RuSMS.AO | 2014 | Strings obfuscated, using Adobe Airpush like name | Custom string encryption |
| SmsSpy.HW!tr.spy | 2014 | "data.xml" file is encrypted with Blowfish algorithm | Custom symmetric encryption |
| Agent.BH!tr.spy | 2014 | Sends encrypted emails using TL security | Custom encryption |
| Rmspy.A!tr | 2013 | Obfuscated with Dexguard | Dexguard obfuscator |
| oBad | 2013 | Obfuscated with Dexguard | Dexguard obfuscator |
| Android.Ginmaster | 2013 | Custom String encryption | Custom encryption |
| Android/GinMaster.L | 2013 | String obfuscation with string table in array | Custom encryption |
| Stels.A!tr | 2013 | Custom encoding URL text with Base64 | Custom encoding |
| Pincer.A!tr.spy | 2013 | Caeser cipher to hide text and telephone No. | Symmetric encryption algorithm |
| GinMaster.B | 2013 | Encrypts IMEI, IMSI and strings with Triple DES | Custom symmetric encryption |
| FakeDefend.A!tr | 2013 | Encrypted fake infections with AES algorithm | Block cipher encryption |
| FakePlay.B!tr | 2013 | Non Dalvik, Javascript payload in resources | Non Dalvik code encryption |
| SmsSend.N | 2012 | Obfuscated with Proguard | Proguard obfuscator |
| Plankton.B!tr | 2012 | Obfuscated with Proguard | Proguard obfuscator |
| DroidKungFu.D!tr | 2012 | Obfuscated with Proguard | Proguard obfuscator |
| FakeInst.A!tr.dial | 2012 | PNG file stores SMS text body and phone numbers | Steganography |
| NotCompatible.Android!tr.bdr | 2012 | Encrypted C&C URL in resource folder with AES | Block Cipher encryption |
| DroidKungFu.G!tr | 2012 | ELF payload stored as "mylogo.jpg " | Non Dalvik code encryption |
| Pjapps.A!tr | 2011 | Encoded URL | URL encryption |
| Android/SmsSpy.HW!tr | 2011 | Encrypted with symmetric key Blowfish algorithm | Symmetric encryption |
| Android/RootSmart | 2011 | Symmetric key encryption DES, AES and Blowfish | Block cipher encryption |
| BaseBridge.A!tr | 2011 | String encrypted in an array | Encrypted strings |
| Android/Geinimi | 2010 | Encrypted with Data Encryption Standard | Symmetric encryption |

Table 3.4: Malware Obfuscation chronology [2–7]

et al. [88] performed Android bytecode de-obfuscation feasibility. The authors evaluated analysis methods to automate de-obfuscation of Dexguard obfuscated code. Rastogi et al. [119] evaluated commercial anti-malware against trivial code obfuscation techniques. Faruki et al. [123] compare the performance of anti-malware, and static analysis tools against popular x86 transformation attacks. Harrison et al. [146] evaluated different Android obfuscation tools. They evaluate the limitations of reverse engineering tools against app repackaging.

Schrittwieser et al. [147] evaluate smartphone code protection techniques. The authors analyze and evaluate software de-obfuscation techniques. The survey is more general targeting software protection techniques and analysis methods. However, our target is, Android specific obfuscation techniques. The proposed review is a comprehensive discussion on source code obfuscation, code protection, Android specific obfuscation, and code protection tools. To the best of our knowledge, we are the first to investigate Android code protection and malware obfuscation techniques. We evaluate Collberg taxonomy [118], and expand source code, and bytecode obfuscation tools and techniques.

## 3.7 Future Research Directions

Malware such as *Android/DroidCoupon.A!r*, and *AndroidSmsZombie.A!.tr* hide the malicious native payloads as JPG, or PNG files [148], [3]. However, the assets are payloads just named as graphic files. Making fake payload with such tricks is prevalent on Android malware applications. Furthermore, authors in [2] developed a Proof-of-Concept (PoC) AngeCryption [2] to illustrate the possibility of encrypting any given input stored as an image (JPG, PNG). In particular, an attacker can develop a benign APK file to hide a malicious image inside resource or asset to evade the anti-malware. The unsuspected image containing malicious payload can be used to execute the malicious code. Such an attack may not be noticed at all, as the APK does not contain obfuscated, protected or wrapped content.

AngeCryption has demonstrated a PoC on the latest Android OS version. Thus, a malicious `dex` file can be embedded inside an image. Furthermore, it can be obfuscated with obfuscation tool. The dynamic code loading techniques can be used to execute the malicious payload. At present, methods to detect such attacks are not available. The functioning of such payloads cannot be determined before runtime image decryption. The suggested remedy that we aim to investigate in the future are: (i) keep tab on an APK that decrypts its resource or assets (such apps can be analyzed dynamically to identify suspicious behavior); (ii) analyze image decryption to an APK as malicious.

The Android devices have constrained processing and limited storage. Obfuscation techniques does have an adverse impact on battery consumption. The power management is an important issue to identify impact of code level modifications. The Android Obfuscation has a APK statistical significance [149]. An important future work is to consider a large set of obfuscated APK empirical evaluation. The same can be extended to different mobile OS and devices. Since the developers do not have access to tools like CARAT [150], they cannot identify the impact on energy consumption. The ability to identify the impact is important for resource constrained Android devices.

The Android apps have a lot of user interaction and string usage. The malware authors use string encryption and obfuscation techniques to hide the plaintext strings. In this chapter, we have discussed notable malicious apps using such encryption techniques. The authors can implement inter-component communication based inter-procedural static analysis to reconstruct the encrypted strings to obtain insight into the string information. Furthermore, authors in [151] empirically evaluated third-party library and obfuscated code usage. To monitor the apps, we propose to identify the third-party libraries to identify APK cloning. The common use of Google advertisement network, Facebook ad libraries impacts the categorization. As a part of future work, we plan to delink the library code from APK files and evaluate obfuscated code.

The existing academic code obfuscation research is heavily concentrated more towards analysis of obfuscated malicious applications [152] [153] [154] [155]. The relevant literature evaluates obfuscation techniques prominently among malicious applications. The real identification of obfuscated code among the normal programs which is important for software protection, is ignored. The non-malicious code reverse engineering is largely unexplored. Targeting program obfuscation and related techniques for protecting the digital rights is an interesting future direction. Inspite of the existing research on obfuscation, evaluation matrices to verify the existing obfuscation technique resilience are not available. Formal analysis techniques to evaluate obfuscation and de-obfuscation techniques is still not available. Hence, we summarize code obfuscation, de-obfuscation tools and techniques to understand the effect in isolation. It would be interesting to combine different class of obfuscation techniques, and evaluate existing de-obfuscation tools.

## 3.8 Obfuscation Code Examples

In the following, we discuss `FakeInstaller`, stealth Android malware employing different class of obfuscation to evade anti-malware. Listing 3.30 reverse engineered code of `FakeInstaller` [133,156,157] at line number 1 checks for the presence of emulator, an alibi for development environment, or automated analysis system. In line number 9, class and method names are obfuscated to erase the program semantics. For example, a random string value “VQIf3AInVTTnSaQI+R]KR9aR9”, is decrypted to Android `android.telephony.SmsManager` class. This class is loaded using reflection API at runtime to evade static analysis. The class sends premium-rate SMS without explicit user consent. The string “BaRIta*9caBBV]a” is decrypted to `SendTextMessage` method. Furthermore, in line number 21 `getMethod` sends the SMS using the text from the parameters p1 and p2, declared in line number 1. Here, the use of multiple obfuscation, evasion and code protection techniques successfully evades the static analysis.

```
 1  // Fakeinstall obfuscated code
 2  public static boolean gdadfjrj(String p1,String p2){ [...]
 3  // Anti analysis check to evade emulator
 4  if (zhfdghfdgd()) return;
 5
 6  // Get class instance
 7  Class clz = Class.
 8
 9  forName(gdadfjrj.gdafbj("VQIf3AInVTTnSaQI+R]KR9aR9"));
10  Object localObject = clz.getMethod(gdadfjrj.gdadfjrj("]a9maFVM.9"), newClass[0])
11  .invoke(null, new Object[0]);
12  // Get the method name
13  String s = gdadfjrj.gdadfjrj("BaRIta*9caBBV]a");
14
15  // Build parameter list
16  Class c = Class.forName(gdadfjrj.gdadfjrj("VQIf3AInVTTnSaQI+R]KR9aR9"));
17  Class[] arr = new Class[]
18  {nglpsq.cbhgc, nglpsq.cbhgc, nglpsq.cbhgc, c, c };
19
20  // Reflection for invoking the method to send SMS
21  clz.getMethod(s, arr).invoke(localObject, newObject[] { p1, null, p2, null,null });
```

Listing 3.30: Fakeinstall Obfuscation [156, 157]

The partial code snippet in Listing 3.31 is a variant of zitmo [158]. As illustrated, when the SMS is received, framework callback `onReceive()` is invoked to stop the broadcast to default SMS app. The `abortBroadcast()` method aborts the current broadcast. Then, an intent that carries the SMS is launched inside the MainService, a background service task. Further, the stored SMS from the intent is accumulated in the array called “pdus”. The sender identification and message parts are extracted by `getOriginatingAddress()` and `getMessageBody()` methods. Furthermore, the available values of the “pdus” object are stored inside variables s1, s2 along with the device Id using method `getDeviceId()`. The information is encoded inside the `UrlEncodedFormEntity` object. Further, the constant URL string is then encoded with `setEntity()` to post the data using `execute()` method.

```
public class mysmsReceiver extends BroadcastReceiver
{
 public void onReceive (context pcontext, intent pintent)
 {
    bundle localBundle=pintent.getExtras();
       if((localBundle != null)&&(localBundle.containsKey("pdus")))
  {
    abortBroadcast();
    Intent.targetService=newIntent(pcontex, MainService.class);
        targetService.putExtra("pdus",localBundle);
        pcontex.startService(ts);
  }
 }
}

public class MainService extends Service
{
   public int onStartCommand(Intent pintent, int pintent1, int pintent2){
   Bundle localBundle=pintent.getBundle("pdus");

   SmsMessage localSMS=SmsMessage.createFromPdu("pdus");
   String S1 = localSMS.getOriginatingAddress();
   String S2 = localSMS.getMessageBody();
   String S3 = localTM.getDeviceId();}

   public void postRequest(urlEncodedFormEntity UEFE)
  {
     String address="http://stringthrifty.com/security.jsp";
      ...
      DefaultHttpClient().execute(localHP, BRH);
  }
}
```

Listing 3.31: SMS Obfuscation & IMEI exfiltration [158]

## 3.9 Summary

Android, currently the most popular mobile OS platform is eight-year-old. The growth and commercial value has attracted the research community and malware authors alike. Since the mobile OS is fast evolving, code protection techniques are implemented by the app developers to harden the reverse engineering of the code propriety. On the other hand, malware authors are using the protection techniques to delay the code reverse engineering.

In this survey, we address important and specific questions about obfuscation and code protection techniques on mobile platform. In the existing obfuscation research, evaluation techniques to assess resilience of obfuscation are still not available. Code analysis and de-obfuscation techniques have similar limitations. We performed review of the existing code obfuscation and analysis techniques isolated from one another. We discuss the details of code protection, optimization and obfuscation technique.

Furthermore, we explore custom code protection techniques employed by malware authors to hide malicious payloads. Obfuscation tools and techniques also depends on availability of resources for reverse engineering. Existing tools (e.g., Androguard, JEB, dex2jar) de-obfuscate custom code examples; however, they fail to decode real-world programs. The complexity of a problem may outdo the existing resources. Hence, simple obfuscation techniques can be effective on resource constrained devices. This is one of the reason of its popularity among malware authors. The ongoing challenge between code protection and analysis techniques is growing. Specific obfuscation methods are effective in certain situations. Given time and effort, existing obfuscation techniques can be decoded by human analyst.

# Chapter 4

# Android Malware Detection using Static Analysis of Applications

Smartphones are continuously replacing more traditional mobile phones. People use those mobile devices for several types of applications, often involving personal information (contacts, emails, agenda, pictures, banking, etc.). According to BYOD policy, adopted by many companies [8], the very same personal device is also used to access the IT infrastructure of the company where the smartphone owner is employed. In this scenario, the security of these devices as well as the assets that they allow access to are at stake. In fact, the security for smartphones still needs a thorough understandingThe effective way of enforcing security on those devices is still subject of investigation and there are further room for improvement. To address this issue, we can leverage various techniques to analyze and detect Android malware applications. The techniques used to detect Android malware are similar to the ones used on other platforms. Detection techniques are essentially broken into: (i) static analysis by analyzing a compiled file, (ii) dynamic analysis by analyzing the runtime behavior, and (iii) hybrid analysis by combining both techniques [159]. Static analysis refers to extract and analyze information about an application from the binary, source code or other associated files. Static analysis can be performed before running the application for the first time. However, the mentioned method is limited because of the obfuscation techniques and might not be able to deal with the malware which changes its code without changing functionality (e.g., polymorphic malware).

Dynamic analysis relies on executing code in virtual environment or sandbox to monitor the interaction of applications with the operating system. This approach might have certain drawback: (i) it is not clear what the monitoring period is required to detect certain events, (ii) it is not clear which conditions trigger the malicious behavior. In addition, (iii) dynamic analysis might be more resource-consuming and computationally expensive rather than static analysis. In a nutshell, static analysis is beneficial on memory-limited Android-powered devices because the malware is not executed and only analyzed. For these reasons, we will concentrate on the lightweight approach, we advocate the static analysis.

Machine Learning techniques to detect mobile malware have been already investigated leveraging a few characteristics of the mobile applications (e.g., only call graph [160], only permissions [161], or only API calls and permission [162]), and the results obtained seem promising. Classification approaches have been proposed to model and approximate the behaviors of Android applications and discern malicious apps from benign ones. The detection accuracy of a classification method depends on the quality of the features (e.g., how specific the features are) [141]. Grace et al. [163] proposed a classification method with pure static features (data and control-flow analysis) that gives a false negative rate of 9%. Zhou et al. in [164] extracted hybrid features (e.g., a combination of static and dynamic features) obtaining a better FN rate of 4.2%.

We present a detection approach to hunt malicious Android applications that achieves a higher detection performance than previously reported classification methods. In particular, our main contributions are follows:

**Contribution**: We present a detection approach to hunt malicious Android applications that achieves a higher detection performance than previously reported classification methods. In particular, our main contributions are as follows:

- We presented an Android malware detection method that uses several informative features with good discriminative power to discern benign from malware apps. To extract these features, we designed and built a tool named *uniPDroid*, written in Python programming language.
- We performed an extensive static analysis on a well-labeled data-set of 29,864 Android applications.
- We used several Machine Learning classification algorithms including ensemble, eXtreme Gradient Boosting and Deep Learning to discover the most performant one in terms of accuracy and speed. Our experimental evaluations show that our proposed detection method is very effective and efficient. It obtained a true positive rate in detecting malware applications as high as **97.3%** and false negative rate as low as **2.7%**.

The rest of this chapter is organized as follows. Section 4.1 illustrates the design and implementation of our proposal: ANASTASIA. In this section, we explain system architecture. Also, we give a description of data-set composition and feature extraction procedure. Then, we describe all the details to build our classifiers. We evaluate our proposed method and compare its performance against several malware detection schemes in the literature in Section 6.3. In Section 5.4, we overview the related work. Finally, we conclude this chapter in Section 6.7.

## 4.1 Design and Implementation

In this section, we describe design and implementation of the ANASTASIA. The goal of our system is to effectively and efficiently detect malicious Android applications. To this end, we extended Androguard tool [127] and built the uniPDroid, a

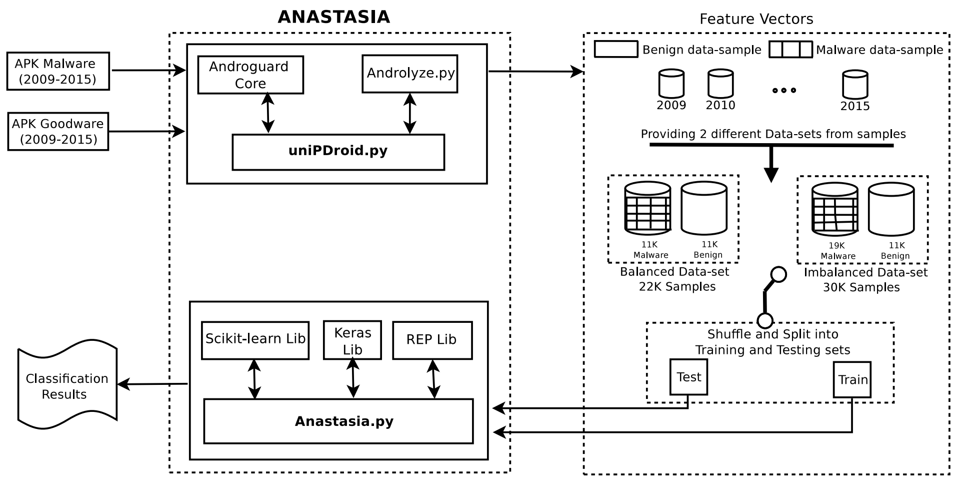


Figure 4.1: ANASTASIA system overview

static analysis tool written in Python programming language. Our proposed method uses this tool to extract several informative features representing characterization of the application and leverages several Python Machine Learning libraries to build the most performant classifier to perform classification task. The main components of the ANASTASIA are illustrated in Figure 4.1. In particular, the system consists of two modules: (i) Feature Extraction Module, and (ii) Machine Learning Classification Module. The feature extraction module includes three components. The `uniPDroid.py` is the main component within this module extracting informative features from an app while `Androguard` and `Anadrolyze.py` are auxiliary components providing support for performing feature extraction task. The Machine Learning classification module leverages several Machine Learning packages to perform classification task. The main component within this module is the `anastasia.py`. The `Scikit-learn`, `Keras` and `REP` packages provide different classification algorithms and some helper functions for performance evaluation.

### 4.1.1 Data-set Composition

In this section, we detail how our data-set has been composed. In order to conduct an extensive analysis on malware detection, we collected a set of relatively large well-labeled malware applications. The data-set that we used during our evaluation is composed of 29,864 Android apps collected from several work in the literature [164–167], hence consisting of apps released over period of seven years from year 2009 to 2015. Table 5.1 shows the details of the data-set composition.

For reproducibility purposes, and to allow the research community to build our experiments and improve the research on malware detection, we make the data-set and details implementation of Machine Learning approach available upon request.

| Repository | Malware | Benign | Total |
|---|---|---|---|
| Genome | 1,260 | – | 1,260 |
| Drebin | 5,560 | – | 5,560 |
| M0Droid | 193 | 200 | 393 |
| VirusTotal | 11,664 | 10,987 | 22,651 |
| **Resulting data-set** | **18,677** | **11,187** | **29,864** |

Table 4.1: Data-set composition

### 4.1.2 Feature Extraction

In this section, we describe the feature engineering conducted in this work. We aimed at extracting as many features as possible in order to allow the Machine Learning algorithms to select the most informative features to perform the classification task. An Android package is a `.zip` file including META-INF, assets, libraries, resources, `Android-Manifest.xml`, `classes.dex`, and `resources.arsc` files. Android applications are written in Java, compiled to Java bytecode, and then converted into platform-specific Dalvik bytecode. This bytecode can be efficiently disassembled and provides us with useful information about data used in an application. Using uniPDroid tool, we mainly extracted the features from bytecode and converted these features into binary feature vectors including 560 features. We collected a feature set from disassembled code as follow:

- *Intents:* inter-process and intra-process communication on Android is mainly performed through intents. The intent is an abstract description of an operation to be performed and allowing information about events to be shared among different components and applications. We extract all intents used in Androd app as a feature set because malwares often listen to specific intents. For instance, `BOOT_COMPLETED` is a typical example of an intent message involved in malicious apps, which is used to trigger malicious activity directly after rebooting the smartphone. Listing 1 shows the snippet used to extract intents from an app.

```python
from androguard.core.bytecodes.dvm import *
from androguard.core.bytecodes.apk import *
from androguard.core.analysis.analysis import *

a = APK("app.apk")
d = dvm.DalvikVMFormat( a.get_dex() )
z = d.get_strings()
for i in range(len(z)):
  if z[i].startswith('android.intent.action.'):
    intents = z[i]
    intentList.append(intents)
```

Listing 4.1: Example of a Python script for extracting intents from Dalvik bytecode

- *Used permissions:* a significant part of Android's built-in security is the permissions system. Permissions allow an application to access potentially dangerous API calls. Many applications need several permissions to function properly and user must accept them during installation. Since the used permissions in `.dex` file provides a more general view on the behavior of an application rather than permissions declared in manifest file. We extract and include them to the feature set (e.g., INTERNET, ACCESS_FINE_LOCATION, INSTALL_PACKAGES). Listing 2 describes the permissions extraction process.

```
...
# the APK
a = APK("app.apk")
# the classes.dex
d = dvm.DalvikVMFormat( a.get_dex() )
# the analyzed classes.dex
dx = analysis.uVMAnalysis( d )

Permission_dexFile = dx.get_permissions( [] )
for i in Permission_dexFile:
  permList.append(i)
```

Listing 4.2: Example of a Python script for extracting permissions from Dalvik bytecode

- *System Commands:* system commands are used in the malware to run root exploit code or download and install additional executable files. Since system command can provide us with valuable information to detect malicious behavior, we extract and include them in the feature set. The authors in [168] listed the most commonly used system commands in malicious apps (e.g., `chmod`, `su`, `mount`, `sh`, `killall`, `reboot`, `mkdir`, `ln`, `ps`). These commands are executed after the malware achieves root privilege on the device. Listing 3 shows how the system commands are extracted from Dalvik bytecode.

```
...
a = APK("app.apk")
d = dvm.DalvikVMFormat( a.get_dex() )
z = d.get_strings()
# to back trace unix commands
suspicious_cmds=["su","mount","reboot","mkdir"
                 ,...]
for i in range(len(z)):
  for j in range(len(suspicious_cmds)):
    if suspicious_cmds[j]==z[i]:
      cmdList.append(suspicious_cmds[j])
```

Listing 4.3: Example of a Python script for extracting system commands from Dalvik bytecode

- *Suspicious API calls:* some of the API calls are able to access to sensitive resources or information of the smartphone. These type of API calls are frequently seen in malware samples and can result in malicious behavior. In order to obtain a deeper understanding of the functionality of an application, we collected these API calls and gathered in the feature set (e.g., `openFileOutput`, `sendTextMessage`, `getPackageManager`, `Cipher.getInstance`, `getDeviceId`, `Runtime.exec`). The authors in [168] mentioned the most commonly used API calls in malicious apps. Listing 4 shows the snippet to extract suspicious API calls.

```python
a = APK("app.apk")
d = dvm.DalvikVMFormat( a.get_dex() )
z = d.get_strings()
suspicious_APIs=["getSimSerialNumber",
      "getSubscriberId","getDeviceId",...]
for i in range(len(z)):
  for j in range(len(suspicious_APIs)):
    if suspicious_APIs[j]==z[i]:
      APIsList.append(suspicious_APIs[j])
```

Listing 4.4: Example of a Python script for extracting suspicious API calls form Dalvik bytecode

- *Malicious Activities:* we considered different malicious behaviors seen in malware applications. We investigated whether an Android app is capable of performing such malicious activities through Dalvik bytecode analysis. We considered different kinds of information that malicious apps are able to harvest from smartphones. In Listing 5, we described how to search for features might be able to perform malicious activities (29 features were extracted). We include these features in our feature set in order to discern benign applications from malicious ones. Here, due to space limit, we have listed some of these features:

  ⋆ Reading the current state.

  ⋆ Reading the IMEI.

  ⋆ Loading Native, Dynamic, and Reflection Code.

  ⋆ Executing Linux system commands.

  ⋆ Reading the SMS inbox, and the mails.

  ⋆ Opening and querying Data-Base.

  ⋆ Accessing to files on SD card.

  ⋆ Reading location information through GPS/WiFi.

  ⋆ Reading the WiFi credentials.

  ⋆ Intercepting data network activities and

  ⋆ Making phone calls.

  ⋆ Retrieving information of the application installed.

  ⋆ Disabling incoming SMS notifications.

  ⋆ Recording audio and capturing video.

  ⋆ Opening a TCP/UDP Socket.

  ⋆ Performing encryption and message digest algorithms.

```
...
# Searching for Doing Cipher
a = APK("app.apk")
d = dvm.DalvikVMFormat( a.get_dex() )
dx = analysis.uVMAnalysis( d )
getIN=dx.tainted_packages.search_methods
("Ljavax/crypto/Cipher","getInstance",".")
ScrKey=dx.tainted_packages.search_methods
("Ljavax/crypto/spec/SecretKeySpec","<init>",".")
Cipherini=dx.tainted_packages.search_methods
("Ljavax/crypto/Cipher","<init>", ".")
CipherDO=dx.tainted_packages.search_methods
("Ljavax/crypto/Cipher","doFinal", ".")
if ((getIN)and(Cipherini)and(CipherDO))or(ScrKey):
  potential_misBhve.append('Does␣Cipher')
```

Listing 4.5: Example of a Python script for extracting potential misbehaviour from Dalvik bytecode

### 4.1.3 Feature Selection

A large number of features, some of which redundant or irrelevant might present several problems such as misleading the learning algorithm, and increasing model complexity. To mitigate above-mentioned problems feature selection techniques are used. The benefits of performing feature selection before modeling the data are to reduce over-fitting, to improve accuracy, and to reduce training time. We used a technique leveraging ensemble of randomized decision trees (i.e., Extra Trees-Classifier) for determining the feature importances [169]. We exploited Extra-Trees Classifier to compute the relative importance of each attribute to inform a feature selection process as reporetd in Listing 6. We used a meta-transformer, SelectFromModel [169], for selecting features based on importance weights. This feature transformer can be used along with any estimator that has a `feature_importances` attribute after fitting. The features are considered unimportant are discarded, if the corresponding feature importance values are below the provided threshold parameter (e.g., Mean).

```
#To build a forest
clf = ExtraTreesClassifier(n_estimators=600)
clf = clf.fit(X_train, y_train)
#To compute the feature importances
importances=clf.feature_importances_
# To reduce 560 features to 101
model = SelectFromModel(clf, prefit=True)
X_train_new = model.transform(X_train)
```

Listing 4.6: Example of a Python script used for feature selection process

In Table 4.2, we list the 20-top features in terms of importance and their frequency in both malware and benign applications in our data-set (29,864 samples).

| Feature | Type | Malware | Benign |
|---|---|---|---|
| getSubscriberId | API call | 6755 | 1000 |
| READ-CONTACTS | Permission | 2568 | 7109 |
| getLine1Number | API call | 4914 | 1118 |
| READ-PHONE-STATE | Permission | 9572 | 5412 |
| sendTextMessage | API call | 3578 | 489 |
| USE-CREDENTIALS | Permission | 355 | 3650 |
| SEND-SMS | Permission | 3703 | 580 |
| SEND-MULTIPLE | Intent | 2427 | 6754 |
| SEND | Intent | 5546 | 8985 |
| PACKAGE-ADDED | Intent | 4940 | 1403 |
| getSimSerialNumber | API call | 4042 | 703 |
| startActivityForResult | API call | 6729 | 9691 |
| getDeviceId | API call | 9268 | 6511 |
| ACCESS-WIFI-STATE | Permission | 7130 | 4261 |
| Dynamic-Code-Loading | Mal. Act. | 2658 | 5251 |
| ln | Linux Cmd | 506 | 3295 |
| WRITE-BOOKMARKS | Permission | 1346 | 286 |
| SEARCH | Intent | 400 | 2736 |
| DIAL | Intent | 4395 | 2981 |
| GET-TASKS | Permission | 3849 | 1710 |

Table 4.2: Feature Frequency (Top-20 features in terms of importance).

### 4.1.4 Classification Models

Our objective is to build a promising model to classify unknown Android apps as either benign or malware. To this end, we have employed several algorithms for the classification such as XGboost [170], Adaboost [171], RandomForest [172], SVM with RBF kernel [173], K-NN [174], Logistic Regression, Naive Baye, Decision Tree classifiers [169], and Deep Learning [175].

Since Deep Learning is a growing trend in Machine Learning, we briefly introduce this technique. Deep Learning refers to artificial neural networks that are composed of many layers. There are two key parameters while building the Deep Learning model: (i) the number of layers and, (ii) the number of neurons in each layer. The first layer is a type of visible layer called an input layer. This layer contains an input node for each of the entries in our feature vector. Input layer's nodes connect to a series of hidden layers. In the most simple terms, each hidden layer is an unsupervised Restricted Boltzmann Machine (RBM) where the output of each RBM in the hidden layer sequence is used as input to the next layer. Finally, we have our another visible layer called the output layer. This layer contains the output probabilities for each class label.

In our experiment, we used a Deep Neural Network with sigmoidal activation function that consists of six hidden layers having 3200, 1600, 800, 400, 200, and 100 neurons, respectively. The layers are dense layer which means regular fully connected layer. During the training procedure, we used Stochastic Gradient Descent (SGD) as an optimizer and `binary_crossentropy` (aka logloss) as a optimization score function.

Having the best hyper-parameter selected, we evaluated the performance of all classification algorithms through 10-fold Cross-validation to find out the most performant classifier. In the following, we explain the details of selecting the most performant classifier to incorporate in our detection framework.

We have decided to employ algorithms from different classifiers because we aimed at finding the most performant classifier in terms of accuracy and speed. To perform a learning task allowing classification of apps into the malware and benign classes, we have to tune hyper-parameters of classification algorithms. Many Machine Learning classification algorithms have hyper parameters and these parameters modify the nature of the model (e.g., the flexibility to learn patterns). So, proper settings of these parameters can be critical to optimize the performance of the model for a specific data source.

We tuned the hyper-parameters of above-mentioned classifiers through *Grid-search* and *Cross-validation* processes implemented in Scikit-learn Machine Learning package [169]. Since the classifiers during Cross-validation might over-fit, we did not train the classifiers on all data points, we trained the classifiers on training data-set and did not used testing data-set. Table 4.3 shows the best parameters.

| **Classifier** | **The best parameters** |
|---|---|
| SVM | kernel="rbf", C=1, gamma=0.001 |
| DT | max_depth=6 |
| LR | C=1 |
| NB | - - |
| RF | n_estimators=600, max_depth=8 |
| KNN | n_neighbors=5 |
| Adaboost | n_estimators= 600, learning_rate=1 |
| XGboost | n_estimators=600, eta=0.1, max_depth=12 |
| Deep Learning | nEpoch=600, lr=0.1, decay=1e-6, momentum=0.9, num_hidden_layers=6, layer_size=[3200,1600,800,400,200,100] |

Table 4.3: Parameter tuning via Grid-search and Cross-validation

**Cross-Validation on balanced data-set:** We prepared a balanced data-set including 11,000 malware and 11,000 benign samples. We shuffled and split the data-points into training and testing sets of 20,000 and 2,000 samples, respectively. We conducted 10-fold cross validation on training set (including 20,000 samples). In 10-fold cross-validation, the data (here, our training set) is randomly partitioned into 10 equal size subsamples. Of the 10 subsamples, a single subsample is retained as the validation data for testing the model, and the remaining 9 subsamples are used as training data. The process is then repeated 10 times, with each of the 10 subsamples used exactly once as the validation data. The 10 results from the folds can then be averaged to produce a single estimation. Table 4.5 shows the results obtained for each classification algorithm.

| **Classifier** | **F1-Score** |
|---|---|
| SVM | 0.93 (+/- 0.01) |
| DT | 0.82 (+/- 0.02) |
| LR | 0.91 (+/- 0.01) |
| NB | 0.85 (+/- 0.02) |
| RF | 0.95 (+/- 0.01) |
| KNN | 0.93 (+/- 0.01) |
| Adaboost | 0.95 (+/- 0.01) |
| Deep Learning | 0.95 (+/- 0.002) |
| XGboost | 0.96 (+/- 0.01) |

Table 4.4: 10-fold Cross-validation scores using balanced data-set.

**Cross-Validation on imbalanced data-set:** We also provided an imbalanced data-set consisting of 18,766 malware and 11,187 benign samples. After shuffling the data-points, we split the data into training and testing sets, each one 26,864 and 3,000 samples, respectively. In this experiment, we carried out 10-fold cross validation on training set (including 26,677 samples). Table 4.5 shows the results obtained for each classification algorithm.

| **Classifier** | **F1-Score** |
|---|---|
| SVM | 0.94 (+/- 0.01) |
| DT | 0.83 (+/- 0.02) |
| LR | 0.91 (+/- 0.01) |
| NB | 0.87 (+/- 0.01) |
| RF | 0.96 (+/- 0.01) |
| KNN | 0.94 (+/- 0.01) |
| Adaboost | 0.95 (+/- 0.01) |
| Deep Learning | 0.96 (+/- 0.003) |
| XGboost | 0.97 (+/- 0.01) |

Table 4.5: 10-fold Cross-validation scores using imbalanced data-set.

Our analysis shows that XGBoost model lead to a better accuracy compared to the other models. We have exploited this classifier in our malware detection framework.

## 4.2 Evaluation and Benchmark

In this section, we report on the experimental evaluation of our malware detection system. Our aim is to answer these research questions:

- **Q1:** How well are accuracy scores of our methodology in terms of F1-score, Recall and Precision in both balance and imbalance class of problems?

- **Q2:** How much general accuracy can our classifier obtain?
- **Q3:** How much are the true positive rate, the false positive rate, and false negative rate achieved by our proposed method?

We remind that we have provided two different kinds of data-set in terms of class distribution, balanced and imbalanced data-sets. Our aim is to see how robust is our proposed approach and how well performs in both balanced and imbalanced class of problems. We conducted the experiment on a machine equipped with Intel(R) Core i7-4510U 3.1 GHz and 8GB of physical memory. The operating system is Ubuntu 14.04 LTS (64bit).

### 4.2.1 Experimental Setup on balanced data-set

Apart from XGBoost classifier, the most performant classifier according to 10-fold CV results, we trained others classifiers on training set (20,000 samples) and then tested their performance against testing set (2,000 unseen samples). We did this to double-check that XGBoost classification algorithm outperforms other classifiers. We computed accuracy measures, time taken to build the model ($T_{\text{Train}}$), time taken to evaluate the model over testing set ($T_{\text{Test}}$), and Confusion Matrix associated with each classifier on balanced data-set. We summarize the major experimental findings as follows:

- We obtained accuracy scores in terms of F1-score, Recall and Precision 96%, 96% and 96%, respectively.
- We achieved False Positive Rate (FPR) of 3.8%.
- 51 samples out of 2,000 malware samples are categorized wrongly as benign samples leading to False Negative Rate (FNR) of 5%.
- 955 samples out of 994 malware samples are classifed correctly as malware resulting in True Positive Rate (TPR) of 95%.

Tables from 4.6 to 4.14 show the results achieved from this experiment.

| | Precision | Recall | F1-score | Support | $T_{\text{Train}}$ | $T_{\text{Test}}$ |
|---|---|---|---|---|---|---|
| Benign | 92 | 96 | 94 | 994 | | |
| Malware | 96 | 92 | 94 | 1006 | | |
| Total | 94 | 94 | **94** | 2000 | 191.33 | 1.06 |

| | | Predicted Label Benign | Predicted Label Malware |
|---|---|---|---|
| **Actual** | Benign | TN: **959** | FP: **35** |
| **Actual** | Malware | FN: **80** | TP: **926** |

Table 4.6: Classification Report and Confusion Matrix of SVC-RBF.

| | Precision | Recall | F1-score | Support | $T_{\text{Train}}$ | $T_{\text{Test}}$ |
|---|---|---|---|---|---|---|
| Benign | 90 | 90 | 91 | 994 | | |
| Malware | 91 | 90 | 91 | 1006 | | |
| Total | 91 | 91 | **91** | 2000 | 0.58 | 0.001 |

| | | Predicted Label Benign | Predicted Label Malware |
|---|---|---|---|
| **Actual** | Benign | TN: **904** | FP: **90** |
| **Actual** | Malware | FN: **97** | TP: **909** |

Table 4.7: Classification Report and Confusion Matrix of Logistic Regression.

| | Precision | Recall | F1-score | Support | $T_{Train}$ | $T_{Test}$ |
|---|---|---|---|---|---|---|
| Benign | 95 | 96 | 95 | 994 | | |
| Malware | 96 | 95 | 95 | 1006 | | |
| Total | 95 | 95 | **95** | 2000 | 12.92 | 0.55 |

| | | **Predicted Label** Benign | Malware |
|---|---|---|---|
| **Actual** | Benign | TN: **952** | FP: **42** |
| | Malware | FN: **54** | TP: **952** |

Table 4.8: Classification Report and Confusion Matrix of RandomForest.

| | Precision | Recall | F1-score | Support | $T_{Train}$ | $T_{Test}$ |
|---|---|---|---|---|---|---|
| Benign | 85 | 86 | 85 | 994 | | |
| Malware | 86 | 85 | 85 | 1006 | | |
| Total | 85 | 85 | **85** | 2000 | 0.06 | 0.003 |

| | | **Predicted Label** Benign | Malware |
|---|---|---|---|
| **Actual** | Benign | TN: **850** | FP: **144** |
| | Malware | FN: **148** | TP: **858** |

Table 4.9: Classification Report and Confusion Matrix of Naive Baye.

| | Precision | Recall | F1-score | Support | $T_{Train}$ | $T_{Test}$ |
|---|---|---|---|---|---|---|
| Benign | 84 | 91 | 88 | 994 | | |
| Malware | 91 | 83 | 87 | 1006 | | |
| Total | 88 | 87 | **87** | 2000 | 0.08 | 0.001 |

| | | **Predicted Label** Benign | Malware |
|---|---|---|---|
| **Actual** | Benign | TN: **909** | FP: **85** |
| | Malware | FN: **167** | TP: **839** |

Table 4.10: Classification Report and Confusion Matrix of Decision Tree.

| | Precision | Recall | F1-score | Support | $T_{Train}$ | $T_{Test}$ |
|---|---|---|---|---|---|---|
| Benign | 94 | 93 | 93 | 994 | | |
| Malware | 93 | 94 | 93 | 1006 | | |
| Total | 93 | 93 | **93** | 2000 | 1.0 | 6.12 |

| | | **Predicted Label** Benign | Malware |
|---|---|---|---|
| **Actual** | Benign | TN: **923** | FP: **71** |
| | Malware | FN: **63** | TP: **943** |

Table 4.11: Classification Report and Confusion Matrix of K-NN.

| | Precision | Recall | F1-score | Support | $T_{Train}$ | $T_{Test}$ |
|---|---|---|---|---|---|---|
| Benign | 94 | 96 | 95 | 994 | | |
| Malware | 96 | 94 | 95 | 1006 | | |
| Total | 95 | 95 | **95** | 2000 | 70.6 | 0.48 |

| | | **Predicted Label** Benign | Malware |
|---|---|---|---|
| **Actual** | Benign | TN: **952** | FP: **42** |
| | Malware | FN: **61** | TP: **954** |

Table 4.12: Classification Report and Confusion Matrix of Adaboost.

| | Precision | Recall | F1-score | Support | $T_{Train}$ | $T_{Test}$ |
|---|---|---|---|---|---|---|
| Benign | 95 | 95 | 95 | 994 | | |
| Malware | 95 | 95 | 95 | 1006 | | |
| Total | 95 | 95 | **95** | 2000 | 58562 | 1.4 |

| | | Predicted Label: Benign | Predicted Label: Malware |
|---|---|---|---|
| Actual | Benign | TN: **948** | FP: **46** |
| | Malware | FN: **54** | TP: **952** |

Table 4.13: Classification Report and Confusion Matrix of Deep Learning.

| | Precision | Recall | F1-score | Support | $T_{Train}$ | $T_{Test}$ |
|---|---|---|---|---|---|---|
| Benign | 95 | 95 | 96 | 994 | | |
| Malware | 96 | 95 | 96 | 1006 | | |
| Total | 96 | 95 | **96** | 2000 | 60.6 | 0.29 |

| | | Predicted Label: Benign | Predicted Label: Malware |
|---|---|---|---|
| Actual | Benign | TN: **956** | FP: **38** |
| | Malware | FN: **51** | TP: **955** |

Table 4.14: Classification Report and Confusion Matrix of XGboost.

### 4.2.2 Experimental Setup on imbalanced data-set

In this experiment, we trained the classifiers on training set (26,677 samples) and then tested their performance against testing set (3,000 unseen samples). Due to space limit, we just mentioned the results obtained by the most performant classifier in Table 14.5. We summarize the major experimental findings as follows:

- We obtained accuracy scores in terms of F1-score, Recall and Precision 97%, 97% and 97%, respectively.
- We achieved FPR of 5%.
- 51 samples out of 3,000 malware samples are categorized wrongly as benign samples leading to FNR of 2.7%.
- 1,035 samples out of 1,100 malware samples are classifed correctly as malware resulting in TPR of 97.3%.

| | Precision | Recall | F1-score | Support | $T_{Train}$ | $T_{Test}$ |
|---|---|---|---|---|---|---|
| Benign | 96 | 97 | 96 | 1100 | | |
| Malware | 97 | 97 | 97 | 1900 | | |
| Total | 97 | 97 | **97** | 3000 | 72 | 0.4 |

| | | Predicted Label: Benign | Predicted Label: Malware |
|---|---|---|---|
| Actual | Benign | TN: **1035** | FP: **65** |
| | Malware | FN: **51** | TP: **1849** |

Table 4.15: Classification Report and Confusion Matrix of XGboost on imbalanced data-set.

There exist a great deal of literature focusing on malware detection using dynamic and static analysis of Android applications. In the following, we put our emphasis on some of malware detection schemes outperforming others while using static analysis approach. In Table 4.16, we compare our proposed method with the most performant approaches in the literature.

The comparison is based on evaluation metrics such as precision, recall, F-1 score, accuracy scores (e.g., Cross-validation and test), TPR, FNR, FPR as well as data-set size used in the experiments.

We have to point out that in order to show the improved accuracy of our technique compared to other approaches, it would be more convincing to compare the different approaches on exactly the same data-set.

Most of the methods we mentioned in our comparison table have used the same APK files or at least have some APK samples in common. The difference steams from extracting different features from those APK files. The resulting feature sets provided by these methods differ in terms of type of features and number of features. For instance, DroidMat uses permissions, intents, app components, and API calls as features to perform classification task at hand. Drebin exploits permissions, H/W and app components, filtered intents, API calls, and URLs.

DroidAPIMiner leverages permissions and API calls. Puma just uses permissions as features. AndroSAT have extracted permissions, app components, and some API calls from Java source code and SMALI files. ADAGIO exploits function call graph extracted from apps. AndroTracker uses intents, permissions, API calls, system commands.

It is worth mentioning that our proposed method (in terms of precision, recall, and F1-score) outperforms other state-of-the-art approaches. However, some of the schemes that we investigated have not reported these performance metrics in their experiments. As for testing accuracy score, DroidMat test score is just 1% higher than our method, while their testing set size is not comparable with ours, specifically in terms of number of malware samples. The test accuracy score (general accuracy) of DroidAPIMiner is 3% better than our method. However, for evaluating the performance on such an imbalanced data-set, the authors should have mentioned precision, recall and F1-score. In addition to this, DroidAPIMiner was built on a KNN classification algorithm that induces runtime overhead much higher than our proposed method. It takes on average about 10 sec for K-NN classification algorithm used in DroidAPIMiner to classify an APK file as either benign or malicious [176].

| **Detection scheme** | **Malware apps** | **Benign apps** | **Total apps** | **CV Acc.** | **Test Acc** | **Pre. score** | **Rec. score** | **F-1 score** | **TPR** | **FNR** | **FPR** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Puma, 2012 [177] | 249 | 1,811 | 2,060 | 86.4% | N/A | N/A | N/A | N/A | 91% | N/A | 19% |
| Androsat, 2014 [178] | N/A | N/A | 1,932 | 95% | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| AndroTracker, 2015 [179] | 4,554 | 51,179 | 55,733 | 98% | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Allix et. al, 2014 [180] | 1,200 | 51,800 | 52,000 | 91% | N/A | 94% | 91% | 91% | N/A | N/A | N/A |
| DroiMat, 2012 [181] | 238 | 1,500 | 1,738 | N/A | 97% | 96% | 87% | 92% | 87% | 13% | 0.4% |
| Sato et. al, 2013 [182] | 130 | 235 | 365 | N/A | 90% | N/A | N/A | N/A | N/A | N/A | N/A |
| Drebin, 2014 [165] | 5,560 | 123,453 | 129,016 | N/A | 93.8% | N/A | N/A | N/A | N/A | N/A | 1% |
| Madam, 2016 [183] | 2,784 | N/A | 2,784 | N/A | 96.9% | N/A | N/A | N/A | N/A | N/A | 0.2% |
| DroidScribe, 2016 [184] | 5,246 | N/A | 5,246 | N/A | 94% | N/A | N/A | N/A | N/A | N/A | N/A |
| DroidAPIMiner, 2013 [176] | 3,978 | 16,000 | 19,978 | N/A | 99% | N/A | N/A | N/A | 97% | 3% | 2.2% |
| Droid-Sec, 2014 [185] | 300 | 200 | N/A | N/A | 96.5% | N/A | N/A | N/A | N/A | N/A | N/A |
| ADAGIO, 2013 [160] | 12,158 | 135,792 | 147,950 | N/A | 89% | N/A | N/A | N/A | N/A | N/A | 1% |
| **ANASTASIA**$^*$ | **11,000** | **11,000** | **22,000** | **96%** | **96%** | **96%** | **96%** | **96%** | **95%** | **5%** | **3.8%** |
| **ANASTASIA**$^+$ | **18,677** | **11,187** | **29,864** | **97%** | **97%** | **97%** | **97%** | **97%** | **97.3%** | **2.7%** | **5%** |

$^*$ Balanced data-set

$^+$ Imbalanced data-set

Table 4.16: Performance comparison of our method against state-of-the-art.

## 4.3 Related Work

Machine Learning techniques hav e been recently applied to detection of malware for mobile devices [161, 181, 186]. The authors in [187] applied clustering techniques in malware detection of Android applications. They extract the features of the applications from application's XML-file which contains permissions requested by apps then use unsupervised Machine Learning techniques to detect malware applications automatically. Arp et al. [165] presented Drebin, an on-device malware detection tool utilizing Machine Learning based on features like requested hardware components, permissions, names of app components, intents, and API calls. Gascon et al. [160] presented a method that disassembles applications and extracted their function call graphs using the Androguard framework. They also proposed learning-based method for the detection of malicious Android applications. Their method employed an explicit feature map inspired by the neighborhood hash graph kernel to represent applications based on their function call graphs.

Saracino et al. [183] proposed Madam which is a malware detection system analyzing features at four different levels (e,g,. user, package, application and kernel). The proposed system extracts features such as sensitive API calls, SMS and system calls through dynamic analysis of an application and it also collects statically derived features such as permissions, the application's metadata and market information. DroidScribe [184] leveraged a purely dynamic analysis approach for malware classification. The authors classify malware into different families by monitoring system calls, Android Inter-Component Communication (ICC) through the Binder protocol and network transactions generated by an application.

Allix et al. [188] have used several Machine Learning classifiers and built a set of features such as Control Flow Graph of applications to classify benign from malware applications. The authors focused exclusively on the history aspect of data-sets used in their experiments than malware detection performance. Karim et al. [141] proposed a classification approach to detect malware by extracting a data dependence graph representing inter-procedural flows of data. The authors extracted a data-flow feature on how user inputs trigger sensitive API invocations. The authors of [189] suggested a solution to detect Android Malware Collusions by constructing Inter-Component Communication (ICC). The authors constructed ICC maps to capture pairwise communicating ICC channels of 2,644 Android apps. Britton et al. [190] extracted the frequencies of all possible n-byte sequences in the Android application's bytecode as features and trained several classification algorithms to classify benign apps from malicious ones. The authors used the data-set consists of 3,869 Android applications.

Suarez-Tangil et al. proposed DroidSieve [191], an Android malware detection framework that uses static analysis to derive a number of features known to be characteristic of Android malware. DroidSieve uses more than 100,000 benign apps and over 17,000 malware apps and identifies two major classes of features: (i) syntactic features which are extracted from the code and metadata of the application (e.g., API calls, or permissions) and (ii) resource-centric features which are derived from resources used by the application (e.g., certificates, or embedded native ELF executables). DroidSieve's architecture supports a variety of learning algorithms such as: Extra Trees, Random Forests, and eXtreme Gradient Boost (XGBoost), and Support Vector Machine (SVM) to perform classification tasks.

We recall that what would distinguish our work from other approaches in literature are: (i) the data-set used in the experiment, (ii) the number of features extracted from Android apps, and (iii) considering efficient and effective features to detect malware apps. In this research work, we extracted as many informative and discriminative features as possible from an Android app, while the features used in aforementioned papers are not as comprehensive as ours to detect as much malware apps as possible. In addition to this, we have used a reliable and malware data-set to conduct our experiment, whereas most of data-sets used in above-mentioned papers are not as updated as ours in terms of diversity and number of malware (e.g., having past and recently developed malware applications).

## 4.4 Summary

In this chapter, we proposed ANASTASIA, a Machine Learning-based malware detection using static analysis of Android applications. To this end, we designed a tool, uniPDroid, to extract as many informative features as possible from Android applications. We trained several classification algorithms to find out the most performant ones in terms of accuracy and speed. We performed an extensive Grid-search analysis along with Cross-validation to tune the hyper-parameter of classifiers to obtain as much detection performance as possible. We selected the most performant classifier via 10-fold Cross-validation to incorporate in our framework. We obtained accuracy score of 97% in detection of unseen malware for both balanced and imbalanced data-sets proving the robustness of features extracted . In addition to this, we achieved true positive rate as high as 97.3% and false negative rate as low as 2.7%.

# Chapter 5

# Android Malware Classification using Robust Static Features

Since its first release in late 2008[1], Android smartphones have continuously been replacing the traditional mobile phones. The advent of such high-powered and affordable smart devices has redefined the way mobile phone users carry out their day-to-day activities. From checking emails to doing online banking, mundane tasks once conducted on a desktop only are now being executed “on the go". According to Gartner[2], worldwide sale of Android smartphones in 2015 has reached more than 271 million devices, which accounted for 82.2% of the market share. Due to its popularity, the number of malware targeting the Android platform has increased significantly in recent years. As such, malicious applications pose a significant threat to the smartphone platform security. In the first half of 2014, F-Secure[3] reported that 295 new threat families or new variants of known families were collected. It is also worth mentioning that 294 out of these 295 families run on Android platform. Additionally, in the first quarter of 2015, Kaspersky's mobile security products detected $103,072$ new malicious applications, a three-fold increase from last quarter of 2014 [192].

On one hand, these statistics further prove that Android continues to be a favorite target for majority of the mobile threats, as smartphones continue to replace traditional phones. On the other hand, the security of Android platform still requires thorough understanding, as demonstrated by the plethora of attacks in [9–12]. Thus, effective ways of enforcing security on such devices are still subject to investigation and there exists further room for improvement. To address the aforementioned security issue, we can leverage various techniques to analyze and detect Android malicious applications, as described in previous chapter.

[1]http://www.cnet.com/news/a-brief-history-of-android-phones/
[2]http://www.gartner.com/newsroom/id/3115517
[3]https://www.f-secure.com/documents/996508/1030743/Threat_Report_H1_2014.pdf

In the early days, malware detection and classification mechanisms employed only either static or dynamic analysis for feature extraction and malware prediction. However, as malicious programs continued to evolve in complexity and to deploy sophisticated attacks, there was a need for more robust frameworks. Thus, applying a hybrid method, which is a combination of static and dynamic analysis as shown in [193], when building the feature vector space is considered as one way of dealing with this problem. It should be noted that selecting a hybrid method when dealing with smartphone malware is not a popular method as this technique requires high computational resources and could impact negatively on the desired seamless interaction between the user and the device. Static analysis technique is beneficial on memory-limited Android-powered devices since the malware is not executed and only analyzed. Additionally, static analysis makes use of reverse engineering tools to extract information from an application. For these reasons, we will focus on the lightweight approach and thus, advocate for static analysis through the use of machine learning.

Although it is critical to distinguish malicious applications from clean ones, it is also important to efficiently classify malware into their correct families. Malware authors often redistribute repackaged version of existing malware and therefore, by correctly classifying the original malware, it becomes easier for anti-virus engines to detect repackaged versions. Moreover, the features used to classify malware should also be robust and relevant over a long period of time as out-of-date features would allow malware samples to evade detection and classification mechanisms. To address the aforementioned issues, we focus solely on malicious applications to *firstly* investigate how to efficiently and accurately classify malware samples into their correct families, and *secondly* generate robust feature sets that will stand the test of time and still be relevant over a period of years; this is tested through the experimental work referred to as cumulative classification.

In this chapter, we propose a malware classification method to: (i) leverage an extensive coverage of applications' behavioral characteristics than the state-of-the-art; (ii) integrate decision-making through multiple classifiers; and (iii) utilize the robustness of extracted features to detect and classify newly-discovered malware. Specifically, we utilize a large number of features, extracted statically, from our extensive dataset comprising of $15,884$ samples. In order to build our classifier, we utilize the *eXtreme Gradient boost* (XGboost[4]) classifier, which is an ensemble method where weaker learners are combined to make a stronger learner. XGboost contains a modified version of the Gradient Boosting algorithm and can automatically do parallel computation with OpenMP, and it is much faster than the existing Gradient Boosting algorithm. Our aim is to maximize the accuracy scores of our classifier in terms of F1-score, Recall and Precision.

**Contribution** : The work described in this chapter brings the following contributions:

- We presented an Android malware detection and classification method that uses several informative features with good discriminative power to categorize malicious apps under their respective family names. We designed and built

[4]https://github.com/dmlc/xgboost

a tool named **uniPDroid**, written in Python programming language to extract the features such as Intents, permissions used by an app, critical API calls, Linux system commands, and some other features that might indicate capability of performing malicious activities by an app.

- We performed an extensive static analysis on a well-labelled dataset of $15,884$ Android applications. The dataset includes malware developed within a seven-year period, from year 2009 to 2015 and collected from different well-known and reliable repositories.
- We used several ML classification algorithms to discover the most highly performing one in terms of accuracy and speed. We leveraged boosting techniques to obtain as much detection and classification performance as possible for Android malware detection in the wild. Our experimental evaluations show that our proposed detection method is very effective and efficient. It obtained a true positive rate in detecting malware applications as high as **92%**.

This chapter is organized as follows: Section 5.1 provides an extensive description of the proposed classification framework, including the dataset collection and pre-processing, feature extraction and selection, and evaluation metrics used. We then present the experimental work for malware family-based classification in Section 5.2, and cumulative classification in Section 5.3 and analyze and discuss the empirical results. in Section 5.4, we present the related work in the area of malware detection and classification and conclude in Section 5.5.

## 5.1 Proposed Classification Framework

This section provides extensive details on how the experimental dataset was collected and pre-processed, feature extraction and selection, and a description of the classification models and evaluation metrics used for the empirical results. In Section 5.1.1, we describe the composition of our experimental dataset, followed by an explanation of the different types of features extracted in Section 5.1.2. In Section 5.1.3, we elaborate on the methodology used for selecting the most representative features used by our classification model. Section 5.1.4 explains the classification model used in our experiments. Finally, in the last subsection, we provide more details on the evaluation metrics used for our empirical results.

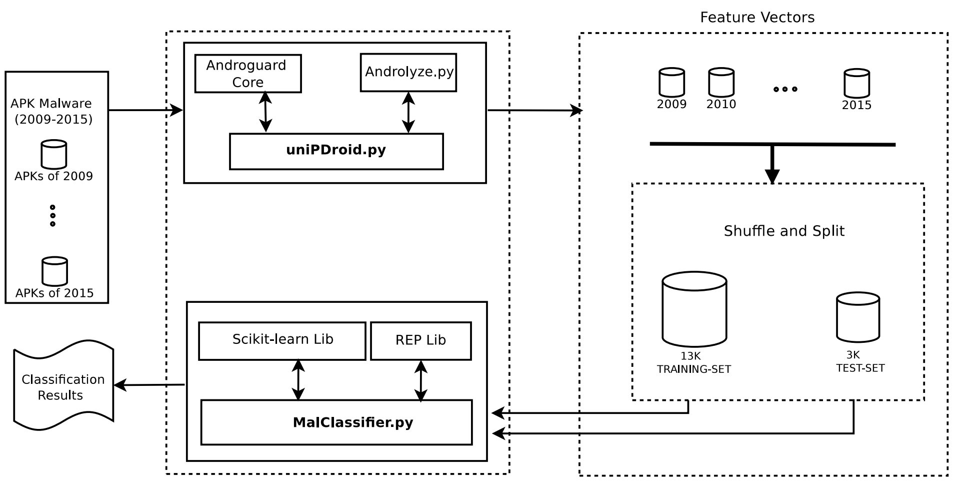


Figure 5.1: Framework of proposed classification methodology

### 5.1.1 Dataset Collection and Pre-processing

In this subsection, we provide further detail on the composition of our experimental dataset. In order to conduct an extensive analysis, we collected a set of well-labeled Android *malicious* applications. The dataset used in our evaluation is composed of 15,884 malicious applications collected from the following existing work in the literature: [166,167,194,195], as we explained them in previous chapter. Table 5.1 shows the details of the dataset composition.

| **Repository** | **No. of samples** |
|---|---|
| Genome [194] | 1,260 |
| Drebin [195] | 5,560 |
| M0Droid [167] | 193 |
| VirusTotal [166] | 8,871 |
| **Total** | **15,884** |

Table 5.1: Dataset composition

To perform malware classification using supervised machine learning classification algorithm (for example, XGBoost classifier), we are required to provide a well-labelled dataset. To find the class label associated with each malware sample in our dataset, we wrote several scripts in `Bash` and `Python` programming languages. We submitted each malware sample to *Virustotal* [166] and made a query to get the malware family name, as shown in listings 5.1 and 5.2. Virustotal then returned an analysis report for the given file in the form of `JSON` object as depicted in Listing 3. We then parsed the `JSON` object and performed text processing to extract the related family names. The names were then used as class labels since there is no agreed-upon malware naming convention among antivirus (AV) companies.

In order to decide on the family name for each class label, we took into account the family names of top eight AV engines[5]. Leveraging these top eight AVs and

[5]http://www.av-comparatives.org/wp-content/uploads/2014/03/security_survey2014_en.pdf

based on majority voting role, we extracted the selected malware family names. The AVs that we exploited are among the top AV engines used on the Android platform and are namely: *MicroWorld-eScan, BitDefender, Kaspersky, Avira, AVG, Emsisoft, AVware,* and *F-Secure*. The reason for considering only these eight AVs is because (i) they are among top AV engines dedicated to the Android Platform; (ii) we observed that these AV outperform others in most cases, particularly when detecting malware; and (iii) we did not further complicate the text processing phase by increasing the number of AV engines.

```python
#imports
import simplejson
import urllib
import urllib2

url = "https://www.virustotal.com/vtapi/v2/file/report"
parameters = {"resource":APK-hasH-name,"apikey":apikey}
data = urllib.urlencode(parameters)
req = urllib2.Request(url, data)
response = urllib2.urlopen(req)
json-object = response.read()
print json-object
```

Listing 5.1: Example of a Python script for submitting malware samples to Virustotal

```
{"scans": {
"Kaspersky":{"detected":true,"version":"15.0","result":"Trojan-Spy.AndroidOS.Adrd.a",.},
"BitDefender":{"detected":true,"version":"7.2","result":"Android.Trojan.Adrd.A",..},
"Emsisoft":{"detected":true,"version":"3.5.642","result":"Android.Trojan.Adrd.A",.. },
"F-Secure":{"detected":true,"version":"11.0.190.45","result":"Trojan:Android/Adrd.A",..},
"Avira":{"detected":true,"version":"8.3.2.4","result":"ANDROID/Spy.Adrd.D.Gen",..},
.
.
"AVG":{"detected":true,"version":"16.0.0.4489","result":"Android/Adr",..},
"resource": "4de0d8997949265a4b5647bb9f9d42926bd88191", "total": 54, "positives": 38,
"md5": "77b0105632e309b48e66f7cdb4678e02",...}
```

Listing 5.2: Example of a JSON file produced by Virustotal

### 5.1.2 Feature Extraction

Android applications are written in `Java`, compiled to `Java` bytecode, and then converted into platform-specific Dalvik bytecode. This bytecode can be efficiently disassembled and provides us with useful information about features used in an application. We mainly extracted the features from bytecode and converted these features into binary feature vectors, which are made up of 560 features. Each feature vector is comprised of the features described in previous chapter, since we found these features effective and efficient while preforming classification tasks .

### 5.1.3 Feature Selection

To avoid from problems such as misleading the learning algorithm, over-fitting, and increasing model complexity, we benefited from feature selection algorithm explained in previous chapter to select the most important features for classification tasks. Figure 5.2 shows the most important features (101 binary features) that we used to train and evaluate our classification algorithms.

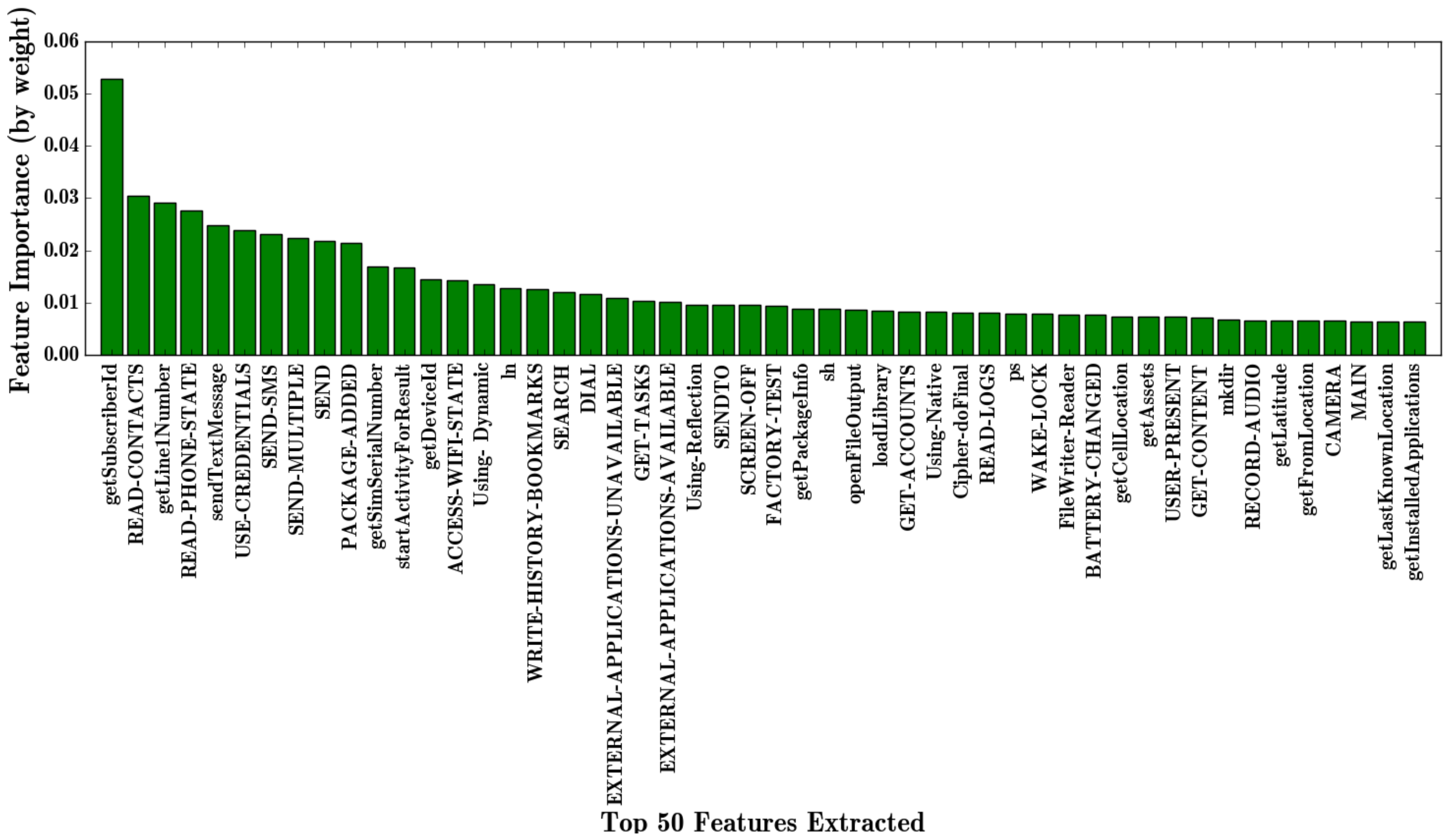


Figure 5.2: Key features extracted from our dataset

### 5.1.4 Classification Models

XGBoost [170] is the abbreviation for eXtreme Gradient Boosting. *Gradient* refers to the use of gradient descent, which can be used as a way to find a local minimum of a function and *Boosting* is a technique which consists of the fact that a set of weak learners is stronger than a single strong learner. XGboost algorithm uses a differentiable loss function to calculate the adjustments needed to be made to a consecutive successor learner in an iterative learning sequence. The algorithm can automatically do parallel computations with OpenMP and it is much faster than existing Gradient Boosting algorithm. Listing 5.1.4 provides an excerpt of the source code for XGBoost.

```
import numpy as np
import xgboost as xgb
from sklearn.metrics import  classification_report

def train():
  data_train = np.genfromtxt(open("train.csv","r"), delimiter=",")
  y_train = data_train[:,0]
  X_train = data_train[:,1:]
  xg_train = xgb.DMatrix(X_train, label=y_train)
  data_test = np.genfromtxt(open("test.csv","r"), delimiter=",")
  y_test = data_test[:,0]
  X_test = data_test[:,1:]
  xg_test = xgb.DMatrix(X_test, label=y_test)
  # setup parameters for xgboost
  param = {}
  param['objective'] = 'multi:softmax'
  param['eta'] = 0.1
  param['max_depth'] = 6
  param['num_class'] = 78 # Number of classes starting from 0
  watchlist = [ (xg_train,'train'), (xg_test, 'test') ]
  num_round =  260
  bst = xgb.train(param, xg_train, num_round, watchlist);
  # get prediction
  y_pred = bst.predict( xg_test );
  print classification_report(y_test, y_pred)

if __name__ == '__main__':
  train()
```

Listing 5.3: Example of code for the Machine Learning classifier, eXtreme Gradient Boosting

The different parts of the proposed classification methodology, explained in previous subsections, can be summarized in Figure 5.1. We extended the `Androguard` tool [127] and built `uniPDroid`, a static analysis tool written in `Python` programming language. Our proposed method uses this tool to extract several informative features representing characteristics of the application and leverages several `Python` ML libraries to build the best performing classifier, XGBoost, in order to perform classification task. In particular, the system consists of two modules: (i) Feature Extraction Module, and (ii) Machine Learning Classification Module. The feature extraction module includes three components. The `uniPDroid.py` is the main component within this module extracting informative features from an application while `Androguard` and `Androlyze.py` are auxiliary components providing support for performing feature extraction task. The ML classification module leverages several ML packages to perform classification. The main component within this module is the `MalClassifier.py`. The `Scikit-learn` and `REP` packages provide different classification algorithms and some helper functions for performance evaluation.

### 5.1.5 Evaluation metrics

Table 5.2 introduces the metrics that we considered in order to assess the performance of the ML classification algorithms in class imbalance problem, that is, the total number of a class of positive data is far less than the total number of negative data. The highest precision means that an algorithm returns substantially more relevant results than irrelevant ones, while the highest recall means an algorithm returns the most of the relevant results. The F1-score combines precision and recall: it is the harmonic mean of precision and recall. We elaborate further on our empirical results in next sections.

| Metric | Description | Formula |
|---|---|---|
| Precision | Measure of exactness or quality | $\frac{T_P}{T_P+F_P}$ |
| Recall | Measure of completeness or quantity | $\frac{T_P}{T_P+F_N}$ |
| F1-score | Harmonic mean of precision and recall | $2\times\frac{Precision\times Recall}{Precision+Recall}$ |

Table 5.2: Performance metrics

## 5.2 Malware Family-based Classification

In this experiment, we carried out family by family malware classification. To this end, we grouped 15, 884 Android malware in our repository into 204 different malware families. To perform an efficient and effective classification task and have sufficient samples to feed our proposed ML classification algorithm, we discard malware families that include less than 10 samples and consequently, ended up with 78 malware families. We shuffled and split the whole data points into training and testing sets, 80% and 20% respectively. We leveraged XGBoost classification algorithm to perform classification task over the 78 different malware families. Tables 5.3, 5.4, and 5.5 show the malware families used in our experiments as well as the infection risks associated with each malware family.[6] Before conducting classification task, in order to achieve a high accuracy in performance, we fine-tuned the hyperparameters of our classification algorithm through Grid-Search procedure combined with 5-fold Cross-Validation over the training set. Having the best parameters selected, we trained our classifier on the training set (13, 000 samples) and tested its performance against 3, 000 unseen samples in classifier point of views. Table 5.6 shows the classification results (F1-score) for each malware family and Table 5.7 illustrates the overall accuracy measures in terms of Precision, Recall, and F1-score over the 78 malware families.

[6]We would have to remind that the major criteria that are used to classify malware are *propagation*, *harm done*, and *resiliency*. Different kind of malware apps do have certain propagation mechanism, do certain type of harm on the system and use specific techniques to stay resilient. Different malware families might have the same risk infection and harm the system in a similar manner but they differ in terms of propagation mechanism and resiliency techniques. that is way in the following tables malware families with the same infection risk are classified differently (in different groups).

| Family Name | Infection Risks |
|---|---|
| AdFlex | An advertisement library may compromise your personal information |
| ADRD | Steals private information |
| Adwo | An advertisement library may compromise your personal information |
| Agilebinary | A Spyware accessing the file system and retrieving app data |
| AirPush | A very aggressive Ad-Network and compromises your personal information |
| Andup | Steals personal information |
| AppQuanta | An advertisement library may compromise your personal information |
| Asroot | Uses Asroot root exploit |
| AutoSMS | Attempts to steal sensitive data by seizing incoming SMS messages and forwards them to a remote site |
| BaseBridge | Sends premium-rate SMS to predetermined numbers |
| Boxer | Sends SMS to premium-rated numbers |
| Cobbler | A monitoring tool and wipes the SD card's contents and everything stored on the device |
| DDLight | Collects information about the device and sends back to a remote server |
| Dianjin | An advertisement library which may compromise your personal information |
| Dianle | Interrupts the normal operations and gains access to private information |
| Dougalek | Steals personal information and uploads these data to a remote server |
| Downloader | Gains root access and downloads additional malicious apps |
| DroidSheep | Captures and hijacks unencrypted web sessions |
| Dropper | Interrupts the normal operations and gains access to private information |
| Ewalls | Steals information from the mobile device |
| Exploid | Exploits vulnerabilities to gain root privileges on devices |
| FakeApp | Downloads configuration files to display advertisements and collects information from the compromised device |
| FakeBank | Opens a back door and steals information from the compromised device |
| FakeDoc | Installs additional applications |
| FakeInstall | Pretends to be an installer for legitimate app, sends premium-rate SMS |
| FakeTimer | Sends personal information to a remote server and opens pornographic websites |

Table 5.3: Infection risks associated with each malware family

| Family Name | Infection Risks |
|---|---|
| Feejar | Sends SMS to premium-rated numbers |
| Geinimi | Opens a back door and transmits private information |
| Gepew | Attempts to replace installed apps with trojanized versions |
| GingerBreak | A root exploit for Android 2.2 and 2.3 |
| GingerMaster | Utilizes a Root Exploit and provides root-level access |
| GoldDream | Steals information from Android devices |
| GoneSixty | Steals private information |
| Hamob | An advertisement library may compromise your personal information |
| HiddenAds | Does not have an icon and runs in a stealth mode and displays various advertising messages |
| Igexin | An advertisement library may compromise your personal information |
| InfoStealer | Secretly collects and uploads sensitive information |
| JSmsHider | Opens a backdoor and sends information to a specific URL |
| Kmin | Attempts to send data to a remote server |
| Kuguo | An advertisement library may compromise your personal information |
| KungFu | Forwards confidential information to a remote server |
| LeadBolt | An advertisement library may compromise your personal information |
| Lovetrap | Sends SMS to premium-rated numbers and steals information |
| Mecor | Monitors and compromises your personal information |
| Metasploit | Exploits vulnerabilities to gain root privileges on devices |
| Minimob | Compromise personal information and distributes via spam email |
| Mobclick | Aggressively pushes unwanted ads and steals personal information |
| MobileTX | Steals information from the compromised device and may send SMS to a premium-rate number |
| Mseg | Steals private data and secretly send SMS to premium-rated numbers |
| MTK | Interrupts the normal operations and gains access to the private information |
| Mulad | Generates income by injecting ads into legitimate free apps |
| NickiSpy | Gathers information from infected user's smartphone and uploads the data to a specific URL |

Table 5.4: Infection risks associated with each malware family (continued)

| Family Name | Infection Risks |
|---|---|
| NoiconAds | Compromises personal information |
| Pentr | A Spyware and hack-tool enables penetration testing |
| RuFraud | Sends SMS to premium rated numbers |
| SecApk | An advertisement library that compromises your personal information |
| SLocker | Encrypts images, documents and videos in the SD Card to later ask for a ransom to decrypt the files |
| SMSKey | Interrupts the normal operations and gains access to the private information |
| SmsPay | Mimics a legitimate app and requires an activation fee through SMS |
| SMSReg | Registers the infected user to non-free services |
| SMSSend | Reaps profit by silently sending SMS to premium-rate numbers |
| SmsSpy | Attempts to steal sensitive data by seizing incoming SMS and forwards them to a remote site |
| SMSZombie | Exploits a vulnerability in the mobile payment system used by China Mobile |
| SndApps | Compromises your personal information |
| SpyHasb | Monitors phone calls, SMS, and GPS locations |
| SpyPhone | Steals personal data |
| Steek | A fraudulent app advertising an online income solution and steals privacy related information and sends SMS |
| Tekwon | Interrupts the normal operations and gains access to the private information |
| Utchi | An advertisement library may compromise your personal information |
| Vdloader | Steals personal information |
| Viser | Opens back door by use of the system loopholes to introduce some adware, browser extensions, spyware or ransomware |
| Wallap | Promises access to a wide collection of wallpapers and uses ads libraries to generate revenue |
| Waps | An advertisement library may compromise your personal information |
| Wapz | An advertisement library may compromise your personal information |
| Youmi | An advertisement library may compromise your personal information |
| YZHCSMS | Sends SMS to a premium-rate number |
| Zdtad | An advertisement library may compromise your personal information |
| Zsone | Sends SMS to premium rated numbers |

Table 5.5: Infection risks associated with each malware family (continued)

| Family Name | Samples | Year Developed | Percentage of apps | Classification F1-score (%) |
|---|---|---|---|---|
| AdFlex | 68 | 2013 | 0.40 | 86 |
| ADRD | 59 | 2010 | 0.37 | 100 |
| Adwo | 388 | 2011 | 2.4 | 83 |
| Agilebinary | 10 | 2010 | 0.06 | 100 |
| AirPush | 787 | 2010 | 4.9 | 93 |
| Andup | 18 | 2013 | 0.11 | 100 |
| AppQuanta | 39 | 2013 | 0.24 | 100 |
| Asroot | 12 | 2009 | 0.07 | 100 |
| AutoSMS | 46 | 2013 | 0.28 | 75 |
| BaseBridge | 608 | 2010 | 3.8 | 97 |
| Boxer | 21 | 2010 | 0.13 | 77 |
| Cobbler | 15 | 2011 | 0.09 | 100 |
| DDLight | 124 | 2011 | 0.78 | 100 |
| Dianjin | 91 | 2012 | 0.57 | 91 |
| Dianle | 54 | 2012 | 0.33 | 77 |
| Dougalek | 22 | 2012 | 0.13 | 93 |
| Downloader | 75 | 2012 | 0.47 | 83 |
| DroidSheep | 11 | 2011 | 0.06 | 100 |
| Dropper | 123 | 2014 | 0.77 | 95 |
| Ewalls | 43 | 2009 | 0.27 | 100 |
| Exploit | 41 | 2010 | 0.25 | 100 |
| FakeApp | 104 | 2011 | 0.65 | 80 |
| FakeBank | 84 | 2014 | 0.52 | 96 |
| FakeDoc | 130 | 2011 | 0.81 | 100 |
| FakeInstall | 1729 | 2011 | 10.8 | 98 |
| FakeTimer | 21 | 2012 | 0.13 | 100 |
| Feejar | 12 | 2014 | 0.07 | 50 |
| Geinimi | 152 | 2010 | 0.95 | 100 |
| Gepew | 13 | 2014 | 0.08 | 100 |
| GingerBreak | 14 | 2011 | 0.08 | 67 |
| GingerMaster | 489 | 2011 | 3 | 90 |
| GoldDream | 126 | 2011 | 0.8 | 77 |
| GoneSixty | 15 | 2011 | 0.09 | 100 |
| Hamob | 35 | 2012 | 0.22 | 80 |
| HiddenAds | 44 | 2014 | 0.28 | 91 |
| Igexin | 42 | 2011 | 0.26 | 91 |
| InfoStealer | 209 | 2010 | 1.3 | 91 |
| JSmsHider | 11 | 2009 | 0.07 | 100 |
| Kmin | 187 | 2010 | 1.1 | 99 |
| Kuguo | 84 | 2012 | 0.52 | 50 |
| KungFu | 1051 | 2011 | 6.6 | 98 |
| LeadBolt | 178 | 2011 | 1.1 | 77 |
| Lovetrap | 11 | 2010 | 0.07 | 100 |
| Mecor | 10 | 2015 | 0.06 | 100 |
| Metasploit | 23 | 2014 | 0.14 | 100 |
| Minimob | 14 | 2013 | 0.09 | 40 |
| Mobclick | 101 | 2010 | 0.63 | 71 |
| MobileTX | 69 | 2011 | 0.43 | 100 |
| Mseg | 20 | 2011 | 0.12 | 67 |
| MTK | 97 | 2013 | 0.61 | 100 |
| Mulad | 1008 | 2012 | 6.3 | 99 |
| NickiSpy | 11 | 2010 | 0.07 | 100 |
| NoiconAds | 882 | 2014 | 5.5 | 99 |
| Pentr | 13 | 2011 | 0.08 | 67 |
| RuFraud | 21 | 2011 | 0.13 | 93 |
| SecApk | 59 | 2012 | 0.37 | 50 |
| SLocker | 22 | 2014 | 0.13 | 100 |
| SMSKey | 34 | 2011 | 0.21 | 100 |
| SmsPay | 1331 | 2010 | 8.4 | 88 |
| SMSReg | 1916 | 2010 | 12.3 | 88 |
| SMSSend | 487 | 2010 | 3 | 84 |
| SMSSpy | 207 | 2010 | 1.3 | 89 |
| SMSZombie | 18 | 2012 | 0.11 | 100 |
| SndApps | 23 | 2011 | 0.14 | 100 |
| SpyHasb | 13 | 2010 | 0.08 | 100 |
| SpyPhone | 23 | 2010 | 0.14 | 91 |
| Steek | 28 | 2011 | 0.17 | 91 |
| Tekwon | 16 | 2013 | 0.10 | 86 |
| Utchi | 26 | 2012 | 0.16 | 100 |
| Vdloader | 17 | 2012 | 0.10 | 77 |
| Viser | 36 | 2012 | 0.22 | 100 |
| Wallap | 88 | 2012 | 0.55 | 92 |
| Waps | 570 | 2011 | 3.5 | 78 |
| Wapz | 231 | 2012 | 1.5 | 75 |
| Youmi | 588 | 2010 | 3.7 | 82 |
| YZHCSMS | 59 | 2010 | 0.37 | 100 |
| Zdtad | 396 | 2015 | 2.5 | 99 |
| Zsone | 31 | 2011 | 0.19 | 86 |
|  |  |  |  |  |
|  |  |  |  |  |

Table 5.6: The number of malware samples, year developed and classification results of 78 malware families from our experimental dataset

Table 5.6 shows the results of classification per malware family, number of samples, the year that those samples have been developed, and the percentage of malware families represented in our dataset. According to the table, the malware families such as `SMSReg`, `FakeInstall`, `SMSPay`, `Kungfu`, and `Mulad` have the biggest share of malware samples in the entire dataset, 12.3%, 10.8%, 8.4%, 6.6%, and 6.3% respectively.

The worst classification results, 40%, belongs to `Minimob` family with 14 samples. It is obvious that by increasing the number of samples in the training set, our proposed ML classification algorithm will be expected to perform the training procedure better. It can be noted in Table 6, as the size of the training set for each malware family increases (that is, number of samples in each family), the accuracy (F1-score) gets better. In other words, with a few amount of samples it is not reasonable to expect to achieve good prediction accuracies from the classification algorithm.

Additionally, the average accuracy in terms of precision, recall, and F1-score for all 78 malware families are reported in Table 5.7. We conducted a 10-fold Cross-Validation experiment to compute Mean Error Rate for both training and testing sets. Table 5.8 shows the results obtained from this experiment. For the 10-fold cross-validation, the data is randomly partitioned into 10 equal size subsamples. Of the 10 subsamples, a single subsample is retained as the validation data for testing the model ($Test_{cv}$), and the remaining 9 subsamples are used as training data ($Train_{cv}$). The process is then repeated 10 times, with each of the 10 subsamples used exactly once as the validation data. The 10 results from the folds can then be averaged to produce a single estimation.

| | Precision | Recall | F1-score | Support |
|---|---|---|---|---|
| Avg / Total | 92 | 92 | 92 | 3000 |

Table 5.7: Classification Report (%) for test set (unseen samples)

| | $Train_{cv}$ Mean Error Rate | $Test_{cv}$ Mean Error Rate |
|---|---|---|
| CV 10-Fold | 0.033359 (+/- 0.000760) | 0.091460 (+/- 0.007298) |

Table 5.8: Cross-Validation result for training set

Comparing the F1-score, as shown in Table 5.7, which has been obtained from evaluating our proposed classifier against unseen samples with $Test_{cv}$, Mean Error rate and the prediction from Cross-Validation (which is equal to 92% accuracy), we can draw this conclusion that our ML classification algorithm is never over-fitted and is able to predict unseen samples with high accuracy rate.

## 5.3 Cumulative Classification

In this experiment, we accumulated Android malware apps and carried out cumulative classification where the classification results are continuously updated as new malware samples are discovered. The number of malware used in our experiment is 15, 884 samples. Figure 5.3 depicts the number of malware collected by month within the period 2009 and 2015 and Figure 5.4 shows the cumulative graph of the malware apps collected each month for that same period. In our cumulative classification, we used 56 different malware groups.

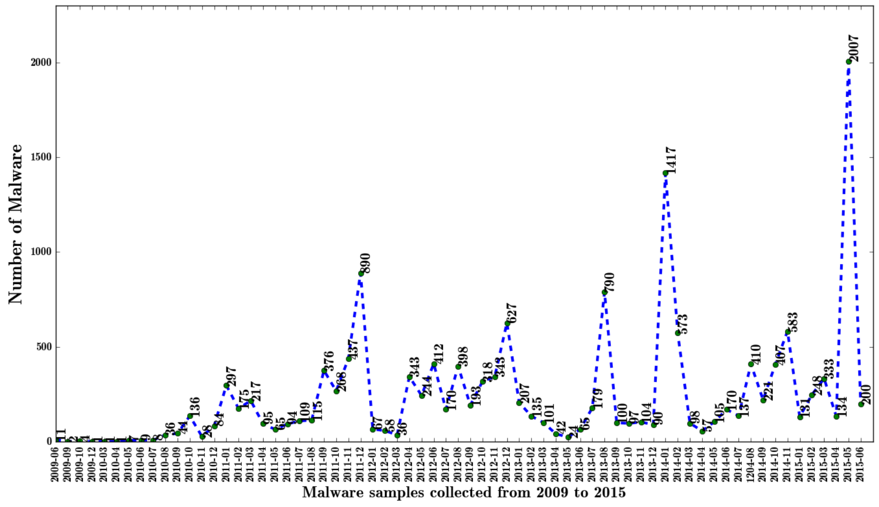


Figure 5.3: Malware number per month

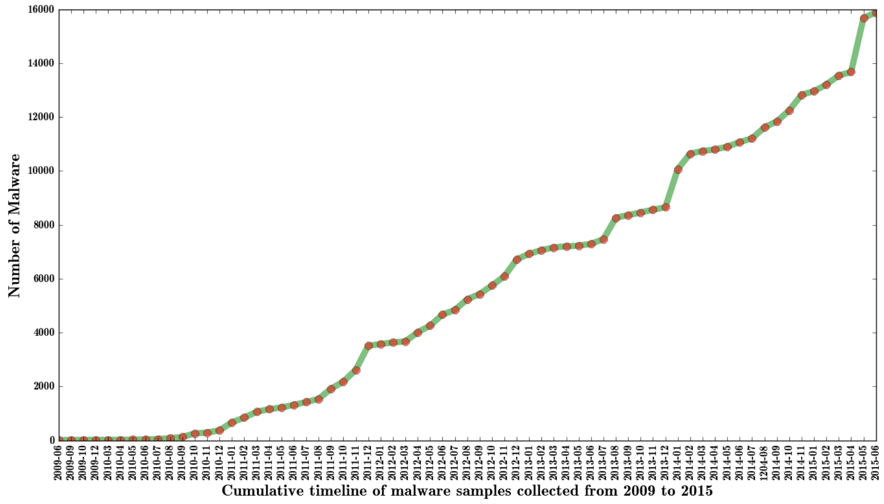


Figure 5.4: Malware number per month

To generate the first malware group, $MG_1$, we take the malware apps from June 2009 and September 2010 which comprises of 124 samples in order to have an initial set of samples enough to perform classification. The second data group, $MG_2$, contains the malware from June 2009 up to October 2010; this is achieved by adding malware belonging to upcoming month to previous months to generate the next malware group). For $MG_3$, we take the malware from June 2009 up to

November 2010. The process is repeated until all the malware in the dataset are incorporated into the malware groups. Finally, we ended up with having 56 groups, $MG_1$,..., $MG_{56}$ altogether, as shown in Figure 5.4.

We trained the classification algorithm, XGBoost, on each malware group, $MG_i$ and tested its performance against malware belonging to upcoming months. We should take this point into account that malware belonging to the next month is unseen for the classification algorithm. We computed accuracy measures in terms of Precision, Recall, and F1-score, Figure 5.5. The aim was to investigate how features of old malware samples can be of help to classify new variant of both known and unknown malware families.

We performed cumulative classification to investigate how well the old malware can help us to detect new malware. In the other word, how old malware can contribute to detecting new variant of both known and unknown malware families. As for the accuracy measures obtained from cumulative classification, Figure 5.5, at some points (e.g., February 2015) accuracy measures drops. The reason for such a decrease in classifier performance is that we have trained the ML algorithm in certain time on data-points belonging to past up to that time and we evaluate its performance against future data-points. In the testing dataset, there exist some samples which are considered as zero-day malware in the wild (that is, recently developed malware). The ML classification algorithm has not been trained on such samples and has no idea about these malware samples which have completely different patterns in terms of features. Consequently, the classifier cannot predict the correct label of these samples based on its past experience. As it can be seen, in next round of cumulative classification by adding the old samples and enriching the training set we let the classifier learn more about past data and as a result the classifier might perform better during classification stage.

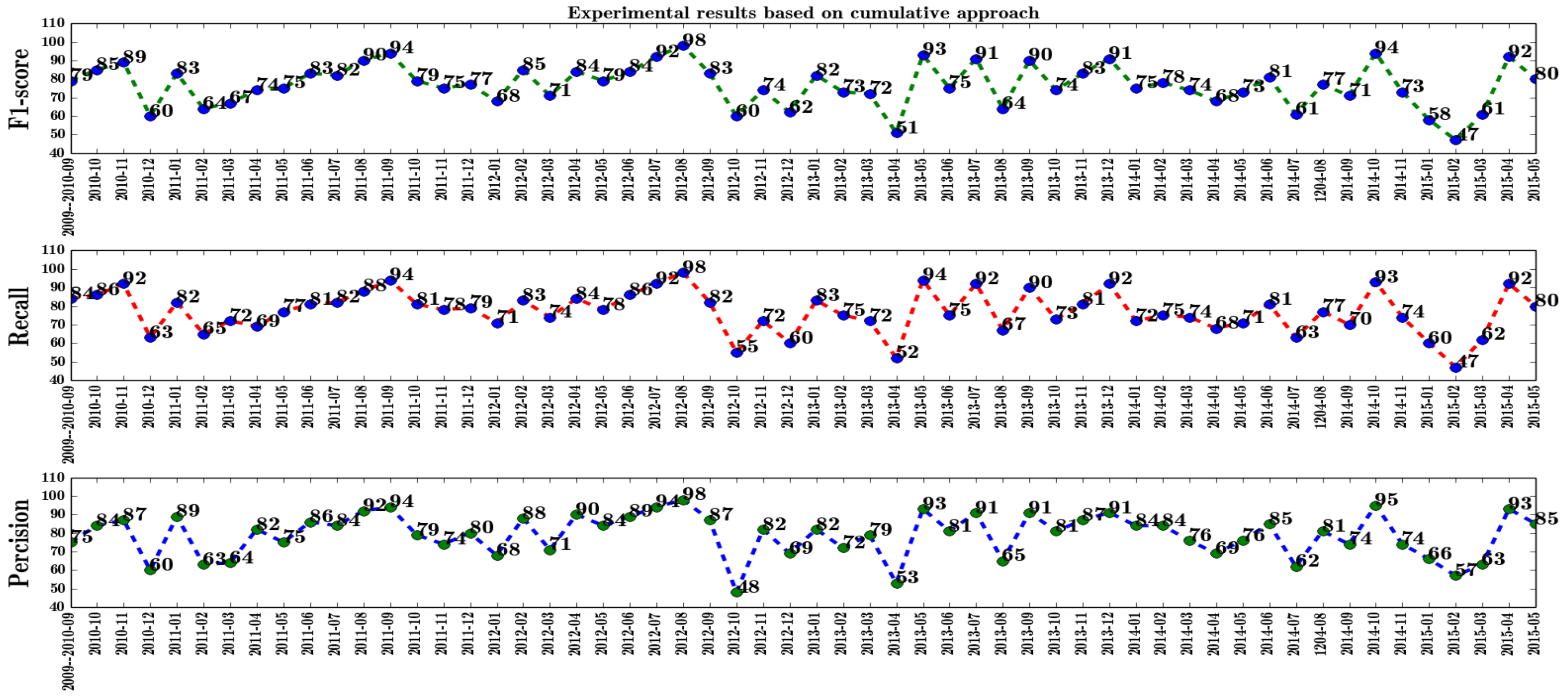


Figure 5.5: Accuracy measures

## 5.4 Related Work

Machine Learning (ML) techniques have been extensively used for detection and classification of malware on mobile devices [161, 186, 196]. In the remainder of this section, we present some of the existing work in the area of Android malware classification.

Sahs et al. [197] presented an ML-based framework for Android malware detection using Support Vector Machines (SVM) algorithms. The authors exploited a single-class SVM model derived from benign samples. They used the Android permissions in the Manifest files and CFGs of applications from the dataset. `Crowdroid` [198] collects behavioral-related data directly from users via crowdsourcing and evaluates the data with a clustering algorithm.

Shabtai et al. [199] proposed a new method for categorizing Android applications through ML techniques. To represent each application, their method extracts different feature sets including the frequency of occurrence of the printable strings, the different permissions of the application itself, and the permissions of the application extracted from the Android Market. Abela et al. [200] presented `AMDA`, an automated malware detection system for the Android platform. The authors extracted features such as system calls form benign and malware applications to provide baseline behavior datasets to feed machine learners. Test applications are then passed through the behavior-based module for identification of presence of malicious payloads. Similarly, `RobotDroid` [201] is a framework that detects smartphone malware based on SVM active learning algorithm. The authors in [202] designed an anomaly detection system that extracts the strings contained in application files in order to detect malware. Their proposed method is based on features that were extracted from string analysis of the application.

Martinelli et al. [203] proposed `CAMAS`, a framework for the analysis and classification of malicious Android applications, through pattern recognition on execution graphs. They extracted a subset of frequent subgraphs of system calls that are executed by most of the malware. The resulting vector of the subgraphs is given to a classifier that returns its decision in terms of whether or not a malware has been detected. `DroidAnalytics` [204] is a malware analytic system for malware collection, signature generation and association based on similarity scores by analyzing the low-level system at the application, class or method level.

The authors in [205] proposed another detection method for Android malware. In particular, they used only manifest files to detect malware. The proposed method extracts six types of information from manifest files such as Permission, Intent (action, priority and category), Process name and Number of redefined permission and then uses them to detect Android malware. `DroidMat` presented by Wu et al. [196], exploits permissions, intents, inter-component communication, and API calls to distinguish malicious apps from benign ones. The detection performance was evaluated on a dataset of $1,500$ benign and 238 malicious applications and compared with the `Androguard` risk ranking tool, with respect to detection metrics such as accuracy rate.

The work in [162] presented a machine learning approach including SVM, Decision Trees (DT), and Bagging predictor to detect malicious Android applications. They trained and tested a classifier by using extracted permissions and API calls as features to identify whether an application is potentially malicious or not. Koundel et al. [206] designed a Naive Bayes classifier to classify applications using various attributes of an application, such as the permissions used by an application, battery usage and rating acquired by the application on Android market. `MAMA` [207] presents Manifest analysis for malware detection in Android. It extracts several features from the Android Manifest of the applications to build machine-learning classifiers such as K-Nearest Neighbors, DT, SVM and Bayesian networks.

The literature presented in this section provides an overview of the existing work in the field of Android malware versus cleanware detection and ML-based classification methodologies. In our work, we focused solely on Android malware, proposing a novel ML-based methodology that can efficiently, and with high accuracy, assign malware samples to their correct family names. We argue that it is not only important to detect malicious applications, but also to label them correctly as malware authors often repackage existing malware. Hence, re-detecting these repackaged samples becomes easier if the correct family names are used. Additionally, we analyzed the robustness of our extracted features used by the proposed methodology by performing a cumulative malware classification. We verified how efficient are features extracted from old malware samples in terms of detecting and classifying newly discovered malware.

## 5.5 Summary

In this chapter, we proposed an ML-based malware detection and classification methodology together with the application of static analysis on an extensive dataset of Android applications. To this end, we designed a tool, **uniPDroid**, to extract as many informative features as possible from our dataset. We considered mainly features from the Dalvik bytecode. The features extracted were converted into feature vectors, each containing 560 binary features. We then applied feature selection on the aforementioned extracted features, which led to the selection of 101 informative binary features suitable to feed our proposed classification methodology.

Moreover, we performed an extensive Grid-search analysis along with a 10-fold Cross-validation to tune the hyper-parameters of the classification algorithm to maximize the prediction accuracy. We performed Family-by-Family classification and obtained an average accuracy score of 92% in classification of unseen malware. In addition to this, we conducted a cumulative classification in order to investigate how well old malware can contribute to the detection of new variants of both known and unknown (zero-day) malware. We achieved reasonable accuracy rate, hence proving the robustness of the features extracted.

# Part II

# Security Analysis on Wearable Fitness Devices

# Chapter 6

# Security Analysis, Reverse Engineering and Spoofing Popular Fitness Devices

There have been several research work in the literature analyzing security and privacy of wearables. Researchers in some of these work attacked to data integrity and user privacy through the analyzing Bluetooth or HTTP(S) communications [208], [209], [28], and [210]. In other papers, they exploited vulnerabilities in Firmware update process to either manipulate the Firmware or inject crafted version of the Firmware into wearable, [211], and [212]. Since the data collected by fitness trackers are used by third party service providers for different type of data analytics, the integrity of the data must be protected such that a malicious user or third-party attacker cannot manipulate the data. Hence, we analyzed the security of a set of fitness trackers that use coding and data integrity check mechanisms and are known to be more secure than their counterparts in the market. We monitored and analyzed communications between the fitness devices and associated back-end services in the cloud and investigated the feasibility of tampering with data collected by the health trackers before it is uploaded to the cloud. We conducted false data injection attacks into remote servers for these wearables. To the best of our knowledge, false data injection attack for wearables employing proprietary coding and data integrity check mechanisms has not been done previously in the literature.

**Contribution**: We conduct an in-depth security analysis of some of the most popular Fitness trackers in the market. We reveal serious security-related vulnerabilities in these devices which can be exploited once identified. Specifically, we analysis the primitives governing the communication between trackers and cloud-based services. We show that designing and deploying security controls such as *end-to-end encryption*, *data integrity check* ,and *digital signature* in a robust and concrete way are overlooked by wearable manufactures. We document successful injection of fabricated data (along with Proof-of-Concept attack) and demonstrate malicious users can inject spoofed activity records to obtain personal benefits.

| **Tracker Name** | **App Name** | **Sync.** | Steps Count | Calories Burned | Heart-rate | Distance | Elevation/Stairs | Sleep Time | Average Price $ |
|---|---|---|---|---|---|---|---|---|---|
| Garmin Vivosmart HR | Garmin Connect | BLE | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 150 |
| Garmin Vivofit2 | Garmin Connect | BLE | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | 100 |
| Garmin Vivofit | Garmin Connect | BLE | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | 55 |
| Polar Electro Loop | Polar Flow | BLE | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | 50 |
| ViFit MEDISANA | VitaDock+ | USB | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | 34 |
| Xiaomi MiBand | Mi Fit | BLE | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | 16 |
| Jawbone UP3 | JAWBONE UP3 | BLE | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | 180 |
| Jawbone Move UP | JAWBONE UP | BLE | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | 50 |
| Misfit Shine | Misfit | BLE | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | 25 |
| Mio Link | Mio GO | BLE | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | 100 |
| Withings Pulse | Health Mate | BLE | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 80 |
| Runtastic Orbit | Runtastic Me | BLE | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | 80 |
| Sony Smartband 2 | SmartBand 2 | BLE/NFC | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | 130 |
| Razor Nabu X | Nabu | BLE | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | 50 |
| Technaxx 39 | My Fitness | BLE | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | 40 |
| Technaxx 37 | My Fitness | BLE | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | 35 |
| Oregon Dynamo 2+ | Ssmart Fit | BLE | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | 130 |

Table 6.1: Fitness trackers used in our experiments.

The rest of this chapter is organized as follows. Section 6.1 explains our system model. Also, we give a description of the adversary model and requirement analysis. Section 6.2 describes some Background preliminaries. In Section 6.3, we describe evaluation and experimental setup and elaborate our findings and technical details. We mention countermeasures and remedies in Section 6.4. Section 6.6 illustrates Related work. Finally, we conclude the chapter in Section 6.7.

**Responsible Disclosure**: we contacted each fitness tracker company in advance of publishing our work. In each case, we informed the respective company about security vulnerabilities that we discovered in their products. We disclosed the security vulnerabilities to them to fix the identified problems before we publish our findings and we received a positive response from some of these companies.

## 6.1 System Model

In this section, we consider attacks targeting the communication between the smartphone and the manufacturer's cloud service. The attack aims to either (a) *disclose fitness data*, or (b) *tamper with the data* in order to inject counterfeit data into the cloud service. Other attacks such as Bluetooth attacks have already in the literature and are out of our scope. The fitness trackers we use in our experiments fall into two categories: (i) some of them have cloud services offered by device manufacturers and (ii) some others have no cloud services and process the fitness data locally on the smartphone. We concentrate on the former category and investigate how the data is transmitted over the Internet, how security practices are employed to safeguard fitness information.

### 6.1.1 Analyzed Devices

We test 17 fitness trackers shown in Table 6.1. Some of these device do not transmit user fitness data to the cloud, others transmit every logged fitness event over the Internet. We select our devices as a representative subset of the fitness trackers. The set included devices from less known manufacturers as well as devices from the most popular brands from top vendors.

### 6.1.2 Adversary Model

We consider an adversarial setting in which entities like mobile devices and users are untrusted but the cloud is considered to be trustworthy. We assume an active adversary model that adversary (malicious user) can view and manipulate all the data uploaded to the server. The user is considered to have full control over his tracker and smartphone and being motivated by possible financial gain. This adversary has both means and motives to try to attack the system. This control allows the adversary to eavesdrop on communications between tracker and the remote server. Figure 7.1 shows the adversarial setting.

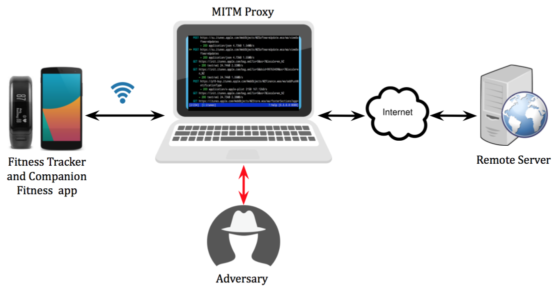


Figure 6.1: Adversary Model.

### 6.1.3 Requirement Analysis

We conduct a Man-In-The-Middle (MITM) attack that targets the communication between the tracker's associated fitness app (installed on the smartphone) and the cloud service, as fitness trackers typically utilize the user's smartphone to upload data to the cloud service.

We analyze several requirements to handle critical data collected by fitness trackers such as: i) Data Confidentiality, 2) Data Integrity, and 3) Data Authenticity. In our experiments, we take several criteria into account to investigate the safety and robustness of fitness trackers' communication protocol (between fitness app and cloud service) such as (i) *Use of End-to-End data encryption*, (ii) *Data In-Transit Encryption (e.g., HTTPS protocol)*, (iii), *Data At Rest Encryption (e.g., Encrypted Data-base)*, (iv) *Existence of Proprietary Encoding*, (v) *Presence of Data Integrity Check Mechanisms*, and (vi) *Use of SSL Certificate Pinning*.

## 6.2 Background Preliminaries

In this preliminary section, we will give definitions, descriptions, concerning the communication requirement analysis described in Section 6.1.3.

- End-to-end encryption (E2EE): In E2EE, the data is encrypted on the sender's device and only the recipient is able to decrypt it. Nobody in between (e.g., an Internet Service Provider or a hacker) can read the data or tamper with it. The cryptographic keys used to perform encryption and decryption are stored exclusively on the endpoints.

- Data In-Transit Encryption: Data in-transit is data actively moving from one device to another such as across the internet or through a private network. Data protection in transit is the protection of this data while traveling using HTTPS protocol. HTTPS typically use one of two secure protocols to encrypt communications - Secure Sockets Layer (SSL) or Transport Layer Security (TLS). Both the TLS and SSL protocols use an *asymmetric* Public Key Infrastructure (PKI) system. An asymmetric system uses two keys to encrypt communications, a public key and a private key. Anything encrypted with the public key can only be decrypted by the private key and vice-versa.

- Data At Rest Encryption: Data at rest is data that is not actively moving from device to device such as data stored on a hard drive. Data protection at rest is protection of data stored in databases, cloud, computer hard drives using robust and strong Encryption algorithms (e.g., AES).

- Data Encoding: Encoding is the process of changing data representation such as binary, hex, decimal, Base64 and so on. Encodings are not meant to conceal data, but may do so, if the encoding-decoding algorithm is secret until somebody reverse-engineers the algorithm.

- Data integrity check: Data integrity refers to the accuracy and consistency of data stored in a database or other construct. There are several data integrity assurance techniques, for instance, Checksumming and Cyclic Redundancy Check (CRC). Checksums can be computed for the data and can be stored persistently. Data integrity can be verified by comparing the stored and the newly computed values on the data read. Checksums are generated using a hash function such as MD5 [213], SHA1 [214], and HMAC [215]. A hashing algorithm computes a fixed size message digest. Keyed-Hash Message Authentication Code (HMAC) [19] is a specific type of a hashing function where the hash generated is cryptographically protected. It is currently one of the predominant means of ensuring that secure data is not corrupted in transit over insecure channels.
  Cyclic Redundancy Check (CRC): CRC check is an easily implemented technique to obtain data reliability in network transmissions. This technique is used to protect blocks of data. Using this technique, the transmitter appends an extra N-bit sequence to every data block and holds redundant information about the data. This redundancy helps the transmitter detect errors data transmitted.

- Certificate Pinning: In this technique, the details of server certificate are hardcoded in the client device (or in the companion application). The client will check whether the certificate received from the server matches the pre-configured certificate. If a different certificate is used the application will normally refuse to establish the connection and will report an error to the user.

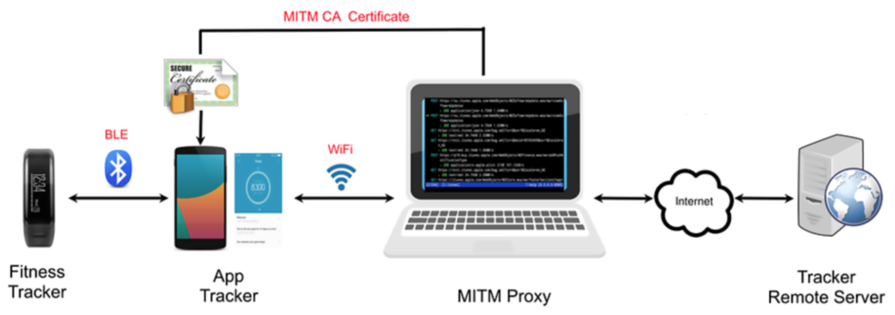


Figure 6.2: Schematic of experimental setup.

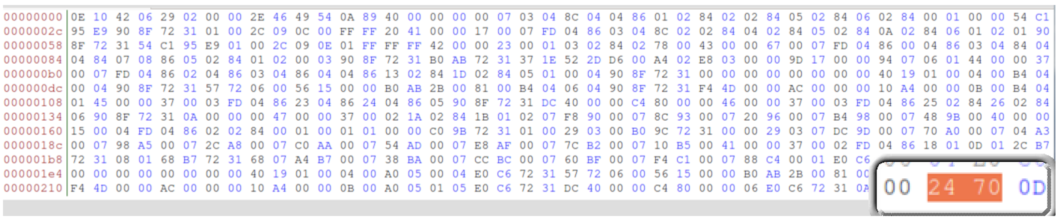
## 6.3 Evaluation

In this section, we evaluate dedicated wearable devices and implement attack against these devices and explain our technical details and findings.

### 6.3.1 Experimental Setup

In our set-up, shown in Figure. 6.2, we connect the various fitness trackers to a LG Nexus 5 smartphone running Android 5.1.1, on which we had installed the corresponding companion applications for the various trackers from the Play Store. We simulate an attacker between the smartphone and the cloud by creating a WiFi hotspot on a Dell XPS Linux laptop. The wireless access point is created using the script *create_ap* [216]. We connect smartphone to WiFi hotspot. The laptop runs Man-In-The-Middle Proxy (mitmproxy) [217], allowing us to intercept all communications between the tracker and the server, as well as between the smartphone and the server.
We install a fake CA certificate on an Android phone and trigger tracker synchronization manually, using fitness trackers' companion application. The application synchronizes the tracker over Bluetooth LE and forwards data between the tracker and the server over the Wi-Fi connection over an HTTPS connection.

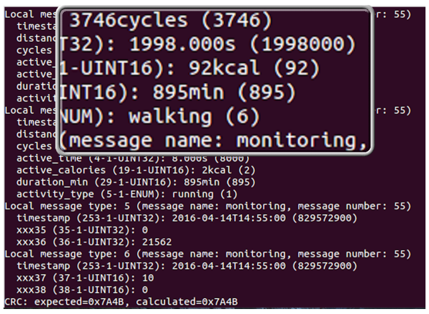
**Garmin Vivosmart HR, Vivofit2, and Vivofit**
The Garmin trackers transmit data trough the HTTPS protocol to Garmin server. The data is not encrypted and leverages FIT protocol to encode the data (Flexible and Inter-operable data Transfer protocol). This protocol is designed to be compact and extensible. Physical activity history (running, cycling, etc.) and user location are transferred to Garmin servers via FIT files. We sniff the HTTPS communications between the tracker and the Garmin server and dump the POST request issued by the tracker using this URL: `https://95.xxx.xxx.183/upload-service/upload/wellness`.



```
00000000 0E 10 42 06 29 02 00 00 2E 46 49 54 0A 89 40 00 00 00 00 07 03 04 8C 04 04 86 01 02 84 02 02 84 05 02 84 06 02 84 00 01 00 00 54 C1
0000002c 95 E9 90 8F 72 31 01 00 2C 09 0C 00 FF FF 20 41 00 00 17 00 07 FD 04 86 03 04 8C 02 02 84 04 02 84 05 02 84 0A 02 84 06 01 02 01 90
00000058 8F 72 31 54 C1 95 E9 01 00 2C 09 0E 01 FF FF FF 42 00 00 23 00 01 03 02 84 02 78 00 43 00 00 67 00 07 FD 04 86 00 04 86 03 04 84 04
00000084 04 84 07 08 86 05 02 84 01 02 00 03 90 8F 72 31 B0 AB 72 31 37 1E 52 2D D6 00 A4 02 E8 03 00 00 9D 17 00 00 94 07 06 01 44 00 00 37
000000b0 00 07 FD 04 86 02 04 86 03 04 86 04 04 86 13 02 84 1D 02 84 05 01 00 04 90 8F 72 31 00 00 00 00 00 00 00 00 40 19 01 00 04 00 B4 04
000000dc 00 04 90 8F 72 31 57 72 06 00 56 15 00 00 B0 AB 2B 00 81 00 B4 04 06 04 90 8F 72 31 F4 4D 00 00 AC 00 00 00 10 A4 00 00 0B 00 B4 04
00000108 01 45 00 00 37 00 03 FD 04 86 23 04 86 24 04 86 05 90 8F 72 31 DC 40 00 00 C4 80 00 00 46 00 00 37 00 03 FD 04 86 25 02 84 26 02 84
00000134 06 90 8F 72 31 0A 00 00 00 47 00 00 37 00 02 1A 02 84 1B 01 02 07 F8 90 00 07 8C 93 00 07 20 96 00 07 B4 98 00 07 48 9B 00 40 00 00
00000160 15 00 04 FD 04 86 02 02 84 00 01 00 01 01 00 00 C0 9B 72 31 01 00 29 03 00 B0 9C 72 31 00 00 29 03 07 DC 9D 00 07 70 A0 00 07 04 A3
0000018c 00 07 98 A5 00 07 2C A8 00 07 C0 AA 00 07 54 AD 00 07 E8 AF 00 07 7C B2 00 07 10 B5 00 41 00 00 37 00 02 FD 04 86 18 01 0D 01 2C B7
000001b8 72 31 08 01 68 B7 72 31 68 07 A4 B7 00 07 38 BA 00 07 CC BC 00 07 60 BF 00 07 F4 C1 00 07 88 C4 00 01 E0 C6
000001e4 00 00 00 00 00 00 00 00 40 19 01 00 04 00 A0 05 00 04 E0 C6 72 31 57 72 06 00 56 15 00 00 B0 AB 2B 00 81 00
00000210 F4 4D 00 00 AC 00 00 00 10 A4 00 00 0B 00 A0 05 01 05 E0 C6 72 31 DC 40 00 00 C4 80 00 00 06 E0 C6 72 31 0A
```

```
00 24 70 0D
```

Figure 6.3: Garmin fitness data including CRC bytes.

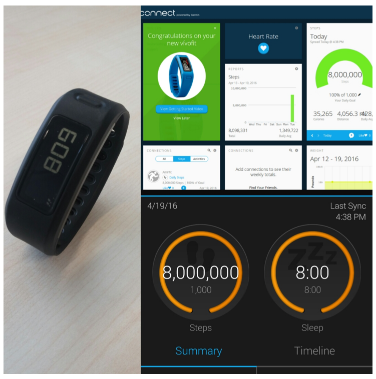


```
 3746cycles (3746)
T32): 1998.000s (1998000)
1-UINT16): 92kcal (92)
INT16): 895min (895)
NUM): walking (6)
(message name: monitoring,
```

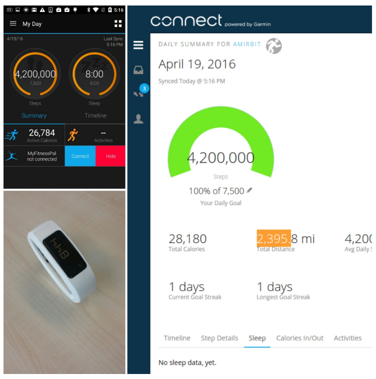
```
Local mes                                                r: 55)
  timesta
  distanc
  cycles
  active_
  active_
  duratio
  activit
Local mes                                                r: 55)
  timesta
  distanc
  cycles
  active_time (4-1-UINT32): 8.000s (8000)
  active_calories (19-1-UINT16): 2kcal (2)
  duration_min (29-1-UINT16): 895min (895)
  activity_type (5-1-ENUM): running (1)
Local message type: 5 (message name: monitoring, message number: 55)
  timestamp (253-1-UINT32): 2016-04-14T14:55:00 (829572900)
  xxx35 (35-1-UINT32): 0
  xxx36 (36-1-UINT32): 21562
Local message type: 6 (message name: monitoring, message number: 55)
  timestamp (253-1-UINT32): 2016-04-14T14:55:00 (829572900)
  xxx37 (37-1-UINT16): 10
  xxx38 (38-1-UINT16): 0
CRC: expected=0x7A4B, calculated=0x7A4B
```

Figure 6.4: Garmin detailed data after decoding.

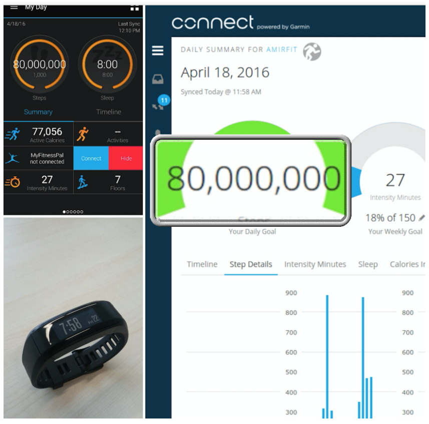
We develop a script that modifies FIT files, our crafted script leverages a *Perl* script named `fitsed` [218] to manipulate and update FIT files. Then, the script uploads new FIT file instead of the original file. Figure 6.3 shows encoded fitness data sent by the tracker to the Garmin server (the original FIT file is not human readable also, contains integrity check such as CRC-16). Figure 6.4 illustrates the detailed fitness data recorded in Garmin FIT file after decoding.



(a) Garmin Vivofit



(b) Garmin Vivofit 2



(c) Garmin Vivosmart HR

Figure 6.5: Injecting fabricated steps into Garmin remote servers.

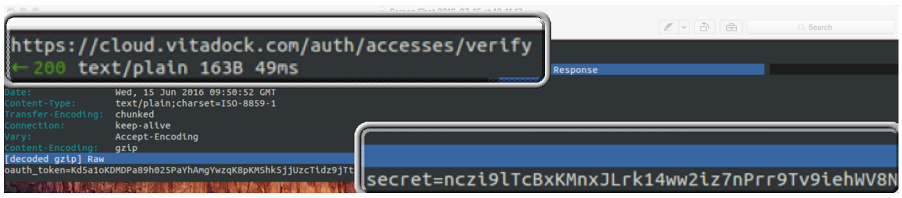
We successfully upload a counterfeit FIT file (indicating 80 million steps to Garmin's server). The server dose not raise any error and the web interface actually shows the counterfeit step count, as shown in Figure 6.5. The Garmin protocol is vulnerable to a motivated user generating false fitness data for his/her own account.

### ViFit MEDISANA

The Vifit tracker uses the HTTPS protocol to make in-transit communication secure. The fitness data is transmitted using a JSON file in plain text format. The payloads (including the fitness data) are protected using a HMAC scheme [215], data integrity check mechanism. HMAC computes a “signature" to verify that the data is actually being sent by the sender we expect also has not been altered. In ViFit protocol, the HMAC-SHA256 variant is used. In order to submit counterfeit data to the server, we compute a new HMAC signature on the modified data. To this end, we use the Base String, which is built by concatenating a number of fields separated by "&". The following list includes the required fields to compute HMAC signature and example data for each of those fields.

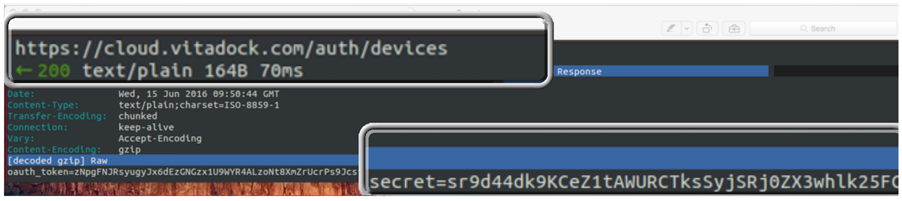
- HTTP_method = POST
- url = https://.../data/tracker/activity/array
- oauth_consumer_key: zNpgFNJRsyugyJx6dE...
- oauth_consumer_secret: sr9d44dk9KCeZ1tAW...
- oauth_token: K8eEFc0W3Pn5irDAv...
- oauth_token_secret: nczi9lTcBxKMnxJLrK1...
- oauth_nonce: k4VdSylUf4OCsOGlaa...
- oauth_timestamp: 1468750792
- Payload (JSON data):
  ["trackerActivityEntries":[ "calories":0.5,"**steps":1000000**,"distance":0.01,...,...]]

We obtain all required parameters from the POST requests, except for `oauth_consumer_secret` and `oauth_token_secret`. We discovered that the server sends these parameters to the tracker in different messages, as shown in (figures 6.6 and 6.7).



```
https://cloud.vitadock.com/auth/accesses/verify
← 200 text/plain 163B 49ms
Date:               Wed, 15 Jun 2016 09:50:52 GMT
Content-Type:       text/plain;charset=ISO-8859-1
Transfer-Encoding:  chunked
Connection:         keep-alive
Vary:               Accept-Encoding
Content-Encoding:   gzip
[decoded gzip] Raw
oauth_token=Kd5a1oKDMDPa89h025PaYhAmgYwzqK8pKMShk5jjUzcTidz9jTt
secret=nczi9lTcBxKMnxJLrk14ww2iz7nPrr9Tv9iehWV8N
```

Figure 6.6: Token secret required to compute HMAC over the data.



```
https://cloud.vitadock.com/auth/devices
← 200 text/plain 164B 70ms
Date:               Wed, 15 Jun 2016 09:50:44 GMT
Content-Type:       text/plain;charset=ISO-8859-1
Transfer-Encoding:  chunked
Connection:         keep-alive
Vary:               Accept-Encoding
Content-Encoding:   gzip
[decoded gzip] Raw
oauth_token=zNpgFNJRsyugyJx6dEzGNGzx1U9WYR4ALzoNt8XmZrUcrPs9Jcs
secret=sr9d44dk9KCeZ1tAWURCTksSyjSRj0ZX3whlk25FC
```

Figure 6.7: Application secret required to compute HMAC over the data.

After sniffing all HTTPS communications and extracting secret parameters, we recompute the HMAC signature over the tampered data and successfully forward the data along with new HMAC to the ViFit server. The ViFit server accepts our fake data, as show in Figure 6.8.

**Polar Electro Loop**:
The Polar Loop tracker uses the HTTPS protocol to communicate with polar servers. The fitness data is uploaded to the Polar server via Protocol buffer [219] which is a mechanism for serializing structured data, as shown in Figure 6.9. Each protocol buffer message is a small logical record of information, containing a series of key-value pairs. The fitness information is not encrypted. But the fitness data is encoded and not human readable and also it is not feasible to edit the fitness data using a text editor or a simple script.

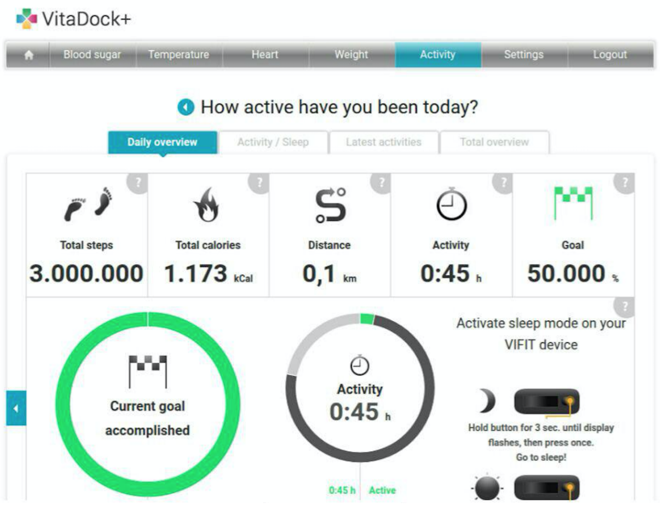
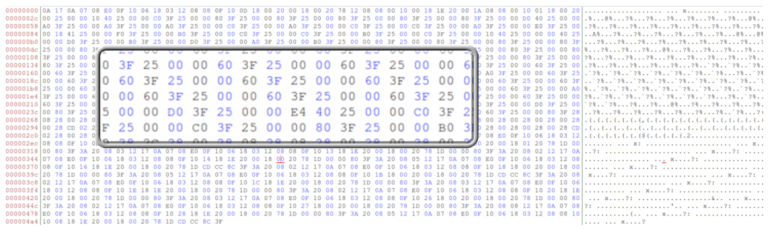
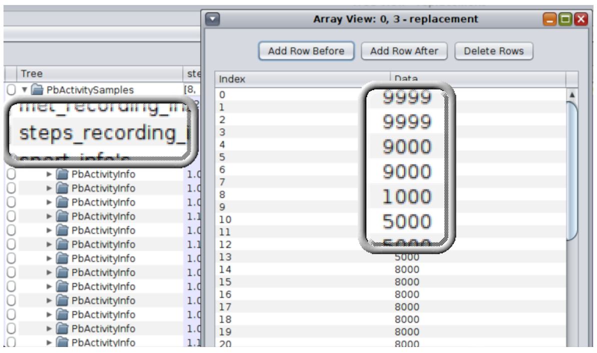


Figure 6.8: Injecting 3 million counterfeit steps into the ViFit server.



```
0 3F 25 00 00 60 3F 25 00 00 60 3F 25 00 00 6
0 60 3F 25 00 00 60 3F 25 00 00 60 3F 25 00 0
0 00 60 3F 25 00 00 60 3F 25 00 00 60 3F 25 0
5 00 00 D0 3F 25 00 00 E4 40 25 00 00 C0 3F 2
F 25 00 00 C0 3F 25 00 00 80 3F 25 00 00 B0 3
```

Figure 6.9: Data encoded using Protocol buffers.



```
9999
9999
9000
9000
1000
5000
```

Figure 6.10: Decoding data and tampering the step count.

Instead, we intercept the encoded fitness data, save it on disk, and then edit the message using the utility *Protocol Buffer Message Editor*, as it can be seen in Figure 6.10. After manipulating various fields (e.g., steps count), we successfully send the counterfeit data to the Polar server and server accepts the falsified data, as shown in Figure 6.11.

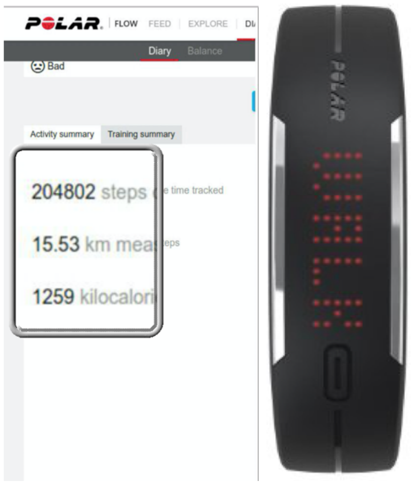
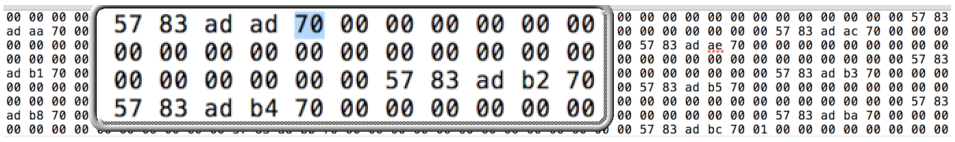


Figure 6.11: Injecting counterfeit data into the Polar remote server.

### Mio Link

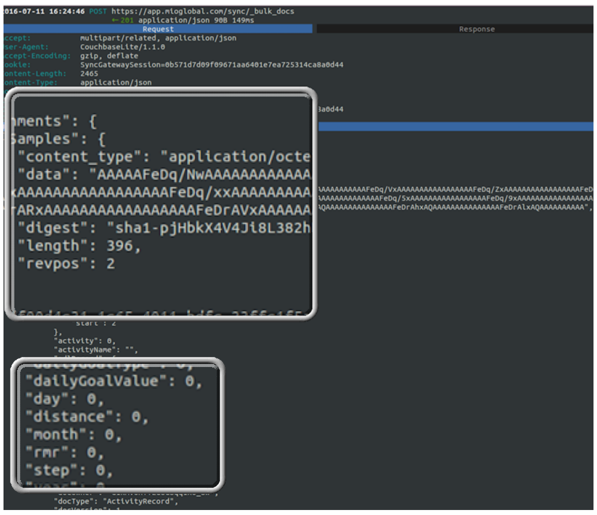
The Mio Link tracker records heart-rate data and sends it to the cloud via HTTPS for backup purposes. No web interface is available for the user for display purpose. We intercept the network traffic and capture heart-rate data. There is no encryption besides HTTPS. The data is encoded using *Base64*. The data containing hear-rate information is shown in Figure 6.12 after Base64 decoding.



```
00 00 00 00 | 57 83 ad ad 70 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 00 00 00 00 00 57 83
ad aa 70 00 |                                  | 00 00 00 00 00 00 00 57 83 ad ac 70 00 00 00
00 00 00 00 | 00 00 00 00 00 00 00 00 00 00 00 | 00 57 83 ad ae 70 00 00 00 00 00 00 00 00 00
00 00 00 00 |                                  | 00 00 00 00 00 00 00 00 00 00 00 00 00 57 83
ad b1 70 00 | 00 00 00 00 00 00 57 83 ad b2 70 | 00 00 00 00 00 00 00 57 83 ad b3 70 00 00 00
00 00 00 00 |                                  | 00 57 83 ad b5 70 00 00 00 00 00 00 00 00 00
00 00 00 00 | 57 83 ad b4 70 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 00 00 00 00 00 57 83
ad b8 70 00 |                                  | 00 00 00 00 00 00 00 57 83 ad ba 70 00 00 00
00 00 00 00 |                                  | 00 57 83 ad bc 70 01 00 00 00 00 00 00 00 00
```

Figure 6.12: The hear-rate data sent after base64 decoding.

As it can be seen in Figure 6.12, the hex string "`57 83 ad ad`" is decoded as a big-endian number representing a UNIX timestamp in `July 11, 2016` and `0x70` equals `112` in decimal (hear-rate value). A SHA-1 digest of the binary data is included in the request as data integrity check mechanism to protect data, as shown in Figure 6.13. However, the digest can be easily modified and crafted to reflect the counterfeit data. We easily manipulate the heart-rate data by recomputing massage digest, as shown in Figure 6.14.



```
2016-07-11 16:24:46 POST https://app.mioglobal.com/sync/_bulk_docs
                         ← 201 application/json 90B 149ms
                  Request                                    Response
ccept:            multipart/related, application/json
ser-Agent:        CouchbaseLite/1.1.0
ccept-Encoding:   gzip, deflate
ookie:            SyncGatewaySession=0b571d7d09f09671aa6401e7ea725314ca8a0d44
ontent-Length:    2465
ontent-Type:      application/json
```

```
ments": {
Samples": {
 "content_type": "application/octe
 "data": "AAAAAFeDq/NwAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAFeDq/xxAAAAAAAAAA
ARxAAAAAAAAAAAAAAAAAAFeDrAVxAAAAAA
 "digest": "sha1-pjHbkX4V4Ji8L382h
 "length": 396,
 "revpos": 2
```

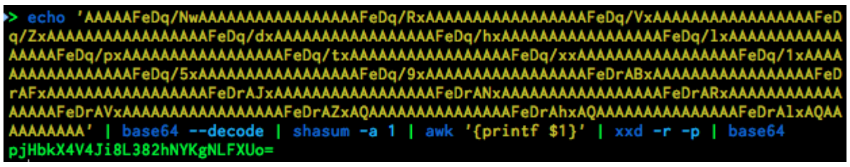
```
AAAAAAAAAAAFeDq/VxAAAAAAAAAAAAAAAAAAFeDq/ZxAAAAAAAAAAAAAAAAAAFeD
AAAAAAAAAAAAAFeDq/5xAAAAAAAAAAAAAAAAAAFeDq/9xAAAAAAAAAAAAAAAAA
AQAAAAAAAAAAAAAAAAFeDrAhxAQAAAAAAAAAAAAAAAFeDrAlxAQAAAAAAAAAA",
```

```
        },
        "activity": 0,
        "activityName": "",
```

```
"dailyGoalValue": 0,
"day": 0,
"distance": 0,
"month": 0,
"rmr": 0,
"step": 0,
```

```
        "docType": "ActivityRecord",
```

Figure 6.13: The data sent by the tracker to its web server.

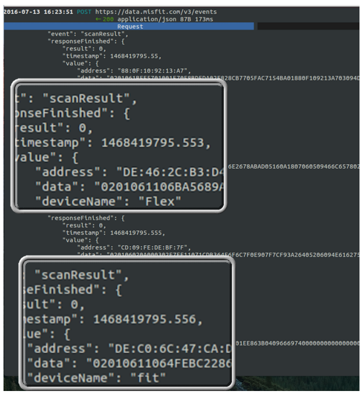


```
> echo 'AAAAAFeDq/NwAAAAAAAAAAAAAAAAAAFeDq/RxAAAAAAAAAAAAAAAAAAFeDq/VxAAAAAAAAAAAAAAAAAAFeD
q/ZxAAAAAAAAAAAAAAAAAAFeDq/dxAAAAAAAAAAAAAAAAAAFeDq/hxAAAAAAAAAAAAAAAAAAFeDq/lxAAAAAAAAAAAA
AAAAAFeDq/pxAAAAAAAAAAAAAAAAAAFeDq/txAAAAAAAAAAAAAAAAAAFeDq/xxAAAAAAAAAAAAAAAAAAFeDq/1xAAAA
AAAAAAAAAAAAAFeDq/5xAAAAAAAAAAAAAAAAAAFeDq/9xAAAAAAAAAAAAAAAAAAFeDrABxAAAAAAAAAAAAAAAAAAFeD
rAFxAAAAAAAAAAAAAAAAAAFeDrAJxAAAAAAAAAAAAAAAAAAFeDrANxAAAAAAAAAAAAAAAAAAFeDrARxAAAAAAAAAAAA
AAAAAFeDrAVxAAAAAAAAAAAAAAAAAAFeDrAZxAQAAAAAAAAAAAAAAAAFeDrAhxAQAAAAAAAAAAAAAAAAFeDrAlxAQAA
AAAAAAAA' | base64 --decode | shasum -a 1 | awk '{printf $1}' | xxd -r -p | base64
pjHbkX4V4Ji8L382hNYKgNLFXUo=
```

Figure 6.14: Recomputing the SHA1 digest from the data.

**MisFit Shine**

The MisFit tracker leverages the HTTPS protocol to encrypt data in-transit and make the communication more secure. The fitness data is transmitted using JSON file in a plain text. Since the data is in plain text, and the MisFit Shine does not take any steps to protect fitness data from being tampered with, we easily inject counterfeit data into MisFit server.



```
2016-07-13 16:23:51 POST https://data.misfit.com/v3/events
                         ← 200 application/json 87B 173ms
                  Request
        "event": "scanResult",
        "responseFinished": {
            "result": 0,
            "timestamp": 1468419795.55,
            "value": {
                "address": "88:0F:10:92:13:A7",
```

```
t": "scanResult",
onseFinished": {
esult": 0,
timestamp": 1468419795.553,
value": {
  "address": "DE:46:2C:B3:D4
  "data": "0201061106BA5689A
  "deviceName": "Flex"
```

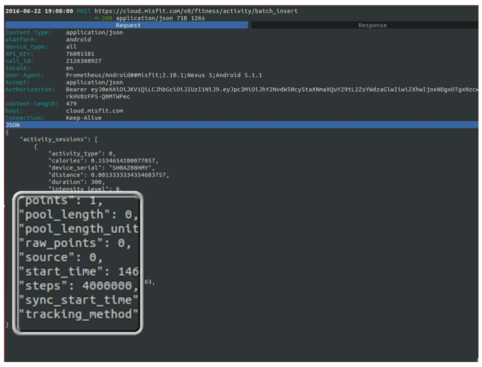
```
6E2678ABAD05160A1807060509466C657802
```

```
        "responseFinished": {
            "result": 0,
            "timestamp": 1468419795.555,
            "value": {
                "address": "CD:09:FE:DE:BF:7F",
```

```
 "scanResult",
eFinished": {
ult": 0,
estamp": 1468419795.556,
ue": {
"address": "DE:C0:6C:47:CA:D
"data": "02010611064FEBC2286
"deviceName": "fit"
```

```
01EE863B04096669740000000000000000
```

Figure 6.15: MisFit Shine Gathers data belonging to other fitness trackers.



```
2016-06-22 19:08:00 POST https://cloud.misfit.com/v0/fitness/activity/batch_insert
                         ← 200 application/json 71B 126s
                  Request                                    Response
Content-Type:    application/json
platform:        android
device_type:     all
API_KEY:         76801581
call_id:         2126360927
locale:          en
User-Agent:      Prometheus/Android##Misfit;2.10.1;Nexus 5;Android 5.1.1
Accept:          application/json
Authorization:   Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJhY2NvdW50cy5taXNmaXQuY29tL2ZsYWdzaGlwIiwiZXhwIjoxNDgxOTgxNzcw
                 rkHV0zFP5-Q0MTWPec
content-length:  479
host:            cloud.misfit.com
Connection:      Keep-Alive
JSON
{
    "activity_sessions": [
        {
            "activity_type": 0,
            "calories": 0.1534654200077057,
            "device_serial": "SH0AZ08HMY",
            "distance": 0.0013333334354683757,
            "duration": 300,
```

```
"points": 1,
"pool_length": 0,
"pool_length_unit
"raw_points": 0,
"source": 0,
"start_time": 146
"steps": 4000000,
"sync_start_time"
"tracking_method"
```

Figure 6.16: Fitness data sent by MisFit to its server.

Additionally, we noticed that the MisFit Shine collects some data during the Bluetooth scanning phase from nearby devices and sends these data to the remote server, as shown in Figure 6.15.

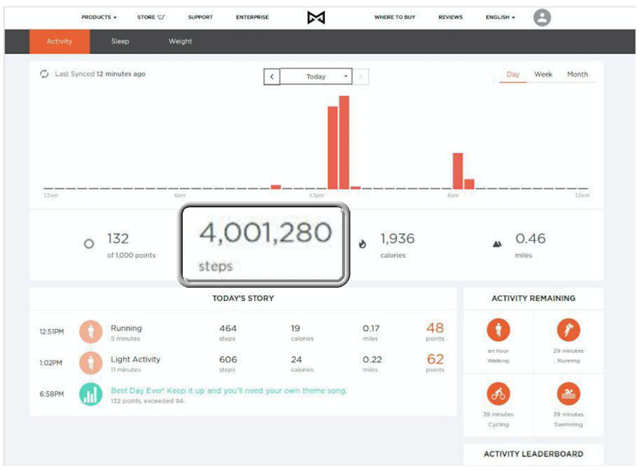
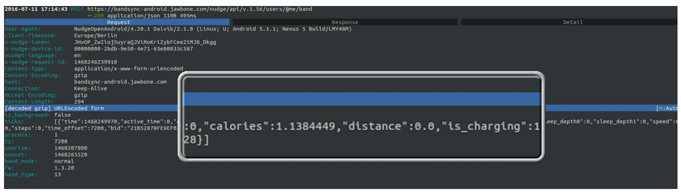


Figure 6.17: Injecting 4 million steps into the Misfit Shine server.

### Jawbone UP3 and MOVE UP

Jawbone trackers also use the HTTPS protocol to communicate with the Jawbone server. Since the fitness data is transmitted using the plain text JSON format, it is easy to tamper with the data. We sniff the HTTPS communications and manipulate the fitness data on the fly then forward the counterfeit data to the server. Jawbone's server accepts our counterfeit data without any check.



```
2016-07-11 17:14:43 POST https://bandsync-android.jawbone.com/nudge/api/v.1.56/users/@me/band
                         ← 200 application/json 110B 495ms
            Request                          Response                          Detail
User-Agent:          NudgeOpenAndroid/4.20.1 Dalvik/2.1.0 (Linux; U; Android 5.1.1; Nexus 5 Build/LMY48M)
Client-Timezone:     Europe/Berlin
x-nudge-token:       JHvOP_ZwIioj5uyraQJViMoKriZybFCee2tMJ8_Dkgg
x-nudge-device-id:   00000000-2bdb-9e50-4e71-63e80033c587
accept-language:     en
x-nudge-request-id:  1468246239910
Content-Type:        application/x-www-form-urlencoded
Content-Encoding:    gzip
host:                bandsync-android.jawbone.com
Connection:          Keep-Alive
Accept-Encoding:     gzip
Content-Length:      294
[decoded gzip] URLEncoded form                                                              [m:Aut
is_background: false
ticks:         [{"time":1468249970,"active_time":0,"  :0,"calories":1.1384449,"distance":0.0,"is_charging":1  eep_depth0":0,"sleep_depth1":0,"speed":
0,"steps":0,"time_offset":7200,"bid":"21B52878FE9EF0  28}]
process:       1
tz:            7200
sunrise:       1468207800
sunset:        1468265520
band_mode:     normal
fw:            1.3.20
band_type:     13
```

Figure 6.18: The fitness data sent by Jawbone trackers to their remote server.

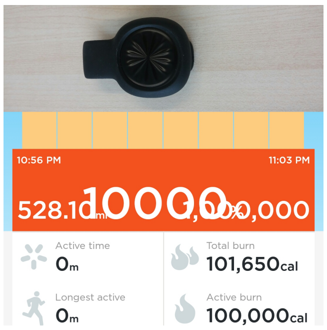


Figure 6.19: Injecting 1 million steps into the server.

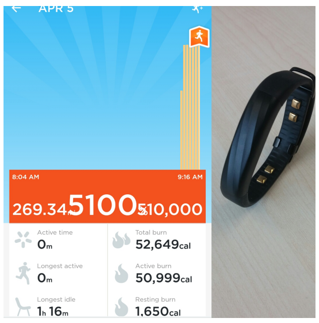
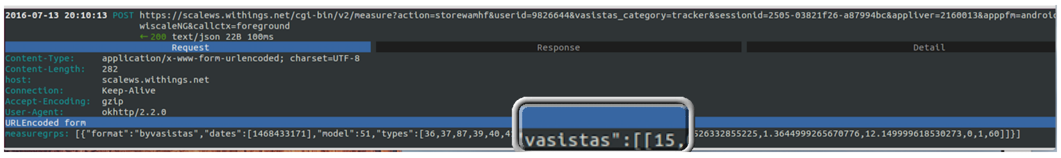


Figure 6.20: Injecting half a million steps into the server.

### Withings Pulse

The Withings tracker also works in a similar fashion to Jawbone's trackers. Since the fitness data sent by the trackers to the server is in plain text and not signed, it can be easily manipulated. We sniff the HTTPS requests, modify the fitness data on the fly, then inject the counterfeit data into the Withings server. The server accepts the counterfeit data without any check and raising error.



```
2016-07-13 20:10:13 POST https://scalews.withings.net/cgi-bin/v2/measure?action=storewamhf&userid=9826644&vasistas_category=tracker&sessionid=2505-03821f26-a87994bc&appliver=2160013&apppfm=androi
                         wiscaleNG&callctx=foreground
                         ← 200 text/json 22B 100ms
                    Request                                        Response                                        Detail
Content-Type:      application/x-www-form-urlencoded; charset=UTF-8
Content-Length:    282
host:              scalews.withings.net
Connection:        Keep-Alive
Accept-Encoding:   gzip
User-Agent:        okhttp/2.2.0
URLEncoded form
measuregrps: [{"format":"byvasistas","dates":[1468433171],"model":51,"types":[36,37,87,39,40,4 vasistas":[[15, 526332855225,1.3644999265670776,12.149999618530273,0,1,60]]}]
```

Figure 6.21: Steps count in plain-text format.

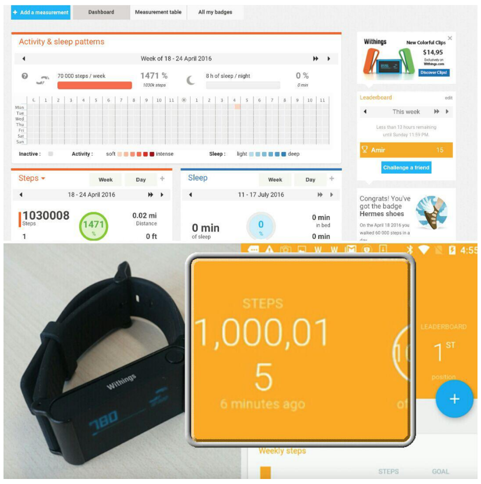
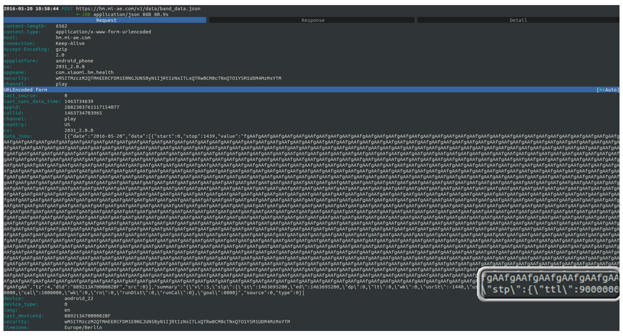


Figure 6.22: Injecting 1 million steps into Withings' web server.

### Xiaomi MiBand

Like other fitness trackers in our experiments, the MiBand tracker leverages the HTTPS protocol to communicate with its own remote server. We sniff the communications and find that no encryption is performed besides HTTPS. Since encryption and data integrity check are not performed, the data can be freely manipulated. We modify the fitness data on the fly, then send it to the Xiaomi server.

The server accepts the counterfeit data and stores it. In contrast with known tracker manufacturers, Xiaomi chose not to develop a web interface for the data stored on its cloud. To confirm that our counterfeit data had actually been accepted, we uninstall the app from the phone (deleting all data as well), then we install it again. The app downloads the data associated with our account from the cloud and shows our counterfeit data.



```
2016-05-20 10:58:44 POST https://hm.mi-ae.com/v1/data/band_data.json
                   ← 200 application/json 86B 80.9s
                 Request                       Response                       Detail
content-length:     6562
content-type:       application/x-www-form-urlencoded
host:               hm.mi-ae.com
Connection:         Keep-Alive
Accept-Encoding:    gzip
v:                  2.0
appplatform:        android_phone
cv:                 2031_2.0.0
appname:            com.xiaomi.hm.health
security:           wM5ITMzczM2QTMmEERCFDM1E0NGJUN50yN1IjRtIzNxITLxQTRw0CM0cTNxQTO1YSM1UDM4MzMxYTM
channel:            play
URLEncoded form                                                                                [:Auto]
last_source:           0
last_sync_data_time:   1463734639
appid:                 2882303761517154077
callid:                1463734703965
channel:               play
country:               US
cv:                    2031_2.0.0
data_json:             [{"date":"2016-05-20","data":[{"start":0,"stop":1439,"value":"fgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfg
AAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgAAfgA
[illegible]
fgAAfgAA","tz":4,"did":"889213A7000002BF","src":0}],"summary":"{\"v\":5,\"slp\":{\"st\":1463695200,\"ed\":1463695200,\"dp\":0,\"lt\":0,\"wk\":0,\"usrSt\":-1440,\"us
0000,\"cal\":1000000,\"wk\":0,\"rn\":0,\"runDist\":0,\"runCal\":0},\"goal\":8000}","source":0,"type":0}]
                                                                                                      fgAAfgAAfgAAfgAAfgAAfgAA
                                                                                                      \"stp\":{\"ttl\":9000000
device:                android_22
device_type:           0
lang:                  en
last_deviceid:         889213A7000002BF
security:              wM5ITMzczM2QTMmEERCFDM1E0NGJUN50yN1IjRtIzNxITLxQTRw0CM0cTNxQTO1YSM1UDM4MzMxYTM
timezone:              Europe/Berlin
```

Figure 6.23: Fitness data sent by Miband to its web server.

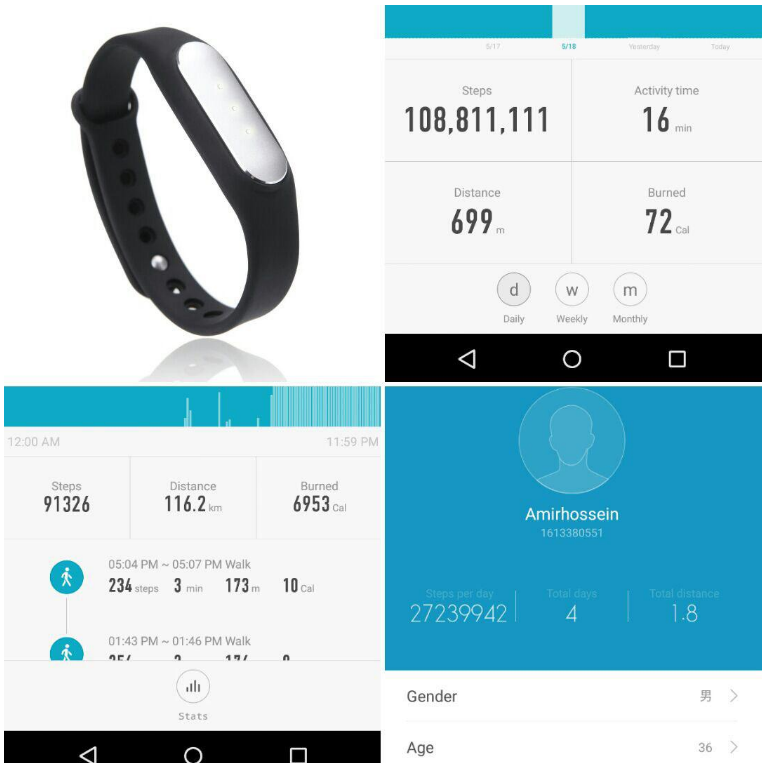
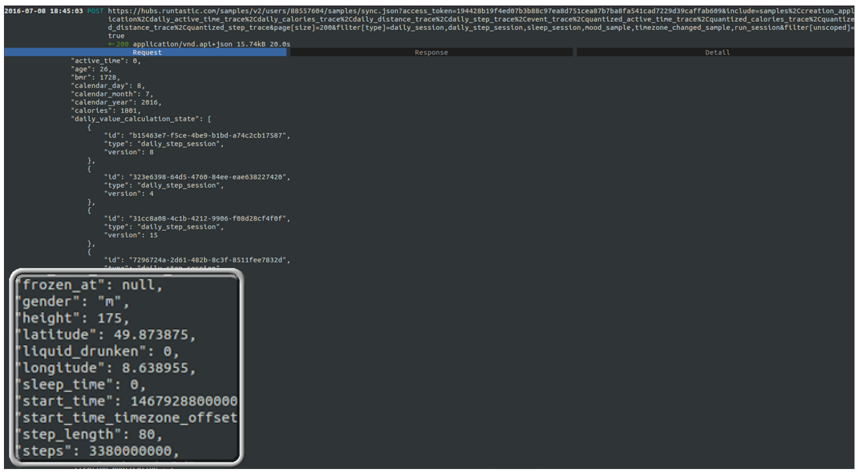


Figure 6.24: Counterfeit data (several million steps) in the Mi Fit app.

### Runtastic Orbit

The Runtastic tracker uses the HTTPS protocol to communicate with the Runtastic server and encrypts data in-transit. The fitness data is transmitted using JSON file in a plain text format. We intercept the requests issued by the trackers, modify them on the fly (using mitmproxy), and send false data to the server and server without doing any data integrity check accepts the counterfeit data.



```
2016-07-08 18:45:03 POST https://hubs.runtastic.com/samples/v2/users/88557604/samples/sync.json?access_token=194428b19f4ed07b3b88c97ea8d751cea87b7ba8fa541cad7229d39caffab609&include=samples%2Ccreation_appl
                         ication%2Cdaily_active_time_trace%2Cdaily_calories_trace%2Cdaily_distance_trace%2Cdaily_step_trace%2Cevent_trace%2Cquantized_active_time_trace%2Cquantized_calories_trace%2Cquantize
                         d_distance_trace%2Cquantized_step_trace&page[size]=200&filter[type]=daily_session,daily_step_session,sleep_session,mood_sample,timezone_changed_sample,run_session&filter[unscoped]=
                         true
                         ← 200 application/vnd.api+json 15.74kB 20.0s
                    Request                                  Response                                  Detail
            "active_time": 0,
            "age": 26,
            "bmr": 1728,
            "calendar_day": 8,
            "calendar_month": 7,
            "calendar_year": 2016,
            "calories": 1801,
            "daily_value_calculation_state": [
                {
                    "id": "b15463e7-f5ce-4be9-b1bd-a74c2cb17587",
                    "type": "daily_step_session",
                    "version": 8
                },
                {
                    "id": "323e6398-64d5-4760-84ee-eae638227420",
                    "type": "daily_step_session",
                    "version": 4
                },
                {
                    "id": "31cc8a08-4c1b-4212-9906-f08d28cf4f0f",
                    "type": "daily_step_session",
                    "version": 15
                },
                {
                    "id": "7296724a-2d61-482b-8c3f-8511fee7832d",

"frozen_at": null,
"gender": "m",
"height": 175,
"latitude": 49.873875,
"liquid_drunken": 0,
"longitude": 8.638955,
"sleep_time": 0,
"start_time": 1467928800000
"start_time_timezone_offset
"step_length": 80,
"steps": 3380000000,
```

Figure 6.25: HTTPS request from the Orbit to its server.

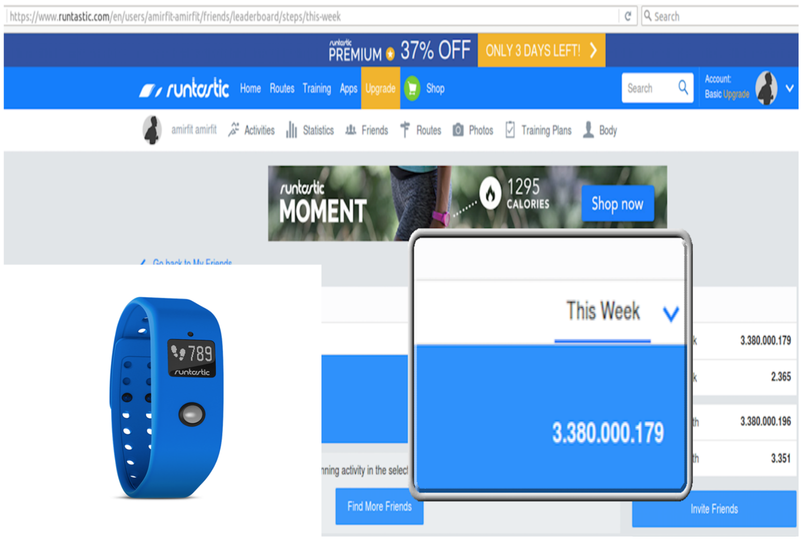

https://www.runtastic.com/en/users/amirfit-amirfit/friends/leaderboard/steps/this-week
Search
PREMIUM 37% OFF ONLY 3 DAYS LEFT!
runtastic Home Routes Training Apps Upgrade Shop
Search
Account: Basic Upgrade
amirfit amirfit Activities Statistics Friends Routes Photos Training Plans Body
runtastic MOMENT
1295 CALORIES
Shop now
Go back to My Friends
789
This Week
3.380.000.179
3.380.000.179
2.365
3.380.000.196
3.351
Find More Friends
Invite Friends

Figure 6.26: Injecting more than 3 million steps into Runtastic's web server.

**Fitness Trackers without Cloud Services**

We test several other fitness trackers, including Razor Nabu X, Technaxx 37 and 39, Sony Smartband 2, and Oregan Dynamo 2+. The companion applications of these devices do not perform any HTTP(S) request, with the exception of the Razor Nabu X which does not send fitness data but just uploads some meta-information. We examine how these applications store the fitness data on the smartphone. The apps for these five trackers, as well as previously mentioned devices, store the data in an unencrypted SQL databases. The fitness data not only can be read or modified by other malicious apps on a rooted phone but also can be easily stolen by an adversary with physical access to the phone.

| No. | **Tracker** | Attacked on Year 2016 | On Market from Year | Injecting Fake Data | Data in-transit Encryption | Data at Rest Encryption | Data Integrity | Proprietary Coding | SSL Pinning | End-to-end Encryption | Cloud Data Storage | Cloud Web Interface |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Garmin Vivosmart HR | Sep. | 2015 | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ |
| 2 | Garmin Vivofit2 | Sep. | 2015 | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ |
| 3 | Garmin Vivofit | Sep. | 2014 | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ |
| 4 | Polar Electro Loop | Sep. | 2013 | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ |
| 5 | ViFit MEDISANA | Sep. | 2014 | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ |
| 6 | Xiaomi MiBand | Aug. | 2014 | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| 7 | Jawbone UP3 | Aug. | 2015 | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| 8 | Jawbone Move UP | July | 2014 | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| 9 | Misfit Shine | June | 2013 | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| 10 | Mio Link | July | 2014 | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ |
| 11 | Withings Pulse | July | 2013 | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| 12 | Runtastic Orbit | July | 2014 | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| 13 | Sony Smartband 2 | — | 2015 | — | — | ✗ | — | — | — | — | ✗ | ✗ |
| 14 | Razor Nabu X | — | 2015 | — | — | ✗ | — | — | — | — | ✗ | ✗ |
| 15 | Technaxx 39 | — | 2015 | — | — | ✗ | — | — | — | — | ✗ | ✗ |
| 16 | Technaxx 37 | — | 2015 | — | — | ✗ | — | — | — | — | ✗ | ✗ |
| 17 | Oregon Dynamo 2+ | — | 2014 | — | — | ✗ | — | — | — | — | ✗ | ✗ |

Table 6.2: Test Results.

### 6.3.2 Findings

It is worth mentioning that out of all fitness trackers examined only Garmin, Polar loop, ViFit MEDISANA, and Mio Link take some minor measures to protect data integrity. Garmin and Polar Loop take benefit from their own proprietary encoding which makes it somewhat more challenging to modify data in-transit. ViFit MIDESANA uses an HMAC scheme to verify the expected sender and check data integrity and Mio Link benefits from SHA-1 (Secure Hash Algorithm) to verify data integrity. The products form Xiaomi, Misfit Shine, Runtastic Orbit, Jawbone, and Withings Pulse do not use proprietary encoding or encryption to prevent data from being read and tampered with. These fitness trackers send fitness data in plain text over HTTPS and for a malicious user it is relatively easy to cheat and inject falsified data into the remote server.

The fitness trackers, Sony Smartband 2, Razor Nabu X, Technaxx 39 and 37, and finally Oregon Dynamo 2+ do not offer cloud-based services for data analytics or storage. Also, the companion applications of these fitness trackers store the data in plain text on the smartphone which introduces the risk of data disclosure. We have summarized our test results in Table 6.2.

## 6.4 Countermeasure & Remedy

Part of the problem with the security of fitness tracking devices is because, wearable makers are rushing to beat their competitors and get their product onto the market first. Fitness tracking manufacturers need to build privacy and security into their existing development process.
When designing wearable devices connected to the Internet, manufacturers must establish a secure hardware and software development process that includes source code analysis to identify security-related vulnerabilities and software delivery mechanisms.

Deploying these devices in a secure way is just as important as their design. Attention should be paid to device provisioning and authentication. The deployment process should cross-authenticate both the device and the network to ensure it does not transmit confidential information in insecure way.

Last but not least, end-to-end encryption should be deployed to ensure the integrity and confidentiality of the data during transmission from device to cloud and vise versa, as shown in Figure 6.27. End-to-end encryption with a *device-specific key* prevents the data from being manipulated using MITM attacks.

One of the benefits of using a device-specific key is to reduce the risk of possible MITM Attack on re-keying and renegotiation in key exchange protocols. The device key itself could be computed using a *master key*, just known to the server. The input for such a computation could for instance be the serial number of the device. The device key is pushed into device during manufacturing process.

In addition to end-to-end encryption (on-chip), Hardware-supported memory readout protection should be applied to add additional layer of security such that device memory cannot be accessible through debugging interferes.

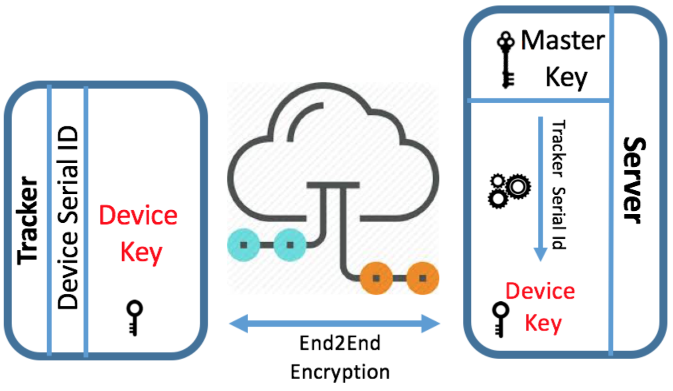


Figure 6.27: Deploying End-to-end encryption mechanism, using a device-specific encryption key.

## 6.5 Discussion

We mainly discuss an adversarial setting in which a malicious user can view and manipulate all the data uploaded to the server. A third-party adversary also can conduct same attacks by setting up a WiFi AP (which runs MITM proxy) and eavesdrops TLS encrypted communications provided that the malicious third-party must convince or deceive victims (e.g., using social engineering techniques) to install CA certificate associated with the MITM proxy. Even if convincing users to install a fake certificate is mostly related to the user's trustworthiness and is a big security concern, but accessing third-party to users' data will be a huge privacy concern.

Wearable devices handle sensitive and critical data showing user health status, Geo-location position and etc. These devices carry health-related information, showing user life habits, health situation and many things that can be abused by third-parties or adversary. For instance, if the adversary has access to the user data, by analyzing the data he can understand about user health situation and even worse user sleep pattern (when user is exactly asleep), Geo-location information (when user is far away from his home), then adversary can break-in user's home or even track user and spy on him.

## 6.6 Related Work

Related work can be categorized into two important groups.

### 6.6.1 Firmware modification attacks

Rieck [211] has identified a vulnerability in the firmware update process for Withings Activity fitness tracker. He used this vulnerability to inject malicious code into the fitness tracker firmware. In the similar work [212], Coppola studied the firmware update procedure of Withings' WS-30 wireless scale. He combined hardware and software reverse engineering to exploit a critical flaw in the firmware update process and he discovered a flow in the firmware allowing him to upload arbitrary firmware to the scale.

### 6.6.2 Data integrity and privacy attacks

In [208] researchers at the University of Toronto investigate transmission security, data integrity, and Bluetooth privacy of eight fitness trackers. They discover several security and privacy issues with the analyzed devices. Five major categories of data transmitted to remote servers over the Internet are investigated. These categories includes basic personal information, fitness information, location information, social information, and device identifiers. They also examine data integrity concentrating on whether or not fitness data can be considered authentic records of activity that have not been tampered with.

Britt et al. [27] analyze the Fitbit Flex ecosystem. They attempt to do a hardware analysis of the Fitbit device but because of difficulties associated with debugging the device they decide to focus on other parts such as data transmission over Bluetooth LE, the associated Android app and network analysis. The authors study the data collected by Fitbit from its users as well as the data Fitbit provided to Fitbit users.

In a report released by AV TEST [209], the authors test nine fitness trackers and evaluate their security and privacy. The authors try to find out how easy it is to get the fitness data from the fitness band through Bluetooth or by sniffing the connection to the cloud during the synchronization process. They list all Bluetooth related security features and their implementation on the tracker as well as the mobile application, considering, e.g., Bluetooth deactivation, pairing, tracker authentication. Also, the aspects of application implementation (e.g., code obfuscation, logging, etc.), data storage and information propagation are analyzed. In another work [28], the same authors evaluate the security of seven different fitness trackers. They again analyze the original application, Bluetooth communication between the tracker and smartphone and on-line communication of the original application.

Margaritelli [210] explores aspects of Bluetooth security and authentication process. Using an implementation for the Nike+ FuelBand, it is shown that a robust protocol can be easily defeated by a poor implementation.

Barcena et al. [220] publish a report in 2014, describing issues related to the handling of private information by fitness trackers. They are able to track individuals using fitness trackers using cheap, off-the-shelf hardware by simply connecting to the fitness trackers on the street.

Our work from many aspects differs from above-mentioned work, as we focus on injecting falsified data into remote servers of fitness products. We do not concentrate on firmware injection attacks but are interested in attacks targeting data integrity. Different research work in the literature have just mentioned that how easy it is to get the fitness data from the fitness band through Bluetooth or by sniffing the connection to the cloud during the synchronization process. They have not shown that how vulnerabilities mentioned in the literature can be exploited to perform serious and real attacks in practice such as false data injection attack, as we perform. Last but not least, we concentrate on fitness trackers that are known to be more secure than their counterparts in the market. For instance, wearable devices that have employed security controls such as proprietary coding and data integrity check mechanisms ( while other work have not taken such devices into account to analyze).

## 6.7 Summary

We analyze the security of many popular and less popular fitness trackers in the malicious user setting that user can manipulate the data. Technical analysis shows that all tracking systems that offer cloud-based services protect the confidentiality of communications through the use of an encrypted protocol like HTTPS to transfer data from the smartphone of the user to the cloud service. However, in all cases, it is still possible for a malicious user to falsify the data. Most fitness devices have no data integrity check. Even for the few of them that use digital signature, the keys that are used for digital signatures can be easily captured by the MITM proxy and hence does not prevent data tampering. None of the devices consider use of end-to-end encryption when synchronizing with the remote server and no effective tamper protection is used. Also, the manufacturers who not offering cloud-based services store the collected fitness data in plain text on the smartphone which introduces a potential risk of unauthorized data leakage should the smartphone be stolen or be infected with malware.

# Chapter 7

# Security Analysis, Reverse Engineering and Spoofing Advanced Fitness Devices

Market forecasts indicate 274 million wrist-based fitness trackers and smartwatches will be sold worldwide by 2020 [221]. Such devices already enable users and healthcare professionals to monitor individual activity and sleep habits, and underpin reward schemes that incentivize regular physical exercise. Fitbit maintains the lead in the wearables market, having shipped more units in 2016 than its biggest competitors Apple, Garmin, and Samsung combined [222].

Fitness trackers collect extensive information which enables infering the users' health state and may reveal particularly sensitive personal circumstances. For instance, one individual recently discovered his wife was pregnant after examining her Fitbit data [223]. Police and attorneys start recognizing wearables as "black boxes" of the human body and use statistics gathered by activity trackers as admissible evidence in court [62,224]. These developments highlight the critical importance of both preserving data privacy throughout the collection process, and ensuring correctness and authenticity of the records stored. The emergence of third-party services offering rewards to users who share personal health information further strengthens the significance of protecting wearables data integrity. These include health insurance companies that provide discounts to customers who demonstrate physical activity through their fitness tracker logs [225], websites that financially compensate active users consenting to fitness monitoring [226], and platforms where players bet on reaching activity goals to win money [227]. Unfortunately, such on-line services also bring *strong incentives for malicious users to manipulate tracking data, in order to fraudulently gain monetary benefits.*

Given the value fitness data has towards litigation and income, researchers have analyzed potential security and privacy vulnerabilities specific to activity trackers [26–29]. Following a survey of 17 different fitness trackers available on the European market in Q1 2016 [228], recent investigations into the security of Fitbit devices (e.g. [29]), and the work we present herein, we found that in comparison to other vendors, Fitbit employs the most effective security mechanisms in their products. Such competitive advantage, giving users the ability to share statistics with friends, and the company's overall market leadership make Fitbit one of the most attractive vendors to third parties running fitness-based financial reward programs. At the same time it motivates us to choose Fitbit trackers as the target of our security study, in the hope that understanding their underlying security architecture can be used to inform the security and privacy of future fitness tracker system designs. Rahman *et al.* have investigated the communication protocols used by early Fitbit wearables when synchronizing with web servers and possible attacks against this [26]. Cyr *et al.* [27] studied the different layers of the Fitbit Flex ecosystem and argued correlation and man-in-the-middle (MITM) attacks are feasible. Recent work documents firmware vulnerabilities found in Fitbit trackers [28], and the reverse engineering of cryptographic primitives and authentication protocols [29]. However, as rapid innovation is the primary business objective, security considerations remain an afterthought rather than embedded into product design. Therefore, wider adoption of wearable technology is hindered by distrust [30, 31].

**Contributions**: We undertake an in-depth security analysis of the Fitbit Flex and Fitbit One fitness trackers and reveal serious security and privacy vulnerabilities present in these devices which, although difficult to uncover, are reproducible and **can be exploited at scale** once identified. Specifically, we reverse engineer the primitives governing the communication between trackers and cloud-based services, implement an open-source tool to extract sensitive personal information in human-readable format, and demonstrate that malicious users can inject fabricated activity records to obtain personal benefits. To circumvent end-to-end protocol encryption implemented in the latest firmware, we perform hardware-based reverse engineering (RE) and document successful injection of falsified data that appears legitimate to the Fitbit cloud. The weaknesses we uncover, as well as the design guidelines we provide to ensure data integrity, authenticity and confidentiality, build foundations for more secure hardware and software development, including code and build management, automated testing, and software update mechanisms. Our insights provide valuable information to researchers and practitioners about the detailed way in which Fitbit operates their fitness tracking devices and associated services. These may help IoT manufacturers in general to improve their product design and business processes, towards developing rigorously secured devices and services.

### Responsible Disclosure

We contacted Fitbit prior to publishing our work and informed the company about the security vulnerabilities we discovered in their products. We disclosed these to them to allow sufficient time to fix the identified problems before the publication of our findings.

## 7.1 Adversary Model

To retrieve the statistics that trackers collect, users predominantly rely on smartphone or tablet applications that extract activity records stored by the devices, and push these onto cloud servers. We consider the adversarial settings depicted in

Figure 7.1, in which users are potentially dishonest, whilst the server is provably trustworthy. We assume an active adversary model in which the wristband user is the primary adversary, who has both the means and motive to compromise the system. Specifically, the attacker (a) views and seeks to manipulate the data uploaded to the server without direct physical control over the device, or (b) inspects and alters the data stored in memory prior to synchronization, having full hardware control of the device. The adversary's motivation is rooted in the potential to obtain financial gains by injecting fabricated fitness data to the remote server. Smartphone and cloud platform security issues are outside the of scope of this paper, therefore not considered in our analysis.

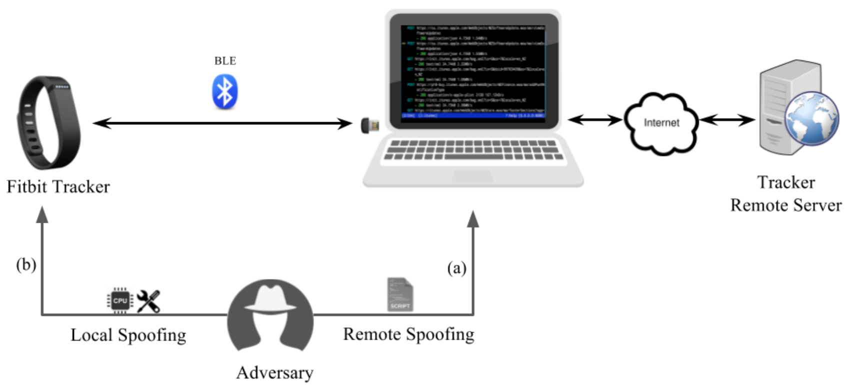


Figure 7.1: Adversary model considered for (a) devices not implementing encryption and (b) trackers using encryption.

## 7.1.1 Target Fitbit Devices

The adversary's target devices are the *Fitbit Flex* and *Fitbit One* wrist-based fitness trackers, which record user step counts, distance traveled, calories burned, floors climbed (Fitbit One), active minutes, and sleep duration. These particular trackers have been on the market for a number of years, they are affordable and their security and privacy has been scrutinized by other researchers. Thus, both consumers and the vendor would expect they are not subject to vulnerabilities.

We subsequently found that other Fitbit models (e.g. Zip and Charge) implement the same communication protocol, therefore may be subject to the same vulnerabilities we identify in this work.

## 7.1.2 End-to-End Communication Paradigms

Following initial pairing, we discover Fitbit trackers are shipped with one of two different firmwares; namely, the latest version (Flex 7.81) which by default encrypts activity records prior to synchronization using the XTEA algorithm and a pre-installed encryption key; and, respectively, an earlier firmware version (Flex 7.64) that by default operates in plaintext mode, but is able to activate message encryption after being instructed to do so by the Fitbit server. If enabled, *encryption is end-to-end* between the tracker and the server, whilst the smartphone app is unaware of the actual contents pushed from tracker to the server. The app merely embeds encrypted records retrieved from the tracker into JSON messages, forwards them to the Fitbit servers, and relays responses back to the tracker. The same functionality can be achieved through software running on a computer equipped with a USB Bluetooth LE dongle, including the open-source Galileo tool, which does not require user authentication [229].

Even though only the tracker and the server know the encryption key, upon synchronization the smartphone app also receives statistic summaries from the server in human readable format over an HTTPS connection. As such, and following authentication, the app and authorized third parties can connect to a user account via the Fitbit API and retrieve activity digests—without physical access to the tracker. We also note that, despite newer firmware enforcing end-to-end encryption, the Fitbit server continues to accept and respond to unencrypted activity records from trackers that only optionally employ encryption, thereby enabling an attacker to successfully modify the plaintext activity records sent to the server.

## 7.2 Protocol Reverse Engineering

In this section, we reverse engineer the communication protocol between Fitbit trackers and servers, uncovering an intricate security through obscurity approach. Once understanding the message semantics, we show that detailed personal information can be extracted and fake activity reports can be created and remotely injected, using an approach that scales, as documented in Section 7.3.

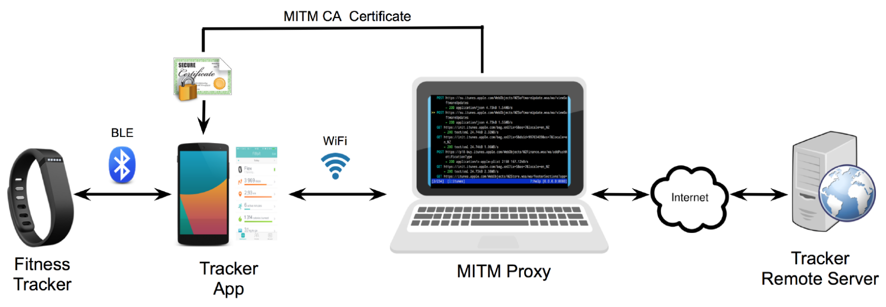


Figure 7.2: Schematic illustration of the testbed used for protocol reverse engineering. Linux-based laptop used as wireless Internet gateway and running MITM proxy.

### 7.2.1 MITM Setup

To intercept the communication between the tracker and the remote server, we deploy an MITM proxy on a Linux-based laptop acting as a wireless Internet gateway, as illustrated in Figure 7.2. We install a fake CA certificate on an Android phone and trigger tracker synchronization manually, using an unmodified Fitbit application. The application synchronizes the tracker over Bluetooth LE and forwards data between the tracker and the server over the Wi-Fi connection, encapsulating the information into JSON messages sent over an HTTPS connection. This procedure resembles typical user engagement with the tracker, however the MITM proxy allows us to intercept all communications between the tracker and the server, as well as between the smartphone and the server. In the absence of end-to-end encryption, we can both capture and modify messages generated by the tracker. Even with end-to-end encryption enabled, we can still read the activity digests that the server provides to logged-in users, which are displayed by the app running on their smartphones.

### 7.2.2 Wireshark Plugin Development and Packet Analysis

To simplify the analysis process and ensure repeatability, we develop a custom frame dissector as stand-alone plugin programmed in C for the Wireshark network analyzer [230].[1] Developing this dissector involves cross-correlating the raw messages sent by the tracker with the server's JSON responses to the client application. After repeated experiments, we infer the many protocol fields that are present in tracker-originated messages and that are encoded in different formats as detailed next. We use the knowledge gained to present these fields in a human-readable format in the protocol analyzer.

There are two types of tracker-originated messages we have observed during our analysis, which will be further described in the following sections:

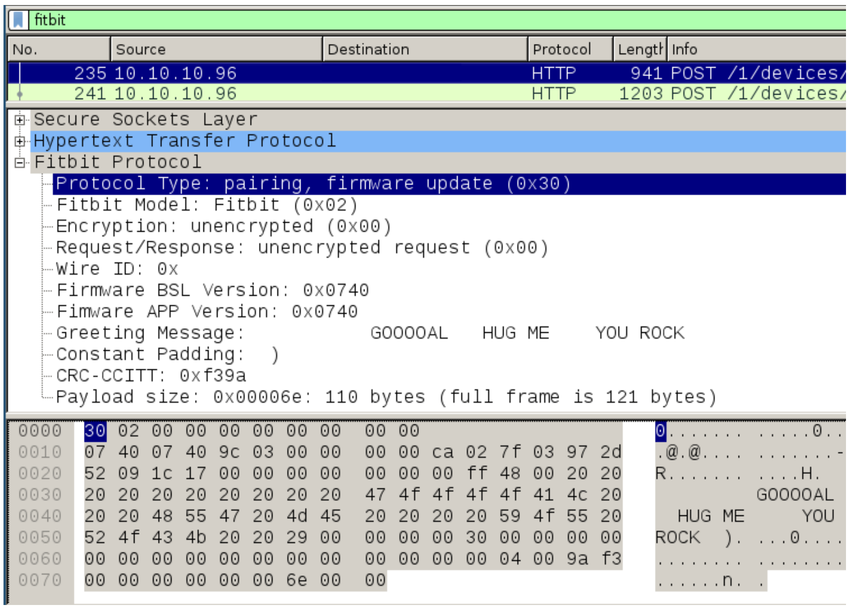
1. **Microdumps:** A summary of the tracker status and configuration.
2. **Megadumps:** A summary of user activity data from the tracker.

### 7.2.3 Microdump

Depending on the action being performed by the user (e.g. authentication and pairing, synchronizing activity records), the smartphone app makes HTTPS requests to the server using specific URLs, e.g. `POST https://<fitbit_server_ip>/1/devices/client/tracker/data/validate.json?btle_Name=Flex&secret=null&btAddress=<6Byte_tracker_ID>` for initial authentication. Each basic action is accompanied by a so-called *microdump*, which is required to identify the tracker, and to obtain its state (e.g. its current firmware version). Irrespective of whether or not the tracker implements protocol encryption, the microdump header includes the tracker ID and firmware version, and is sent in plain-text. Figure 7.3 illustrates a microdump sent along with a firmware update request, as interpreted by our Wireshark dissector.

We also note that the only validation feature that plain-text messages implement is a CRC-CCITT checksum, presumably used by the server to detect data corruption in tracker-originated messages. In particular, this acquired knowledge will allow us to inject generic messages into the server and obtain replies, even when a valid

[1]We shall make the source code of our plug-in available upon publication.



```
fitbit
No.         Source            Destination       Protocol  Length Info
      235 10.10.10.96                           HTTP        941 POST /1/devices/
      241 10.10.10.96                           HTTP       1203 POST /1/devices/
⊞ Secure Sockets Layer
⊞ Hypertext Transfer Protocol
⊟ Fitbit Protocol
    Protocol Type: pairing, firmware update (0x30)
    Fitbit Model: Fitbit (0x02)
    Encryption: unencrypted (0x00)
    Request/Response: unencrypted request (0x00)
    Wire ID: 0x
    Firmware BSL Version: 0x0740
    Fimware APP Version: 0x0740
    Greeting Message:            GOOOOAL    HUG ME     YOU ROCK
    Constant Padding:  )
    CRC-CCITT: 0xf39a
    Payload size: 0x00006e: 110 bytes (full frame is 121 bytes)

0000  30 02 00 00 00 00 00 00  00 00                    0....... .....0..
0010  07 40 07 40 9c 03 00 00  00 00 ca 02 7f 03 97 2d  .@.@.... .......-
0020  52 09 1c 17 00 00 00 00  00 00 00 ff 48 00 20 20  R....... ....H.
0030  20 20 20 20 20 20 20 20  47 4f 4f 4f 4f 41 4c 20          GOOOOAL
0040  20 20 48 55 47 20 4d 45  20 20 20 20 59 4f 55 20    HUG ME       YOU
0050  52 4f 43 4b 20 20 29 00  00 00 00 30 00 00 00 00  ROCK  ). ...0....
0060  00 00 00 00 00 00 00 00  00 00 00 00 04 00 9a f3  ........ ........
0070  00 00 00 00 00 00 6e 00  00                       ......n. .
```

Figure 7.3: Generic microdump in plain-text, as displayed by the wireshark dissector we implement. Note the ability to filter by 'fitbit' protocol type in the analyzer.

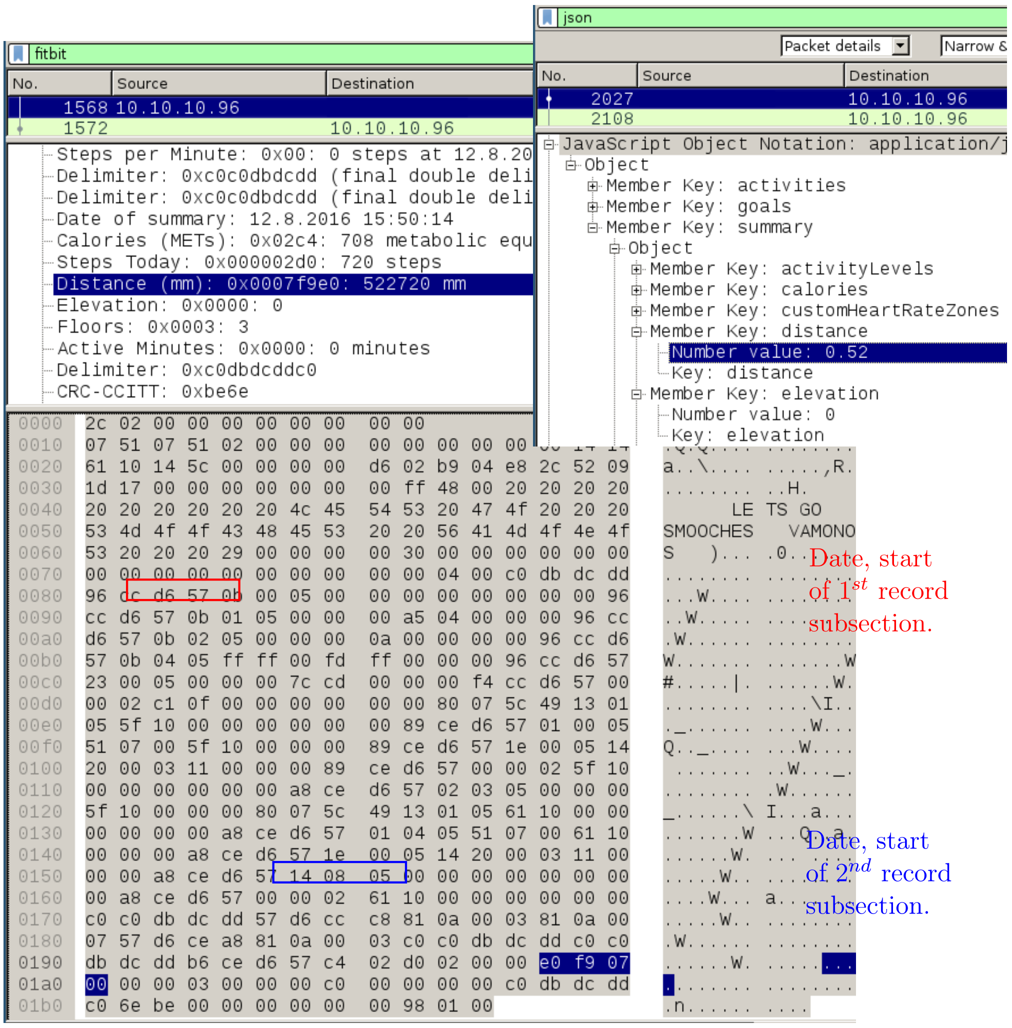
tracker ID is already associated with a person's existing account. Yet, microdumps only contain generic information, which does not allow the spoofing of user activity records. In what follows, we detail the format of messages sent to the server to synchronize the tracked user activity.

Note that the plain-text format does not provide measures for verifying the integrity and authenticity of the message contents except for a checksum, which is deterministically calculated from the values of the message fields. This allows the adversary to inject generic messages to the server and receive replies, including information about whether a tracker ID is valid and associated with a user account.

### 7.2.4 Megadump Synchronization Message

Step counts and other statistics are transmitted by the tracker in the form of a so-called *megadump*. Independent of encrypted or plain-text mode, neither the Fitbit smartphone application nor the Galileo synchronization tool are aware of the exact meaning of this payload. The megadump is simply forwarded to the server, which in turn parses the message and responds with a reply. This reply is then forwarded (by the corresponding application) back to the tracker, confirming to the tracker that the data was synchronized with the server successfully.

Despite this behavior, the Fitbit smartphone application—in contrast to Galileo—is aware of the user's statistics. However, this is due to the application making requests to the Fitbit Web API. Once authenticated, this API can be used to retrieve user information from the server in JSON format. The Fitbit smartphone application periodically synchronizes its display via the Fitbit Web API, allowing the user to see the latest information that was uploaded by the most recent tracker megadump. A plain-text example of this is shown in Figure 7.4. Note that the


fitbit
No.
Source
Destination
json
Packet details
Narrow &
Date, start of 1st record subsection.
Date, start of 2nd record subsection.

```
1568 10.10.10.96
1572              10.10.10.96

Steps per Minute: 0x00: 0 steps at 12.8.20
Delimiter: 0xc0c0dbdcdd (final double deli
Delimiter: 0xc0c0dbdcdd (final double deli
Date of summary: 12.8.2016 15:50:14
Calories (METs): 0x02c4: 708 metabolic equ
Steps Today: 0x000002d0: 720 steps
Distance (mm): 0x0007f9e0: 522720 mm
Elevation: 0x0000: 0
Floors: 0x0003: 3
Active Minutes: 0x0000: 0 minutes
Delimiter: 0xc0dbdcddc0
CRC-CCITT: 0xbe6e

0000  2c 02 00 00 00 00 00 00  00 00
0010  07 51 07 51 02 00 00 00  00 00 00 00 00                 
0020  61 10 14 5c 00 00 00 00  d6 02 b9 04 e8 2c 52 09   a..\.... .....,R.
0030  1d 17 00 00 00 00 00 00  00 ff 48 00 20 20 20 20   ........ ..H.
0040  20 20 20 20 20 20 4c 45  54 53 20 47 4f 20 20 20         LE TS GO
0050  53 4d 4f 4f 43 48 45 53  20 20 56 41 4d 4f 4e 4f   SMOOCHES   VAMONO
0060  53 20 20 20 29 00 00 00  00 30 00 00 00 00 00 00   S   )... .0......
0070  00 00 00 00 00 00 00 00  00 00 04 00 c0 db dc dd   ........ ........
0080  96 cc d6 57 0b 00 05 00  00 00 00 00 00 00 00 96   ...W.... ........
0090  cc d6 57 0b 01 05 00 00  00 a5 04 00 00 00 96 cc   ..W..... ........
00a0  d6 57 0b 02 05 00 00 00  0a 00 00 00 00 96 cc d6   .W...... ........
00b0  57 0b 04 05 ff ff 00 fd  ff 00 00 00 96 cc d6 57   W....... .......W
00c0  23 00 05 00 00 00 7c cd  00 00 00 f4 cc d6 57 00   #.....|. ......W.
00d0  00 02 c1 0f 00 00 00 00  00 00 80 07 5c 49 13 01   ........ ....\I..
00e0  05 5f 10 00 00 00 00 00  00 89 ce d6 57 01 00 05   ._...... ....W...
00f0  51 07 00 5f 10 00 00 00  89 ce d6 57 1e 00 05 14   Q.._.... ...W....
0100  20 00 03 11 00 00 00 89  ce d6 57 00 00 02 5f 10    ....... ..W..._.
0110  00 00 00 00 00 00 a8 ce  d6 57 02 03 05 00 00 00   ........ .W......
0120  5f 10 00 00 00 80 07 5c  49 13 01 05 61 10 00 00   _......\ I...a...
0130  00 00 00 00 a8 ce d6 57  01 04 05 51 07 00 61 10   .......W ...Q.a.
0140  00 00 00 a8 ce d6 57 1e  00 05 14 20 00 03 11 00   ......W. ........
0150  00 00 a8 ce d6 57 14 08  05 00 00 00 00 00 00 00   .....W.. ........
0160  00 a8 ce d6 57 00 00 02  61 10 00 00 00 00 00 00   ....W... a.......
0170  c0 c0 db dc dd 57 d6 cc  c8 81 0a 00 03 81 0a 00   .....W.. ........
0180  07 57 d6 ce a8 81 0a 00  03 c0 c0 db dc dd c0 c0   .W...... ........
0190  db dc dd b6 ce d6 57 c4  02 d0 02 00 00 e0 f9 07   ......W. ........
01a0  00 00 00 03 00 00 00 c0  00 00 00 00 c0 db dc dd   ........ ........
01b0  c0 6e be 00 00 00 00 00  00 98 01 00               .n...... ....
```

```
2027                     10.10.10.96
2108                     10.10.10.96

JavaScript Object Notation: application/j
  Object
    Member Key: activities
    Member Key: goals
    Member Key: summary
      Object
        Member Key: activityLevels
        Member Key: calories
        Member Key: customHeartRateZones
        Member Key: distance
          Number value: 0.52
          Key: distance
        Member Key: elevation
          Number value: 0
          Key: elevation
```

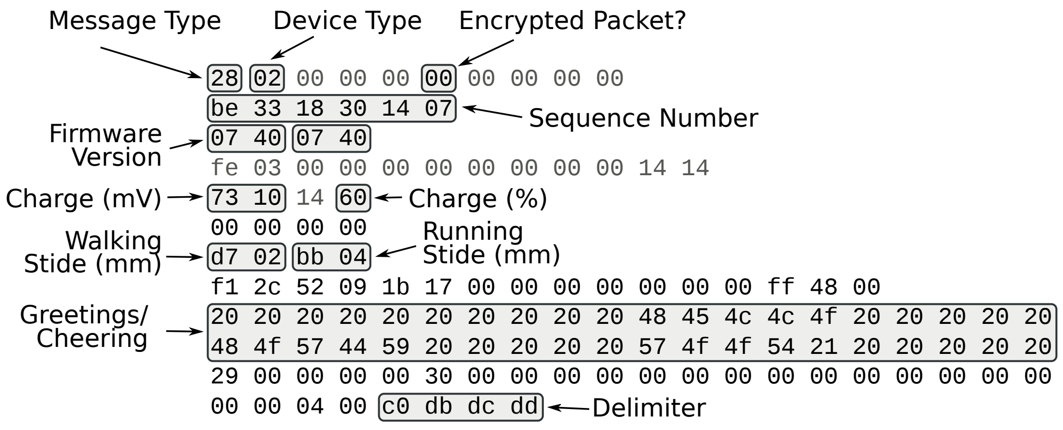
Figure 7.4: Megadump frame in plain-text format as transmitted to the Fitbit server (main window) and the human-readable JSON status response by the Fitbit Web API (top right).

Fitbit Web API separates data by type, such that not all information transmitted within one megadump is contained within one JSON response. From the megadump a total distance of 522 720 mm can be extracted, which equals to the 0.52 km from the JSON.

We use this information to reverse engineer and validate the megadump packet format, and have identified that each megadump is split into the following sections: a header, one or more *data sections*, and a footer. These sections start with a *section start* sequence of bytes: `c0 cd db dc`; and end with a *section terminator* byte: `c0`. If the byte `c0` is required to be used within a data section, it is escaped in a manner similar to RFC 1055.[2]

[2]A Non-standard for transmission of IP Data-grams over Serial Lines: SLIP

**Message Header** The megadump header is very similar to the microdump header, but contains a few differences. Figure 7.5 shows how this header is structured.



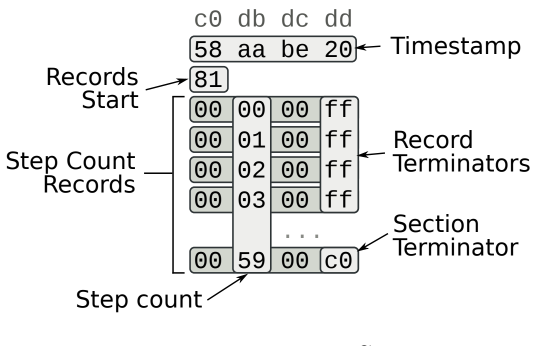
```
28 02 00 00 00 00 00 00 00 00
be 33 18 30 14 07
07 40 07 40
fe 03 00 00 00 00 00 00 00 00 14 14
73 10 14 60
00 00 00 00
d7 02 bb 04
f1 2c 52 09 1b 17 00 00 00 00 00 00 00 ff 48 00
20 20 20 20 20 20 20 20 20 20 48 45 4c 4c 4f 20 20 20 20 20
48 4f 57 44 59 20 20 20 20 20 57 4f 4f 54 21 20 20 20 20 20
29 00 00 00 00 30 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 04 00 c0 db dc dd
```

Figure 7.5: Megadump Header Structure

**Data Sections** Following the header are one or more *data sections*. Each *data section* contains various statistics in a particular format, and may even be blank. As previously mentioned, each data sections start with `c0 cd db dc`, and are terminated by a single c0 character. Therefore, the data sections are of variable length. From the packets we have analyzed, it has been observed that there are typically four data sections, which all appear in the following order, and have the following format:

(1) Daily Summary: The first data section contains activity information across a number of different absolute timestamps. This section contains a series of fixed-length records that begin with a little-endian timestamp, and end with a section terminator byte (`c0`).

(2) Per-minute Summary: The next data section is a *per-minute summary*, comprising a series of records that indicate user activity on a per-minute granularity. The structure of this data section is shown in Figure 7.6.



```
c0 db dc dd
58 aa be 20
81
00 00 00 ff
00 01 00 ff
00 02 00 ff
00 03 00 ff
      . . .
00 59 00 c0
```

Figure 7.6: Per-minute Summary

The section begins with a timestamp (unlike other timestamps, this field is big-endian), which acts as the *base* time for this sequence of step counts. Each step count record is then an increment of a time period (typically two minutes), from this base time. Following the timestamp is a byte indicating the start of the step count records. The full meaning of this byte is unclear, but we believe it indicates the time period between each step count record. Following this, a series of records consisting of four bytes state the number of steps taken per-time period. The second byte indicates the number of steps taken, and the fourth byte is either `ff` to indicate another record follows, or `c0` (for the last record) to terminate the data section.

(3) Overall Summary: This data section contains a summary of the previous records, although as will be demonstrated later it is not validated against "per-minute" or "per-day" data. The format of this section is shown in Figure 7.7.

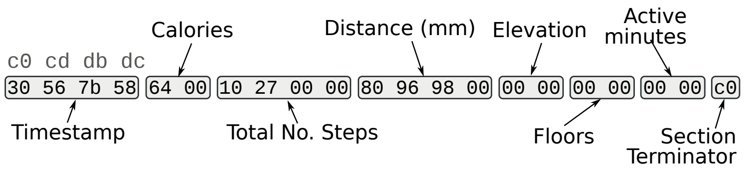


Figure 7.7: Megadump Summary Fields

This section starts with a timestamp, indicating the base time for this summary data. Following this timestamp is a 16-bit value that holds the number of calories burned. Following on from this is a 32-bit value containing the number of steps taken, and a 32-bit value containing the distance travelled in millimeters. Finally, the summary ends with elevation, floors climbed and active minutes—all 16-bit values.

(4) Alarms: The final data section contains information about what alarms are currently set on the tracker, and is typically empty unless the user has instructed the tracker to create an alarm.

**Message Footer** The megadump footer contains a checksum and the size of the payload, as shown in Figure 7.8.

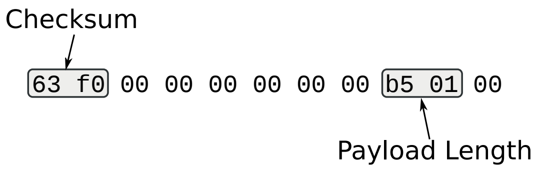


Figure 7.8: Megadump Footer Fields

## 7.3 Protocol-based Remote Spoofing

This section shows that the construction of a megadump packet containing fake information and the subsequent transmission to the Fitbit server is a viable approach for inserting fake step data into a user's exercise profile. This attack does not actually require the possession of a physical tracker, but merely a known tracker ID to be associated with the user's Fitbit account. This means that one can fabricate fake data for any known and actively used tracker having a firmware version susceptible to this vulnerability. In order to construct a forged packet, however, the format of the message must be decoded and analyzed to determine the fields that must be populated.

### 7.3.1 Submission of Fake Data

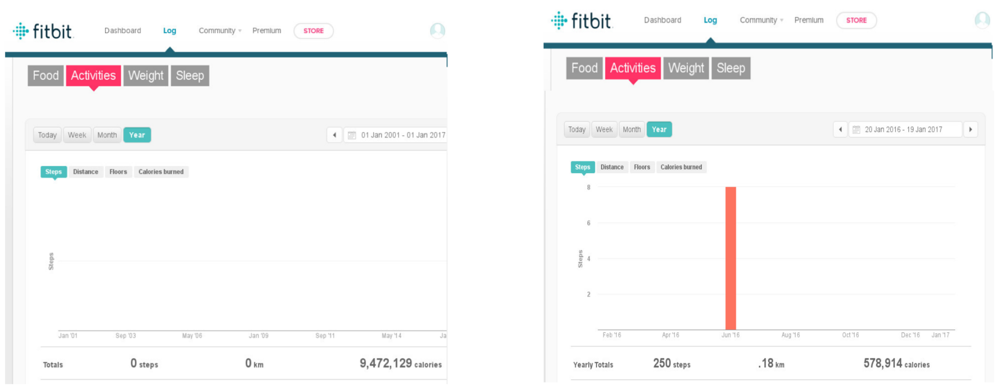
The Fitbit server has an HTTPS endpoint that accepts raw messages from trackers, wrapped in an XML description. The raw message from the tracker is Base64 encoded, and contains various fields that describe the tracker's activity over a period of time.

The raw messages of the studied trackers may or may not be encrypted, but the remote server will accept either. Even though the encryption key for a particular tracker is unknown, it is possible to construct an unencrypted frame and submit it to the server for processing, associating it with an arbitrary tracker ID. Provided that all of the fields in the payload are valid and the checksum is correct, the remote server will accept the payload and update the activity log accordingly. In order to form such a message, the raw Fitbit frame must be Base64 encoded and placed within an XML wrapper as shown in Listing 7.1:

```xml
<?xml version="1.0"?>
<galileo-client version="2.0">
 <client-info>
 <client-id>
     6de4df71-17f9-43ea-9854-67f842021e05
 </client-id>
 <client-version>1.0.0.2292</client-version>
 <client-mode>sync</client-mode>
 <dongle-version major="2" minor="5" />
 </client-info>
 <tracker tracker-id="F0609A12B0C0">
  <data>*** BASE64 PACKET DATA ***</data>
 </tracker>
</galileo-client>
```

Listing 7.1: Fitbit frame within an XML wrapper

The fabricated frame can be stored in a file, e.g. `payload`, and then submitted with the help of an HTTP `POST` request to the remote server as shown in Listing 7.2, after which the server will respond with a confirmation message.



(a) Before submission (b) After submission

Figure 7.9: The result of replaying data from another Fitbit tracker to a different tracker ID. Figure 7.9a shows the Fitbit user activity screen before the replay attack, and Figure 7.9b shows the results after the message is formed by changing the tracker ID, and submitted to the server.

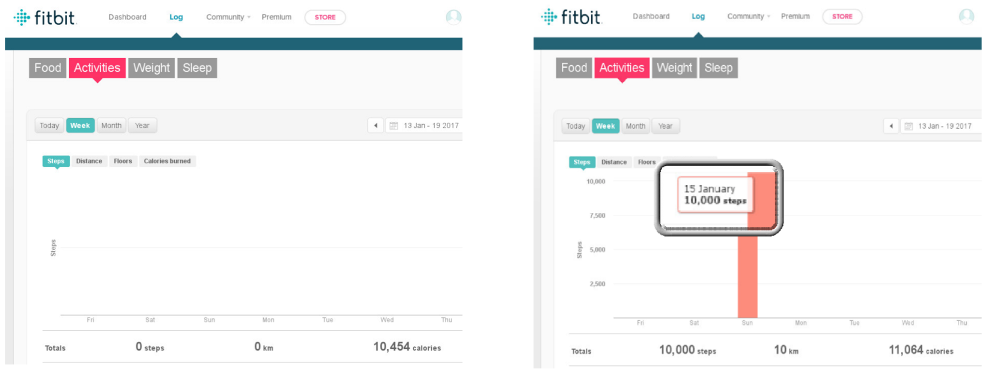
```
$ curl -i -X POST https://client.fitbit.com/tracker/client/message
-H "Content-Type: text/xml"
--data-binary @payload
```

Listing 7.2: Submitting fake payload to the server

**Impersonation Attack:**

In order to test the susceptibility of the server to this attack, a frame from a particular tracker was captured and re-submitted to the server with a *different* tracker ID. The different tracker ID was associated with a *different* Fitbit user account. The remote server accepted the payload, and updated the Fitbit user profile in question with identical information as for the genuine profile, confirming that simply altering the tracker ID in the submission message allowed arbitrary unencrypted payloads to be accepted. Figure 7.9 shows the Fitbit user activity logs before and after performing the impersonation attack. The fact that we are able to inject a data report associated to any of the studied trackers' IDs reveals both a severe DoS risk and the potential for a paid rogue service that would manipulate records on demand. Specifically, an attacker could arbitrarily modify the activity records of random users, or manipulate the data recorded by the device of a target victim, as tracker IDs are listed on the packaging. Likewise, a selfish user may pay for a service that exploits this vulnerability to manipulate activity records on demand, and subsequently gain rewards.

**Fabrication of Activity Data:** Using the information gained during the protocol analysis phase (see section 7.2), we constructed a message containing a frame with fake activity data and submitted it to the server, as discussed above. To do this, the payload of a genuine message was used as a *skeleton*, and each data section within the payload was cleared by removing all data bytes between the delimiters.



(a) Before submission (b) After submission

Figure 7.10: Figure 7.10a shows the Fitbit user activity screen before fake data were submitted, and Figure 7.10b shows the screen after the attack. In this example, 10000 steps and 10 km were injected for the date of Sunday, January 15th, 2017 by fabricating a message containing the data shown in Table 7.1.

Then, the summary section was populated with fake data. Using only the summary section was enough to update the Fitbit user profile with fabricated step count and distance traveled information. The format of the summary section is shown in Table 7.1, along with the fake data used to form the fabricated message.

| Range | Usage | Value | |
|---|---|---|---|
| `00-03` | Timestamp | `30 56 7b 58` | 15/01/17 |
| `04-05` | Calories | `64 00` | 100 |
| `06-09` | Number of Steps | `10 27 00 00` | 10000 |
| `0A-0D` | Distance in mm | `80 96 98 00` | 10000000 |
| `0E-0F` | Elevation | `00 00 00 00` | 0 |

Table 7.1: Data inserted into the packet summary section

Figure 7.10 again shows a before and after view of the Fitbit user activity screen, when the fake message is submitted. In this example, the packet is constructed so that 10000 steps and a distance traveled of 10 km were registered for the 15th of January 2017. This attack indicates that it is possible to create an arbitrary activity message and have the remote server accept it as a real update to the user's activity log.

**Exploitation of Remote Server for Field Deduction:** A particular problem with the unencrypted packets was that it was not apparent how the value of the CRC field is calculated (unlike the CRC for encrypted packets). However, if a message is sent to the server containing an invalid CRC, the server responds with a message containing information on what the correct CRC should be (see Listing 7.3).

```
$ curl -i -X POST <target-url> --data-binary @payload
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<galileo-server version="2.0">
  <error>INVALID_DEVICE_DATA:com.fitbit.protocol.serializer.
      DataProcessingException: Parsing field
   [signature] of the object of type CHECKSUM. IO error -&gt; Remote
      checksum [2246|0x8c6] and local
  checksum [60441|0xec19] do not match.</error>
</galileo-server>
```

Listing 7.3: Response from the Fitbit server when a payload with an invalid checksum is submitted.

This information can be used to reconstruct the packet with a valid CRC. Such an exploit must be used sparingly, however, as the remote server will refuse to process further messages if an error threshold is met, until a lengthy timeout (on the order of hours) expires.

## 7.4 Hardware-Based Local Spoofing

We now demonstrate the feasibility of hardware-based spoofing attacks focusing on Fitbit Flex and Fitbit One devices. We first conducted an analysis of the Fitbit protocol as previously described in Section 7.2. However, since the newest firmware (Fitbit 7.81) uses end-to-end encryption with a device-specific key, the data cannot be manipulated using MITM attacks, as described in the previous section. Therefore, we resort to a physical attack on the tracker's hardware. We reverse engineered the hardware layout of the devices to gain memory access, which enabled us to inject arbitrary stepcount values into memory, which the tracker would send as valid encrypted frames to the server.

### 7.4.1 Device Tear-Down

In order to understand how to perform the hardware attack, we needed to tear down the devices. In the following section, we give an overview of the tools required for this process.

**Tools:** The tools to perform the hardware attack were quite cheap and easy to purchase. To accomplish the attack, we used:

1. digital multimeter,
2. soldering iron, thin gauge wire, flux
3. tweezers,
4. soldering heat gun,
5. debugger/programmer in circuit ST-LINK/v2, and
6. STM32 ST-LINK utility.

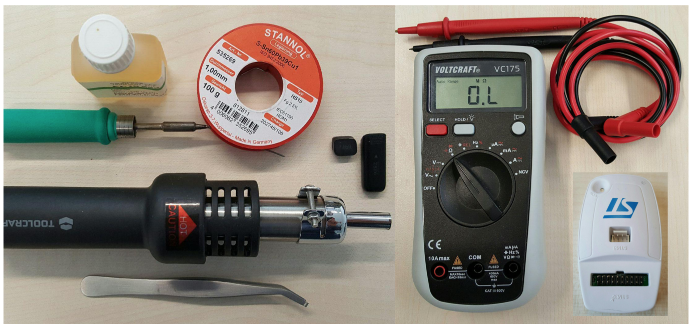


Figure 7.11: Tools for device tear-down and hardware RE.

, as shown in Figure 7.11.

The digital multimeter was used once to locate the testing pins associated with the debug interface of the microcontroller. However, attackers performing the attack do not require a multimeter, as long as the layout of the testing pins is known to them. The soldering heat gun and tweezers were utilized to perform the mechanical tear-down of the device casing. The soldering iron and accessories were used to solder wires to the identified testing pins. We used the ST-LINK/v2 and STM32 ST-LINK utilities to connect to the device in order to obtain access to the device's memory.

**Costs:**

The required tools for performing the hardware attack are relatively cheap. The STLINK/v2 is a small debugger/programmer that connects to the PC using a common mini-USB lead and costs around \$15. The corresponding STM32 ST-LINK utility is a full-featured software interface for programming STM32 microcontrollers, using a mini-USB lead. This is free Windows software and that can be downloaded from ST[3]. General-purpose tools (e.g. hair dryer) can be employed to tear-down the casing. Therefore the total costs make the attack accessible to anyone who can afford a fitness tracker. We argue that hardware modifications could also be performed by a third party in exchange of a small fee, when the end user lacks the skills and/or tools to exploit hardware weaknesses in order to obtain financial gains.

**Tear-Down Findings:** According to our tear-down of the Fitbit trackers (Fitbit Flex and Fitbit One), as shown in Figure 7.12, the main chip on the motherboard is an ARM Cortex-M3 processor. This processor is an ultra-low-power 32-bit MCU, with different memory banks such as 256KB flash, 32KB SRAM and 8KB EEPROM. The chip used for Fitbit Flex is *STM32L151UC WLCSP63* and for Fitbit One *STM32L152VC UFBGA100*. The package technology used in both micro-controllers is ball grid array (BGA) which is a surface-mount package with no leads and a grid array of solder balls underneath the integrated circuit. Since the required specifications of the micro-controller used in Fitbit trackers are freely available, we were able to perform hardware reverse-engineering (RE).

[3] `http://www.st.com/en/embedded-software/stsw-link004.html`

### 7.4.2 Hardware RE to Hunt Debug Ports

We discovered a number of testing points at the back of the device's main board. Our main goal was to identify the testing points connected to debug interfaces. According to the IC's datasheet, there are two debug interfaces available for *STM32L*: (1) SWD, and (2) JTAG.

We found that the Fitbit trackers were using the SWD interface. However, the SWD pins were obfuscated by placing them among several other testing points without the silkscreen identifying them as testing points. SWD technology provides a 2-pin debug port, a low pin count and high-performance alternative to JTAG. The SWD replaces the JTAG port with a clock and single bidirectional data pin, providing test functionality and real-time access to system memory. We selected a straightforward approach to find the debug ports (other tools that can be exploited include *Arduino+JTAGEnum* and *Jtagulator*). We removed the micro-controller from the device PCB. Afterward, using the IC's datasheet and a multimeter with continuity tester functionality, we traced the debug ports on the device board, identifying the testing points connected to them.

### 7.4.3 Connecting Devices to the Debugger

After discovering the SWD debug pins and their location on the PCB, we soldered wires to the debug pins. We connected the debug ports to ST-LINK v2 pin header, according to Figure 7.13.

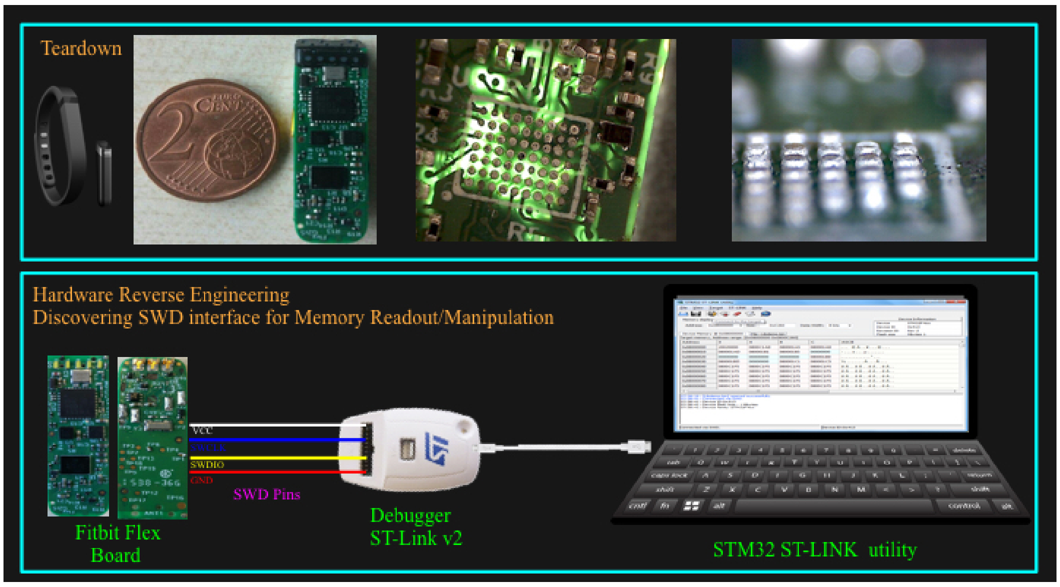
**Dumping the Firmware:** After connecting to the device micro-controller, we were able to communicate with MCU as shown in Figure 7.12. We extracted the entire firmware image since memory readout protection was not activated. There are three levels of memory protection in the STM32L micro-controller:

1. level 0: *no readout protection*,
2. level 1: *memory readout protection*, the Flash memory cannot be read from or written to, and
3. level 2: *chip readout protection*, debug features and boot in RAM selection are disabled (JTAG fuse).

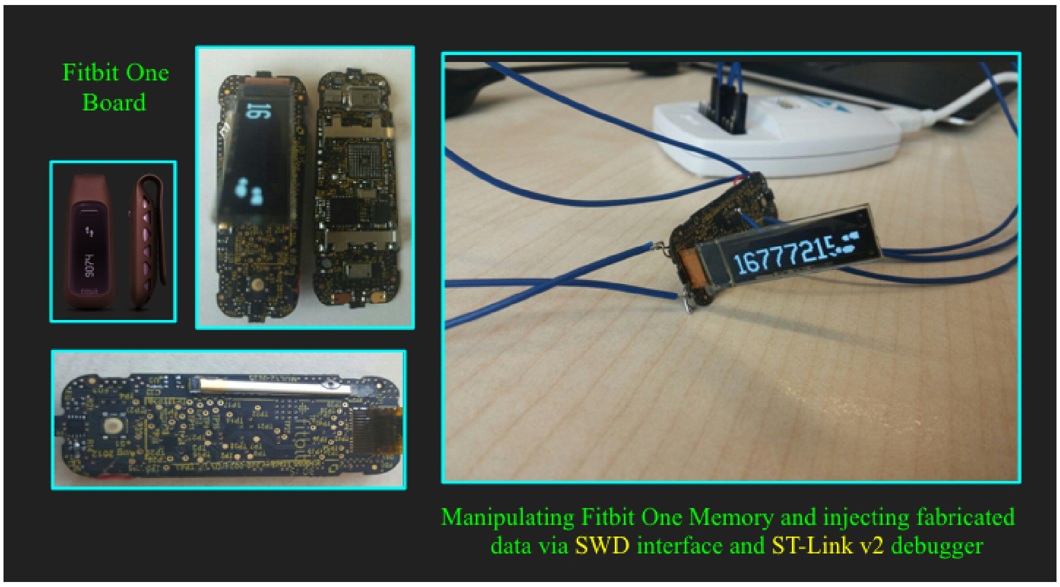
We discovered that in the Fitbit Flex and the Fitbit One, memory protection was set to *level 0*, which means there is no memory readout protection. This enabled us to extract the contents of the different memory banks (e.g., FLASH, SRAM, ROM, EEPROM) for further analysis.

Note that it is also possible to extract the complete firmware via the MITM setup during an upgrade process (if the tracker firmware does not use encryption). In general, sniffing is easier to perform, but does not reveal the memory layout and temporal storage contents. Moreover, hardware access allows us to change memory contents at runtime.

**Device Key Extraction:** We initially sniffed communications between the Fitbit tracker and the Fitbit server to see whether a key exchange protocol is performed, which was not the case. Therefore, we expected pre-shared keys on the Fitbit trackers we connected to, including two different Fitbit One and three different Fitbit Flex devices. We read out their EEPROM and discovered that the device


Teardown
Hardware Reverse Engineering
Discovering SWD interface for Memory Readout/Manipulation
VCC
SWDIO
GND
SWD Pins
Fitbit Flex Board
Debugger ST-Link v2
STM32 ST-LINK utility

(a) Fitbit Flex.


Fitbit One Board
16777215
Manipulating Fitbit One Memory and injecting fabricated data via SWD interface and ST-Link v2 debugger

(b) Fitbit One.

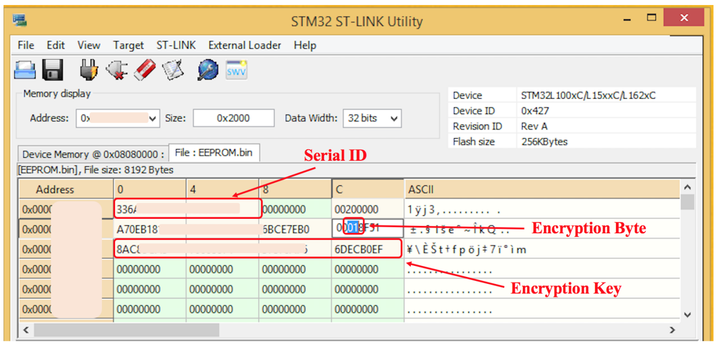
Figure 7.12: Fitbit tear-down and connecting Fitbit micro-controller to the debugger.

| ST-LINK/V2 | SWD Pins | Description |
|---|---|---|
| Pin 1 | Vcc | Target board Vcc |
| Pin 7 | SWDIO | The SWD Data Signal |
| Pin 8 | GND | Ground |
| Pin 9 | SWCLK | The SWD Clock Signal |
| Pin 15 | RESET | System Reset |

Figure 7.13: Connecting the tracker to the debugger.

encryption key is stored in their EEPROM. Exploring the memory content, we found the exact memory addresses where the 6-byte serial ID and 16-byte encryption key are stored, as shown in Figure 7.14. We confirm that each device has a *device-specific key* which likely is programmed into the device during manufacturing [29].

**Disabling the Device Encryption:** By analyzing the device memory content, we discovered that by flipping one byte at a particular address in EEPROM, we were able to force the tracker to operate in unencrypted mode and disable the encryption. Even trackers previously communicating in encrypted mode switched to plaintext after modifying the encryption flag (byte). Figure 7.14 illustrates how to flip the byte, such that the the tracker sends all sync messages in plaintext format (Base64 encoded) disabling encryption.



Memory display
Address: 0x[redacted] Size: 0x2000 Data Width: 32 bits

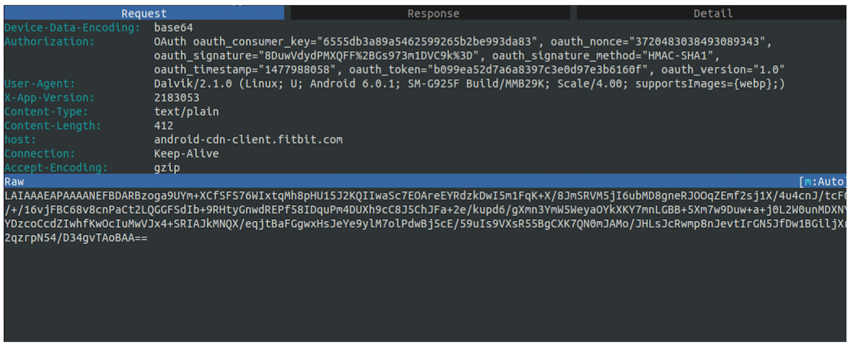
| | |
|---|---|
| Device | STM32L100xC/L15xxC/L162xC |
| Device ID | 0x427 |
| Revision ID | Rev A |
| Flash size | 256KBytes |

Device Memory @ 0x08080000 : File : EEPROM.bin
[EEPROM.bin], File size: 8192 Bytes

| Address | 0 | 4 | 8 | C | ASCII |
|---|---|---|---|---|---|
| 0x000[redacted] | 336[redacted] | [redacted] | 00000000 | 00200000 | 1ÿj3,......... . |
| 0x000[redacted] | A70EB18[redacted] | [redacted] | 5BCE7EB0 | 00018F51 | ±.§Iše˜~îkQ.. |
| 0x000[redacted] | 8AC[redacted] | [redacted] | [redacted]5 | 6DECB0EF | ¥\ÈŠt+fpöj‡7ï°ìm |
| 0x000[redacted] | 00000000 | 00000000 | 00000000 | 00000000 | ................ |
| 0x000[redacted] | 00000000 | 00000000 | 00000000 | 00000000 | ................ |
| 0x000[redacted] | 00000000 | 00000000 | 00000000 | 00000000 | ................ |

Figure 7.14: Device key extraction and disabling encryption.

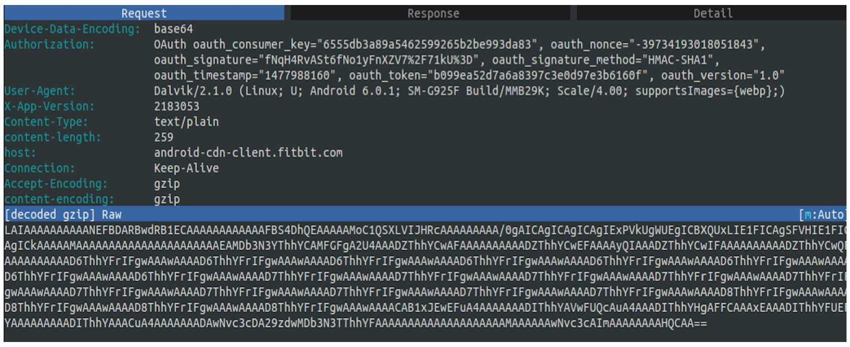
**Injecting Fabricated Data Activities:** We investigated the EEPROM and SRAM content to find the exact memory addresses where the total step count and other data fields are stored. Based on our packet format knowledge and previously sniffed megadumps, we found that the activity records were stored in the EEPROM in the same format. Even encrypted frames are generated based on the EEPROM plaintext records. Therefore, oblivious falsified data can be injected, even with the newest firmware having encryption enabled.



```
Request                              Response                              Detail
Device-Data-Encoding:  base64
Authorization:         OAuth oauth_consumer_key="6555db3a89a5462599265b2be993da83", oauth_nonce="3720483038493089343",
                       oauth_signature="8DuwVdydPMXQFF%2BGs973m1DVC9k%3D", oauth_signature_method="HMAC-SHA1",
                       oauth_timestamp="1477988058", oauth_token="b099ea52d7a6a8397c3e0d97e3b6160f", oauth_version="1.0"
User-Agent:            Dalvik/2.1.0 (Linux; U; Android 6.0.1; SM-G925F Build/MMB29K; Scale/4.00; supportsImages={webp};)
X-App-Version:         2183053
Content-Type:          text/plain
Content-Length:        412
host:                  android-cdn-client.fitbit.com
Connection:            Keep-Alive
Accept-Encoding:       gzip
Raw                                                                                                              [m:Auto]
```

```
LAIAAAEAPAAAANEFBDARBzoga9UYm+XCfSFS76WIxtqMh8pHU15J2KQIIwaSc7EOAreEYRdzkDwI5m1FqK+X/8JmSRVM5jI6ubMD8gneRJOOqZEmf2sj1X/4u4cnJ/tcF
/+/16vjFBC68v8cnPaCt2LQGGFSdIb+9RHtyGnwdREPf58IDquPm4DUXh9cC8J5ChJFa+2e/kupd6/gXmn3YmW5WeyaOYkXKY7mnLGBB+5Xm7w9Duw+a+j0L2W0unMDXN
YDzcoCcdZIwhfKwOcIuMwVJx4+SRIAJkMNQX/eqjtBaFGgwxHsJeYe9ylM7olPdwBj5cE/59uIs9VXsR55BgCXK7QN0mJAMo/JHLsJcRwmp8nJevtIrGN5JfDw1BGiljX
2qzrpN54/D34gvTAoBAA==
```

(a) Encrypted sync message before memory modification.



```
Request                              Response                              Detail
Device-Data-Encoding:  base64
Authorization:         OAuth oauth_consumer_key="6555db3a89a5462599265b2be993da83", oauth_nonce="-397341930180518443",
                       oauth_signature="fNqH4RvASt6fNo1yFnXZV7%2F71kU%3D", oauth_signature_method="HMAC-SHA1",
                       oauth_timestamp="1477988160", oauth_token="b099ea52d7a6a8397c3e0d97e3b6160f", oauth_version="1.0"
User-Agent:            Dalvik/2.1.0 (Linux; U; Android 6.0.1; SM-G925F Build/MMB29K; Scale/4.00; supportsImages={webp};)
X-App-Version:         2183053
Content-Type:          text/plain
content-length:        259
host:                  android-cdn-client.fitbit.com
Connection:            Keep-Alive
Accept-Encoding:       gzip
content-encoding:      gzip
[decoded gzip] Raw                                                                                               [m:Auto]
```

```
LAIAAAAAAAAAANEFBDARBwdRB1ECAAAAAAAAAAAAFBS4DhQEAAAAAMoC1QSXLVIJHRcAAAAAAAAA/0gAICAgICAgICAgIExPVkUgWUEgICBXQUxLIE1FICAgSFVHIE1FI
AgICkAAAAAMAAAAAAAAAAAAAAAAAAAAAAEAMDb3N3YThhYCAMFGFgA2U4AAADZThhYCwAFAAAAAAAAAAADZThhYCwEFAAAAyQIAAADZThhYCwIFAAAAAAAAAAADZThhYCwQ
AAAAAAAAAAD6ThhYFrIFgwAAAwAAAAD6ThhYFrIFgwAAAwAAAAD6ThhYFrIFgwAAAwAAAAD6ThhYFrIFgwAAAwAAAAD6ThhYFrIFgwAAAwAAAAD6ThhYFrIFgwAAAwAAAA
D6ThhYFrIFgwAAAwAAAAD6ThhYFrIFgwAAAwAAAAD7ThhYFrIFgwAAAwAAAAD7ThhYFrIFgwAAAwAAAAD7ThhYFrIFgwAAAwAAAAD7ThhYFrIFgwAAAwAAAAD7ThhYFrI
gwAAAwAAAAD7ThhYFrIFgwAAAwAAAAD7ThhYFrIFgwAAAwAAAAD7ThhYFrIFgwAAAwAAAAD7ThhYFrIFgwAAAwAAAAD7ThhYFrIFgwAAAwAAAAD8ThhYFrIFgwAAAwAAAA
D8ThhYFrIFgwAAAwAAAAD8ThhYFrIFgwAAAwAAAAD8ThhYFrIFgwAAAwAAAACAB1xJEwEFuA4AAAAAAAADIThhYAVwFUQcAuA4AAADIThhYHgAFFCAAAxEAAADIThhYFUE
YAAAAAAAAADIThhYAAACuA4AAAAAAAADAwNvc3cDA29zdwMDb3N3TThhYFAAAAAAAAAAAAAAAAAAAAAMAAAAAAAwNvc3cAImAAAAAAAAAHQCAA==
```

(b) Unencrypted sync message (Base64 encoded) after memory modification.

Figure 7.15: Disabling the Device Encryption.

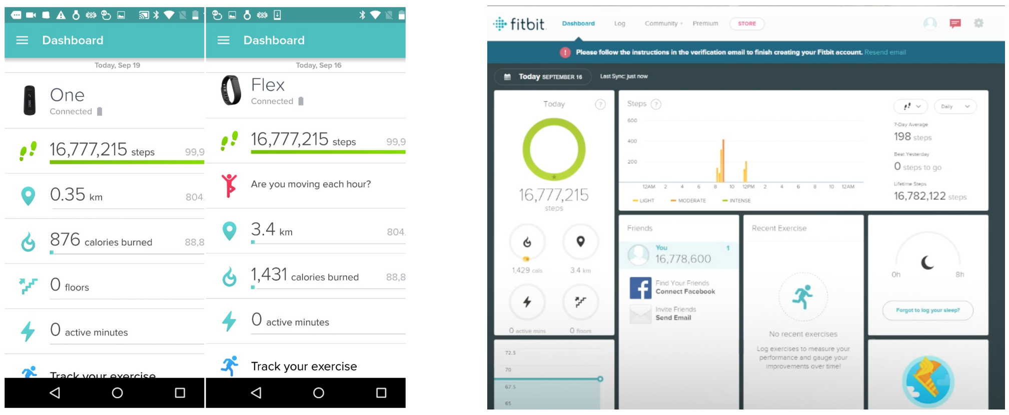
As it can be seen in Figure 7.16a and Figure 7.16b, we managed to successfully inject `0X00FFFFFF` steps equal to 16 777 215 in decimal into Fitbit server by modifying the corresponding address field in the EEPROM and subsequently synchronising the tracker with the server.

## 7.5 Discussion

In this section we give a set of implementation guidelines for fitness trackers. While Fitbit is currently the only manufacturer that puts effort into securing trackers [228], our guidelines also apply to other health-related IoT devices. We intend to transfer the lessons learned into open security and privacy standards that are being developed.[4]

False data injection as described in the previous sections is made possible by a combination of sub-optimal design choices in the implementation of the Fitbit trackers and in the communication protocol utilized between the trackers and Fitbit application servers. These design choices relate to how encryption techniques have been applied, the design of the protocol messages, and the implementation of the hardware itself. To overcome such weaknesses in future system designs, we propose the following mitigation techniques.

[4]See https://www.thedigitalstandard.org



(a) Fitbit app (b) Fitbit web interface

Figure 7.16: The results of injecting fabricated data. Figure 7.16a shows the Fitbit app screenshot, and Figure 7.16b demonstrates the Fitbit web interface.

**Application of encryption techniques:** The examined trackers support full end-to-end encryption, but do not enforce its use consistently.[5] This allows us to perform an in-depth analysis of the data synchronization protocol and ultimately fabricate messages with false activity data, which were accepted as genuine by the Fitbit servers.

**Suggestion 1.** *End-to-end encryption between trackers and remote servers should be consistently enforced, if supported by device firmware.*

**Protocol message design:** Generating valid protocol messages (without a clear understanding of the CRC in use) is enabled by the fact that the server responds to invalid messages with information about the expected CRC values, instead of a simple "invalid CRC", or a more general "invalid message" response.

**Suggestion 2.** *Error and status notifications should not include additional information related to the contents of actual protocol messages.*

CRCs do not protect against message forgery, once the scheme is known. For authentication, there is already a scheme in place to generate subkeys from the device key [29]. Such a key could also be used for message protection.

**Suggestion 3.** *Messages should be signed with an individual signature subkey which is derived from the device key.*

**Hardware implementation:** The microcontroller hardware used by both analyzed trackers provides memory readout protection mechanisms, but were not enabled in the analyzed devices. This opens an attack vector for gaining access to tracker memory and allows us to circumvent even the relatively robust protection provided by end-to-end message encryption as we were able to modify activity data directly in the tracker memory. Since reproducing such hardware attacks given the necessary background information is not particularly expensive, the available hardware-supported memory protection measures should be applied by default.

[5]During discussions we had with Fitbit, the company stressed that models launched after 2015 consistently enforce encryption in the communications between the tracker and server.

**Suggestion 4.** *Hardware-supported memory readout protection should be applied.*

Specifically, on the MCUs of the investigated tracking devices, the memory of the hardware should be protected by enabling chip readout protection level 2.

**Fraud detection measures:** In our experiments we were able to inject fabricated activity data with clearly unreasonably high performance values (e.g. more than 16 million steps during a single day). This suggests that data should be monitored more closely by the servers before accepting activity updates.

**Suggestion 5.** *Fraud detection measures should be applied in order to screen for data resulting from malicious modifications or malfunctioning hardware.*

For example, accounts with unusual or abnormal activity profiles should be flagged and potentially disqualified, if obvious irregularities are detected.

## 7.6 Related Work

In 2013, Rahman et al. [26] studied the communication between Fitbit Ultra and its base station as well as the associated web servers. According to Rahman et al., Fitbit users could readily upload sensor data from their Fitbit device onto the web server, which could then be viewed by others online. They observed two critical vulnerabilities in the communication between the Fitbit device's base station, and the web server. They claimed that these vulnerabilities could be used to violate the security and privacy of the user. Specifically, the identified vulnerabilities consisted of the use of plaintext login information and plaintext HTTP data processing. Rahman et al. then proposed FitLock as a solution to the identified vulnerabilities. These vulnerabilities have been patched by Fitbit and no longer exist on contemporary Fitbit devices.

In the report released by AV TEST [209], the authors tested nine fitness trackers including Fitbit Charge and evaluated their security and privacy. The authors tried to find out how easy it is to get the fitness data from the fitness band through Bluetooth or by sniffing the connection to the cloud during the synchronization process. AV-TEST reported some security issues in Fitbit Charge [28]. They discovered that Fitbit Charge with firmware version 106 and lower allows non-authenticated smartphones to be treated as authenticated if an authenticated smartphone is in range or has been in range recently. Also, the firmware version allowed attackers to replay the tracker synchronization process.

Zhou et al. [231] followed up on Rahman's work by identifying shortcomings in their proposed approach named FitLock, but did not mention countermeasures to mitigate the vulnerabilities that they found. In 2014, Rahman et al. published another paper detailing weaknesses in Fitbit's communication protocol, enabling them to inject falsified data to both the remote web server and the fitness tracker. The authors proposed SensCrypt, a protocol for securing and managing low power fitness trackers [232]. Note that Fitbit's communication paradigm has changed considerably since Fitbit Ultra, which uses ANT instead of Bluetooth, and is not supported by smartphone applications, but only by a Windows program last updated in 2013. Neither the ANT-based firewalls FitLock nor SensCrypt would work on recent Fitbit devices. Transferring their concept to a Bluetooth-based firewall would not help against the attacks demonstrated in this paper, since hardware attacks are one level below such firewalls, while our protocol attacks directly target the Fitbit servers.

In [29], the authors captured the firmware image of the Fitbit Charge HR during a firmware update. They reversed engineer the cryptographic primitives used by the Fitbit Charge HR activity tracker and recovered the authentication protocol. Moreover, they obtained the cryptographic key that is used in the authentication protocol from the Fitbit Android application. The authors found a backdoor in previous firmware versions and exploiting this backdoor they extracted the device specific encryption key from the memory of the tracker using Bluetooth interface.

Principled understanding of the Fitbit protocol remains open to investigation as the open-source community continues to reverse-engineer message semantics and server responses [229].

## 7.7 Summary

Trusting the authenticity and integrity of the data that fitness trackers generate is paramount, as the records they collect are being increasingly utilized as evidence in critical scenarios such as court trials and the adjustment of healthcare insurance premiums. In this paper, we conducted an in-depth security analysis of two models of popular activity trackers commercialized by `Fitbit`, the market leader, and we revealed serious security and privacy vulnerabilities present in these devices. Additionally, we reverse engineered the primitives governing the communication between these devices and cloud-based services, implemented an open-source tool to extract sensitive personal information in human-readable format and demonstrated that malicious users could inject spoofed activity records to obtain personal benefits. To circumvent the end-to-end protocol encryption mechanism present on the latest firmware, we performed hardware-based RE and documented successful injection of falsified data that appears legitimate to the Fitbit cloud. We believe more rigorous security controls should be enforced by manufacturers to verify the authenticity of fitness data. To this end, we provided a set of guidelines to be followed to address the vulnerabilities identified.

## Acknowledgments

We thank the Fitbit Security Team for their professional collaboration with us, and their availability to discuss our findings and address the vulnerabilities we identified.

# Chapter 8

# Conclusions

This chapter presents the conclusions of this thesis. We first summarize the main contributions and discuss how these contributions satisfies the objectives of the dissertation. Next, we identify and discuss a number of challenging open issues and research problems that should be tackled in future work.

As current trends indicate, the proliferation of Android malware is going to be a continued problem. As malware authors develop newer and more sophisticated means of intrusions, old methods of protection will no longer work. To this end, Android users need a security solution that is not only tailored to protect against the threats of today, but also of tomorrow as well. Android users must be extremely cautious in how they download applications. One of the best practices is not to trust third-party apps, and whatever apps users download should be scanned locally by running a mobile security suite on their devices. It is not just individual Android users who have to take Android security into account, it is businesses as well. With the increase in company BYOD policies, businesses are operating with a great deal of employee Android devices accessing their network. While BYOD has the potential to drive up productivity, it also opens the door to vulnerabilities, and it is the business enterprise's responsibility to keep threats under control and prevent them from causing problems.

As for wearable smart devices (e.g., fitness trackers), part of the problem with the security of these devices is because wearable manufacturers are rushing to beat their competitors and get their product onto the market first. Fitness tracking manufacturers need to build privacy and security into their existing development process, because patching security vulnerabilities, fixing errors and dealing with the investigation is significantly more costlier when compared to if companies had done it right the first time. When designing wearable devices connected to the IoT, manufacturers must establish a secure hardware and software development process that includes code management, build management, automated testing, streamlined packaging and software delivery mechanisms. It should include source code analysis to identify vulnerabilities as well as security-related testing to identify runtime vulnerabilities.

Deploying these devices in a secure way is just as important as their design. Attention should be paid to device provisioning and authentication. The deployment process should cross-authenticate both the device and the network to ensure it does not transmit confidential information to a malicious party. Similarly, strong encryption should be deployed to ensure the integrity and privacy of the data on the device, in the cloud or during transmission from device to cloud. Also, utilizing authentication and authorization for devices, users and applications must be taken into account.

In the following, we analyze the main conclusions and summarize our contributions in Section 8.1 and discuss open research problems and future work in Section 8.2.

## 8.1 Summary of Contribution

This dissertation portrays our contributions towards tackling mobile and smart devices security as follows.

### 8.1.1 Tackling Mobile Malware

In Part I, we illustrated an important part of our research toward tackling mobile malware with focus on Android OS. The goal of this part was to inspire malware researchers to come up with new and innovative ways of tackling mobile malware threats. The problems of malware detection are not solvable by simple iterations of existing technology. It will require novel and revolutionary approaches for detecting malicious application. In this area our contributions are threefold:

- *Secure Message Delivery Games for D2D Communications:* We design the SMD protocol in Chapter 2 with primary objective of selecting the most secure path to deliver a message from a sender to a destination in a multi-hop D2D network. In addition, we consider the energy cost and quality-of-service of each route. We formulate secure message delivery game so as to derive an optimal behavior for our protocol. Simulation results demonstrate the degree of improvement that SMD introduces as opposed to a shortest path routing protocol. This improvement has been measured in terms of the defender's expected cost as defined in SMDGs. This cost includes security expected damages, energy consumption incurred due to messages inspection, and the quality-of-service of the D2D message communications. The outcome of our contribution has been published in [13].
- *Android Code Obfuscation Techniques:* Advanced and sophisticated malware application tend to stay hidden during infection and operation to prevent removal and analysis. Malware applications achieve this using many techniques to thwart detection and analysis. In more advanced cases, the malware might attempt to subvert modern detection systems to prevent being found, hiding

running processes and network connections. Malware applications use some additional layers of defense to protect themselves from analysis and reverse engineering. By implementing additional protection mechanisms, malware can be more difficult to detect and even more resilient to takedown. Obfuscation techniques are used to hide malware's internals. To address this issue, in Chapter 3, we investigate concrete and relevant questions concerning Android code obfuscation and protection techniques and review code obfuscation and code protection practices, as well as evaluating the efficacy of existing code de-obfuscation tools. Part of the research presented in this chapter appears in [16].

- *Android Malware Detection:* In Chapter 4, we propose a system to detect Android malicious applications. To do so, we conduct an extensive static analysis on a well-labelled data-set of Android applications and extract several informative features from malware applications. We leverage several Machine Learning classification algorithms to discover the most performant one in terms of accuracy and speed. We evaluate the performance of our proposal on large-scale malware data-set (including 18,677 malware and 11,187 benign apps). Our experimental results show a true positive rate of **97.3%** and a false negative rate of **2.7%**. These results are better than what are reported by state-of-the-art Android malware detection methods. This contribution to the filed of malware detection has been publihsed in [17].

- *Android Malware Classification:* Malware authors introduce polymorphism to the malicious components to evade detection. This means that malicious apps belonging to the same malware "family", with the same forms of malicious behaviour, are constantly modified or obfuscated using various techniques, such that they look like many different applications. In order to classify malware applications effectively, we group them into 78 groups and identify their respective families. In addition, such grouping criteria applies to new malware encountered on smartphones, in order to detect them as malicious and of a certain family.
  In Chapter 5, we carry out family-by-family malware classification. Then, we accumulate Android malware apps and perform cumulative classification where the classification results are continuously updated as new malware samples are discovered. We leverage boosting techniques to obtain as much classification performance as possible for Android malware detection in the wild. In malware family classification, we obtain an average classification accuracy of 92%. We also present the empirical results for our cumulative classification which investigates how good features from old malware can contribute to the detection of new variants of both known and unknown malware. Part of the work presented in this chapter appeared in [18].

### 8.1.2 Security Analysis on Wearable Fitness Devices

In Part II of this dissertation, we analyze a representative subset of different wearable fitness tracking devices including devices from top manufacturers and less well-known brands with divers security mechanism (ranging form the most secure fitness trackers to less secure ones). We reveal that many challenges need to be addressed in order to develop consistent, robust, flexible, safe and secure systems. This thesis provides two contributions as follows:

- *Security Analysis, Reverse Engineering and Spoofing Popular Fitness Devices:* In Chapter 6, we conduct hardware and software security analysis of these devices and related communication protocols to understand how they work internally. we focus on a malicious user setting that aims to inject false data into the cloud-based services leading to erroneous data analytics. We primarily consider attacks aimed to either (i) disclose fitness data, or (ii) tamper with data in order to inject counterfeit information into the cloud service. We conduct a Man-In-The-Middle (MITM) attack that targets the communication between the fitness App (installed on smartphones) and the manufacturer's cloud service, as fitness trackers typically utilise the user's smartphone for uploading data to the cloud service. The attack is successful in all tested activity trackers and consequence of the mentioned attack is to inject counterfeit information into the cloud service.
  Quite a few fitness products include End-to-End (On-device) encryption to protect data from being sniffed or tampered. Since the data is End-to-End encrypted, it cannot be manipulated using MITM attack and to compromise the data, hardware access is needed. We deal with such devices in chapter 7. We show that none of these products can provide data integrity, authenticity and confidentiality. As a result of our research, we discover a number of vulnerabilities in selected fitness tracking products. These vulnerabilities could be potentially used by malicious users to fabricate false activity record data and upload them to the respective cloud services. We published the outcome of this work in [25].

- *Security Analysis, Reverse Engineering and Spoofing Fitbit Devices:* We provide an in-depth security analysis of the operation of fitness trackers commercialized by Fitbit, the wearables market leader in Chapter 7. We reveal an intricate security through obscurity approach implemented by the user activity synchronization protocol running on these devices. Although non-trivial to interpret, we reverse engineer the message semantics, demonstrate how falsified user activity reports can be injected, and argue that based on our discoveries, such attacks can be performed at scale to obtain financial gains.
  Regarding the studied Fitbit products, we found that they are susceptible to impersonation attack in which activity reports are modified by changing the tracker ID to a different tracker ID associated with a different Fitbit user account. In addition, we have also discovered that it is possible to modify the contents of activity reports in transit in a way that would allow an attacker to fabricate arbitrary activity information. In both cases, the modified activity reports are accepted by the Fitbit servers as genuine. Last but not least, our security analysis of the tracker hardware revealed details about the internal state of the tracker device, which would potentially allow an attacker to directly modify activity information in tracker memory and thereby fabricate activity information. The activity reports are accepted by the Fitbit servers

as genuine, even if end-to-end encryption between the tracker and the server is in-place. Finally, we give guidelines for avoiding similar vulnerabilities in future system designs. Part of the research work conducted in this chapter is published in [32].

## 8.2 Open Issues and Future Work

The research work presented in Part I of this dissertation aims to overcome some of the limitations that affect current malware analysis and detection solutions. However, the techniques we described are not free from limitations. In this section we sketch possible improvements and extensions over the ideas we proposed, together with some directions for future work. Our future contributions include but not limited to:

- **Android Obfuscation Future Research Directions:** The Android devices have constrained processing and limited storage. Obfuscation techniques have an adverse impact on battery consumption. The power management is an important issue to identify impact of code level modifications. The Android Obfuscation has a APK statistical significance [149]. An important future work is to consider a large set of obfuscated APK empirical evaluation. The same can be extended to different mobile OS and devices. Since the developers do not have access to tools like CARAT [150], they cannot identify the impact on energy consumption. The ability to identify the impact is important for resource constrained Android devices.
  The existing academic code obfuscation research is heavily concentrated more towards analysis of obfuscated malicious applications [152] [153] [154] [155]. The relevant literature evaluates obfuscation techniques prominently among malicious applications. The real identification of obfuscated code among the normal programs which is important for software protection, is ignored. The non-malicious code reverse engineering is largely unexplored. Targeting program obfuscation and related techniques for protecting the digital rights is an interesting future direction. In spite of the existing research on obfuscation, evaluation matrices to verify the existing obfuscation technique resilience are not available. Formal analysis techniques to evaluate obfuscation and de-obfuscation techniques is still not available. Hence, we summarize code obfuscation, de-obfuscation tools and techniques to understand the effect in isolation. It would be interesting to combine different class of obfuscation techniques, and evaluate existing de-obfuscation tools.

- **Extending Android Applications Analysis**: We will extend our analysis and combine static analysis with dynamic analysis to overcome to limitation associated with static analysis that we already explained in Chapter 4. We plan to leverage more fine-grained features, and structured data (e.g., string, set, and graph) to characterize Android apps. We will extract more features related to behavior of Android applications such as CPU and Memory consumption, Network traffic activities, Inter-Process Communications (IPC) and system calls made by applications so as to interact with Android OS.

We also plan to concentrate more on tackling the problem of automatically detection of evasive malware. One of the most problematic classes of malware is evasive malware. These malware applications are considered as one of the most problematic malicious apps. An example of evasive malware, we can mention logic bombs which are executed, or triggered, only under certain circumstances. These malicious software applications are written with the specific intent of evading currently analysis systems and this aspect makes the automatic detection an open research problem. We intend to focus more on analyzing this class of malware to devise a performant malware detection framework.

- **Deep Learning for Malware detection and classification**: Deep Learning as a new frontier in data mining and machine learning, is starting to be used in automatic malware detection. A multilayer deep learning architecture takes advantage of superior ability in feature learning and takes control of the learning difficulty via layerwise pretraining. A deep learning model exploits of feature detectors from the lowest level to the highest level to construct the final classification model [233].
  We will construct deep neural networks and apply them to hunt Android malicious applications. We leverage *Convolutional Neural Networks (CNN)*, and *Recurrent Neural Network (RNN)* using architectures such as: (i) Long Short Term Memory (LSTM), and (ii) Gated Recurrent Unit (GRU). These deep neural networks have shown state-of-the-art performance in various field such as Computer Vision, Pattern Recognition, and Natural language Processing (NLP). To increase the model performance and optimize malware detection architecture, we will combine the above-mentioned deep neural networks

- **Malware Detection in IoT Smart Devices:** The experience obtained from Android-powered smartphones recommends that malware will also hit other smart IoT devices. There will be a new type of threat in which adjacent IoT devices will infect each other with a worm that will spread explosively over large areas [234]. Recently, Linux Malware has clearly started to target IoT Devices. Mirai malware enabled the largest DDoS attack ever by targeting not just Linux servers, but also IoT devices on the Internet.[1]

- **Security and Privacy Analysis of Smartwatches**: In chapters 6 and 7, we evaluated security of fitness tracking devices like smart bands rather than more general-purpose smartwatches. While fitness tracking bands in general are limited in terms of computational power, memory, connectivity and functionality, smartwatches are more complex and powerful devices. Most of smartwatches on the market use the most advanced hardware and software solutions available for mobile operating system that was developed specifically for wearable devices. Therefore, these devices can leverage more robust, safe and efficient security controls. As a future work, we are interested in conducting an in-depth security and privacy analysis of smartwatches.

[1] https://github.com/jgamblin/Mirai-Source-Code

# Bibliography

[1] Giovanni Russello, Mauro Conti, and Bruno Crispo andand Earlence Fernandes. Moses: Supporting operation modes on smartphones. *Proceedings of the17th ACM Symposium on Access Control Models and Technologies*, pages 3–12, 3-12, Newark, NJ, US, June 20-22, 2012.

[2] Eurograbber. `http://www.checkpoint.com/products/downloads/whitepapers/Eurograbber`. white paper.pdf, 2013.

[3] Earlence Fernandes, Bruno Crispo, and Mauro Conti. Fm 99.9, radio virus: Exploiting fm radio broadcasts for malware deployment. In *In IEEE Transactions on Information Forensics and Security*, 2013.

[4] Roman Schlegel, Kehuan Zhang, Xiao yong Zhou, Mehool Intwala, Apu Kapadia, , and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proceedings of the NDSS*, 2011.

[5] V. Laxmi V. Ganmoor M.S. Gaur M. Conti P. Faruki, A. Bharmal and M. Rajarajan. Android security: A survey of issues, malware penetration, and defenses. *IEEE Communications Surveys & Tutorials*, pages 998–1022, 2015.

[6] Emmanouil Panaousis, Tansu Alpcan, Hossein Fereidooni, and Mauro Conti. Secure message delivery games for device-to-device communications. In *In Proceedings of 5th International Conference, on Decision and Game Theory for Security*. IEEE, Los Angeles, CA, USA, November 6-7, 2014.

[7] Hui Y. Wei Z. Jianting Y., Chuan M. Secrecy-based access control for device-to-device communication underlaying cellular networks. *IEEE Communications Magazine*, pages 2068–2071, 2013.

[8] A.L. Fakoorian S.A.A. Wei X. Chunming Z Daohua Z., Swindlehurst. Device-to-device communications: The physical layer security advantage. *IEEE Communications Magazine*, pages 1606–1610, 2014.

[9] Parvez Faruki, Hossein Fereidooni, Vijay Laxmiand, Manoj Singh Gaur, and Mauro Conti. Android code protection via obfuscation techniques: Past, present and future directions. *ACM Computing Surveys (CSUR).*

[10] Hossein Fereidooni, Mauro Conti, Alessandro Sperduti, and Danfeng Yao. Anastasia: Android malware detection using static analysis of applications. In *In Proceedings of 8th IFIP International Conference on New Technologies, Mobility & Security*. IEEE, Cyprus, 21-23th November 2016.

[11] Hossein Fereidooni, Veelasha Moonsamy, Mauro Conti, and Lejla Batina. Efficient classification of android malware in the wild using robust static features. *In Protecting Mobile Networks and Devices: Challenges and Solutions, CRC Press - Taylor & Francis*, (Editors: Weizhi Meng, Xiapu Luo, Jianying Zhou, Steven Furnell), 2016.

[12] L. Lee, S. Egelman, J. H. Lee, and D. Wagner. Risk perceptions for wearable devices. In *arXiv pre-print arXiv:1504.05694*, 2015.

[13] X. Liu, Z. Zhou, W. Diao, Z. Li, and K. Zhang. When good becomes evil: Keystroke inference with smartwatch. In *In Conference on Computer and Communications Security, ACM*, page 1273âĂŞ1285, 2015.

[14] A. Maiti, M. Jadliwala, J. He, and I Bilogrevic. (smart)watch your taps: side-channel keystroke inference attacks using smartwatches. In *In International Symposium on Wearable Computers, ACM*, page 27âĂŞ30, 2015.

[15] A. Migicovsky, Z. Durumeric, J. Ringenberg, and J. A. Halderman. Outsmarting proctors with smartwatches: A case study on wearable computing security. In *In Financial Cryptography and Data Security. Springer*, page 89âĂŞ96, 2014.

[16] Forbes Magazin. `http://www.forbes.com/sites/parmyolson/2014/11/16/fitbit-data-court-room-personal-injury-claim/#a169f02209f8`, [Online]; Accessed June 2016.

[17] Daily News. `http://www.nydailynews.com/news/national/police-attorneys-fitness-trackers-court-evidence-article-1.2607432`, [Online]; Accessed June 2016.

[18] Hossein Fereidooni, Tommaso Frassetto, Markus Miettinen, Ahmad-Reza Sadeghi, and Mauro Conti. Fitness trackers: Fit for health but unfit for security and privacy. In *In Proceedings of the 2nd IEEE International Workshop on Safe, Energy-Aware, & Reliable Connected Health (CHASE 2017 workshop: SEARCH 2017)*. IEEE, Philadelphia, USA, July 17-19, 2017.

[19] Mahmudur Rahman, Bogdan Carbunar, and Madhusudan Banik. Fit and Vulnerable: Attacks and Defenses for a Health Monitoring Device. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, Bloomington, Indiana, USA, July 2013.

[20] Britt Cyr, Webb Horn, Daniela Miao, and Michael Specter. Security analysis of wearable fitness devices (fitbit). Technical report, Massachusetts Institute of Technology, USA. Available at: `https://courses.csail.mit.edu/6.857/2014/files/17-cyrbritt-webbhorn-specter-dmiao-hacking-fitbit.pdf`, Accessed April 2017.

[21] Eric Clausing, Michael Schiefer, and Maik Morgenstern. AV-TEST Analysis of Fitbit Vulnerabilities. Available at: `https://www.av-test.org/fileadmin/pdf/avtest_2016-04_fitbit_vulnerabilities.pdf`, Accessed June 2016.

[22] Maarten Schellevis, Bart Jacobs, , and Carlo Meijer. Security/privacy of wearable fitness tracking IoT devices. Radboud University. Bachelor thesis: Getting access to your own Fitbit data., August 2016.

[23] Accenture. Digital trust in the IoT era, 2015.

[24] PwC. Use of wearables in the workplace is halted by lack of trust. `http://www.pwc.co.uk/who-we-are/regional-sites/northern-ireland/press-releases/use-of-wearables-in-the-workplace-is-halted-by-lack-of-trust-pwc-research.html`, Accessed June 2016.

[25] Hossein Fereidooni, Jiska Classen, Tom Spink, Paul Patras, Markus Miettinenand Ahmad-Reza Sadeghi, Matthias Hollick, and Mauro Conti. Breaking fitness records without moving: Reverse engineering and spoofing fitbit. In *In Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2017)*. IEEE, Atlanta, Georgia, USA, 2017, *(Submitted)*.

[26] Lu-L. Yuan-Wu Y. Ye Li G. Li S. Feng G. Feng, D. Device-to-device communications in cellular networks. *IEEE Communications Magazine*, pages 49–55, 2014.

[27] Dahlman-E. Mildh-G. Parkvall S. Reider N. Miklos G. Turanyi Z. Fodor, G. Design aspects of network assisted device-to-device communications. *IEEE Communications Magazine*, pages 170–177, 2012.

[28] Ito-M. Kato-N. Nishiyama, H. Relay-by-smartphone: realizing multihop device-to-device communications. *IEEE Communications Magazine*, pages 56–65, 2014.

[29] Rinne-M. Wijting-C. Ribeiro C.B. Hugl K Doppler, K. Device-to-device communication as an underlay to lte-advanced networks. *IEEE Communications Magazine*, pages 42–49, 2009.

[30] Conti-M. Leone-M. Stefa J. Ardagna, C.A. An anonymous end-to-end communication protocol for mobile cloud environments. *IEEE Transactions on Services Computing*, 2014.

[31] F-Secure. Bluetooth-worm:symbos/cabir. Technical report, accessed June 2016.

[32] Courtney T.-Sanders W. H. Stevens F. Van Ruitenbeek, E. Quantifying the effectiveness of mobile phone virus response mechanisms. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 790–800, 2007.

[33] K.G. Bose, A. Shin. On mobile viruses exploiting messaging and bluetooth services. In *Proceedings of the ecurecomm and Workshops*, pages 1–10, 2006.

[34] Martinelli F.-Sgandurra D. La Polla, M. A survey on security for mobile devices. *IEEE Communications Surveys and Tutorials*, pages 446–471, 2013.

[35] Halonen P. Hatonen-K. Miettinen, M. Host-based intrusion detection for advanced mobile devices. In *Proceedings of the 20th International Conference on Advanced Information Networking and Applications (AINA)*, pages 72–76, 2006.

[36] Basar T. Alpcan, T. Network security: A decision and game-theoretic approach.

[37] Schapire R.E. Freund, Y. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, pages 119–139, 1997.

[38] J.F. Nash. Equilibrium points in n-person games. In *Proceedings of the National Academy of Sciences*, volume 36, pages 48–49, 1950.

[39] Olsder G. J. Basar, T. Dynamic noncooperative game theory. London Academic press, 2nd Edition, 1995.

[40] Maltz D.A. Johnson, D.B. Dynamic source routing in ad hoc wireless networks.

[41] Yu W. Han-Z. Liu K.J.R. Sun, Y.L. Information theoretic framework of trust modeling and evaluation for ad hoc networks. *IEEE Journal on Selected Areas of Communication*, 24:305–317, 2006.

[42] Ji Z. Liu-K.J.R. Yu, W. Securing cooperative ad-hoc networks under noise and imperfect monitoring: strategies and game theoretic analysis. *IEEE Transactions on Information Forensics and Security*, 2:240–253, 2007.

[43] Liu K.J.R. Yu, W. Game theoretic analysis of cooperation stimulation and security in autonomous mobile ad hoc networks. *IEEE Transactions on Mobile Computing*, 6:507–52, 2007.

[44] Liu K.J.R. Yu, W. Secure cooperation in autonomous mobile ad-hoc networks under noise and imperfect monitoring: a game-theoretic approach. *IEEE Transactions on Information Forensics and Security*, 3:317–330, 2008.

[45] Buttyan L. Felegyhazi, M. and J.-P. Hubaux. Nash equilibria of packet forwarding strategies in wireless ad hoc networks. *IEEE Transactions on Mobile Computing*, pages 463–476, 2006.

[46] Chen I.R. Feng-P.G. Cho, J.H. Effect of intrusion detection on reliability of mission-oriented mobile group systems in mobile ad hoc networks. *IEEE Transactions on Reliability*, 59:231–241.

[47] Debbabi M. Assi-C. Bhattacharya P. Otrok, H. A cooperative approach for analyzing intrusions in mobile ad hoc networks. In *Proceedings of the International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 48–49, 2007.

[48] Saranyan R. Senthil-K.P. Vetriselvi V. Santosh, N. Cluster based co-operative game theory approach for intrusion detection in mobile ad-hoc grid. In *Proceedings of the International Conference on Advanced Computing and Communications (ADCOM)*, pages 273–278, 2008.

[49] Park J. M. Patcha, A. A game theoretic approach to modeling intrusion detection in mobile ad hoc networks. In *Proceedings of the 5th Annual IEEE SMC Information Assurance Workshop*, pages 280–284, Switzerland (2004).

[50] Park J. M Patcha, A. A game theoretic formulation for intrusion detection in mobile ad hoc networks. *International Journal of Network Security*, 2:131–137, 2006.

[51] Comaniciou C. Man-H. Liu, Y. A bayesian game approach for intrusion detection in wireless ad hoc networks. In *Proceedings of the of the Workshop on Game theory for communications and networks (GameNets)*, 2006.

[52] Comaniciou C. Man-H. Liu, Y. Modeling misbehaviour in ad hoc networks: A game theoretic approach for intrusion detection. *International Journal of Security and Networks*, 1:243–254, 2006.

[53] Politis C. Panaousis, E.A. A game theoretic approach for securing aodv in emergency mobile ad hoc networks. In *Proceedings of the 34th IEEE Conference on Local Computer Networks (LCN)*, pages 985–992, Switzerland (2009).

[54] Tripathi R. Marchang, N. A game theoretical approach for efficient deployment of intrusion detection system in mobile ad hoc networks. In *Proceedings of the International Conference on Advanced Computing and Communications (ADCOM)*, pages 60–464, 2007.

[55] The Guardian. Three graphs to stop smartphone fans fretting about "market share.". `http://www.theguardian.com/technology/2014/jan/09/market-sharesmartphones-iphone-android-windows`, [Online]; Accessed 17 Sept., 2014.

[56] Techology Research Gartner. Worldwide traditional pc, tablet, ultramobile and mobile phone shipments are on pace to grow 6.9 percent in 2014. `http://www.gartner.com/newsroom/id/2692318`, [Online]; Accessed 17 Sept., 2014.

[57] Google Play. All your entertainment, anywhere you go. `http://googleblog.blogspot.co.uk/2012/03/introducing-google-play-all-your.html`, [Online]; Accessed January, 2015.

[58] SlideME. All your entertainment, anywhere you go. `http://slideme.org/`, [Online]; Accessed, Nov. 2014.

[59] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M.S. Gaur, M. Conti, and M. Rajarajan. Android security: A survey of issues, malware penetration, and defenses. *Communications Surveys Tutorials, IEEE*, 17(2):998–1022, Secondquarter 2015.

[60] Carlos A. Castillo. Android Malware Past, Present, and Future. Technical report, Mobile Working Security Group McAfee, 2012.

[61] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting Runtime Data in Android Applications for Identifying Malware and Enhancing Code Analysis. Technical report, EC SPRIDE, 2015.

[62] Michael Franz. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms*, NSPW '10, pages 7–16, New York, NY, USA, 2010. ACM.

[63] Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. XIFER: A software diversity tool against code-reuse attacks. In *ACM International Workshop on Wireless of the Students, by the Students, for the Students*, August 2012.

[64] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. Efficient privilege de-escalation for ad libraries in mobile apps. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 89–103, New York, NY, USA, 2015. ACM.

[65] Mark Rogers. Dendroid malware can take over your camera, record audio, and sneak into google play. `https://blog.lookout.com/blog/2014/03/06/dendroid/`, [Online]; Accessed October, 2014.

[66] Contagio Minidump. Backdoor.AndroidOS.Obad.a. `http://contagiominidump.blogspot.in/2013/06/backdoorandroid\osobada.html`, [Online]; Accesed December, 2014.

[67] Google Android. Proguard | android developers. `http://developer.android.com/tools/help/proguard.html`, [Online]; accessed September, 2015.

[68] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy'12*, pages 95–109, 2012.

[69] Appbrains. Number of available android apps. `http://www.appbrain.com/stats/number-of-android-apps`, [Online]; Accessed January, 2015.

[70] Giovanni Russello, Arturo Blas Jimenez, Habib Naderi, and Wannes van der Mark. Firedroid: Hardening security in almost-stock android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 319–328, New York, NY, USA, 2013. ACM.

[71] Parvez Faruki, Vijay Ganmoor, Vijay Laxmi, M. S. Gaur, and Ammar Bharmal. Androsimilar: Robust statistical feature signature for android malware detection. In *Proceedings of the 6th International Conference on Security of Information and Networks*, SIN '13, pages 152–159, New York, NY, USA, 2013. ACM.

[72] Yury Zhauniarovich. *Improving the Security of the Android Ecosystem.* PhD thesis, University of Trento, April 2014.

[73] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4(4):2:1–2:36, January 2008.

[74] Android. Class to Dex Conversion with Dx. `http://developer.android.com/tools/help/index.html`, Online;Last Accessed March 5 2013.

[75] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. Dexhunter: Toward extracting hidden code from packed android applications. In GÃijnther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl, editors, *ESORICS (2)*, volume 9327 of *Lecture Notes in Computer Science*, pages 293–311. Springer, 2015.

[76] Mila Dalla Preda. *Code Obfuscation and Malware Detection by Abstract Interpretation.* Phd thesis, Universit'a degli Studi di Verona, Dipartimento di Informatica, 2007.

[77] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. `https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf`, [Online]; 2002.

[78] Paul C. van Oorschot. Revisiting software protection. In Colin Boyd and Wenbo Mao, editors, *ISC*, volume 2851 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[79] Douglas Low. Java control flow obfuscation. Technical report, [Online]; 1998.

[80] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *In Proceedings of the 12th Working Conference on Reverse Engineering*, pages 45–54. IEEE Computer Society, 2005.

[81] Hannes Schulz. Automated de-obfuscation of android bytecode. Master's thesis, Technische UniversitÃďt MÃijnchen, July 2014.

[82] Collberg. Obfuscation techniques for enhancing software security. `www.patents.com/us-6668325.html`, [Online]; Accessed June 24., 2015.

[83] Egil Aspevik Martinsen. Detection of junk instructions in computer viruses. Master's thesis, Technische UniversitÃďt MÃijnchen, July 2008.

[84] Aleksandrina Kovacheva. Efficient code obfuscation for android. In Borworn Papasratorn, Nipon Charoenkitkarn, Vajirasak Vanijja, and Vithida Chongsuphajaisiddhi, editors, *IAIT*, volume 409 of *Communications in Computer and Information Science*, pages 104–119. Springer, 2013.

[85] Mark Stamp and Wing Wong. Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3):211–229, December 2006.

[86] Michael Batchelder and Laurie J. Hendren. Obfuscating java: The most pain for the least gain. In Shriram Krishnamurthi and Martin Odersky, editors, *CC*, volume 4420 of *Lecture Notes in Computer Science*, pages 96–110. Springer, 2007.

[87] Michael R. Batchelder. Java bytecode obfuscation. Master's thesis, School of Computer Science, McGill University, MontrÌĄeal, 2007.

[88] Sable Mcgill. Java obfuscation techniques. `www.sable.mcgill.ca/JBCO/examples.html`, [Online]; Accessed June 24., 2015.

[89] freepatents. FPO IP Research and Communities. `http://www.freepatentsonline.com/`, [Online]; 2015.

[90] Marjanne Plasmans. White-box cryptography for digital content protection. Master's thesis, TECHNISCHE UNIVERSITEIT EINDHOVEN, July 2005.

[91] Sonali Gupta. Obfuscating data structures. `http://palisade.plynt.com/issues/2005Sep/code-obfuscation-continued`, [Online]; September 2005.

[92] Gregory Wroblewski. General method of program code obfuscation. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.9052`, [Online]; 2002.

[93] DexGuard. Dexguard, android dalvik executable protector. `https://www.saikoa.com/dexguard`, [Online]; Accessed September, 2015.

[94] Strazzare. ANDROID HACKER PROTECTION LEVEL. `https://www.defcon.org/images/defcon-22/\dc-22-presentations/Strazzere-Sawyer/DEFCON-22-Strazzere-and-Sawyer-Android-Hacker-Protection-\Level-\UPDATED.pdf`, 2014.

[95] Erik Ramsgaard Wognsen, Henrik SÃÿndberg Karlsen, Mads Chr. Olesen, and RenÃľ Rydhof Hansen. Formalisation and analysis of dalvik bytecode. *Science of Computer Programming*, 92, Part A(0):25 – 55, 2014. Special issue on Bytecode 2012.

[96] Erik Ramsgaard Wognsen. Static analysis of dalvik bytecode and reflection in android. Master's thesis, Software Engineering, Aalborg University, 2012.

[97] Jonathan Crussell, Clint Gibler, and Hao Chen. Andarwin: Scalable detection of semantically similar android applications. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security âĂŞ ESORICS 2013*,

volume 8134 of *Lecture Notes in Computer Science*, pages 182–199. Springer Berlin Heidelberg, 2013.

[98] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. Adrob: examining the landscape and impact of android application plagiarism. In Hao-Hua Chu, Polly Huang, Romit Roy Choudhury, and Feng Zhao, editors, *MobiSys*, pages 431–444. ACM, 2013.

[99] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. Appink: Watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIACCS '13, pages 1–12, New York, NY, USA, 2013. ACM.

[100] APKTool. Reverse Engineering with ApkTool. `https://code.google.com/android/apk-tool`, [Online]; Accessed March 20, 2015.

[101] Heqing Huang, Sencun Zhu, Peng Liu, and Dinghao Wu. A framework for evaluating mobile app repackaging detection algorithms. In *Trust and Trustworthy Computing - 6th International Conference, TRUST 2013, London, UK, June 17-19, 2013. Proceedings*, pages 169–186, 2013.

[102] Symantec. Remote Access Tool Takes Aim with Android APK Binder. http://www.symantec.com/connect/blogs/remote-access-tool-takes-aim-android-apk-binder, [Online]; Accessed September, 2015.

[103] Axelle Apvrille and Ruchna Nigam. Obfuscation in android malware, and how to fight back. `https://www.virusbtn.com/pdf/magazine/2014/vb201407-Android-obfuscation.pdf`, [Online]; Accessed January, 2016.

[104] Patrick Schulz. Code protection in android. `https://net.cs.uni-bonn.de/fileadmin/user_upload/plohmann/2012-Schulz-Code_Protection_in_Android.pdf`, [Online], Available 2012.

[105] Strazzere. Dex education: Practicing safe dex. `http://www.strazzere.com/papers/DexEducation-PracticingSafeDex.pdf`, [Online]; Available.

[106] Axelle Apvrille. Angecryption: Hide android applications in images. `https://www.blackhat.com/docs/eu-14/materials/eu-14-Apvrille-Hide-Android-Applications-In-Images-wp.pdf`,, [Online]; 2014.

[107] Karen Kent Frederick. Network intrusion detection signature. `http://securityfocus.org/infocus/1553`, [Online]; 2002.

[108] Patrick Schulz. Android bytecode obfuscation. `http://dexlabs.org/blog/bytecode-obfuscation`, [Online]; Accessed November, 2015.

[109] Allatori. Allatori obfuscator. `http://www.allatori.com/doc.html`, [Online]; Accessed November, 2015.

[110] Patrick Schulz. Dalvik-obfuscator project github page. `https://github.com/thuxnder/dalvik-obfuscator`, Available Online 2012.

[111] Tim Strazzere. Apkfuscator project github page. `https://github.com/strazzere/APKfuscator`, [Online]; Accessed September, 2015.

[112] Google. Art and dalvik | android open source project. `https://source.android.com/devices/tech/dalvik/`, [Online]; accessed June 2016.

[113] SecNeo. The professional service provider for the mobile application security. `http://www.bangcle.com/`, [Online]; accessed September, 2015.

[114] APK Protect. Apk protect: Android apk security protection. `http://www.apkprotect.com/`, [Online]; accessed September, 2015.

[115] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *SOFTWARE ENGINEERING, IEEE TRANSACTIONS ON*, 28(8):735–746, 2002.

[116] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108, 2014.

[117] Min Zheng, Patrick P. C. Lee, and John C. S. Lui. ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-virus Systems. In *DIMVA*, pages 82–101, 2012.

[118] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin T. S. Chan. On tracking information flows through JNI in android applications. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 180–191, 2014.

[119] Android tools: Adb, dx, aapt, emulator, dumpstate, monkey, logcat, sqlite3, ptrace. `http://elinux.org/Android_Tools`, [Online]; Accessed September, 2015.

[120] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Evaluation of android anti-malware techniques against dalvik bytecode obfuscation. In *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, China, September 24-26, 2014*, pages 414–421, 2014.

[121] Android. Android Open Source Project. Android sdk. `http://developer.android.com/sdk/index.html`, [Online]; Accessed November, 2015.

[122] smali. Smali Dalvik bytecode assembler. `http://code.google.com/p/smali/`, [Online]; Accessed October, 2015.

[123] DexGuard. How does dalvikvm handle switch and try smali code. `http://stackoverflow.com/questions/14100992/how-does-dalvikvm-handle-switch-and-try-smali-code`, [Online]; Accessed November, 2015.

[124] Desnos. Androguard, android static analysis tool. `http://code.google.com/p/androguard/`, [Online]; Accessed May, 2015.

[125] ded: Decompiling Android Applications. `http://siis.cse.psu.edu/ded/`, [Online]; Accessed September, 2015.

[126] Dex2Jar. Android Decompiling with Dex2jar. `http://code.google.com/p/dex2jar/`, Online;Last Accessed May 15 2013.

[127] Damien Octaeu, Patrick McDaniel, and Somesh Jha. DARE: Dalvik Retargeting. http://siis.cse.psu.edu/dare/, [Online]; Accessed September, 2015.

[128] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting Android applications to Java bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 6. ACM, 2012.

[129] Gabor Paller. Dedexer. `http://dedexer.sourceforge.net/`, [Online]; accessed September, 2015.

[130] JEB Decompiler. http://www.android-decompiler.com/, Online; Last Accessed 11th February, 2016 2013.

[131] HexRays. IDA Pro, Disassembler. `http://www.hex-rays.com/products/ida/index.shtml`, [Online]; Accessed March, 2015.

[132] Dexter Labs. Dexter, android static analysis tool. `http://dexter.dexlabs.org/`, [Online]; Accessed November, 2015.

[133] Jurrian Bremer. Automated deobfuscation of android applications. `http://jbremer.org/automated-deobfuscation-of-android-applications/`, [Online]; Accessed October, 2015.

[134] QuantumG and Mike Van Emmerik. Unix-like reverse engineering framework and commandline tool. `https://github.com/radare/radare2`, [Online]; Accessed September, 2014.

[135] QuantumG and Mike Van Emmerik. A general, open source, retargetable decompiler of machine code programs. `http://boomerang.sourceforge.net/index.php`, [Online]; Accessed 17 Sept., 2014.

[136] Jurrian bremer. Abusing dalvik beyond recognition. `http://jbremer.org/wp-posts/AbusingDalvikBeyondRecognition.pdf`, In Hack.Lu, 2013.

[137] Axelle Apvrille. Playing hide and seek with dalvik executables. `http://www.fortiguard.com/uploads/general/hidex_insomni.pdf.`, In Hack.Lu, 2013.

[138] Axelle Apvrille. Playing hide and seek with dalvik executables. `Hidex. https://github.com/cryptax/dextools/tree/master/hidex`, In Hack.Lu, 2013.

[139] Karim O. Elish, Xiaokui Shu, Danfeng (Daphne) Yao, Barbara G. Ryder, and Xuxian Jiang. Profiling user-trigger dependence for android malware detection. *Comput. Secur.*, 49(C):255–273, March 2015.

[140] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer. Google android: A comprehensive security assessment. *IEEE Security and Privacy*, 8(2):35–44, 2010.

[141] Timothy Vidas, Daniel Votipka, and Nicolas Christin. All your droid are belong to us: A survey of current android attacks. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[142] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[143] G. Suarez-Tangil, J.E. Tapiador, P. Peris-Lopez, and A. Ribagorda. Evolution, detection and analysis of malware for smart devices. *Communications Surveys Tutorials, IEEE*, 16(2):961–987, Second 2014.

[144] Rowena Harrison. Investigating the Effectiveness of Obfuscation Against Android Application Reverse Engineering. Technical report, Information Security Group, Royal Holloway University of London, 2015.

[145] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM.

[146] Lookout Inc. Current World of Mobile Threats. Technical report, Lookout Mobile Security, 2013.

[147] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv.*, 49(1):4:1–4:37, April 2016.

[148] Ruchna Nigam Axelle Apvrille. Obfuscation in android malware, and how to fight back. `http://www.strazzere.com/papers/DexEducation-PracticingSafeDex.pdf`, [Online]; Accessed September, 2015.

[149] Cagri Sahin, Mian Wan, Philip Tornquist, Ryan McKenna, Zachary Pearson, William G.J. Halfond, and James Clause. How does code obfuscation impact energy usage? *Journal of Software: Evolution and Process*, 2016. To Appear.

[150] Ella Peltonen, Eemil Lagerspetz, Petteri Nurmi, and Sasu Tarkoma. Constella: Crowdsourced system setting recommendations for mobile devices. *Pervasive and Mobile Computing*, 26:71 – 90, 2016. Thirteenth International Conference on Pervasive Computing and Communications (PerCom 2015).

[151] Mario Linares VÃąsquez, Andrew Holtzhauer, Carlos Bernal-CÃąrdenas, and Denys Poshyvanyk. Revisiting Android reuse studies in the context of code obfuscation and library usages. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 242–251. ACM, 2014.

[152] Junyuan Zeng, Yangchun Fu, Kenneth A. Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 487–498, New York, NY, USA, 2013. ACM.

[153] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A security framework for the analysis and design of software attestation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer 38; Communications Security*, CCS '13, pages 1–12, New York, NY, USA, 2013. ACM.

[154] Luyi Xing, Xiaorui Pan, Rui Wang, Kan Yuan, and XiaoFeng Wang. Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 393–408, Washington, DC, USA, 2014. IEEE Computer Society.

[155] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 353–368, Washington, D.C., 2013. USENIX.

[156] Fakeinstaller leads the attack on android phones. `https://blogs.mcafee.com/mcafee-labs/fakeinstaller-leads-the-attack-on-android-phones`, Online; 2012.

[157] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime data in android applications for identifying malware and enhancing code analysis. Technical Report TUD-CS-2015-0031, EC SPRIDE, February 2015.

[158] Yongfeng Li, Tong Shen, Xin Sun, Xuerui Pan, and Bing Mao. *Security and Privacy in Communication Networks: 11th International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015*, chapter Detection, Classification and Characterization of Android Malware Using API Data Dependency, pages 23–40. Springer International Publishing, Cham, 2015.

[159] N. Idika and A. P. Mathur. A survey of malware detection techniques. *Purdue University*, 2007.

[160] FHugo Gascon, Fabian Yamaguchi, and Daniel Arp. Structural detection of android malware using embedded call graphs. booktitle =.

[161] Zarni Aung and Win Zaw. Permission-based android malware detection. *Int. J. of Scientific and Technology Research.*, pages 228–234, 2013.

[162] Naser Peiravian and Xingquan Zhu. Machine learning for android malware detection using permission and api calls. In *Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 300–305, IEEE Computer Society Washington, DC, USA, 2013.

[163] Grace Michael, Zhou Yajin, Zhang Qiang, Zou Shihong, and Jiang Xuxian. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[164] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. *2012 IEEE Symposium on Security and Privacy (SP)*, pages 95–109, 20-23 May, 2012.

[165] Daniel Arp, Michael Spreitzenbarth, Malte HÃijbner, Hugo Gascon, and Konrad Rieck. Drebin: Efficient and explainable detection of android malware in your pocket. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS)*, February 2014.

[166] VirusTotal. `http://www.virustotal.com`.

[167] M0droid. `http://m0droid.netai.net/modroid/`.

[168] Seo, Gupta, Mohamed Sallam, Bertino, and Yim K. Detecting mobile malware threats to homeland security through static analysis. *Journal of Network and Computer Applications*, pages 43–53, 2014.

[169] Scikit-learn. `https://github.com/scikit-learn/`.

[170] Tianqi Chen. `https://github.com/dmlc/xgboost`.

[171] Yoav Freund and E. Scha. Experiments with a New Boosting Algorithm. In proceedings of the Thirteenth International Conference, 1996.

[172] Leo Breiman. Random Forests. University of California, Berkeley, Journal of Machine Learning, Volume 45 Issue 1, Pages 5 - 32, October, 2001.

[173] V. Vapnikp. The Nature of Statistical Learning Theory. Springer-Verlag, NY, 1995.

[174] D. Kibler D. W. Aha and M. K. Albert. Instance-Based Learning Algorithms. Machine Learning, 6:3766, 1991.

[175] Keras. `http://keras.io/#keras-deep-learning-library-for-theano-and-ten,`.

[176] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. pages 86–103, Sydney, NSW, Australia, September 25-28, 2013.

[177] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P.G. Bringas, and G. Alvarez. Puma: permission usage to detect malware in android. In *Proceedings of the CISIS'12-ICEUTE'12-SOCO'12 Special Sessions*, 2012.

[178] Saurabh Oberoi, Weilong Song, and Amr M. Youssef. Androsat: Security analysis tool for android applications. *The 8th International Conference on Emerging Security Information, Systems and Technologies*, 2014.

[179] Hyunjae Kang, Jae wook Jang, Aziz Mohaisen, and Huy Kang Kim. Detecting and classifying android malware using static analysis along with creator information. *International Journal of Distributed Sensor Networks*, 2015.

[180] Kevin Allix, TegawendÃľ F. BissyandÃľ, Quentin JÃľrome, Jacques Klein, Radu State, and Yves Le Traon. Empirical assessment of machine learning-based malware detectors for android. *Journal of Empirical Software Engineering*, 2014.

[181] Dong Jie Wu, Ching Hao Mao, Te En Wei, Hahn Ming Lee, and Kuo Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. pages 62–69, Washington, DC, USA, 2012.

[182] Ryo Sato, Daiki Chiba, and Shigeki Goto. Detecting android malware by analyzing manifest files. *Proceedings of the Asia-Pacific Advanced Network*, pages 23–31, 2013.

[183] G. Dini A. Saracino, D. Sgandurra and F. Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. January, 2016.

[184] S. Khan K. Tam M. Ahmadi J. Kinder S. K. Dash, G. Suarez-Tangil and L. Cavallaro. Droidscribe: Classifying android malware based on runtime behavior. San Jose, CA, USA, May 22-26, 2016.

[185] Zhaoguo Wang Yibo Xue Zhenlong Yuan, Yongqiang Lu. Droid-sec: deep learning in android malware detection. In *In Proceedings of the 2014 ACM conference on SIGCOMM, pp 371-372.*, 2014.

[186] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the 6th European Workshop on Systems Security (EuroSec) Prague, Czech Republic*, April 14, 2013.

[187] A.A. Samra, O.A. Ghanem, and Kangbin Yim. Analysis of clustering technique in android malware detection. In *Proceedings of the IMIS*, pages 729–733, 2013.

[188] Kevin Allix, TegawendÃľ F. BissyandÃľ, Jacques Klein, and Yves Le Traon. Are your training datasets yet relevant?. *Proceedings of the 7th International Symposium, ESSoS 2015*, pages 51–67, Milan, Italy, March 4-6, 2015.

[189] Karim Elish, Danfeng Yao, and Barbara Ryder. On the need of precise inter-app icc classification for detecting android malware collusions. In *Proceedings of the Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy*, San Jose, CA. May 2015.

[190] Britton Wolfe, Karim Elish, and Danfeng Yao. High precision screening for android malware with dimensionality reduction. *Proceedings of the 13th International Conference on Machine Learning and Applications (ICMLA'14)*, Detroit, MI. Dec. 2014.

[191] Mansour Ahmadi Johannes Kinder Giorgio Giacinto Guillermo Suarez-Tangil, Santanu Kumar Dash and Lorenzo Cavallaro. Droidsieve: Fast and accurate classification of obfuscated android malware. In *In Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy (CODASPY '17), pp 309-320.*, Scottsdale, Arizona, USA, March 22-24, 2017.

[192] D. Makrushin M. Garnaeva, V. Chebyshev and A. Ivanov. `https://securelist.com/analysis/quarterly-malware-reports/69872/it-threat-evolution-in-q1-2015/`, IT Threat Evolution in Q1 2015. [Online]; Accessed June, 2016.

[193] S. Etalle B. P. S. Rocha, M. Conti and B. Crispo. Hybrid static-runtime information flow and declassification enforcement. *IEEE Transactions on Information Forensics & Security*, pages 1294–1305, 2013.

[194] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. *IEEE Symposium on Security and Privacy*, pages 95–109, San Francisco, CA, USA, 2012.

[195] M. H D. Arp, M. Spreitzenbarth.

[196] T.E. Wei H.M. Lee D.J. Wu, C.H. Mao and Wu K. Droidmat: Android malware detection through manifest and api calls tracing. pages 62–69, Tokyo, Japan, August 2012.

[197] J. Sahs and L. Khan. A machine learning approach to android malware detection. In *Proceedings of the EISIC*, pages 141–147, 2012.

[198] U. Zurutuza I. Burguera and S. Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. pages 15–26, Illinois, USA, October 2011.

[199] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. Andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, pages 161–190, 2012.

[200] J. R. D. Alas R. J. Tolentino K.J. Abela, D. K. Angeles and M. A. Gomez. An automated malware detection system for android using behavior-based analysis - amda. *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, pages 1–11, 2013.

[201] F. Ge M. Zhao, T. Zhang and Z. Yuan. Robotdroid: A lightweight malware detection framework on smartphones. *Journal of Networks*, pages 1–8, 2012.

[202] X. Ugarte-Pedrero C. Laorden J. Nieves B. Sanz, I. Santos and P. G. Bringas. Anomaly detection using string analysis for android malware detection. pages 1–10, Salamanca, Spain, September 2013.

[203] A. Saracino F. Martinelli and D. Sgandurra. Classifying android malware through subgraph mining. pages 1–15, Egham, UK, September 2013.

[204] Min Zheng, Mingshen Sun, and John C. S. Lui. Droidanalytics: A signature based analytic system to collect, extract, analyze and associate android malware. 2013.

[205] D. Chiba R. Sato and S. Goto. Detecting android malware by analyzing manifest files. pages 23–31, 2013.

[206] V. Khobaragade D. Koundel, S. Ithape and R. Jain. Malware classification using naives bayes classifier for android os. *International Journal of Engineering And Science (IJES)*, pages 59–63, 2014.

[207] C. Laorden-X. Ugarte-Pedrero J. Nieves P.G. Bringas B. Sanz, I. Santos and G. Álvarez. Mama: Manifest analysis for malware detection in android. *Cybernetics and Systems, Intelligent Network Security and Survivability*, pages 469–488, 2013.

[208] Andrew Hilts, Christopher Parsons, and Jerey Knockel. Every step you fake: A comparative analysis of fitness tracker privacy and security. Technical report, Open Effect Report. Available at: `https://openeffect.ca/reports/Every_Step_You_Fake.pdf`, 2016.

[209] Eric Clausing, Michael Schiefer, and Maik Morgenstern. Internet of Things: Security Evaluation of nine Fitness Trackers. AV TEST, The Independent IT-Security institue, Magdeburg, Germany, June 2015.

[210] Margaritelli: Nike+ fuelband se ble protocol reversed, Accessed April 2017.

[211] Jakob Rieck. Attacks on fitness trackers revisited: A case-study of unfit firmware security. Technical report, Available at: `http://subs.emis.de/LNI/Proceedings/Proceedings256/33.pdf`, Accessed July 2016.

[212] M Coppola. Attacks on fitness trackers revisited: A case-study of unfit firmware security. Technical report, Available at: `http://poppopret.org/2013/06/10/summercon-2013-hacking-the-withings-ws-30/`, Accessed June 2016.

[213] MD5 2104. `https://www.ietf.org/rfc/rfc1321.txt`, [Online]; Accessed April 2017.

[214] SHA RFC. `https://tools.ietf.org/html/rfc3174`, [Online]; Accessed April 2017.

[215] HMAC RFC. https://tools.ietf.org/html/rfc2104, [Online]; Accessed April 2017.

[216] WiFi Access Point. https://github.com/oblique/create_ap, [Online]; Accessed June 2016.

[217] MITM Proxt. http://docs.mitmproxy.org/en/latest/mitmproxy.html, [Online]; Accessed June 2016.

[218] Fitsed. http://pub.ks-and-ks.ne.jp/cycling/fitsed.shtml, [Online]; Accessed June 2016.

[219] Protocol Buffers Message Editor. https://developers.google.com/protocol-buffers, [Online]; Accessed July 2016.

[220] Wueest C. Barcena, M. B. and H. Lau. How safe is your quantified self? Technical report, Available at: https://www.symantec.com/content/dam/symantec/docs/white-papers/how-safe-is-your-quantified-self-en.pdf, Accessed April 2017.

[221] Forbes. Wearable tech market to be worth $34 billion by 2020. https://www.forbes.com/sites/paullamkin/2016/02/17/wearable-tech-market-to-be-worth-34-billion-by-2020, February 2016.

[222] International Data Corporation. Worldwide quarterly wearable device tracker. https://www.idc.com/tracker/showproductinfo.jsp?prod_id=962, March 2017.

[223] Mashable. Husband learns wife is pregnant from her Fitbit data. http://mashable.com/2016/02/10/fitbit-pregnant/, Feb. 2016.

[224] The Wall Street Journal. Prosecutors say Fitbit device exposed fibbing in rape case. http://blogs.wsj.com/law/2016/04/21/prosecutors-say-fitbit-device-exposed-fibbing-in-rape-case/, April 2016.

[225] VitalityHealth. https://www.vitality.co.uk/rewards/partners/activity-tracking/.

[226] AchieveMint. https://www.achievemint.com.

[227] StepBet. https://www.stepbet.com/.

[228] Markus Miettinen-Ahmad-Reza Sadeghi Hossein Fereidooni, Tommaso Frassetto and Mauro Conti. Fitness trackers: Fit for health but unfit for security and privacy. In Proc. IEEE International Workshop on Safe, Energy-Aware, & Reliable Connected Health (CHASE workshop: SEARCH 2017), in press, Philadelphia, Pennsylvania, USA, July 17-19, 2017., 2017.

[229] Galileo project. https://bitbucket.org/benallard/galileo/.

[230] Wireshark network protocol analyzer. https://www.wireshark.org/.

[231] S. Zhou, W.; Piramuthu. Security/privacy of wearable fitness tracking iot devices. In *Proceedings of the 9th Iberian Conference on IEEE Information Systems and Technologies (CIST)*, pages 1–5, 2014.

[232] Mahmudur Rahman, Bogdan Carbunar, and Umut Topkara. Secure Management of Low Power Fitness Trackers. Published in IEEE Transactions on Mobile Computing, Volume 15 Issue 2, Pages 447-459, February 2016.

[233] W. Kang-Z. Li Y. Lv, Y. Duan and F. Wang. Traffic flow prediction with big data: A deep learning approach. In IEEE Transactions on Intelligent Transportation Systems, Vol 16(2), pp 865-873., 2015.

[234] Eyal Ronen, Colin O'Flynn, Adi Shamir, and Achi-Or Weingarten. Iot goes nuclear: Creating a zigbee chain reaction. *IACR Cryptology ePrint Archive*, 2016:1047, 2016.