Head Office: Università degli Studi di Padova

Human Inspired Technology Research Centre

Ph.D. COURSE IN: Brain, Mind & Computer Science (BMCS)
CURRICULUM: Computer Science and Innovation for Societal Challenges
XXX SERIES

Security and Privacy Threats on Mobile Devices through Side-Channels Analysis

Coordinator: Prof. Giuseppe Sartori
Supervisor: Prof. Mauro Conti
Co-Supervisor: Prof. Luciano Gamberini and Prof. Giuseppe Sartori

Ph.D. student : Riccardo Spolaor

University of Padua

Human Inspired Technology Research Centre

Doctorate Degree in Brain, Mind and Computer Science
Curriculum in Computer Science and Innovation for Societal Challenges

Security and Privacy Threats on Mobile Devices through Side-Channels Analysis

Candidate
Riccardo Spolaor

Supervisor
Prof. Mauro Conti

Co-Supervisors
Prof. Luciano Gamberini
Prof. Giuseppe Sartori

University of Padova, Italy

October 31st, 2017

Acknowledgements

I want to express my sincere gratitude to my advisor Prof. Mauro Conti for his valuable support, motivation, for letting me discover the amazing world of research, and for guiding me with his inexhaustible enthusiasm since the very beginning of my research career. It is an honor working with him and I really hope to keep learning from him in my future scientific career.

Thanks also to my Ph.D. co-advisors Luciano Gamberini and Giuseppe Sartori. Despite they are from another research field, they taught me that psychology and computer science have strong boundings.

A sincere thank to my parents Renzo and Sonia, my sister Martina, and my grandma Lina. They supported me throughout my life and they always encouraged me to follow my aspirations. Without their support, I would have never achieved this goal.

A special thanks to my girfriend and colleague QianQian Li, for her patient and continuous encouragement to improve myself as a researcher and as a man.

I would like to sincerely thank Prof. Ivan Martinovic from University of Oxford for being a precious mentor for me, especially during my visiting period in his university.

Also, I would like to thank Prof. Radha Poovendran, Prof. Christina Popper, Prof. Marco Ferrante, Prof. Lejla Batina, Prof. Cristiano Giuffrida, Prof. Luigi Vincenzo Mancini, and Dr. Michele Nati for accompanying me during important periods of my Ph.D. studies. Additionally, I would like to give a special thank to the members of the Ph.D. Thesis Committee for the valuable comments and suggestions they have provided.

I would like to thank my friends, collegues and officemates at the University of Padua (Moreno Ambrosin, Riccardo Lazzeretti, Daniele Lain, Luca Pasa, Ankit Gangwal, Alberto Compagno, and Franchino P.) for the help, stimulating discussions and beautiful (life) moments that we have had in the last three years. Moreover, a big thank to the other members of the SPRITZ

Research Group (Gabriele Tolomei, Tooska Dargahi, Giuseppe Cascavilla, Hossein Fereidooni, Fabio De Gaspari, Masoom Rabbani, Muhammad Hassan Khan, Pallavi Kaliyar, Sarada Prasad Gochhayat, Chhagan Lal, and Vinod P.) and other collegues of BM&CS Ph.D. school (Merylin Monaro, Mirko Polato, Dinh Van Tran, Reza Khosh Kangini, and Kiyana Bahadori).

A special thanks to the PostDocs and Ph.D. students that collaborated with me and helped me during these years, in particular, Veelasha Moonsamy, Nino Vincenzo Verde, Vincent F. Taylor, Michal Piskozub, Katharina Kohl, and Valeriu Daniel Stanciu.

I would like to thank the numerous MSc and BCs thesists and friends that I had the pleasure to supervise (Elia Dal Santo, Lorenzo Bordoni, Claudio Guarisco, Alberto Maragno, Stefano Sandonà, Riccardo Bonafede, Enrico Rotundo, Maddalena Favaretto, Marco Baesso, Giulia Calzavara, Laila Abudahi, Pasquale Capuozzo, and Zahra Alimadadi). I honestly feel that I learned from them more than they learned from me.

I would also want to thank my collegues of SSL Research group at University of Oxford (Martin Strohmeier, Bushra AlAhmadi, Richard Baker, Simon Birnbach, Simon Eberz Giulio Lovisotto, Michal Piskozub, Marc Roeschlin, Ivo Sluganovic, Matt Smith, Vincent F. Taylor, and Christian Vaas) for their friendly support during my visiting period in Oxford.

Last but not least, I want to thank my long-time friends Davide, Mariachiara, Ludovico, Luigi, Jenny, and Carlo.

Riccardo Spolaor

Padua, October 31st, 2017

Abstract

In recent years, mobile devices (such as smartphones and tablets) have become essential tools in everyday life for billions of people all around the world. Users continuously carry such devices with them and use them for daily communication activities and social network interactions. Hence, such devices contain a huge amount of private and sensitive information. For this reason, mobile devices become popular targets of attacks. In most attack settings, the adversary aims to take local or remote control of a device to access user sensitive information. However, such violations are not easy to carry out since they need to leverage a vulnerability of the system or a careless user (i.e., install a malware app from an unreliable source). A different approach that does not have these shortcomings is the side-channels analysis. In fact, side-channels are physical phenomenon that can be measured from both inside or outside a device. They are mostly due to the user interaction with a mobile device, but also to the context in which the device is used, hence they can reveal sensitive user information such as identity and habits, environment, and operating system itself. Hence, this approach consists of inferring private information that is leaked by a mobile device through a side-channel. Besides, side-channel information is also extremely valuable to enforce security mechanisms such as user authentication, intrusion and information leaks detection.

This dissertation investigates novel security and privacy challenges on the analysis of side-channels of mobile devices. This thesis is composed of three parts, each focused on a different side-channel: (i) the usage of network traffic analysis to infer user private information; (ii) the energy consumption of mobile devices during battery recharge as a way to identify a user and as a covert channel to exfiltrate data; and (iii) the possible security application of data collected from built-in sensors in mobile devices to authenticate the user and to evade sandbox detection by malware.

In the first part of this dissertation, we consider an adversary who is able to eavesdrop the network traffic of the device on the network side (e.g., controlling a WiFi access point). The fact that the network traffic is often encrypted makes the attack even more challenging. Our work proves that it is possible to leverage machine learning techniques to identify user activity and apps installed on mobile devices analyzing the encrypted network traffic they produce. Such insights are becoming a very attractive data gathering technique for adversaries, network administrators, investigators and marketing agencies.

In the second part of this thesis, we investigate the analysis of electric energy consumption. In this case, an adversary is able to measure with a power monitor the amount of energy supplied to a mobile device. In fact, we observed that the usage of mobile device resources (e.g., CPU, network capabilities) directly impacts the amount of energy retrieved from the supplier, i.e., USB port for smartphones, wall-socket for laptops. Leveraging energy traces, we are able to recognize a specific laptop user among a group and detect intruders (i.e., user not belonging to the group). Moreover, we show the feasibility of a covert channel to exfiltrate user data which relies on temporized energy consumption bursts.

In the last part of this dissertation, we present a side-channel that can be measured within the mobile device itself. Such channel consists of data collected from the sensors a mobile device is equipped with (e.g., accelerometer, gyroscope). First, we present DELTA, a novel tool that collects data from such sensors, and logs user and operating system events. Then, we develop MIRAGE, a framework that relies on sensors data to enhance sandboxes against malware analysis evasion.

Contents

Chapter 1

Introduction

Nowadays, mobile devices like smartphones and tablet have spread rapidly becoming more and more pervasive; this is due to their feature-richness, mobility, and affordable price. In fact, market studies reported that in 2011 mobile device sales surpassed those of desktop PCs [97]. At the time of writing, smartphone and tables remain the most used handheld devices. Such outbreaking success is mostly due to the fact that these devices embed advanced operating systems and allow users to perform a variety of tasks, which in the past would have been only possible on desktop PCs. In order to carry out such tasks, mobile platforms make available to users a plethora of multi-purpose applications, commonly known as apps. Since mobile devices are also equipped with networking capabilities, many apps rely on Internet access to provide extra functionalities and fresh contents. In addition to this, other mobile devices such as laptops are becoming year after year more popular than desktop workstations [188], since they offer portability in addition to equivalent computational capabilities.

Users continuously rely on mobile devices in their everyday life to perform even the most sensitive tasks from the point of view of privacy. For example, users can perform tasks such as private messaging, manage appointments, purchase of goods or services, finance or bank operations. Mobile devices became a receptacle for personal information. Moreover, with the recent diffusion of cloud services, users private data are kept updated and available anytime and anywhere from all the devices that belong to a user.

Unfortunately, this worldwide technological success of mobile devices and their access to users private information have not gone unnoticed by malicious users. Despite some of them could be also motivated by political interests, attackers are mostly motivated to obtain financial advantages from their malicious behavior. Such malicious behavior may either directly hinder users' financial assets or sell users' private information to interested third-parties. To perpetrate their malicious behaviors, attackers can adopt different approaches, which can be either active or passive

On one hand, in an active attack, the adversary's main target is to gain control of the mobile device in order to exfiltrate information she aims at. In most scenarios, the attacker relies on vulnerabilities exposed by flaws on the operating system or application software. Alternatively, adversaries leverage also distracted users, or users without sufficient knowledge of security. Indeed, another attack method consists of inducing the victim to install malware via untrusted or repackaged apps. Active attacks are the most suitable choice for the adversaries that want since immediately access to stolen information. Luckily, attackers may not always succeed in their malicious intent without being detected and neutralized, since such violations have to meet requirements that are beyond their control. In fact, active attacks are only possible when the victim does not adopt good security practices or does not deploy any prevention mechanisms (e.g., anti-viruses software). Nevertheless, even the most motivated attackers have to invest a lot of effort and money in order to discover novel zero-day vulnerabilities.

On the other hand, passive attacks do not face the shortcomings described above. In this kind of attacks, the adversary passively collects data leaked by mobile devices through side-channels. We can define a side-channel as a physical phenomenon that can be measured, and it is due to user actions or to processes running in a mobile device. In particular, with *user actions* we mean the operations actively performed by a user, such as sending a message, reading the news, or watching a video. It is worth noticing that, despite some common issues, our definition of side-channel differs from the traditional one given in cryptanalysis. In particular, we use such channels to infer knowledge about the user and the mobile device itself, rather than focus on inferring a cryptographic key. Side-channels analysis is the ideal approach for adversaries that aim to not raise suspiciousness in the victim and remain undetected, because she does not intervene directly on the mobile device. Besides, the adversary can analyze side-channels information to inferring valuable insights about the user and the mobile device itself.

1.1 Research Motivation and Contribution

The multi-purpose capabilities offered by mobile devices are inducing users to develop a symbiotic bound with them, which is becoming stronger year by year. Users rely on such devices to carry out sensitive tasks and to store even the most private information. At the same time, the usage of mobile devices generates side-channels from which an attacker can infer such information. In this dissertation, we investigate on side-channels analysis techniques and applications in the field of security and privacy. Besides, we not only present user privacy attacks that can be perpetrated relying on side-channel information, but also propose some possible applications that aim to enhance mobile devices' security. We highlight that most of the applications of side-channels analyses that we report in this dissertation are novel contributions to the state of the art.

Figure 1.1 depicts the type of mobile devices considered in this thesis, the side-channels analyzed and their possible applications of such analyses. On the left side of Figure 1.1, we report the mobile devices we consider in our work, namely smartphones, tablets, and laptops. Side-channels can be categorized according to the point in which they can be measured: inside or outside the mobile device. The former category includes the data about touch gestures (i.e., user inputs on the touchscreen) and built-in sensors that measure environmental properties, such as motion and orientation of the mobile device. The latter category considers physical phenomenon measurable externally to the device, such as the electric energy provided to recharge the battery, and the wireless network traffic. Given the data collected from the aforementioned side-channels, it is possible to apply machine learning techniques to extract some valuable knowledge. On the right side of Figure 1.1, we list some possible outcomes and applications of such analyses.

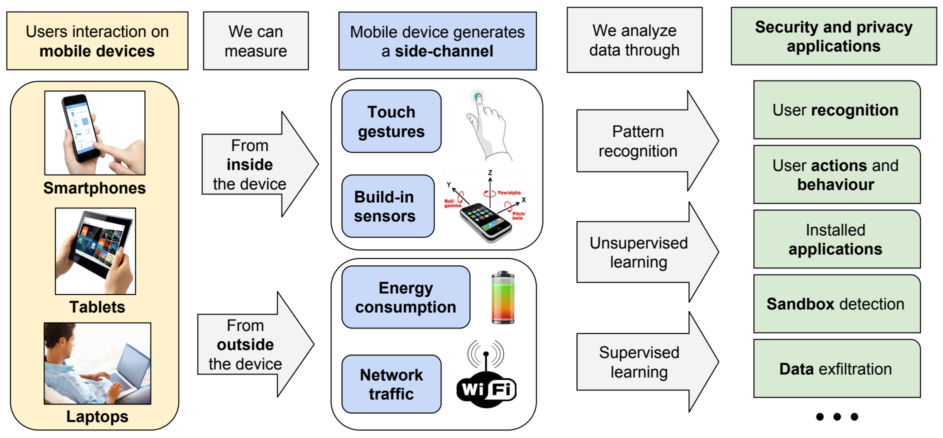


Figure 1.1: Schema for mobile devices side-channels analysis and possible applications.

The research work presented in this thesis is composed of three main parts, each related to a different side-channel:

- *Network traffic analysis* investigates on possible information that could be inferred in wireless encrypted communications between mobile device and remote service.
- *Energy consumption analysis* considers the applications of electric current retrieved by a mobile device as a side-channel.

- *Built-in sensors analysis* presents work that consider data collected from mobile devices' sensors as a side-channel to be used to enforce security.

In what follows, we briefly introduce the aforementioned parts, highlighting our contributions.

In this dissertation, some figures have been re-used and some passages have been quoted verbatim from the following works [34, 51–53, 183, 185, 195, 196] (all co-authored by the author of this dissertation).

1.1.1 Network Traffic Analysis

In order to access to web services, mobile devices are equipped with WiFi or cellular network adapters, that provide Internet connectivity. An attacker could intercept the communications between mobile devices and services in several ways, such as by eavesdropping wireless transmission with an antenna, by wiretapping the transmission cable, or by being the deployer of Internet access point. Such kind of attack is called Man-In-The-Middle (MITM) attack since the attacker places herself in the middle of the communicating parties, being able to observe their messages. In Figure 1.2, we depicts a possible MITM attack that involves an Android device and a web server. This kind of attack can be either active or passive.

On one hand, an active attacker can intercept a web service's real certificate and provide the victim a fake certificate, pretending to be the real web service. Nonetheless, active MITM attack can be easily revealed whether the victim adopts effective security countermeasures [49]. On the other hand, at the time of the writing, passive MITM attack are hard, if not impossible, to detect since it only listens to the channel without performing any alteration.

In the first part of this thesis, we consider a passive MITM attacker that listen to TCP/IP network traffic generated by the victim's mobile device. It is worth noticing that such network traffic is encrypted with SSL/TLS encryption protocol, thus the attacker is not able to rely on Packet Inspection techniques. Even the network traffic is encrypted, a passive MITM attacker can still infer a significant amount of information from the analysis of the properly encrypted network traffic. For example, work leveraging analysis of encrypted traffic already highlighted the possibility of understanding the set of apps installed on a mobile device [190], or identify the presence of a specific user within a network [198].

In our work, we focus on inferring two private information about the victim: (i) the actions performed by the victim with an app; and (ii) the list of installed apps on the victim's mobile device.

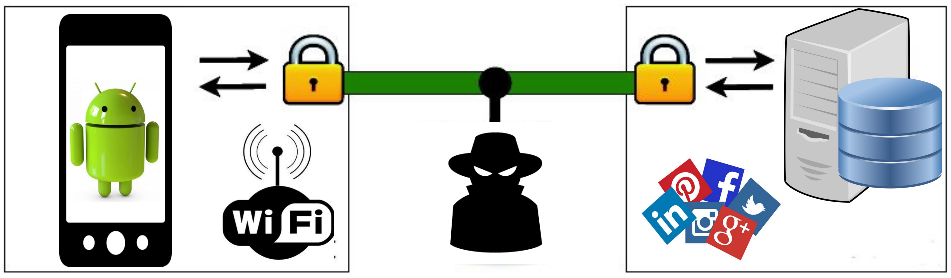


Figure 1.2: Man-In-The-Middle attack.

User Actions Recognition

The actions a user performs on an app could reveal many insights about her. She installs and uses several apps to communicate with friends or acquaintances. Through her smartphone, she gets information about sensitive topics such as diseases, sexual or religious preferences. As a consequence, several concerns have been raised about the capabilities of these portable devices to invade the privacy of users actually becoming “tracking devices”.

This work focuses on understanding whether the user profiling made through analyzing encrypted traffic [198] can be enhanced to understand exactly what actions the user is doing on her phone: as concrete examples, we aim at identifying actions such as the user sending an email, receiving an email, browsing someone’s profile on a social network, publishing a post or a tweet, or “tagging” someone in a picture. The underlying issue we leverage in our work is that SSL and TLS protect the content of a packet, while they do not prevent the detection of networks packets patterns that instead may reveal some sensitive information about the user behavior.

Contributions: We investigate to which extent such an external attacker can identify the specific actions that a user is performing on her mobile apps. Our work analyzes the network communications and leverages information available in TCP/IP packets (like IP addresses and ports), together with other information like the size, the direction (incoming/outgoing), and the timing. By using an approach based on machine learning, each app that is of interest is analyzed independently. To set up our system, for each app we first pre-process a dataset of network packets labeled with the user actions that originated them, we cluster them in flow typologies that represent recurrent network flows, and finally we analyze them in order to create a training set that will be used to feed a classifier. The trained classifier will then be able to classify new traffic traces that have never been seen before. We run a thorough set of experiments to evaluate our solution considering seven popular apps: Facebook, Gmail, Twitter, Tumblr, Dropbox, Google+ and Evernote. The results show that it can achieve accuracy and precision higher than 95%, for most of the considered actions. We also run a thorough comparison of our solution with three algorithms, showing that our solution outperforms them in all of the cases. We present this work in Chapter 2

Mobile Apps Fingerprinting

Automatic fingerprinting and identification of mobile apps are becoming a very attractive data gathering technique for adversaries, network administrators, investigators, and marketing agencies. In fact, the list of apps installed on a device can be used to identify vulnerable apps for an attacker to exploit, uncover a victim's use of sensitive apps, assist network planning, and aid marketing. However, app fingerprinting is challenging because of the vast number of apps available for download, the wide range of devices they may be installed on, and the use of payload encryption protocols such as HTTPS/TLS.

Contributions: In Chapter 3, we propose a novel methodology and a framework implementing it, called AppScanner, for the automatic fingerprinting and real-time identification of Android apps from their encrypted network traffic. To build app fingerprints, we run apps automatically on a physical device to collect their network traces. We apply various processing strategies to these network traces before extracting the features that are used to train our supervised learning algorithms. Our fingerprint generation methodology is highly scalable and does not rely on inspecting packet payloads; thus our framework works even when HTTPS/TLS is employed. We built and deployed this lightweight framework and ran a thorough set of experiments to assess its performance. We automatically profiled 110 of the most popular apps in the Google Play Store and were later able to re-identify them with more than 99% accuracy.

1.1.2 Energy Consumption Analysis

The Internet of Things (IoT) paradigm, in conjunction with the one of smart cities, is pursuing toward the concept of smart buildings, i.e., "intelligent" buildings able to receive data from a network of sensors and thus to adapt the environment. IoT sensors can monitor a wide range of environmental features such as the energy consumption inside a building at fine-grained level (e.g., for a specific wall-socket). Some smart buildings already deploy monitoring electric energy consumption of appliances. Besides, charger stations deployed in public places provide electric current to recharge the batteries of mobile devices. Such stations could monitor the energy provided as well. While electric consumption measurements could be carried out in order to optimize the energy use for good purposes (e.g., to save money, to reduce pollution), they also raise a significant amount of privacy concerns.

Energy consumption can be considered as another side-channel of mobile devices because it is directly related to the way a user interacts with a device. In the second part of this dissertation, we contribute to the state of the art of energy consumption analysis with two works having the following targets: (i) identifying the user laptop plugged on a smart meter; and (ii) exfiltrating data via USB charging cable using energy consumption as a covert channel.

User Laptops Identification

Laptops became a useful mobile device thanks to their portability and computational capabilities. Despite these positive features make them a valuable tool for work and entertainment, laptops share the same shortcoming of smartphones and tablet: they are limited by the life of their battery if not connected to a power source. For this reason, when laptops are used in a stationary location, users connect them to a power source rather than rely on the battery. In our threat model, we assume a smart building where a user plugs her laptop to a wall-socket smart meters, thus we are able to measure the energy consumption of such laptop.

Contributions: We investigate the feasibility of recognizing the pair laptop-user (i.e., a user using her own laptop) from the energy traces produced by her laptop in Chapter 4. We design MTPlug, a framework that achieves this goal relying on supervised machine learning techniques as pattern recognition in multivariate time series. We present a comprehensive implementation of this system and run a thorough set of experiments. In particular, we collected data by monitoring the energy consumption of two groups of laptop users, some office employees and some intruders, for a total of 27 people. We show that our system is able to build an energy profile for a laptop user with an accuracy above 80%, in less than 3.5 hours of laptop usage. To the best of our knowledge, this is the first research that assesses the feasibility of laptop users profiling relying uniquely on fine-grained energy traces collected using wall-socket smart meters.

Data Exfiltration

Thanks to the technological innovation and to the high user demand, mobile devices are integrating extensive battery-draining functionalities, which results in a surge of energy consumption of these devices. This scenario leads many people to look for opportunities to charge their devices at public charging stations: the presence of such stations is already prominent around public areas such as hotels, shopping malls, airports, gyms and museums, and is expected to significantly grow in the future. While most of the times the electric power comes for free, there is no guarantee that the charging station is not maliciously controlled by an adversary, with the intention to exfiltrate data from the devices that are connected to it.

Contributions: We show for the first time how an adversary could leverage a maliciously controlled charging station to exfiltrate data from a smartphone via a USB charging cable (i.e., without using the data transfer functionality), controlling our simple app (named PowerSnitch) running on the device. It is worth noticing that PowerSnitch app does not any permission to be granted by the user to send data out of the device. We show the feasibility of our attack through a prototype implementation in Android, which is able to send out potentially sensitive information, such as `IMEI` and contacts' phone number. We present and evaluate our covert-channel in Chapter 5

1.1.3 Built-in Sensors Analysis

Along with other aforementioned features, mobile devices are equipped with a wide range of sensors to measure the surrounding environment (e.g., orientation, location, lightness) and provide an advanced user experience. The data measured by those sensors are made available to all apps without any permission and it can be considered as a side-channel. To some extent, we can also consider mobile device's touchscreen as a sensor, more than simply an input device. Differently from the side-channels presented in the two previous parts, built-in sensors side-channel can be directly measured from inside a mobile device. Since the analysis of such data may require time and computational power for a battery-powered device, it is possible to first collect sensors data and then send it out to be analyzed in a more suitable facility.

In our work, we investigate different aspects related to built-in sensors side-channel and we give two contributions to the state of the art: (i) we present a usable and multi-purpose logging tool for Android; and (ii) we use sensors data collected from real devices to build a malware analysis sandboxes robust against emulator detection heuristics.

Logging and Data Extraction Tool

The use of smartphones has increased exponentially, and so have their sensing and functional capabilities. Indeed, together with an increase in processing power, smartphones and tablets are equipped with a variety of sensors and provide an extensive set of API. Such capabilities allow mobile devices to collect data related to environment, user habits, and operating system itself. This data is extremely valuable in many research fields such as information leaks detection and user authentication [54, 85, 184, 187]. For these reasons, researchers need a solid and reliable logging tool to collect data from mobile devices.

Contributions: First, we survey the existing logging tools available on the Android platform, comparing their features and their impact on the system. Second, we present DELTA, a framework for Data Extraction and Logging Tool for Android. DELTA improves the existing Android logging solutions in terms of flexibility, fine-grained tuning capabilities, extensibility, and available set of logging features. We fully implement DELTA and we run a thorough performance evaluation. The results show that our tool has a low impact on the performance of the system, on battery consumption, and on user experience. DELTA is open source and its code is available to the research community. We presented this work in Chapter 6.

Robust Sandbox Against Malware Analysis Evasion

Malware developers attention is gradually moving from PCs to mobile devices. This because the latter ones can access and store personal information (e.g., location, photos, and messages). The most promising approach to analyze malware is by monitoring its execution in a sandbox (i.e., via dynamic analysis). In particular, most malware sandboxing solutions for Android rely on an emulator, rather than a real device. This motivates malware authors to include runtime checks in order to detect whether the malware is running in a virtualized environment. In that case, the malware does not trigger the malicious payload. The presence of differences between real devices and Android emulators (also called artifacts) started an arms race between security researchers and malware authors, where the former want to hide these differences and the latter try to seek them out.

Contributions: We present Mirage, a malware sandbox architecture for Android focused on dynamic analysis evasion attacks in Chapter 7. We designed the components of Mirage to be extensible via software modules, in order to build specific countermeasures against such attacks. Mirage is the first modular sandbox architecture that is robust against sandbox detection techniques. As a representative case study, we present a proof of concept implementation of Mirage with a module that tackles evasion attacks based on sensors API return values. In particular, such values can be constant or stream of sensor data previously collected from real mobile devices via a data collection app (e.g., DELTA). Mirage replays such data injecting them into the emulator that, such a way, pretends to be a real device does.

Finally, Chapter 8 we draw some conclusions and possible future directions.

1.2 Publications

The research presented in this dissertation and carried out during my Ph.D. program produced peer-reviewed conference, journal and workshop publications. In what follows, we report the complete list of published and currently submitted works: In Section 1.2.1, journal papers; in Section 1.2.2, conference and workshop papers; and international patents and knowledge transfer in Section 1.2.3.

1.2.1 Journal Publications

[J01] M. Conti, L. V. Mancini, R. Spolaor and N. V. Verde, "Analyzing Android Encrypted Network Traffic to Identify User Actions", in *IEEE Transactions on Information Forensics & Security*, 2016 **(JCR IF 2013: 2.065; IT-ANVUR Class 1)**.

[J02] R. Spolaor, Q-Q. Li, M. Monaro, M. Conti, L. Gamberini and G. Sartori, "Biometric Authentication Methods on Smartphones: A Survey", in *PsychNology Journal*, 2017.

[J03] R. Spolaor, E. Dal Santo and M. Conti, "DELTA: Data Extraction and Logging Tool for Android", in *IEEE Transactions on Mobile Computing*, 2017. **(JCR IF 2016: 3.822; IT-ANVUR Class 1)** (In press).

[J04] V. F. Taylor, R. Spolaor, M. Conti and I. Martinovic, "Robust Smartphone App Identification Via Encrypted Network Traffic Analysis", in *IEEE Transactions on Information Forensics & Security*, 2017. **(JCR IF 2016: 4.332; IT-ANVUR Class 1)** (In press).

[J05] M. Conti, Q-Q. Li, A. Maragno, R. Spolaor, "The Dark Side(-Channel) of Mobile Devices: A Survey on Network Traffic Analysis", CoRR abs/1708.03766, 2017. Under submission at: *IEEE Communications Surveys and Tutorials.*

[J06] M. Monaro, C. Galante , R. Spolaor, Q-Q. Li , L. Gamberini, M. Conti, and G. Sartori, "Covert lie detection using keyboard dynamics", Under MINOR revision at: *Scientific Reports.*

1.2.2 Conference and Workshop Publications

[C01] M. Conti, L. V. Mancini, R. Spolaor and N. V. Verde, "Can't you hear me knocking: Identification of user actions on Android apps via traffic analysis", in *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2015. **(acceptance rate 21%)**.

[C02] V. F. Taylor, R. Spolaor, M. Conti and I. Martinovic, "AppScanner: Automatic Fingerprinting of Smartphone Apps From Encrypted Network Traffic", in *Proceedings the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016. **(acceptance rate 17%)**.

[C03] V. D. Stanciu, R. Spolaor, M. Conti and C. Giuffrida, "On the Effectiveness of Sensor-enhanced Keystroke Dynamics Against Statistical Attacks", in *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2016. **(acceptance rate 19%)**.

[C04] M. Conti, M. Nati, E. Rotundo and R. Spolaor, "Mind The Plug! Laptop User Recognition Through Power Consumption", in *Proceedings of the International Workshop on IoT Privacy, Trust, and Security (IoTPTS) @ AsiaCCS*, 2016.

[C05] M. Conti, C. Guarisco and R. Spolaor, "CAPTCHaStar! A novel CAPTCHA based on interactive shape discovery", in *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*, 2016 **(acceptance rate 19%)**.

[C06] R. Spolaor, L. Abudahi, V. Moonsamy, M. Conti, R. Poovendran, "No Free Charge Theorem: A Covert Channel via USB Charging Cable on Mobile Devices", in *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*, 2017.

[C07] L. Bordoni, M. Conti and R. Spolaor, "Mirage: Toward a Stealthier and Modular Malware Analysis Sandbox for Android", in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)* 2017. **(acceptance rate 15.88%)**.

[C08] Merylin Monaro, R. Spolaor and Q-Q. Li, M. Conti, L. Gamberini and G. Sartori, "Type Me the Truth!: Detecting Deceitful Users via Keystroke Dynamics", in *Proceedings of the International Conference on Availability, Reliability and Security (IWCC) @ ARES*, 2017.

[C09] M. Favaretto, R. Spolaor, M. Conti, and M. Ferrante, "You Surf so Strange Today: Anomaly Detection in Web Services via HMM and CTMC", in *Proceedings of the International Conference on Green, Pervasive and Cloud Computing (GPC)*, 2017.

[C10] R. Spolaor, M. Monaro, P. Capuozzo, M. Baesso, M. Conti, L. Gamberini, and G. Sartori, “You Are How You Play: Authenticating Mobile Users via Game Playing”, in *Proceedings of the International Workshop on Communication Security (WCS) @ EuroCrypt*, 2017.

[C11] M. Piskozub, R. Spolaor, M. Conti and I. Martinovic, “NetVerifier: Analysis of Host Behavior Profiles using CompactFlow Data” (submitted).

[C12] S. Wang, Z. Chen, Q. Yan, L. Wang, R. Spolaor, B. Yang and M. Conti, “Lexical Clustering of Malicious URLs for Mining Mobile Malware” (submitted).

1.2.3 Patents and Knowledge Transfer

[P01] M. Conti, C. Guarisco and R. Spolaor, “Metodo per riconoscere se un utente di un terminale elettronico e’ un umano o un robot” (International Patent Submitted), First prize at Notte dei Brevettatori 2015, Padua.

[P02] M. Conti, R. Spolaor an G. Tolomei “CAPTCHAd: interactive CAPTCHA with Monetizetion via Ads” (Spin-off Request Submitted), 2017.

Part I

Network Traffic Analysis

Chapter 2

User Actions Recognition

With people becoming more familiar with mobile devices and their related privacy threats, users have started adopting good practices that better adapt to their understanding privacy. Unfortunately, even adopting such good practices would not close the door to malicious adversaries. Indeed, several attacks may violate user privacy even when the adversary does not physically or remotely control the user device. In this chapter, we consider a passive attacker that is able to sniff the network traffic generated by mobile apps. Obviously, if the network traffic is not encrypted, the attacker can simply read the payload of each packet. However, many apps use the Secure Sockets Layer (SSL) – and its successor Transport Layer Security (TLS) – as a building block for encrypted communications. Even when such encryption are in place, the adversary can still infer a significant amount of information from encrypted network traffic analysis. As an example, work leveraging such analysis highlighted the possibility of identify the presence of a specific user within a network [198].

In this chapter, we present a work that focuses on identifying whether user profiling can be enhanced to understand what actions a user is performing on a specific app. In particular, we aim at identifying actions such as the user sending an email, browsing friend's profile on a social network, publishing a tweet or a post. Our framework can carry out such task by leveraging an analysis on encrypted network traffic. Indeed, SSL and TLS do not prevent us to detect patterns of networks packets that may reveal some sensitive information about the user behavior.

An adversary may use our framework in several practical ways to threaten the privacy of the user. In the following, we present some possible scenarios:

- A censorship government trying to identify a dissident who spreads anti-government propaganda using an anonymous social network account. Comparing the time of the public posts with the time of the actions (inferred with our method), the government can guess the identity of that anonymous dissident.

- By tracing the actions performed by two users, and taking into account the communication latency, an adversary can guess (even with some probability of error) whether there is a communication between them. However, multiple observations could reduce the probability of errors.
- An adversary can build a behavioral profile of a target victim based on the habits of the latter one (e.g., wake up time, work time). For example, this could be used to improve user fingerprinting methods, to infer the presence of a particular user in a network [198], even when she accesses the network with different types of devices.

Contributions: Our framework analyzes the network communications and leverages information available in TCP/IP packets, such as IP addresses and ports, packet size, direction (i.e., incoming or outgoing), and timing. We analyze traffic of several apps of interests using an approach based on machine learning techniques. For each app, we build a dataset in three steps: (i) we pre-process a dataset of network packets labeled with the user actions that generated them; (ii) we cluster them in flow typologies that represent recurrent network flows; and (iii) we analyze them in order to create a training set that will be used to feed a classifier. The trained classifier will then be able to classify new traffic traces that were never seen before. We run a thorough set of experiments to evaluate our framework considering seven popular apps: Facebook, Gmail, Twitter, Tumblr, Dropbox, Google+, and Evernote. For most of the considered actions, our proposal achieves an accuracy and a precision more than 95%.

In addition to that, we discuss about the key idea underneath our traffic analysis approach. In particular, we examine thoroughly examine the concept of network flow and the metric to evaluate the similarity between them. We also report details of the machine learning techniques we leverage in our method. Furthermore, we run a thorough comparison of our solution with three state of the art algorithms, showing that our solution outperforms them in all of the scenarios.

Organization: The rest of this chapter is organized as follows. We revise the state of the art around our research topic in Section 2.1. In Section 2.1.1, we introduce some background knowledge on machine learning and data mining tools used in our work. We present our framework describing all its different components in Section 2.2, and we present the evaluation of our solution for identifying user actions in Section 2.3. In Section 2.4, we discuss about possible countermeasures against the proposed attack. Finally, we summarize the work described in this chapter in Section 2.5.

2.1 Related Work

Our main claim in this chapter is that network traffic analysis and machine learning can be used to infer private information about the user, i.e., the actions that she executes with her mobile phone, even though the traffic is encrypted. To position our contribution with respect to the state of the art, we survey the works that belong to two main research areas that focus on similar issues: *privacy attacks via traffic analysis* (not necessarily focusing on mobile devices) and *traffic analysis of mobile devices* (not necessarily focusing on privacy).

Security and privacy on smartphones Privacy is an important matter also sensed by smartphones' users, even more than using a laptop [46]. Malware are a serious threat to security on smartphone [72], because they can cause device malfunctioning and user personal information disclosure, like her position, contacts, health condition, etc. Some works propose apps profiling frameworks to detect malicious behavior or potential privacy-related informations exposure [64, 172, 210]. In particular Wei et al. [204] present system to profile the behavior of Android apps. App profile is done from four different points of view (layer): static, user interaction, OS and network. On Android OS, malicious apps could obtain the access to device resources by given permissions. But in most cases, users don't understand what these permissions really mean [73]. Possible countermeasures to this problem consist to replace sensitive data with shadowed copy [95] or use taint analysis [66], to track and understand how these informations are used by apps. Another possible approach to this problem is MockDroid [27], a modified version of Android OS, that gives to apps a fake access to resources, trading more privacy with a reduction of functionality.

Application classification with traffic analysis In recent years it was produced lots of works about traffic analysis, aiming to identify which application produces a flow. With applications, in these works, authors mean the last layer in ISO/OSI model, those cannot be classified using only the ports or protocols used. Particular interest is focused on P2P traffic recognition (emule, bittorrent), but also in application level protocols like HTTP, POP3, SSH and so on. Traffic analysis can be done in several ways exploiting machine learning algorithms on different network traffic features [104, 118]. Works on traffic analysis could be distinguished from each other by the machine learning methods: Naive Bayes [138, 149], Hidden Markov Models (HMM) [29, 181, 198, 206], unsupervised clustering and supervised classification [41, 88, 129], Support Vector Machines (SVM) [88, 177, 212], or custom classification algorithms [56,102,213]. Many of these works' analysis could be done also on encrypted TCP/IP traffic, because they consider only statistics of network flows (with no access to payloads). In our work we use a method known as early traffic analysis, that consist in consider only a limited the number of packets of a traffic flow. [29,88,177]. An example of custom traffic classifier is proposed by Crotti et al. in [56]. They build a fingerprint for a protocol using statistics about packets size, inter-arrival time and order. So they propose a classification algorithm made ad-hoc for classify these fingerprints. Another example of custom TCP traffic classification is BLINC [102]. In their *multi-level* analysis Karagiannis et al. consider network traffic from three different points of view: hosts popularity (social), hosts role in the network (client/server), and traffic flow features (application level). Finally, they propose a multi-level classifier that combines features from each level.

In our framework we use a classification in two steps, first we regroup flows by typology, and then we use those typologies to classify user actions. A similar approach, but with different purpose, is used in [129] on HTTP traffic and in [41] on TCP traffic. In [88, 177] Sena et al. describes two on-line methods to classify encrypted traffic. Starting from a payload traffic analysis ground truth, they compare Centroid clustering and SVM applied on statistical flow analysis. Like us, they doesn't consider a whole flow, but only the first N packets (early) in both directions. Although in their flow representation, informations about packets sequence isn't take in concern. Early flows analysis is also used in [29] for application recognition. First studying early TCP connection features, then evaluating performance of HMM and clustering methods like GMM on this domain. Some others, like [149], do the same classification but timely and continuously, oriented to QoS management. Their traffic classifier uses C4.5 Decision Tree and Naive Bayes Machine learning algorithms on statistics of sub-flows (a limited number of packets taken at any point on a flow). They are able to classify interactive traffic (on-line gaming and VOIP) among other TCP/IP traffic.

Privacy attacks via traffic analysis

In the literature, several works proposed to track user activities on the web by analyzing unencrypted HTTP requests and responses [20, 26, 176]. With this analysis it was possible to understand user actions inferring interests and habits. More recently, Neasbitt et al. proposed ClickMiner [146], a tool that reconstructs user-browser interactions. However, in recent years, websites and social networks started to use SSL/TLS encryption protocol, both for web and mobile services. This means that communications between endpoints are encrypted and this type of analysis cannot be performed anymore.

Different works surveyed possible attacks that can be performed using traffic analysis assuming a very strong adversary (e.g., a national security agency) which is able to observe all communication links [30, 168]. In [116], Liberatore et al. evaluated the effectiveness of two traffic analysis techniques based on naive Bayes and on Jaccards coefficient for identifying encrypted HTTP streams. Such an attack was outperformed by [94], where the authors presented a method that applies common text mining techniques to the normalized frequency distribution of observable IP packet sizes, obtaining a classifier that correctly identifies up to 97% of requests. Similarly, in [156] the authors presented a support vector machine classifier that was able to correctly identify web pages, even when the victim used both encryption and anonymization networks such as Tor. Finally, Cai et al. [38] and Dyer et al. [62] presented a web pages fingerprinting attack and proved its effectiveness despite traffic analysis countermeasures, such as HTTPOS [124].

More recently still, Panchenko et al. [155] present a website fingerprinting attack on Tor that outperforms related work (including their own earlier proposal) while requiring less computational resources. Using SVMs as the classifiers, they proposed a feature extraction technique that samples features starting from a cumulative representation of a web page's network flow. Muehlstein et al. [139] show that HTTPS traffic can be used to identify operating system, browser, and application. Similar to our work, the authors generate features using the concept of an encrypted network traffic flow. Miller et al. [133] use traffic analysis to identify individual web pages within websites with approximately 90% accuracy. However, the authors make two assumptions: (i) they rely on feature extraction using whole network bursts (multiple flows), assuming all the flows in the same burst belong to the same web page; and (ii) they can also rely on multiple bursts to build a website graph through an Hidden Markov Model (HMM). These assumptions, while reasonable for web page fingerprinting, fail for mobile app fingerprinting. On smartphones, multiple apps can concurrently send network traffic and so a single burst could contain flows generated by different apps.

Unfortunately, none of the aforementioned works was designed for (or could easily be extended) to mobile devices. In fact, all of them focus on web pages identification in desktop environment (in particular, in desktop browsers), where the generated HTTP traffic strictly depends on how web pages are designed. Conversely, mobile users mostly access the contents through the apps installed on their devices [86]. These apps communicate with a service provider (e.g., Facebook) through a set of APIs. An example of such differences between desktop web browsers and mobile apps is the validation of SSL certificates [49, 84].

Traffic analysis has been applied not only to HTTP but also to other protocols. For example, Song et al. [181] prove that several versions of SSH are not secure. In particular, they show that even very simple statistical techniques suffice to reveal sensitive information such as login passwords. More importantly, the authors show that by using more advanced statistical techniques on timing information collected from the network, the eavesdropper can also learn significant information about what users type in SSH sessions. SSH is not the only protocol that has been target of such attacks. Another example is Voice Over IP (VoIP). In [206], the authors show how the length of encrypted VoIP packets can be used to identify spoken phrases of a variable bit rate encoded call. Their work indicates that a profile Hidden Markov Model trained using speaker- and phrase-independent data can detect the presence of some phrases within encrypted VoIP calls with recall and precision exceeding 90%.

In [45], the authors show that despite encryption, web applications also suffer from side-channel leakages. The system model considered is different from ours. In particular, their focus is on web applications. On the contrary, we focus on mobile applications. More importantly, the authors leverage three fundamental features of web applications: stateful communication; low entropy input; significant traffic distinction. We believe that in most mobile applications two of these features (stateful communication, low entropy input) are not very useful to characterize user actions. In contrast to the work in [45], we adopt a solution that only needs information about packet sizes and their order.

Traffic analysis of mobile devices Focusing on mobile devices, traffic analysis has been successfully used to detect information leaks [66], to profile users by their set of installed apps [190], and to generate network profiles to identify Android apps in the HTTP traffic [57]. Traffic analysis has also been used to understand network traffic characteristics, with particular attention to energy saving [67]. Stober et al. [190] show that it is possible to identify the set of apps installed on an Android device, by eavesdropping the 3G/UMTS traffic that those apps generate. Similarly, Tongaonkar et al. [57] introduce an automatic app profiler that creates the network fingerprint of an Android app relying on packet payload inspection. Unfortunately, their solution is viable only for apps that do not use encrypted traffic. In [214], Zhou et al. discovered three unexpected channels of information leaks on Android: per-app data-usage statistics, ARP information, and speaker status. In particular, the authors used a suite of inference techniques to reveal a phone user's identity from the network-data consumption of Twitter app, by also leveraging online resources such as tweets published by Twitter. Unfortunately, the authors focused only on a specific user action (i.e., send a tweet) without distinguish that action from the other ones a user could perform. More recently, Coull et al. in [55] presented a work similar to ours. The authors inferred information analyzing payload lengths of network packets produced by Apple iMessage and other messaging apps on iOS and OSX. In particular, the purpose of their work is to infer the OS version, user actions and language used in instant messaging. The author focused on five actions strictly related to instant messaging apps: start writing, stop writing, message sending, attachment sending and read notification. In this chapter, we consider social network and email service apps on Android. Those apps permit us to investigate a wider set of actions than the one offered by instant messaging apps. We believe that the interest from researchers to aim at different targets (i.e., OS version, actions, language) and the results obtained so far, underlines the feasibility of those attacks, the relevance of this issue and the importance to foster further research in this domain.

2.1.1 Machine Learning and Data Mining Background

In this section, we briefly recall several machine learning and data mining concepts that we use in our works, while we point the reader to appropriate references for a complete introduction on those topics.

Dynamic Time Warping

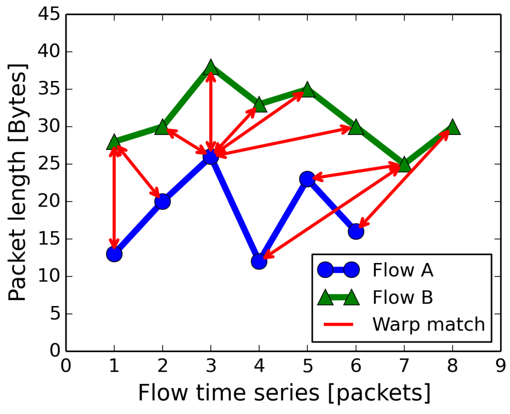
Dynamic Time Warping (DTW) [140] is a useful method to find alignments between two time-dependent sequences (also referred as time series) which may vary in time or speed. This method is also used to measure the distance or similarity between time series.

Let us consider two sequences that represent two discrete signals: $X = (x_1, \ldots, x_N)$ of length $N \in \mathbb{N}$; and $Y = (y_1, \ldots, y_m)$ of length $M \in \mathbb{N}$. DTW uses a local distance measure $c : \mathbb{R} \times \mathbb{R} \to \mathbb{R}_{\geq 0}$ to calculate a cost matrix $C \in \mathbb{R}^{N \times M}$, s.t., each cell $C_{i,j}$ reports the distance between x_i and y_j. The goal is to find an alignment between X and Y having minimal overall distance. Intuitively, such an optimal alignment runs along a "valley" of low cost cells within the cost matrix C. More formally, a *warping path* is defined as a sequence $p = (p_1, \ldots, p_L)$ with $p_l = (n_l, m_l) \in [1 : N] \times [1 : M]$, $l \in [1 : L]$ satisfying the following three conditions:

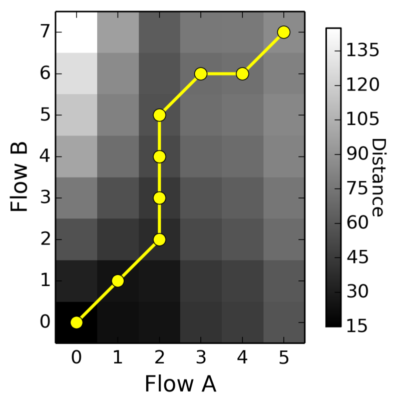
1. Boundary condition:
   $p_1 = (1, 1)$ and $p_L = (N, M)$;
2. Monotonicity condition:
   $n_1 \leq n_2 \leq \ldots \leq n_M$ and $m_1 \leq m_2 \leq \ldots \leq m_L$;
3. Step size condition:
   $p_{l+1} - p_l = \{(0, 1), (1, 0), (1, 1)\}$ for $l \in [1 : L - 1]$.

The total cost of a warping path is calculated as the sum of all the local distances of its elements. An *optimal warping path* is a warping path p^* having minimal total cost among all possible working paths. The total cost of an *optimal warping path* is also used as a distance measure between two sequences X and Y. In this chapter, we will indicate the cost of an *optimal warping path* with $DTW(X, Y)$.

Figure 2.1a shows an example of alignment between two signals (indicated in the figure with *Flow A* and *Flow B*). The arrows show the matched points which are given by the DTW algorithm. The same two flows have been used to calculate the heat matrix shown in Figure 2.1b. In this representation, the color of a cell (i, j) represents the minimum distances to reach



(a) Alignment of two discrete signals.



(b) Representation of an optimal warping path.

Figure 2.1: Example of DTW algorithm applied to two discrete signals: *Flow A* and *Flow B*.

cell (i, j) when starting from cell $(0, 0)$. An *optimal warping path* is then highlighted with a line that runs from cell $(0, 0)$ to cell $(12, 13)$. It can be noticed that this warping path satisfies boundary, monotonicity, and step size conditions reported above.

2.1.2 Supervised and Unsupervised Learning

Generally, machine learning approaches can be classified in two classes: unsupervised and supervised algorithms. Unsupervised learning algorithms try to find hidden structure in unlabeled data. Since the examples given to the learner are unlabeled, there is no error or reward signal to evaluate a potential solution. On the contrary, supervised machine learning algorithms learns from labeled instances or examples, which are collected in the past and represent past experiences in some real-world applications. They produce an inferred model, which can be then used for mapping or classifying new instances. An optimal scenario will allow for the algorithm to correctly determine the class labels for unseen instances.

In this chapter, we will use both supervised and unsupervised learning algorithms. We use supervised learning by applying an ensemble classifier that is called Random Forest [35]. The main principle behind ensemble methods is that a group of "weak learners" can be combined together to form a "strong learner". Random forest leverages a standard machine learning technique called "decision tree", which, in ensemble terms, corresponds to the weak learner. In practice, it combines together the results of several decision trees trained with different portions of the training dataset and different subsets of features. More details about the Random Forest classifier can be found in [35].

We use unsupervised learning by applying a clustering algorithm called hierarchical clustering. Hierarchical clustering is a cluster analysis method which seeks to build a hierarchy of clusters. This clustering method has the distinct advantage that any valid measure of distance can be used. In fact, the observations themselves are not required: all that is used is a matrix of distances.

In the following we will use a type of hierarchical clustering that is called agglomerative: each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy. In order to decide which clusters should be combined, a metric (a measure of distance between pairs of observations) and a linkage criterion are required. Since we will clusterize time-dependent sequences, we will use the total cost of an *optimal warping path* as distance metric. As for the linkage criterion, that determines the distance between sets of observations as a function of the pairwise distances between observations, we will use the average distance, that is defined as:

$$d(u,v) = \sum_{\substack{1 \le i \le n \\ 1 \le j \le m}} \frac{d(u[i], v[j])}{|u| * |v|},$$

where $d()$ is a distance function, and u and v are two clusters of n and m elements, respectively. More details about Hierarchical clustering can be found in [93].

2.2 Our Proposed Methodology

Our framework is logically composed by two components: the "pre-processor" and the "traffic classifier". The former has the task of executing all the pre-processing steps that allow us to model the network traffic into data that the traffic classifier can easily handle. The latter executes the actual classification task. Before using the traffic classifier, it has to be trained with labeled traffic data that we are able to generate by artificially stimulating the analyzed apps. We detail the steps executed by the pre-processor in Section 2.2.1, while in Section 2.2.2, we describe the methodology used to generate our training dataset, as well as the procedure used to classify user actions.

2.2.1 Network Traffic Pre-Processing Steps

Mobile apps generally rely on SSL/TLS to securely communicate with peers. These protocols are built on the top of the TCP/IP suite. The TCP layer receives encrypted data from the above layer, it divides data into chunks

ID	Type	Time series
Flow 1	Incoming	[1514, 1514, 315, 113, 477]
	Outgoing	[282, 188, 514, 96, 1514, 179, 603, 98, 801, 98]
	Complete	[282, -1514, -1514, -315, 188, -113, 514, 96, 1514, 179, 603, 98, 801, 98, -477]
Flow 2	Incoming	[1514, 1514, 1266, 582, 113, 661]
	Outgoing	[282, 188, 692, 423]
	Complete	[282, -1514, -1514, -1266, -582, 188, -113, 692, 423, -661]
Flow 3	Incoming	[1245, 1514, 107, 465, 172, 111]
	Outgoing	[926, 655, 136, 913, 1514, 1514, 863]
	Complete	[926, 655, 136, -1245, 913, 1514, 1514, 863, -1514, -107, -465, -172, -111]

Table 2.1: Example of time series generated from three network flows. Values within square brackets represent the amount of bytes exchanged per packet: negative values in complete time series indicate incoming bytes, while positive values indicate outgoing bytes.

if the packets exceeds a give size. Then, for each chunk it adds a TCP header creating a TCP segment. Each TCP segment is encapsulated into an Internet Protocol (IP) datagram, and exchanged with peers. Since TCP packets do not include a session identifier, both endpoints identify a TCP session using the client's IP address and the port number.

A fundamental entity considered in this work is the traffic *flow*: with this term we indicate a time ordered sequence of TCP packets exchanged between two peers during a single TCP session. The pre-processor takes in input the network traffic, it builds the network flows that represent that network traffic, and it generates a set of time series: (i) a time series is obtained by considering the bytes transported by incoming packets only; (ii) another one is obtained by considering bytes transported by outgoing packets only; (iii) a third one is obtained by combining (ordered by time) bytes transported by both incoming and outgoing packets. Hence, we use this set of time series as an abstract representation of a connection between two peers. Note that additional time series may be added to this set for example by considering other parameters such as the time-gap between different packets. For the sake of simplicity, in the following we will only consider the first three types of time series mentioned above.

Table 2.1 reports an example of time series generated from three network flows, while Figure 2.2 graphically represents these flows through a cumulative chart. The lower side of the chart represents incoming traffic, while the upper side represents outgoing traffic. This is only one of the possible representations, and it shows that the "shapes" of these three network flows are quite different. Intuitively, our classification approach tries to learn the "shape" of network flows related to particular user actions, and successively it aims to identify user actions by classifying the "shape" of previously unseen network flows.

Before generating for each flow the corresponding set of time series, a few pre-processing steps have to be performed. In particular: 1) we apply

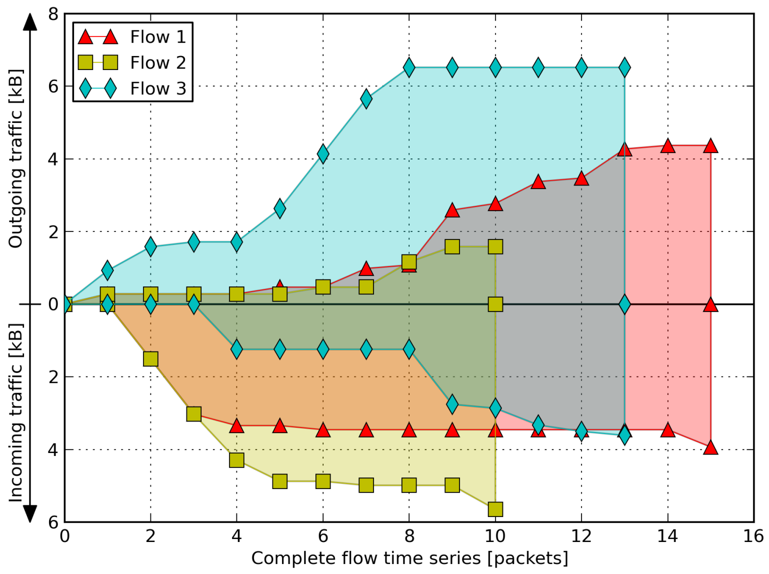


Figure 2.2: Representation of flows time series.

a domain filtering to select only flows belonging to the analyzed app; 2) we filter the remaining flows, in order to delete packets that may degrade the precision of our approach (i.e., we filter out ACK and retransmitted packets); 3) we limit the length of the generated time series. For each flow, the result of these three pre-processing steps will be a set of time series that will be passed to the next component of the framework, which is the traffic classifier. In the following, we will detail the three pre-processing steps.

Domain filtering

The network traffic generated by an application is generally directed toward a back-end infrastructure. The back-end infrastructures might be composed of a single server, or a set of servers. The set of servers might even be behind a load balancer. Since we analyze each app independently, we need to make sure that traffic generated from apps other than the considered one (or traffic generated by the OS) does not interfere with the analysis. Different methods can be used in order to identify the app that generated each network flows. The destination IP address is a trivial discriminating parameter. However, in case of a load balanced back-end, we should know all the individual IP addresses that can be involved in the communication. The same happens when the back-end is composed of several components such as different web services, databases, etc. To overcome this problem we use another strategy: we take into consideration for further analysis only the flows which destination IP addresses owners have been clearly identified as related to the considered app. In the implementation of our framework, we leverage the WHOIS protocol for this purpose, but we want to highlight that this is only one of the possible ways. Business and other context information may be used in order to perform the domain filtering. We also take into consideration the traffic related to third party services (such as Akamai or Amazon) that are indeed used by several applications [204].

Packets filtering

Due to network congestion, traffic load balancing, or other unpredictable network behavior, IP packets can be lost, duplicated, or delivered out of order. TCP detects these problems, hence requesting retransmission of lost data, and reordering out-of-order data. As a results, several TCP packets that do not carry data, may hinder the analysis process. In the data exchange phase, for example, the receiver sends a packet with the ACK flag set to notify the correct reception of a chunk of data. These ACK packets are transmitted in asynchronous mode so they are affected by many factors related to round trip time of the connection link. The order of the received packets may hinder the evaluation of the similarity between two network flows. For this reason, we filter out all packets retransmissions, as well as packets marked with the ACK flag. Note that the metric that we will use in order to measure similarity between flows (see Section 2.2.2) will mitigate the consequences of missing packets. We also filter out other packets that do not bring any additional information helpful in characterizing flows. In particular, we filter out the three way handshake executed to open a TCP connection, and the packets exchanged to close it.

Timeout and packets interval

Two different techniques are used to limit the length of the generated time series: a *timeout* mechanism and the specification of a *packets interval*. The *timeout* mechanism is used to terminate the flows that did not receive any new packet since 4.5 seconds. Indeed, it has been proved experimentally that 95% of all packets arrive at most 4.43 seconds after their predecessors [190]. The *packets interval* specifies the first and the last packet to be considered.

For example, considering a flow f composed by l packets, and the interval $[x, y]$ with $x \leq y$ and $y \leq l$, the corresponding time series will be composed by $y - x + 1$ values that report the bytes of the x^{th} to the y^{th} packet. This simple mechanism allows us to focus on particular portions of the flow. The first part, for example, is often the more significant. In the experimental part, we report the results for different configurations of packets intervals, showing that the best configuration is app dependent.

2.2.2 Classification of Network Traffic

Since we use a supervised learning approach, it is necessary to create a labeled dataset that describes the user actions that we want to classify. The labeled dataset is used to train the traffic classifier component allowing it to correctly classify previously unseen data instances. In order to build the training dataset, we simulate a series of user actions by interacting with the app to analyze. For each performed action we intercept and label the flows generated after the execution of the action itself. For each app that we analyze we focus on actions that are significant for that particular app.

In most cases, a single user action generates a set of different flows (i.e., not just a single one). Furthermore, different user actions may generate different sets of flows. Our classification method is based on the detection of the sets of flows that are distinctive of a particular user action. In order to elicit these distinctive sets of flows, we build clusters of flows by using the agglomerative clustering approach described in Section 2.1.2. Similar flows will be grouped together in the same cluster, while dissimilar flows will be assigned to different clusters. The average distance is used as linkage criterion, while the computation of the distance between two flows combines the distances of the corresponding time series. Supposing that each flow f_i is decomposed into a set of n time series $\{T_1^i, \ldots, T_n^i\}$, the distance between f_i and f_j is defined as:

$$dist(f_i, f_j) = \sum_{k=1}^{n} w_k \times DTW(T_k^i, T_k^j),$$

where w_k is a weight assigned to the particular time series. Weights can be assigned in such a way as to give more importance to some type of time series with respect to others. For example, it is possible to give more weight to the time series that represent incoming packets, and less weight to those that represent outgoing packets.

In order to reduce the computational burden of the subsequent classification, a leader is elected for each cluster. Leaders will be the representative flows of their clusters. Given a cluster C containing the flows $\{f_1, \ldots, f_n\}$, the leader is elected by selecting the flow f_i that has the minimum overall distance from the other members of the cluster, that is:

$$\underset{f_i \in C}{\operatorname{argmin}} \left(\sum_{j=1}^{n} dist(f_i, f_j) \right).$$

Clustering is executed on the set of flows that will be used to build the training dataset. In particular, after performing the clustering the training dataset will be composed as follows. The user actions will be the instances of the datasets, while the class of each instance is a label representing the action. We will have one integer feature for each cluster identified through the agglomerative clustering. The value of each feature is determined by analyzing the flows related to an action. Each flow f captured after the execution of an action will be assigned to the cluster that minimizes the distance between f and the leader of the cluster. The k^{th} feature will therefore indicate the number of flows that have been assigned to the cluster C_k after the execution of that action. For example, for the action *send mail*, the k^{th} feature will be equal to 2 if there are 2 flows labeled with *send mail*assigned to the cluster C_k. Finally, we execute the classification with the Random Forest algorithm. The main idea behind the overall approach is that different actions will "trigger" different sets of clusters. The classification algorithm will therefore learn which are these sets, and will be able to correctly determine the class labels for unseen instances.

2.3 Experimental Evaluation

In order to assess the performance of our proposal, we considered several widespread apps that have different purposes: Gmail, Facebook, Twitter, Tumblr, Dropbox, Google+ and Evernote. We selected these apps because of their high popularity [1]. Indeed, Gmail is one of the largest email services and its Android app has over one billion downloads. On the other hand, Facebook and Twitter are not only the most popular Online Social Networks [2], but they also had a leading role in the Arab spring and the Istanbul's Taksim Gezi Park protests (when Turkish government blocked Twitter). Tumblr is a widely used micro-blogging platform owned by Yahoo! Inc., while Dropbox is one of the most used cloud storage services. Google+ is the social network and social layer for Google services owned and operated by Google Inc. Finally, Evernote is an app designed for note-taking and archiving. Given the wide set of apps we considered, we believe that the results of our analysis also hold for any other app that generates network traffic as a consequence of a user action. Note that most of the apps make use of a back-end service to implement the logic of the service, and thus they must generate network traffic as a consequence of almost any user interaction. To collect the network traffic related to different user actions, we set up a controlled environment. In this section we present the elements that compose this environment (Section 2.3.1), the methodology used to collect the data (Section 2.3.2), and the results of the evaluation (Section 2.3.3).

2.3.1 Hardware and Network Configuration

For the evaluation of our solution, we used a Galaxy Nexus (GT-I9250) smartphone, running the Android 4.1.2 (Jelly bean) operative system. We enabled the "*Android Debug*" option in order to allow the usage of the ADB

(Android Debug Bridge) interface via USB cable. We used a WiFi access point (U.S. Robotics USR808054) to provide wireless connectivity to the mobile phone. Finally, we used a server (Intel Pentium Processor dual core E5400 2.7GHz with 4 GB DDR2 RAM) with two network cards running Ubuntu Server 11.04 LTS to route the traffic from the access point to the Internet, and vice versa.

To eavesdrop network packets flowing through the server, we used Wireshark software. From a Wireshark capture file, we created a comma separated file (csv), where each row describes a packet captured from the access point's interface. For every packet we reported source and destination IP addresses, ports, size in bytes and time in seconds from Unix epoch[1], protocol type and TCP/IP flags. Since the payload is not relevant to our analysis, it has been omitted. This data has then been used to generate the time series as explained in Section 2.2.1.

2.3.2 Dataset Collection and Analysis

For our study we considered seven apps installed from the official Android market: Gmail v4.7.2, Facebook v3.8, Twitter v4.1.10, Tumblr v3.8.6.08, Dropbox v2.4.9.00, Google+ v5.3.0.91034052 and Evernote v7.0.2. For the social apps, we created ten accounts that have been divided in two different categories of users: "*active*" and "*passive*" users. "*Active*" users simulated the behavior of users that actively use the app by sending posts, email, tweets, surfing the various menus, etc. "*Passive*" users simulated the behavior of users that passively use the app, just by receiving messages or posts. The accounts of both passive and active users have been configured in such a way as to have several friends/followers within the group. We avoided configuring the accounts with actual friends or followers, in order to avoid interference due to notifications of external users activities that were not under our control.

To reach a particular target, a user may have to perform several actions in a precise order. An action could be simple (e.g., a tap on a button, a swipe, or a selection of edit box), or complex (e.g., type a text, which is a sequence of keyboard inputs). For example, a user has to perform three actions in a precise sequence to post a message on her Facebook wall. He has to be sure that the Facebook app shows the "*user's wall*", then she has to tap on the "*write a post*" button (1), fill the edit box with some text (2), and finally tap on the "*post*" button (3). It is important to highlight that we do not use static text to fill in text boxes, but the text is randomly selected from a large set of sentences. A script submits the sequence of actions to the mobile phone through the ADB commands, and it captures the network traffic that is generated. The script also records the execution

[1]00:00:00 UTC, 01 January 1970

time of each action. By using the recorded execution time of each action, it is then possible to label the flows extracted from the network traffic with the user action that produced it. For each app, we choose a set of actions that are more sensitive than others from user privacy point of view (e.g., send an email or a message). The list of these actions is reported in tables 2.3 and 2.4 . We underline that we do not ignore other user actions, but we label them as *other*. In this way we have several benefits [134]: we obtain a greater representation of data in terms of variety and variance of examples; we reduce the chances of overfitting; we improve the performance of the classifier on relevant user actions.

We collected and labeled the traffic generated by 220 sequences of actions for each app, where a sequence is composed by 50 types of actions (for a total of 11660 examples of actions for Gmail, 6600 for Twitter, 10120 for Facebook, 16070 for Tumblr, 15104 for Dropbox, 7813 for Google+ and 8740 for Evernote). The user action examples in the dataset were divided into a training set and a test set. We use the training set to train the classifier, while we use the test set to evaluate its accuracy. We underline that to build the test set we used accounts that have not been used to create the training set. By using different accounts to generate the training and the test set, it is possible to assure that the results of the classification do not depend on the specific accounts that have been analyzed.

As explained in Section 2.2.1, each network flow is modeled as a set of time series. Table 2.2 reports the weights and the intervals for several configurations ("Conf." in the table) used to limit the length of the time series generated by each app. We used different weights configurations, and we selected the packets intervals by analyzing the statistical length of the flows. Figure 2.3 reports the statistical distribution of the length of the flows app by app. The first quartile, the median and the third quartile are highlighted by using a notched box plot. In particular, the median value and the third quartile have been used as thresholds to limit the maximum length of the generated time series. For the Twitter app, in some cases we set the interval in such a way to focus only on the last three or four packets. Indeed, we noticed that the first part of the time series was identical for each flow.

To confirm this statement we report in Figure 2.4a and Figure 2.4b the graphical representation of the flows that occur when executing three different actions in Gmail and Twitter respectively. Comparing the two figures, it can be noticed that the shapes of the actions drastically change for Gmail, while they are almost unvaried for Twitter. As a matter of fact, different Twitter actions just differ in their last packets. Nevertheless, our approach reaches very good performance for this app too. In our experiments, we used the Random forest classifier implemented by the Python library *scikit-learn* [37]. The classifier is trained using 40 estimators (or weak learners). Each estimator consists of a decision tree without any restrictions on its

Apps	Sets	Weights	In	Out	Complete
Gmail	Configuration 1	0.80	[1,4]	[1,2]	[1,6]
		0.20	[1,6]	[1,3]	[1,9]
	Configuration 2	0.66	[1,4]	[1,2]	[1,6]
		0.33	[1,6]	[1,3]	[1,9]
	Configuration 3	0.33	[1,4]	[1,2]	[1,6]
		0.66	[1,6]	[1,3]	[1,9]
Facebook	Configuration 1	0.66	[1,3]	[1,5]	[1,7]
		0.33	[1,6]	[1,7]	[1,12]
	Configuration 2	0.33	[1,3]	[1,5]	[1,7]
		0.66	[1,6]	[1,7]	[1,12]
	Configuration 3	0.20	[1,3]	[1,5]	[1,7]
		0.80	[1,6]	[1,7]	[1,12]
Twitter	Configuration 1	0.95	-	-	[7,10]
		0.05	-	-	[1,10]
	Configuration 2	0.95	-	-	[8,11]
		0.05	-	-	[1,11]
	Configuration 3	0.95	-	-	[8,10]
		0.05	-	-	[1,10]
Tumblr	Configuration 1	1.00	-	-	[1,11]
Dropbox	Configuration 1	1.00	-	-	[1,9]
Google+	Configuration 1	1.00	-	-	[1,16]
Evernote	Configuration 1	1.00	-	-	[1,23]

Table 2.2: Weights set configurations and packets intervals for the considered apps.

depth limit. The number of features for each estimator is equal to the square root of the maximum number of available features.

2.3.3 Classification Performance

Before considering the classification of the user actions, it is worth discussing how to choose the number of clusters that should be used. In order to establish a reasonable value for this parameter, we used a validation dataset to study the accuracy of the classification when varying the number of clusters. Figure 2.5 reports the achieved results. For each app, we therefore considered the number of clusters that maximized the accuracy, in terms of averaged F-measure. In the following, we report the results of the classification app by app, and we discuss the average accuracy reached when detecting each sensitive user action. In tables 2.3 and 2.4, we report detailed results for the precision, the recall and the F-measure metrics achieved by the best configuration of all the analyzed apps. Since we are space constrained, we report the corresponding confusion matrices only for some of the analyzed apps.

Facebook: We focused on seven different actions that may be sensitive when using the Facebook app. On average, the F-measure is equal to 99%, with a precision and a recall of 99% and 98% respectively. Performance

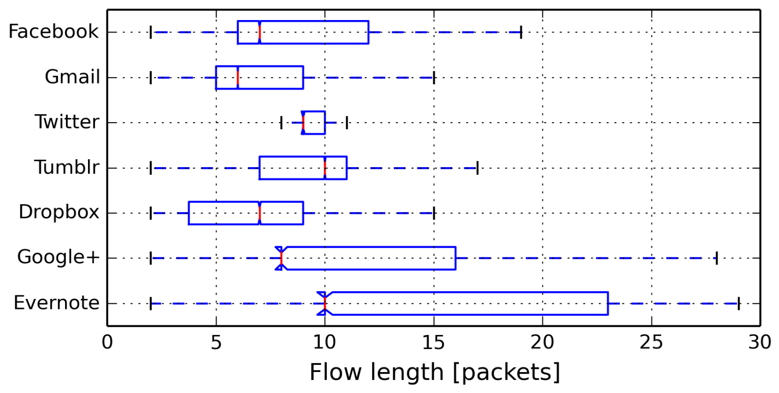


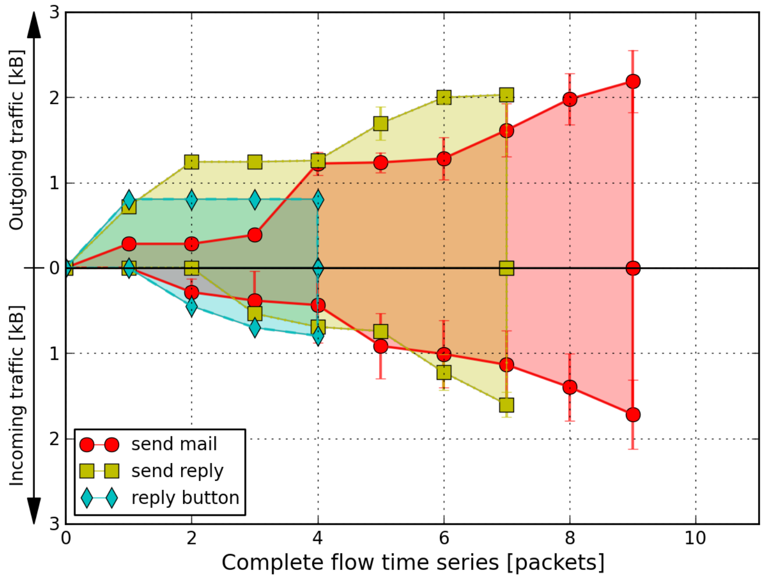
Figure 2.3: Statistical distribution of the length of the complete time series extracted from the network traffic. First and third quartile are represented as the left and right side of the notched box. The notch of the box represents the median value. Lines that extend horizontally from the boxes indicate the 2^{nd} percentile (left) and the 98^{th} percentile (right).

reached with different configurations of weights and packets intervals constraints are reported in Figure 2.6a. For each action at least one of the configurations exceeds 94% of accuracy, while the worst performing is always higher than 74%.

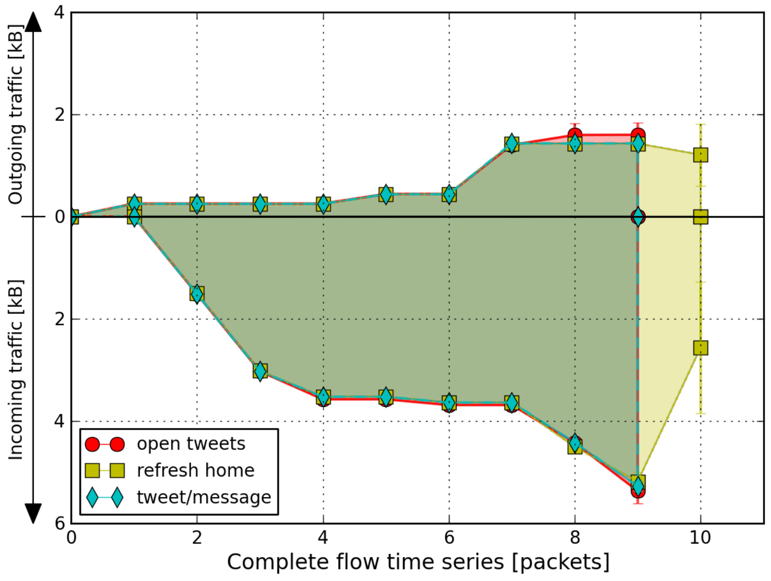
Table 2.3 reports precision, recall and F-measure reached by using Configuration 3. We noticed that all the actions have a precision higher 96%. The recall is higher than 95% for all the actions apart from the *open user profile*, that reaches 91%. In fact, we realized that this particular action is classified as *other* in 9% of the examples, as we can see from the confusion matrix reported in Figure 2.6b.

Gmail: We analyzed four specific user actions of the Gmail app: *send mail*, *reply button*, *open chats* and *send reply*. Figure 2.6c shows the classification accuracy that has been reached for each configuration of weights and packet interval constraints. We observe that we are able to distinguish with high accuracy the action of sending of a new mail, from that of replying to a previously received message, as well as the tap on the reply button. The *open chats* action is instead more difficult to distinguish. Table 2.3 reports precision, recall and F-measure for Configuration 1. We can observe that the action *open chats* (that allows to read past chats) achieves a low precision but a high recall. Analyzing the confusion matrix depicted in Figure 2.6d it is possible to notice that 16% of *other* actions are wrongly classified as *open chats*. This is the reason of such a low precision.

Twitter: During the analysis we noticed that Twitter actions may be more difficult to classify than Gmail and Facebook actions. Indeed, different Twit-



(a) Representation of three different Gmail actions.



(b) Representation of three different Twitter actions.

Figure 2.4: Comparison of three different Gmail and Twitter actions. It can be noticed that Twitter actions are more similar than Gmail actions, indeed their shapes are largely overlapped.

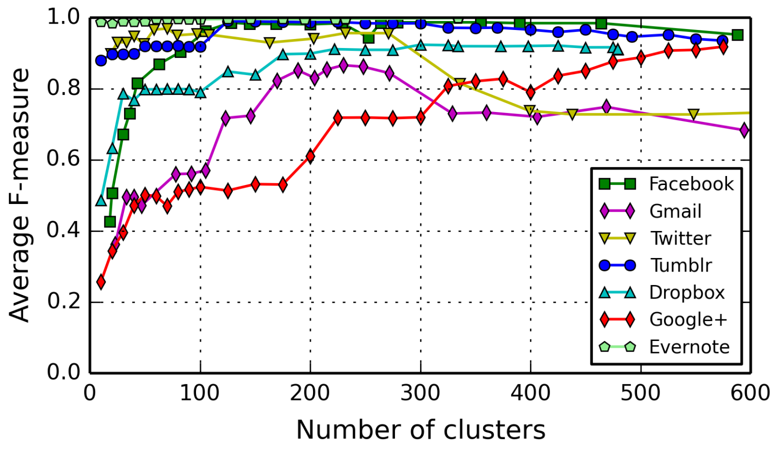


Figure 2.5: Classification accuracy over number of clusters.

ter actions generate similar time series that have a large portion in common. Only the last three or four packets of each time series show some difference. Nevertheless, we have been able to reach outstanding results for this app as well. In particular, we focus on six specific user actions: *refresh home*, *open contacts*, *tweet/message*, *open messages*, *open twitter*, *open tweets*. Performance reached for all the analyzed configurations are reported in Figure 2.6e. For each action at least one of the configurations exceeds 96% of accuracy, while the worst configuration has an accuracy in any case higher than 91%. The best performing configuration is Configuration 1, that on average, reached an F-measure value equal to 97%, with a precision and a recall of 98% and 97% respectively (see Table 2.3). The action *open twitter* has accuracy and recall equal to 100%, independently of the Configuration set used for the clustering phase. As a consequence, none of the examples of the test set have been wrongly classified. Figure 2.6f reports the confusion matrix obtained by considering the Twitter actions. Three of the six analyzed actions are correctly classified in more than the 99% of the cases, while the other three actions, that are *open contacts*, *open messages* and *open tweets*, are correctly classified in more than 95% of the cases.

Tumblr: We analyzed ten different user actions of the Tumblr app (see Table 2.3). On average precision, recall and F-measure is equal to 99%. Precision is always greater than 97% for the individual actions, while recall is greater than 96% in all the cases but one: *home page*.

Dropbox: As for Dropbox, we analyzed eight different user actions. On average, we reached a precision of 95% and a recall of 92%. Only for two individual actions, we reached precision or recall lower than 80%. This is

Apps	Actions	Description	P	R	F1
Facebook	*send message*	send a direct message to a friend	1.00	1.00	1.00
	post user status	post a status on the user's wall	1.00	0.95	0.97
	open user profile	select user profile page from menu	0.96	0.91	0.94
	open message	select a conversation on messages	0.98	1.00	0.99
	status button	select "*write a post*" on user's wall	1.00	1.00	1.00
	post on wall	post a message on a friend's wall	1.00	0.98	0.99
	open facebook	open the Facebook app	1.00	1.00	1.00
	other facebook	other Facebook network traffic	0.99	1.00	0.99
	Average Facebook		0.99	0.98	0.99
Gmail	*send mail*	send a new mail	1.00	1.00	1.00
	reply button	tap on the reply button	0.85	1.00	0.92
	open chats	select chats page from menu	0.36	0.94	0.52
	send reply	send a reply to a received mail	0.98	1.00	0.99
	other gmail	other Gmail network traffic	0.99	0.82	0.90
	Average Gmail		0.83	0.85	0.86
Twitter	*refresh home*	refresh the home page	0.94	0.99	0.96
	open contacts	select contacts page on menu	0.97	0.96	0.97
	tweet/message	publish a tweet or a message	0.97	1.00	0.98
	open messages	select direct messages page	1.00	0.95	0.97
	open twitter	open the Twitter app	1.00	1.00	1.00
	open tweets	select tweets page	1.00	0.95	0.97
	other twitter	other Twitter network traffic	0.96	0.96	0.96
	Average Twitter		0.98	0.97	0.97
Tumblr	*open tumblr*	open the Tumblr app	1.00	1.00	1.00
	post/reblog/quote	publish a post/reblog/quote	1.00	0.99	0.99
	delete post	delete a post/reblog/quote	0.97	1.00	0.99
	refresh	refresh of the current page	1.00	1.00	1.00
	home page	select home page	1.00	0.93	0.96
	likes page	select user's likes page	1.00	1.00	1.00
	user page	select user info page	0.98	0.96	0.97
	following page	select following page	1.00	1.00	1.00
	tag input	input a tag in a post	1.00	1.00	1.00
	add tag	confirm a tag in a post	0.97	0.99	0.98
	other tumblr	other Tumblr network traffic	1.00	1.00	1.00
	Average Tumblr		0.99	0.99	0.99

Table 2.3: First part of the description of user actions for the considered apps and relative classification performance in terms of precision (P), recall (R) and F-measure (F1).

the case of *folder creation* or *delete file*. However, the average F-measure is still greater than 92%.

Google+: We analyzed ten different user actions of the Google+ app (see Table 2.4). On average precision, recall and F-measure are equal to 90%, 94%, 92% respectively. Precision values range from 75% to 100% for the individual actions, while recall is greater than 84% in all the cases. The actions *delete post* and *send comment* have both precision and recall equal to 100%.

Evernote: We analyzed six different user actions of the Evernote app.

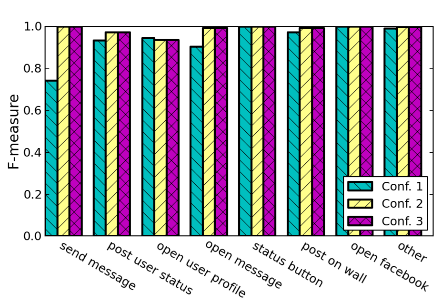
Apps	Actions	Description	P	R	F1
Dropbox	*open dropbox*	open the Dropbox app	1.00	0.97	0.98
	file favorited	mark a file as favorite	1.00	1.00	1.00
	content of file/folder	browse content of file/folder	1.00	0.81	0.90
	operation on file/folder	modify a file or a folder	0.81	0.95	0.87
	folder creation	creation of a new folder	0.75	0.99	0.86
	favorites page	select favorite file page	0.98	0.97	0.98
	delete file	deletion of a file	1.00	0.60	0.75
	delete folder	deletion of a folder	0.99	0.99	0.99
	other dropbox	other Dropbox network traffic	1.00	0.98	0.99
	Average Dropbox		0.95	0.92	0.92
Google+	*open gplus*	open the Google+ app	0.65	0.85	0.73
	refresh	refresh the current page	0.98	0.84	0.91
	user page	select user page	0.92	0.96	0.94
	new selection	open editor for a new post	0.75	0.96	0.84
	send post	publish a post or share a content	0.95	0.98	0.97
	delete post	delete a post	1.00	1.00	1.00
	post selection	select and open a post	0.77	0.85	0.81
	send comment	publish a comment	1.00	1.00	1.00
	back to main	back to the main page	0.92	0.96	0.94
	plus post	like or unlike a post	1.00	0.98	0.99
	other gplus	other Google+ network traffic	0.98	0.96	0.97
	Average Google+		0.90	0.94	0.92
Evernote	*open evernote*	open the Evernote app	1.00	1.00	1.00
	market page	select market page on menu	1.00	1.00	1.00
	note title input	input the title of a note	1.00	1.00	1.00
	text note done	save a text note	0.99	1.00	0.99
	note edit done	save modifies on a text note	1.00	0.99	1.00
	audio note done	save an audio note	1.00	1.00	1.00
	other evernote	other Evernote network traffic	1.00	1.00	1.00
	Average Evernote		1.00	1.00	1.00

Table 2.4: Second part of the description of user actions for the considered apps and relative classification performance in terms of precision (P), recall (R) and F-measure (F1).

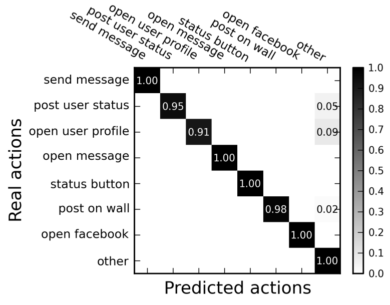
Evernote is definitely the app that achieved better performance among those we analyzed. Indeed, we achieved an average precision, recall and F-measure equal to 100%.

2.3.4 Comparison with other methods

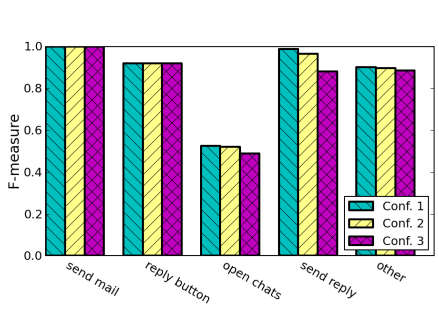
To confirm the validity of the proposed approach, we compared the results achieved by our solution with three traffic analysis techniques. The solutions we compare with have been proposed to face a problem similar to the one we consider, i.e., the identification of the websites the user is retrieving under the cover of an encrypted tunnel. Two of them are due to Liberatore et al. [116]. They proposed two methods that are based on naive Bayes and Jaccards coefficient respectively. The third one is due to Herrmann et al. [94]. They applied common text mining techniques to the frequency distribution of observable IP packet sizes. Since all these algorithms require



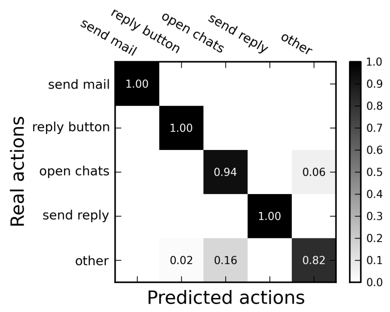
(a) Classification accuracy of the Facebook actions.



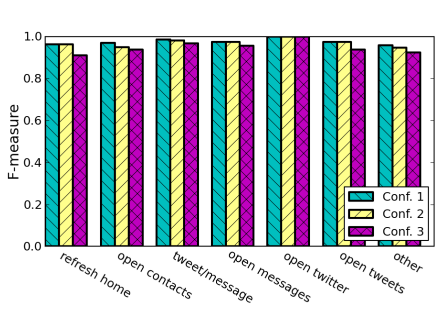
(b) Facebook actions confusion matrix for Configuration 3.



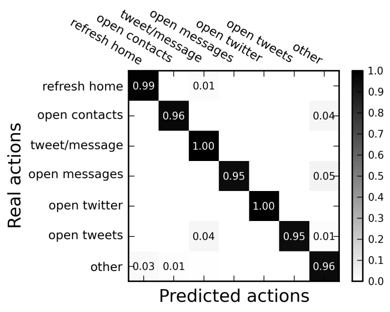
(c) Classification accuracy of the Gmail actions.



(d) Gmail actions confusion matrix for Configuration 1.



(e) Classification accuracy of the Twitter actions.



(f) Twitter actions confusion matrix for Configuration 1.

Figure 2.6: Classification accuracy and confusion matrices of Facebook, Gmail and Twitter actions.

several parameters to be tuned, we analyzed different configuration and in the following we report only the results for the best configuration that we found. Figure 2.7 reports the results of the comparison. Because we are space constrained, we report the results of the comparison only for some of the apps we analyzed. The other apps do not show a significantly different behavior. In particular, the performance of our solution is always comparable or significantly better than the performance of the other proposed approaches.

In particular, Figure 2.7a shows the averaged F-measure for Facebook, Gmail and Twitter. The averaged F-measure is the average of the F-measures reached by classifying the actions considered for that specific app. It can be noticed that in all the cases our classifier outperforms the other approaches. Figures 2.7b, 2.7c and 2.7d show the results for each app more in depth. In particular, each figure compares the F-measures reached when classifying the individual actions of that app. As it turns out, our classifier significantly outperforms the other three approaches in the majority of cases, while the results are comparable in the remaining cases. This indicates a higher level of reliability with respect to the other approaches.

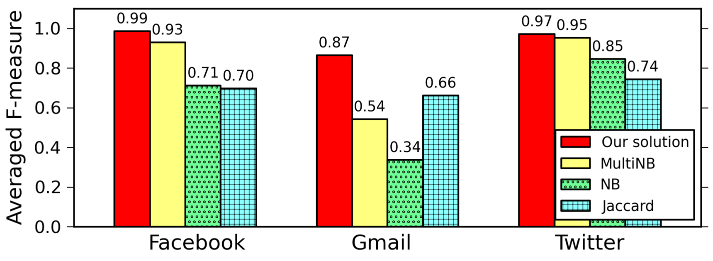
In contrast with the other algorithms, our solution uses more advanced machine learning techniques such as ensemble methods, Dynamic Time Warping, and hierarchical clustering. Furthermore, our solution uses information such as the packet order that is not considered in the other cases. Finally, our approach is resilient to packet retransmissions that might be significant in mobile apps. We believe that these features make our classifier more reliable than its competitors, especially for the mobile scenario. However, we want to highlight that our solution may also be competitive in desktop scenarios.

2.4 Possible Countermeasures and Limitations

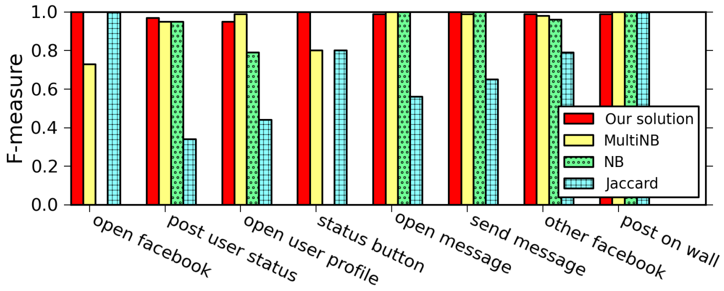
Users and service providers might believe that their two parties communications are secure if they use the right encryption and authentication mechanisms. Unfortunately, current secure communication mechanisms limit their traffic encryption actions to the syntax of the transmitted data. The semantic of the communication is not protected in any way [107]. For this reason, it has been possible for example to develop classifiers for TLS/SSL encrypted traffic that are able to discriminate between applications.

The contribution of this work was to investigate to which extent it is feasible to identify the specific actions that a user is doing on her mobile device, by simply eavesdropping the device's network traffic. While it is out of the scope of the work to investigate possible countermeasures to the proposed attack, we discuss in the following some related issues.

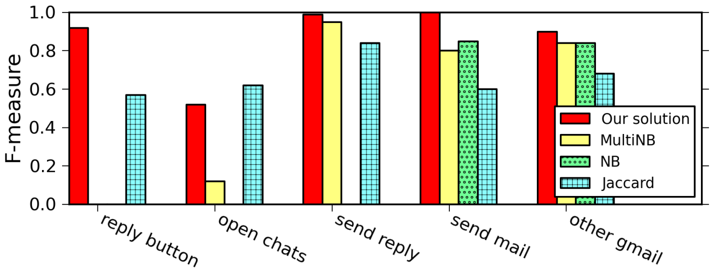
The common belief is that simple padding techniques may be effective against traffic analysis approaches. However, it has to be considered that padding countermeasures are already standardized in TLS, explicitly to "frustrate attacks on a protocol that are based on analysis of the lengths of exchanged messages" [59]. Nevertheless, our attack worked against TLS encrypted traffic. More advanced techniques have been proposed in the literature, such as traffic morphing and direct target sampling [206, 207].



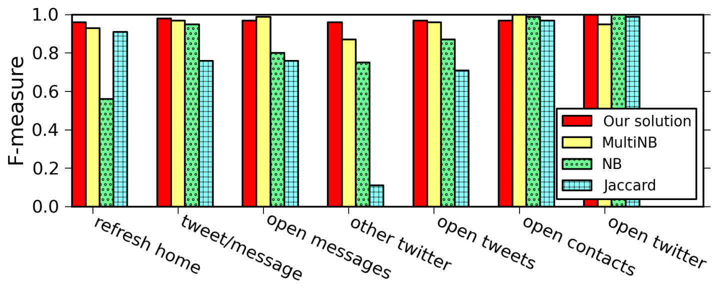
(a) Comparison of averaged F-measures.



(b) Comparison of performance classification of Facebook actions.



(c) Comparison of performance classification of Gmail actions.



(d) Comparison of performance classification of Twitter actions.

Figure 2.7: Comparison of our solution with Herrmann Multinomial Naive Bayes (*MultiNB*) [94], Liberatore Naive Bayes (*NB*) [116], and Jaccard (*Jaccard*) [116].

However, a recent result showed that none of the existing countermeasures are effective [62].

The intuition is that coarse information is unlikely to be hidden efficiently, and the analysis of these features may still allow an accurate analysis. On the light of these results, we believe it is not trivial to propose effective countermeasures to the attack we showed in this chapter. Indeed, it is the intention of the authors to highlight a problem that is becoming even more alarming after the revelation about the mass surveillance programs that are nowadays adopted by governments and nation states.

In our opinion, the main limitation of our approach is related the usage of supervised learning algorithms. It has to be considered that this technique is generally more efficient than the unsupervised learning since it takes advantage of the knowledge of each class of interest. However, it has two main drawbacks: (i) the training dataset has to be labeled with the intervention of a human, (ii) it is not possible to recognize classes of events that have not been used during the training phase. We mitigated the first limitation using an automatic approach to label the network traces collected for the training phase (see Section 2.3.2 for the details). However, the second limitation cannot be addressed without revising the entire approach. Furthermore, it has to be noticed that even applying an unsupervised learning technique the selection of the actions that could originate more privacy concerns should always be evaluated by a human.

A possible limitation in our framework is that is not able to recognize user actions that generate the same identical traffic pattern. In practice, given a flow with a fixed shape, a countermeasure may consists to apply a sort of padding on the entire flow (i.e., the sequence of packets in a flow must have the same number of packets, and each packet in the sequence must have the same size). Unfortunately, such countermeasure is not feasible since it will heavily hinder the scalability and the efficiency of data transmission, especially on energy constrained devices such as smartphones and tablets.

2.5 Summary

The proposed framework is able to analyze encrypted network traffic and to infer which particular actions the user executed on some apps installed on her mobile-phone. We demonstrated that despite the use of SSL/TLS, our network traffic analysis approach is an effective tool that an eavesdropper can leverage to undermine the privacy of mobile users. With this tool an adversary may easily learn habits of the target users. The adversary may aggregate data of thousands of users in order to gain some commercial or intelligence advantage against some competitor. In addition to that, a powerful attacker such as a Government, could use these insights in order to de-anonimize user actions that may be of particular interest.

Chapter 3

Mobile Apps Fingerprinting

Mobile devices allow users to install apps that provide various functionalities. In addition to that, many apps utilize Internet access and thus they generate network traffic. The combination of increased app usage coupled with app-generated network traffic makes the smartphone an attractive target for anyone seeking to uncover users' habits. Smartphone users typically install and use apps that are in line with their interests. As a result, the apps installed on typical smartphones may reveal sensitive information about a user [178], such as medical conditions, hobbies, and sexual and religious preferences. An adversary could also infer who a user banks with, what airline do they fly on, and which company provides them insurance. This information may be particularly useful in "spear phishing" attacks. In addition to uncovering the aforementioned sensitive information, an adversary can also use app identification to enumerate and exploit potentially vulnerable apps in an attempt to gain privileges on a smartphone.

Despite network traffic analysis is not a new area of research on traditional computers [148], on the domain of mobile devices it present some new challenges [50]. Unfortunately, app fingerprinting and identification on smartphones is frustrated in several ways. Port-based fingerprinting fails because apps deliver their data predominantly using HTTP/HTTPS. Typical web page fingerprinting fails since apps usually send data back and forth using text formats such as XML and JSON, thus removing rich information (such as the number of files and file sizes) that aid web page classification. Additionally, many apps use content delivery networks (CDNs) and third-party services, thus eliminating domain name resolution or IP address lookup as a viable strategy. Observing domain name resolution or TLS handshakes also proves less useful due to the use of CDNs. Moreover, DNS and TLS exchanges may not be observed at all due to the use of client-side caching, or simply due to the mobile nature (i.e., transient connectivity) of smartphones.

App fingerprinting may be useful in a variety of scenarios:

1. An adversary in possession of exploits for particular apps may use app fingerprinting to identify these vulnerable apps on a network to narrow their list of target devices.

2. An adversary on the same WiFi network as the victim could surreptitiously monitor the victim's network traffic to identify what apps are installed on a device for the purposes of blackmail.

3. App fingerprinting in the current era of bring-your-own-device (BYOD) can provide valuable data about the types of apps and usage patterns of these apps within an organization.

4. App fingerprinting can aid market research since app usage can be measured within a target population.

In this chapter, we present AppScanner, a highly-scalable and extensible framework for the fingerprinting and identification of apps from their network traffic. The framework is encryption-agnostic, and only analyzes side-channel data, thus making it perform well whether the network traffic is encrypted. Similarly to the work presented in Chapter 2, we exploit the fact that while SSL/TLS protects the payload of a network connection, it fails to hide other coarse information such as packet lengths and direction. Additionally, we evaluate the robustness of our app fingerprinting framework by measuring how it is affected by different devices, different app versions, or the mere passage of time. We make the following contributions to the state of the art:

1. Strategies for network traffic pre-processing that enable accurate extraction of features that can be reliably used to re-identify an app.

2. A method of obtaining perfect ground truth of what app is responsible for each network transmission using a novel demultiplexing strategy.

3. Design and full implementation[1] of an app classification system incorporating a novel machine learning strategy to identify ambiguous network traffic, i.e., traffic that is similar across apps.

4. An analysis of the robustness of app fingerprinting across different devices and app versions. We also analyze the time invariability of app fingerprints, by measuring how performance is affected when attempting to identify apps using fingerprints generated six months earlier.

[1]The code and datasets used for this chapter are available at `https://github.com/vftaylor/appscanner`.

5. Evidence that app fingerprints are time, app version, and device invariant to several extents. This lends support to the idea that app classification can be useful in real-world settings.

The rest of the chapter is organized as follows: In Section 3.1, we survey of related work. We elucidate how our system works at a high-level and explains key terminology in Section 3.2. In Section 3.2, we outline our approach to identifying ambiguous traffic and in Section 3.3 we overview the datasets that were collected We evaluate system performance in Section 3.4 and we present some ways of improving classifier accuracy using post-processing strategies in Section 3.5. In Section 3.6, we discuss our results. Finnaly, we conclude the chapter in Section 3.7.

3.1 Related Work

Since the attack scenario is the same we presented in the previous chapter, we point the reader to Section 2.1 for the review of the related work about traffic analysis for privacy attacks and for mobile devices. In what follow, we focus on the state of the art of mobile app fingerprinting.

In early work on the topic, Dai et al. [57] propose NetworkProfiler, an automated approach to profiling and identifying Android apps using dynamic methods. They use user-interface fuzzing (UI fuzzing) to automatically explore different activities and functions within an app, while capturing and logging the resulting network traffic. The authors inspect HTTP payloads in their analysis and thus this technique only works with unencrypted traffic. The authors did not have the full ground truth of the traffic traces they were analyzing, so it is difficult to systematically quantify how accurate NetworkProfiler was in terms of precision, recall, and overall accuracy.

Qazi et al. [163] present Atlas, a framework for identifying apps using network flows. Atlas uses crowd-sourcing to obtain ground truth. The authors tested their system on 40 Android applications and achieved an average performance of 94%. However, it remains unclear whether Atlas maintains good performance as the number of apps to be classified increases. Le et al. [113] propose AntMonitor, a system that also uses crowd-sourcing but for the fine-grained collection of network data from Android devices. In contrast, AppScanner does not leverage crowd-sourcing approaches. Indeed, AppScanner is able to obtain perfect ground truth and does so in a scalable way using UI fuzzing (see Section 3.2.2).

Stöber et al. [190] propose a scheme for identifying a mobile device using the characteristic traffic patterns it produces. The authors posit that 3G transmissions can be realistically intercepted and demodulated to obtain side channel information such as the amount of data and timing information. The authors leverage network bursts from which they extract features since they cannot analyze the TCP payload directly. Using supervised learning, the authors build a model of the background traffic coming from devices. With their system, using approximately 15 minutes of captured traffic can result in a classification accuracy of over 90%. A major drawback with this work is that the system needs six hours of training and 15 minutes of monitoring to achieve reliable fingerprint matching.

Mongkolluksamee et al. [135, 136] use packet size distribution and communication patterns for identifying mobile app traffic. The authors achieve an F-score of approximately 95%. Unfortunately, they only consider five apps so it remains unclear how scalable their approach is. The authors also fail to collect perfect ground truth because their methodology calls for running one app at a time on a device to reduce noise instead of more robust approaches (see Section 3.2.2). Alan and Kaur [5] use TCP/IP headers for identifying apps. The authors identified apps with up to 88% accuracy using packet size information from the first 64 packets generated upon app launch. The authors found that performance decreases when training and testing devices are different. They also found that performance decreases only slightly when several days have passed between training and testing. Complementary to this, we investigate the problem using all network traffic coming from apps. By collecting data over a period of six months instead of several days, we show that traffic classification is more severely impacted by time and that additional strategies to improve performance need to be employed.

Wang et al. [203] propose a system for identifying smartphone apps from encrypted 802.11 frames. They collect data from target apps by running them dynamically and training classifiers with features from Layer 2 frames. The authors test 13 arbitrarily chosen apps from eight distinct app store categories and collect network traces for five minutes. By taking into account a larger set of apps, we show that increasing the number of apps negatively impacts classifier accuracy. Wang et al. also fail to collect perfect ground truth. Indeed, our methodology minimizes noise by running a single app at a time, and we still had to filter 13% of the traffic collected because it was background traffic from other apps. AppScanner solves the aforementioned problems by using a larger sample of apps from a wider set of categories and collecting network traffic for substantially more time.

3.2 System Overview

As an overview, AppScanner fingerprints smartphone apps by using machine learning to understand the network traffic that has been generated by them. Patterns in app-generated traffic, when later seen, are used to identify apps.

Unfortunately, apps sometimes have common traffic patterns because they share libraries, such as ad libraries, that generate similar traffic[2] across distinct apps. This can frustrate attempts at app classification using traffic analysis, since it may generate false positives. Thus, a strategy is needed to first identify traffic that is shared among apps, so that it can be appropriately labeled before being passed to classifiers. We call traffic shared among apps *ambiguous traffic* and the remaining traffic *distinctive traffic*.

In what follows, we introduce the concepts of *burst* and *flow*, which are central to our fingerprinting methodology. We point out that the concept of flow is slightly different from the one we defined in Section 2.2.1.

- **Burst:** A burst is the group of all network packets (irrespective of source or destination address) occurring together that satisfies the condition that the most recent packet occurs within a threshold of time, the *burst threshold*, of the previous packet. In other words, packets are grouped temporally and a new group is created only when no new packet has arrived within the amount of time set as the burst threshold. This is visually depicted as *Traffic burstification* in Figure 3.1(c), where we can see Burst A and Burst B separated by the *burst threshold*. We use the concept of a burst to logically divide the network traffic into discrete, manageable portions, which can then be further processed.

- **Flow:** A flow is a sequence of packets (within a burst) with the same remote IP address. That is, within a flow, all packets will either be going to (or coming from) the same remote IP address. A flow is not to be confused with a TCP session. A flow ends at the end of a burst, while a TCP session can span multiple bursts. Thus, flows typically last for a few seconds, while TCP sessions can continue indefinitely. AppScanner leverages flows instead of TCP sessions to achieve real-time/near-to-real-time classification. From *Flow separation* in Figure 3.2(d), it can be seen that a burst may contain one or more flows. Flows may overlap in a burst if a single app, *App X*, initiates TCP sessions in quick succession or if another app (e.g., *App Y*), happens to initiate a TCP session at the same time as *App X*.

Our app identification framework first elicits network traffic from an app, generates features from that traffic, trains classifiers using these features, and finally identifies apps when the classifiers are later presented with unknown traffic.

[2]Traffic generated by third-party libraries will typically be common among apps using that particular library.

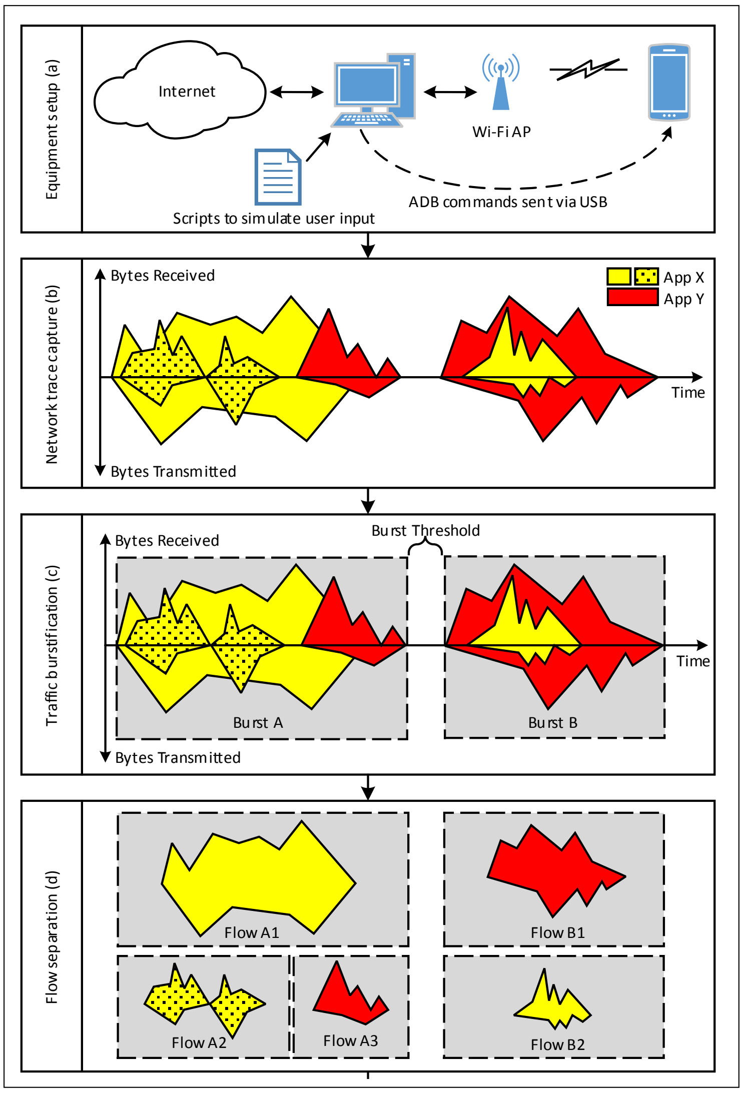


Figure 3.1: First part of the high-level representation of classifier training, and a visualization of bursts and flows within network traffic. (a) Equipment setup. (b) Network trace capture. (c) Traffic burstification. (d) Flow separation.

3.2.1 Equipment Setup

The setup used to collect network traces from apps is depicted as *Equipment setup* in Figure 3.1(a). The workstation was configured to forward traffic between the WiFi access point (AP) and the Internet. To generate traffic from which to capture our training/testing data, we used scripts that communicated with the target smartphone via USB using the Android Debug Bridge (ADB). These scripts were used to simulate user actions within apps

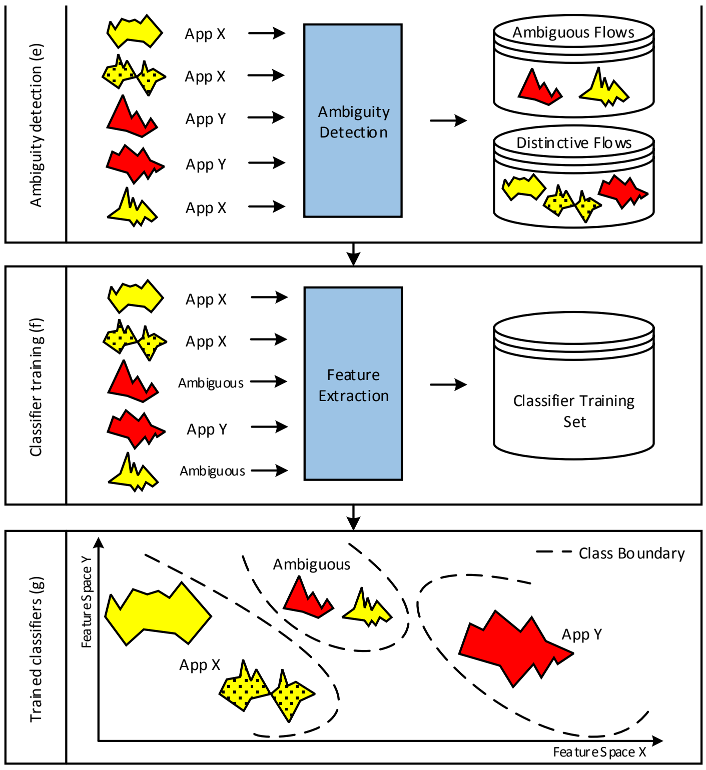


Figure 3.2: Second part of the High-level representation of classifier training, and a visualization of bursts and flows within network traffic. (e) Ambiguity detection. (f) Classifier training. (g) Trained classifiers.

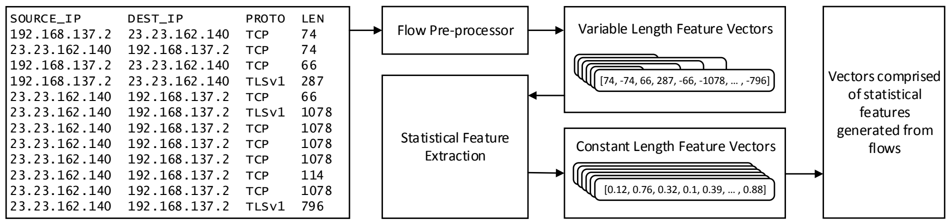
and thus elicit network flows from the apps. This technique is called UI fuzzing.

The traffic generated by the smartphone was captured and exported as network traffic dumps containing details of captured packets. We collected packet details such as time, source address, destination address, source port, destination port, packet size, protocol and TCP/IP flags. The payload for each packet was also collected but was not used to provide features since it may or may not be encrypted. Although physical hardware was used for network traffic generation and capturing, this process can be massively automated and parallelized by running apps within Android emulators on virtual machines.

3.2.2 Construction of an App Fingerprint

In what follows, we describe the stages in the fingerprint making process.

Network Trace Capture: Network traffic from apps was elicited automat-



```
SOURCE_IP      DEST_IP        PROTO  LEN
192.168.137.2  23.23.162.140  TCP    74
23.23.162.140  192.168.137.2  TCP    74
192.168.137.2  23.23.162.140  TCP    66
192.168.137.2  23.23.162.140  TLSv1  287
23.23.162.140  192.168.137.2  TCP    66
23.23.162.140  192.168.137.2  TLSv1  1078
23.23.162.140  192.168.137.2  TCP    1078
23.23.162.140  192.168.137.2  TCP    1078
23.23.162.140  192.168.137.2  TCP    1078
23.23.162.140  192.168.137.2  TCP    114
23.23.162.140  192.168.137.2  TCP    1078
23.23.162.140  192.168.137.2  TLSv1  796
```

Flow Pre-processor
Variable Length Feature Vectors
[74, -74, 66, 287, -66, -1078, ... , -796]
Statistical Feature Extraction
Constant Length Feature Vectors
[0.12, 0.76, 0.32, 0.1, 0.39, ... , 0.88]
Vectors comprised of statistical features generated from flows

Figure 3.3: Generating features from flows for classifier training.

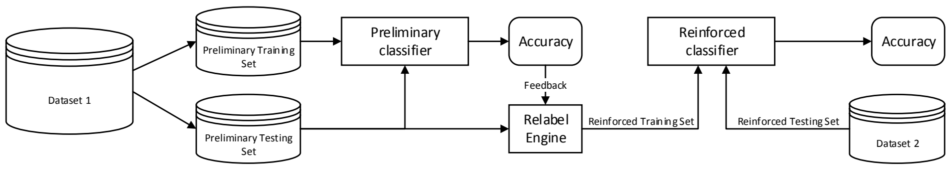


Figure 3.4: Using reinforcement learning to obtain robustness against ambiguous flows.

ically using UI fuzzing. UI fuzzing involves using scripts on a workstation to interact with the target device through the Android Debug Bridge (ADB). User interface events such as button presses, swipes, and key-presses are sent to apps in an automated fashion. These touchscreen input events cause the logic within apps to execute, thus generating network traffic.

We performed UI fuzzing on one app at a time to minimize 'noise' (i.e., traffic generated simultaneously by other apps) in the network traces. Traffic from other apps or the Android operating system itself could interfere with and taint the fingerprint making process. To combat the problem of noise, the Network Log tool [160] was used to identify the app responsible for each network flow. Using data from Network Log combined with a 'demultiplexing' script, all traffic that did not originate from the target app was removed from the traffic dump for that app. In this way, and in contrast to related work, we obtained perfect ground truth of what flows came from what app.

After data collection, the network traffic dumps were filtered to include only TCP traffic that was error free. For example, we filtered to remove packet retransmissions that were as a result of network errors.

Traffic Burstification and Flow Separation: The next step was to parse the network dumps to obtain network traffic bursts. Traffic was first discretized into bursts to obtain ephemeral chunks of network traffic that could be sent immediately to the next stage of AppScanner for processing. This allows us to meet the design objective of real-time or near real-time classification of network traffic. Falaki et al. [67] observed that 95% of packets on smartphones "are received or transmitted within 4.5 seconds of the previous packet". During our tests, we observed that setting the burst threshold to one second instead of 4.5 seconds only slightly increased the number of bursts seen in the network traces. This suggests that network performance (in terms of bandwidth and latency) has improved since the original study. For this reason, we opted to use a burst threshold of one second to favor more overall bursts and nearer-to-real-time performance. Bursts were separated into individual flows (as defined at the beginning of this section and depicted in Figure 3.3) using remote IP address. We enforced a maximum flow length that would be considered by the system. This is simply to ensure that abnormal traffic can be safely ignored in the real-world.

It is important to note that while destination IP addresses were used for flow separation, they were not leveraged to assist with app identification. We also opted not to use information gleaned from DNS queries or flows with unencrypted payloads. We took this design decision to avoid the reliance on domain-specific knowledge that frequently changes, thus making our framework useful in the long term.

Ambiguity Detection: As mentioned at the beginning of this section, many apps have third-party libraries in common (especially ad libraries) and these libraries themselves generate network traffic. Unfortunately, it is not possible to discriminate traffic coming from libraries (as opposed to the app that embeds the library) in a scalable way, i.e., without an intrusive approach such as reverse-engineering or modifying apps. Indeed, as far as the operating system is concerned, apps and their bundled libraries are one entity within the same process. Since network traffic generated by common libraries across apps is similar, this will frustrate the fingerprinting process because classifiers will be given contradictory training examples. This problem of *ambiguous flows* poses a challenge to naive machine learning approaches. To mitigate negative effects, we introduce *Ambiguity Detection* as detailed in Section 3.2.4. Ambiguity detection uses simple reinforcement learning techniques to identify similar flows coming from different apps. In the training phase, ambiguous flows are detected and re-labeled as belonging to the "ambiguous" class, so that the system is later able to properly identify and handle them.

Classifier Training: Statistical features were generated from flows and used to train classifiers. Statistical feature extraction involves deriving 54 statistical features from each flow as shown in Figure 3.3. For each flow, three vectors are considered: size of incoming packets only, size of outgoing packets only, and size of both incoming and outgoing packets. For each vector (3 in total), the following values were computed: minimum, maximum, mean, median absolute deviation, standard deviation, variance, skew, kurtosis, percentiles (from 10% to 90%), and the number of elements in the series (18 in total). These statistical features are computed using the Python pandas [131] library. Thus, arbitrary length flows are converted to feature vectors of length 54. These feature vectors and their corresponding ground truth are used as training examples.

3.2.3 App Identification

Unknown flows are passed to the trained classifiers. Ambiguous flows are identified and labeled as such, since the classifiers were trained to understand ambiguous flows. Flows that are not labeled by the classifiers as ambiguous next go through *classification validation* as described in Section 3.5.2. The classification validation stage is crucial for one primary reason. Machine learning algorithms will always attempt to place an un-labeled example into the class it most closely resembles, even if the match is not very good. Given that our classifiers will never be trained with the universe of flows from apps, it follows that there will be some flows presented to AppScanner which are simply unknown or never-before-seen. If left unchecked, this can cause an undesirable increase in the false positive (FP) rate.

To counteract these problems, we leverage the prediction probability metric (available in many classifiers) to understand how certain the classifier is about each of its classifications. For example, if the classifier labeled an unknown sample as *com.facebook.katana*, we would check its prediction probability value for that classification to determine the classifier's confidence. If this value is below the classification validation threshold, AppScanner will not make a pronouncement. However, if this value exceeds the threshold, AppScanner would report it as a match for that particular app. In Section 3.5, we discuss how varying this threshold impacts the precision, recall, and overall accuracy of AppScanner, as well as how this affects the percentage of total flows that the classifiers are confident enough to classify.

3.2.4 Ambiguity Detection

The ambiguity detection phase aims to identify and relabel ambiguous flows. This phase involves a reinforcement learning strategy that is leveraged during classifier training. As outlined in Figure 3.4, classifier training is divided into two stages: the *preliminary classifier* stage, and the *reinforced classifier* stage.

The main training set considered in the analysis is first randomly shuffled and divided into halves: the *preliminary training set* and the *preliminary testing set*. The preliminary training set is used to train the preliminary classifier. The preliminary testing set is used to measure the accuracy of the preliminary classifier, and as a basis for generating the training set for the reinforced classifier. In this way, we can first identify which flows are

Name	**Device**	**Android version**	**N. of apps**	**App versions**	**Time of data collection**
`Dataset-1`	Motorola XT1039	4.4.4	110	Latest versions at T_0	T_0
`Dataset-1a`	Motorola XT1039	4.4.4	65	Latest versions at T_0	T_0
`Dataset-2`	Motorola XT1039	4.4.4	65	Latest versions at T_0	T_0 + 6 months
`Dataset-3`	LG E960	5.1.1	65	Latest versions at T_0	T_0 + 6 months
`Dataset-4`	Motorola XT1039	4.4.4	110	Latest versions at T_0 + 6 months	T_0 + 6 months
`Dataset-4a`	Motorola XT1039	4.4.4	65	Latest versions at T_0 + 6 months	T_0 + 6 months
`Dataset-5`	LG E960	5.1.1	110	Latest versions at T_0 + 6 months	T_0 + 6 months
`Dataset-5a`	LG E960	5.1.1	65	Latest versions at T_0 + 6 months	T_0 + 6 months

Table 3.1: Descriptions of the devices, operating systems, number of apps, app versions, and time of data collection for each dataset used.

incorrectly classified by the preliminary classifier. We validated that these incorrectly labeled flows are to a large extent library traffic, as expected.

The Relabel Engine leverages feedback on the accuracy of the preliminary classifier to identify ambiguous flows. Flows in the preliminary testing set that are incorrectly classified are re-labeled as "ambiguous" by the *Relabel Engine*. On the other hand, flows that are correctly classified by the preliminary classifier keep their original label (i.e., the app that generated them). This relabeled dataset is now used as the reinforced training set and is passed to the reinforced classifier. The reinforced classifier is thus equipped to identify ambiguous flows since it is trained with examples of ambiguous flows.

We emphasize to the reader that no flows from the preliminary training set are used in the reinforced training set. The preliminary classifier and the preliminary training set are only used as a means of identifying ambiguous flows so that additional knowledge can be provided to the reinforced classifier.

3.3 Dataset Collection

To test the performance of AppScanner, we considered a random 110 of the 200 most popular free apps as listed by the Google Play Store. We chose the most popular apps because they form a large part of the install-base of apps across the world. Additionally, we chose free apps because free apps tend to be ad-supported and thus use ad libraries. There is a small set of major ad libraries and thus ad libraries tend to be shared across apps. This suggests that free apps will be more likely to generate ambiguous flows than paid apps. Being able to properly fingerprint and identify free apps thus implies that AppScanner is robust enough to handle paid apps as well.

Smartphones in our testbed were connected to the Internet via a Linksys E1700 WiFi router/AP that had its Internet connection routed through a workstation. UI fuzzing was performed on each app for 30 minutes using the `MonkeyRunner` tool from the Android SDK. UI fuzzing simulated user actions by invoking UI events such as touches, swipes, and button presses. These UI events were generated randomly and sent to apps. It is worth noting that some apps presented login screens upon first launch. In such cases, we first manually created accounts for those apps before logging in. We did this to ensure that traffic generation using UI fuzzing was not hindered by a login screen. Greater coverage of all the network flows in an app may theoretically be obtained by using advanced UI fuzzing techniques provided by frameworks such as Dynodroid [125], or by recruiting human participants. However, we consider these approaches to be out of the scope of our research.

A major contribution of this work is to understand how app fingerprinting is affected by time, the device used, app versions, and combinations of these variables. For this reason, we collected several datasets as outlined in Table 3.1. In what follows, we describe these datasets in detail.

The dataset we consider as our baseline is `Dataset-1`, which was collected using *Device-A*, a Motorola XT1039 running Android version 4.4.4. This dataset contains network traffic from 110 apps using the latest version of each app at the time of initial data collection. We refer to this time of initial data collection as T_0. All other main datasets (`Dataset-2` to `Dataset-5`) were collected six months after T_0, i.e., at time $T_0 + 6$ months.

`Dataset-2` differs from `Dataset-1` only by the time of data collection. `Dataset-2` contains data from only 65 apps (instead of 110), because the remaining 45 apps refused to run without being updated. We hereafter refer to the 65 apps in `Dataset-2` that ran without being updated as the *run-without-update* subset.

`Dataset-3` was collected using *Device-B*, an LG E960 running Android version 5.1.1. `Dataset-3` also used the *run-without-update* subset.

`Dataset-4` and `Dataset-5` were obtained by collecting network traffic from the latest versions (at the time of data collection six months after initial data collection) of the original 110 apps and were collected using *Device-A* and *Device-B* respectively.

Additionally, we consider variants of the datasets that had 110 apps. These variants consider only apps in the *run-without-update* subset. We denote these dataset variants as `Dataset-1a`, `Dataset-4a`, and `Dataset-5a` for `Dataset-1`, `Dataset-4` and `Dataset-5`, respectively. These datasets were

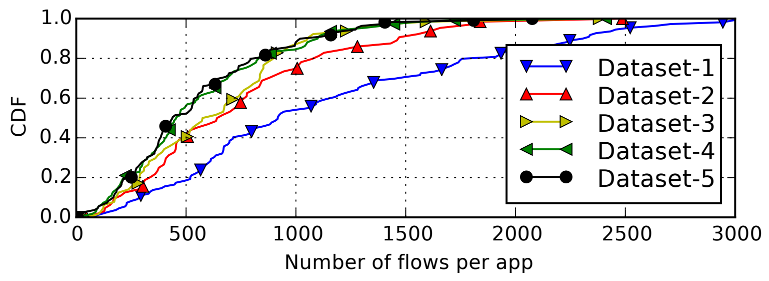


Figure 3.5: CDF plot showing the number of flows per app in each dataset.

generated in order to offer a balanced analysis in the presence of datasets with less than 110 apps (i.e., `Dataset-2` and `Dataset-3`).

Figure 3.5 shows a cumulative distribution function (CDF) of the number of flows per app in each of the five main datasets. Using `Dataset-1` as an example, an average of 1132 flows were collected per app during UI fuzzing. In the same dataset, approximately 80% of apps had 500 or more flows. Other datasets contained fewer flows per app on average. Nonetheless, recall that AppScanner identifies individual flows. Thus, even one flow is sufficient to successfully identify an app, if that flow is distinctive for that app.

3.4 Evaluation

In evaluating our system, we followed a two-step procedure. First, we report the results of a baseline evaluation of system performance using training and testing sets derived from single datasets. Second, to obtain a more representative measurement of system performance, we performed a comprehensive suite of tests (as outlined in Table 3.3) using completely independent training and testing sets. Measurements were taken to understand how factors such as time, device (including operating system), app version, and a combination of device and app version affected performance.

We leveraged the *scikit-learn* [37] machine learning libraries to implement the classifiers in our framework. All classifiers were set to use default parameters. Random forest classifiers were chosen since they gave superior performance over support vector machines (SVMs) in our preliminary tests. Random forest classifiers use aggregated decision trees which in turn reduce bias. Additionally, these classifiers are intrinsically multi-class classifiers, making them suitable for tasks such as app classification. Moreover, random forest classifiers natively give the probability of belonging to a class, a feature we use in classification validation (Section 3.5.2). Although SVMs can handle multi-class problems and output the probability of belonging to a class, these features are not native to SVMs.

We highlight to the reader that any results reported in this section should be considered as lower bounds of system performance. Indeed, the results presented in this section show the performance of the system before any performance-enhancers, such as ambiguity detection and classification validation (Section 3.5), have been applied. The tests performed in this section are merely to assess default system performance before post-processing is applied.

For our baseline results, we split each dataset into a training set (75% of flows) and a testing set (25% of flows) and used them to train classifiers as detailed in Section 3.2. Each test was run 50 times with randomly shuffled datasets and the results were averaged. We report the performance of our system in Table 3.2. Accuracy within datasets fell between 65.5% and 73.7%. These results are fairly good but may overestimate the performance of the system. This is because the training and testing sets in each case were generated from one original dataset.

In what follows, we do more robust measurements by using completely independent datasets for training and testing to make a more real-world assessment of system performance. Each test was run 50 times with randomly shuffled datasets and the results averaged. The results in each case are summarized in Table 3.3.

3.4.1 Effect of Time

To measure the effect of time on classification performance, we trained a classifier with `Dataset-1a` and tested with `Dataset-2`. This combination of training and testing sets assessed the effect of keeping device and app versions constant, but causing six months to pass between collection of data for training and testing. The overall accuracy for this test, called the `TIME` test, was 40.9% and was the highest performance of our tests that used completely separate training and testing sets.

Among tests with completely independent training and testing sets, `TIME` gave the highest performance. This result is not surprising, since the app versions and device (including operating system version) were constant. The logic (app and operating system) that generates traffic seems to generate the same traffic even after some amount of time (in this case six months) has elapsed. Since the underlying logic does not change, it would be reasonable to expect app fingerprints to also remain constant.

3.4.2 Effect of a Different Device

To assess the impact of a different device on app classification we did three tests: `D-110`, `D-110A`, and `D-65`. `D-110` used `Dataset-4` as a training set and `Dataset-5` as a testing set. That is, we trained with 110 apps on one device and tested with the same 110 apps on a different device. The overall accu-

Dataset	Precision(%)	Recall(%)	F1(%)	Accuracy (%)
Dataset-1	74.5	72.8	73.1	73.1
Dataset-1a	74.4	73.2	73.4	73.7
Dataset-2	68.8	67.6	67.7	65.5
Dataset-3	71.3	69.5	69.8	70.4
Dataset-4	68.6	66.6	66.9	67.4
Dataset-4a	68.5	66.8	67.1	66.9
Dataset-5	69.7	68.1	68.3	69.6
Dataset-5a	67.7	65.5	66.0	67.5

Table 3.2: Baseline performance of app classification for each dataset without any post-processing techniques applied.

Test Name	Training Set	Test Set	P(%)	R(%)	F1 (%)	A(%)	Independent Variable	Apps
TIME	Dataset-1a	Dataset-2	44.5	43.1	42.6	40.9	Time	65
D-110	Dataset-4	Dataset-5	40.9	36.6	36.3	37.6	Device	110
D-110A	Dataset-4a	Dataset-5a	38.4	35.2	35.1	37.7	Device	65
D-65	Dataset-2	Dataset-3	43.5	38.3	39.0	39.6	Device	65
V-LG	Dataset-3	Dataset-5a	33.0	31.0	30.0	30.3	App	65
V-MG	Dataset-2	Dataset-4a	34.8	32.1	32.1	32.7	App	65
DV-110	Dataset-1	Dataset-5	23.3	19.5	19.3	19.2	Device & App	110
DV-65	Dataset-1a	Dataset-5a	22.3	19.7	19.3	19.5	Device & App	65

Table 3.3: Summary of the comprehensive suite of tests used to measure the performance of the app classification system in terms of precision (P), recall (R), F-measure (F1), and Accuracy (A). All training and testing sets were completely independent of each other. The independent variables for each test are identified.

racy for D-110 was 37.6%. D-110A used the *run-without-update* subsets of the datasets used in D-110 and had an overall accuracy of 37.7%. D-65 was conducted with a training set of Dataset-2 and testing set of Dataset-3. That is, we trained with 65 apps on one device and tested with 65 apps on another device. The overall accuracy for this test was 39.6%. We note that this test, with 65 apps, gives performance comparable to the TIME test, which also had 65 apps. This insight suggests that device model and operating system version does not have a significant effect on app fingerprinting performance.

3.4.3 Effect of Different App Versions

We carried out two tests to understand the impact that different app versions had on app fingerprinting. V-LG involved training with Dataset-3 and testing with Dataset-5a. For this test, the same device was used but with different versions of the same apps. The overall accuracy of this test was 30.3%. V-MG used a training set of Dataset-2 and testing set of Dataset-4a. The overall accuracy for this test was 32.7%. We note that the accuracy for both of these tests were fairly similar but markedly lower than the `TIME`, `D-110` or `D-110A` or `D-65` tests. This insight suggests that changes in app versions affects the reliability of app fingerprinting. We believe that this phenomenon could be due to changes in app code or logic that has direct consequences on the way that an app generates network flows. Thus there is a need to keep app fingerprint databases up-to-date as app developers release new app versions.

3.4.4 Effect of a Different Device and Different App Versions

A final two tests were conducted to measure the impact of changing both device and app versions. The first test, `DV-110`, used a training set of `Dataset-1` and a testing set of `Dataset-5`, i.e., using a total of 110 apps. The second test, `DV-65`, used a training set of `Dataset-1a` and testing set of `Dataset-5a`. These tests yielded overall accuracies of 19.2% and 19.5% respectively. As expected from the results of our previous tests, changing both device and app versions together more severely impacted classification performance. It is interesting to note, however, that the number of apps in training and testing sets did not seem to impact overall classification accuracy in a negative way under these adverse conditions. This result lends support to the idea that app fingerprinting can be a scalable endeavor. We note that despite `DV-110` and `DV-65` having approximately half the accuracy of the `TIME` test, they still perform approximately 20 times better than pure random guessing.

3.5 Improving Accuracy

Our results so far show the performance of AppScanner without any post-processing applied. Additionally, these results simulate laboratory conditions since they are taken from datasets that have been filtered of noise (using NetworkLog as described in Section 3.2.2). In this section, we look at two post-processing strategies that have proven effective in improving the accuracy of the system: ambiguity detection and classification validation. Ambiguity detection is detailed in Section 3.2.4/3.5.1 and classification validation is discussed in Section 3.5.2. In general, both of these strategies aim to identify network flows that are not reliable for app fingerprinting.

Also in the section, we analyze the impact of noise on classification accuracy. Noise was filtered in our previous tests. While an attacker can easily filter noise during training, they are unable to filter noise during testing. Thus, we examine the impact of noise using three (one laboratory and two real-world) experimental settings:

1. The *noise-filtered* setting removes all noise from training and testing sets using NetworkLog. This gives the performance of the system assuming that devices do not have any network traffic being generated from non-app sources (such as the operating system).

2. The *noise-ignored* setting removes noise from the training sets, but leaves noise in the testing sets. This gives a realistic estimation of performance that might be expected in a real-world attack scenario. This is because the attacker can remove noise from their training sets but is unable to remove noise during an attack.

3. The *noise-managed* setting goes a step further by actually identifying and labeling noise in both training and testing sets. This allows the classifiers to understand what network traffic coming from the Android operating system itself looks like. Thus, classifiers are better able to identify noise in the real-world, which further improves accuracy.

Unless otherwise stated, subsequent results are generated using the *noise-managed* experimental setting.

3.5.1 Ambiguity Detection

As mentioned in Section 3.2.4, many apps have traffic in common and this can hinder app classification if left unhandled. Our reinforcement learning approach identifies and relabels ambiguous flows so that the classifiers have a model to identify them. When measuring performance with ambiguity detection in use, unknown flows that are labeled as ambiguous are omitted from calculations of classifier performance. That is, ambiguous flows are identified and ignored, and thus do not affect the measurement of the performance of our system.

In what follows, we report on the improvements that can be made by using our reinforcement learning approach to identify ambiguous traffic flows. Table 3.4 shows the improvement in performance obtained by applying ambiguity detection as outlined in Figure 3.4. Additionally, the table shows the number of flows in each testing set and the percentage of flows that were considered ambiguous. Each test used the training and testing sets described in Table 3.3 but with varying noise handling to provide results for each of the *noise-filtered*, *noise-ignored*, and *noise-managed* experimental settings. For these tests, reinforced classifiers are used so that ambiguous traffic can be managed. Ambiguity detection was applied to the training sets of these reinforced classifiers as detailed in Section 3.2.4. Each test was run 50 times with randomly shuffled data and the results were averaged.

Reinforced classifiers received an approximately twofold boost in overall accuracy. The most challenging tests, `DV-110` and `DV-65` (using different physical devices, Android versions, and app versions between training and

Test	Noise filtered				Noise ignored				Noise managed				Testing Set Details	
Name	**P(%)**	**R(%)**	**F1(%)**	**A(%)**	**P(%)**	**R(%)**	**F1(%)**	**A(%)**	**P(%)**	**R(%)**	**F1(%)**	**A(%)**	**Flows**	**Ambiguous**
TIME	66.9	66.4	65.7	72.9	61.4	65.3	61.9	64.5	66.4	65.3	64.6	74.8	54240	58.3%
D–110	62.2	57.2	56.3	66.2	56.1	56.9	53.0	55.4	59.5	53.5	53.0	64.7	73811	59.6%
D–110A	60.5	55.9	55.6	65.9	53.7	55.0	51.6	54.5	57.3	54.0	53.1	63.4	51995	60.4%
D–65	64.4	59.8	59.4	67.5	59.4	59.0	56.0	59.0	63.8	58.8	58.5	68.1	47018	58.6%
V–LG	47.8	47.3	44.6	52.8	41.4	46.6	41.4	43.9	44.7	45.6	42.5	54.5	51995	61.9%
V–MG	52.4	50.6	49.5	58.1	48.9	50.0	47.2	50.2	52.8	50.3	49.3	62.4	51731	61.8%
DV–110	37.0	35.0	33.2	41.0	32.8	34.8	31.0	32.4	35.0	33.7	32.0	46.3	73811	67.5%
DV–65	35.8	34.8	32.9	39.8	30.9	34.0	30.2	31.3	33.8	33.8	31.5	44.0	51995	67.2%

Table 3.4: The reinforcement learning strategy, for each of the tests that were conducted, improved classifier performance in terms of precision (P), recall (R), F-measure (F1), and Accuracy (A).

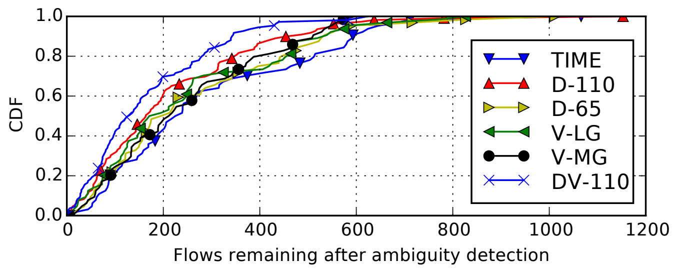


Figure 3.6: CDF plot showing the number of flows remaining per test after ambiguity detection was applied.

testing sets), had the greatest percentage increases in performance and saw accuracy more than double when using reinforced classifiers. For example, in `DV-110`, accuracy was increased from 19.2% to 41.0-46.3% using ambiguity detection. Improving performance using reinforced classifiers highlights the prevalence of ambiguous flows in app traffic and reiterates the need for systems that can address them.

Ambiguity detection improves accuracy at the expense of the number of flows that are classified by the system. In the worst case, many apps would have all flows considered as ambiguous, and thus could not be classified by the system if ambiguity detection is used. Figure 3.6 shows the number of flows per app that were considered to not be ambiguous flows. For clarity, we show results from only the *noise-managed* experimental setting in this plot. Other settings gave similar plots. For additional clarity, we omit `D-110A` and `DV-65` since they are scaled-down versions of `D-110` and `DV-110` respectively. For all tests except `DV-110`, all apps had some flows that were unambiguous, i.e., there were some flows remaining for every app after ambiguity detection.

For test `DV-110`, there was one app that did not have unambiguous flows. In order to identify this app, ambiguity detection would have to be abandoned (at the expense of system accuracy), or the app would have to be re-fingerprinted in an attempt to obtain additional flows that may be unambiguous. There also remains the possibility that some apps simply cannot be identified by AppScanner because they do not generate unambiguous network flows at all. This is a limitation of app identification using network traffic analysis.

3.5.2 Classification Validation

Classification validation is another effective strategy that can be leveraged to improve app classification performance. Classifiers can be made to output their confidence when labeling and unknown example. In simple terms, a classifier may be very confident about a classification if the class boundaries within its models are distinct, i.e., with sufficient separation between classes. In other cases, this distinction may be less clear.

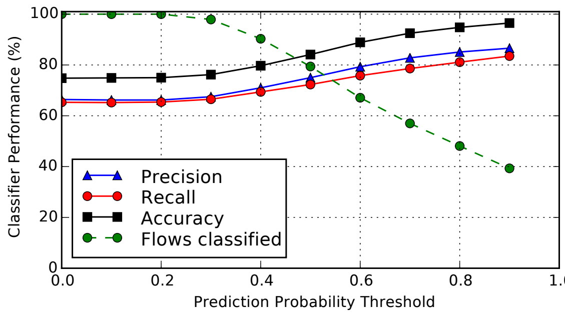
By assessing the confidence that a classifier reports with its classification, a judgment can be made as to whether the classification will be considered as valid by the system. We call the cut-off for what is considered a valid classification the prediction probability threshold (PPT). A higher PPT will lead to more conservative predictions, and thus higher accuracy, at the expense of the number of flows with classifications considered as valid. On the other hand, a lower PPT reduces accuracy but maximizes the number of flows with classifications considered as valid. For a system concerned with accurate identification of apps, false positives are usually undesirable and thus higher PPTs are likely to be suitable.

Classification validation reduces the number of flows that are considered as being "correctly" classified, but it is important to note that there is no inherent requirement to label all unknown flows. Apps typically send tens or hundreds of flows per minute when they are being used, so there remains significant opportunity to identify apps from their more distinctive flows. Thus, classification validation can be an effective technique for improving app classification performance while incurring negligible drawback. In what follows, we report on the improvements provided by applying classification validation to our previously described reinforced classifiers.

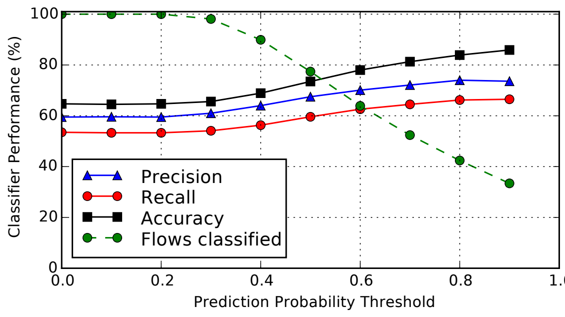
Figure 3.7 shows the improvement provided by classification validation for the `TIME`, `D-110`, `D-110A`, and `D-65` tests. We highlight some results by considering a PPT of 0.9. Figure 3.7a shows that the `TIME` test had a preliminary accuracy of 74.8% which was improved to 96.5% using classification validation. The results for the `D-110` and `D110-A` tests are shown in Figure 3.7b and 3.7c respectively. Overall accuracy was improved from 64.7% to 85.9% for test `D-110` and from 63.4% to 85.5% for `D-110A`. The final test in this group, `D-65` saw accuracy go from 68.1% to 89.9% when using classification validation.

Figure 3.8 shows the improvement provided by classification validation for the `V-LG`, `V-MG`, `DV-110`, and `DV-65` tests. Once again, we report our results considering a PPT of 0.9. Figure 3.8a shows that classification validation improved accuracy for the `V-LG` test from 54.5% to 83.9%. Figure 3.8b shows the results for the `V-MG` test, which is similar to `V-LG` but with a different device. Classification validation improved accuracy from 62.4% to 85.5% in this case.

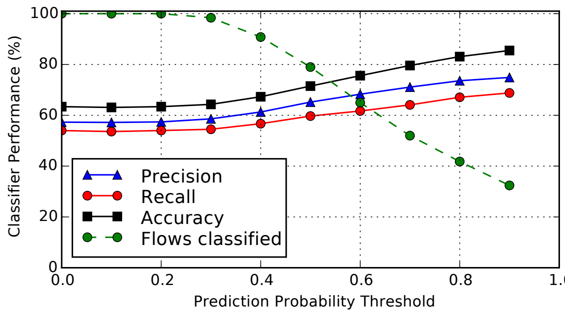
Figure 3.8c and 3.8d show the results for our most challenging tests: `DV-110` and `DV-65`. Classification validation was able to increase the accuracy of `DV-110` from 46.3% to 76.7%. Likewise, for test `DV-65`, accuracy was increased from 44.0% to 73.5%. This demonstrates that classification validation can be a useful tool to improve system performance under difficult conditions.



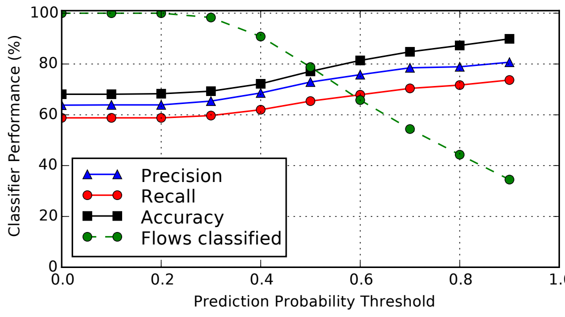
(a) Performance for the `TIME` test.



(b) Performance for the `D-110` test.

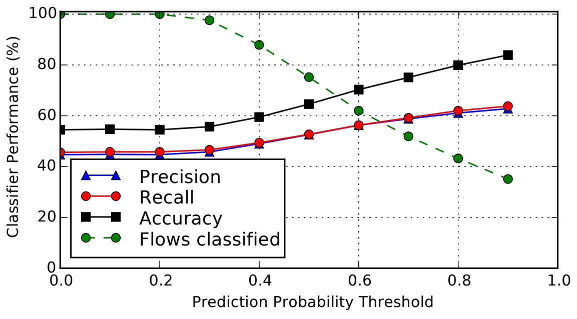


(c) Performance for the `D-110A` test.

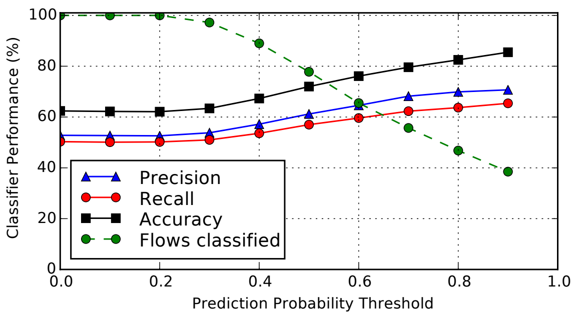


(d) Performance for the `D-65` test.

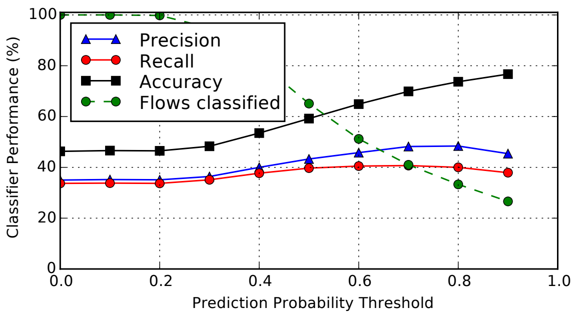
Figure 3.7: Performance of the reinforced classifiers on the `TIME`, `D-110`, `D-110A`, and `D-65` tests.



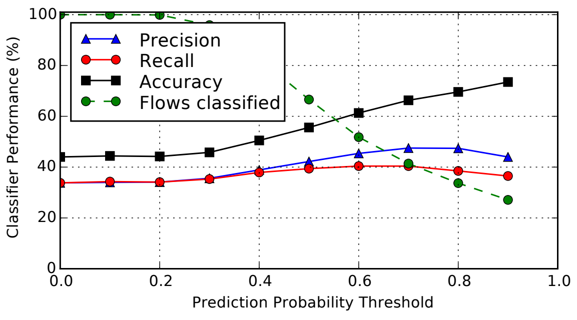
(a) Performance for the `V-LG` test.



(b) Performance for the `V-MG` test.



(c) Performance for the `DV-110` test.



(d) Performance for the `DV-65` test.

Figure 3.8: Performance of the reinforced classifiers on the `V-LG`, `V-MG`, `DV-110`, and `DV-65` tests.

3.5.3 Considerations for Parameter Tuning

Given that classification validation disregards some classifier predictions if the classifier is not confident enough, it is possible that setting the PPT too high will result in the system being no longer able to classify flows from a particular app. We measure the effect of classification validation and PPTs on the number of apps the system is able to classify. For brevity, we show results for our tests with the best and worst baseline performance, i.e., `TIME` and `DV-110` respectively. Additionally, we show the different PPTs of 0.5, 0.7, and 0.9. These results are summarized in the CDF plots of Figure 3.9.

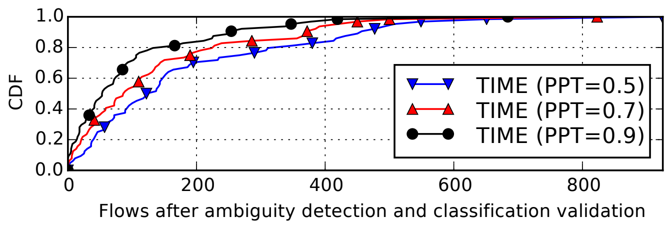
In general, higher thresholds for PPT reduced the number of flows that remained (and were correctly classified) after classification validation. Indeed, setting the PPT too high resulted in the system no longer being able to positively identify some apps. At the extreme end, setting the PPT to 0.9 for the `DV-110` test resulted in the system not being confident enough to classify approximately half of the apps. We remind the reader, however, that `DV-110` does not represent a real-world attack scenario since the test uses outdated app signatures for some apps. The test more representative of a real-world scenario, `TIME`, failed to identify 6 apps at a PPT of 0.9 and only 2 apps at a PPT of 0.5. Thus classification validation, while useful, must be tuned only after understanding how the system performs for particular *apps of interest*.

There is a trade-off between the number of apps that the system is able to classify and the overall accuracy that the system can identify apps with. Experimental settings should be tuned according to the usage scenario of AppScanner. More apps can be detected with less accuracy, or less apps can be detected with higher accuracy. An attacker can also tune her system using knowledge of how her *apps of interest* behave in the AppScanner system.

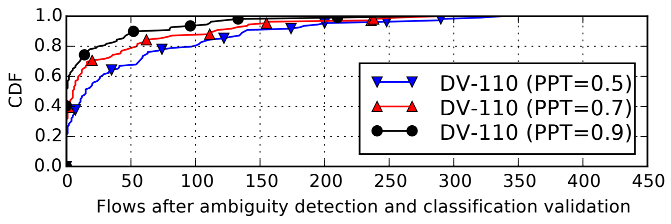
3.6 Discussion

Smartphone app fingerprinting is challenging because of a variety of variables that are likely to change between fingerprint building and final deployment. Such variables include device, operating system version, app version, and time. Any mismatch between variables during app fingerprinting and app identification has the potential to reduce the performance of our app classification system. To this end, we assessed how the aforementioned variables affected system performance. Apps were fingerprinted and later re-identified under a thorough suite of experimental settings.

In Table 3.2, we report app classification performance when training and testing sets are generated from the same dataset. In the other tests, we used completely independent datasets for training and testing. System performance when using independent sets was seen to be notably lower than the baseline experiments. This highlights the need for completely indepen-



(a) `TIME` test.



(b) `DV-110` test.

Figure 3.9: CDF plots showing the number of flows correctly classified per app after ambiguity detection and classification validation.

dent training and testing sets if one wants to get a more accurate estimate of the performance of an app fingerprinting system.

Training with specific app versions and device with six months between the collection of training and testing data had the highest baseline accuracy. This suggests that time (at the six month timescale) introduces the least variance in app fingerprints. This insight suggests that although the content returned by the app's servers may have changed, our models are fairly resilient to those changes and still give good performance. Our analysis on datasets collected using different devices (and operating system version) gave performance slightly lower than the previous test. This suggests that device or operating system characteristics of different devices can introduce some additional noise that affects classification performance to a small extent. Such reduction in performance is expected, since apps are known to change their behavior depending on the version of Android operating system that they are run on. Additionally, differences in the operating system itself may also contribute additional noise that affects classifier performance.

Fingerprinting a set of apps and identifying new versions of the same apps incurred a further performance penalty. This phenomenon is not unexpected, since apps routinely receive changes to their logic during updates [194], which may cause changes in their network traffic flows. However, our classification system shows that it is able to cope with such changes to an extent. This, however, motivates the need to re-fingerprint apps whenever they are updated, but suggests that old fingerprints may be useful, although presumably less so as apps receive more updates. Changing both device and app versions (and time) provided the greatest performance penalty for our classification system. This is an expected penalty since time, device, operating system version, and app versions have all changed between training and testing. Even under these most severe of constraints, our classifier was able to achieve a baseline performance 20 times that of pure random guessing.

The majority of the performance hit appears to come from so-called ambiguous flows. These flows are traffic that is similar across apps and typically comes from third-party libraries that are in common among apps. Such ambiguous traffic frustrates naive machine learning approaches, since the classifiers are given effectively the same training examples with different labels. Using a novel two-stage classification strategy with reinforcement learning, we were able to approximately double the baseline performance of our classifiers. Using the additional post-processing technique of classification validation, further accuracy could be extracted from the system, but at the expense of the number of flows that the classifiers were able to give a confident enough prediction. We remind the reader here that in app classification there is no inherent requirement to label all network flows, but rather to positively identify apps with high accuracy.

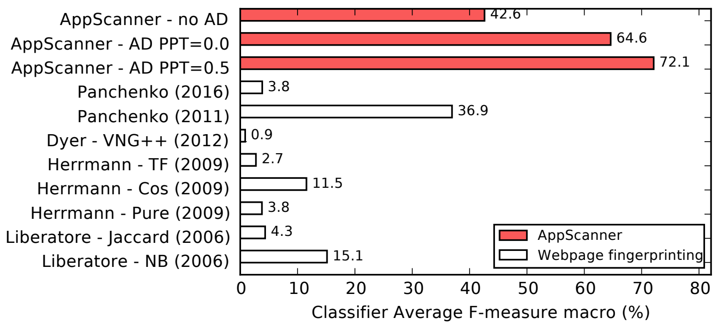
3.6.1 Comparison with Related Work

AppScanner works in the domain of fingerprinting network traffic from smartphones and so a direct comparison to website fingerprinting cannot be made. However, to motivate why different approaches such as AppScanner need to be taken when fingerprint app traffic, we show how existing work on website fingerprinting has reduced accuracy when run on app traffic. For a comprehensive comparison, we used our best and worst performing datasets: `TIME` and `DV-110`.

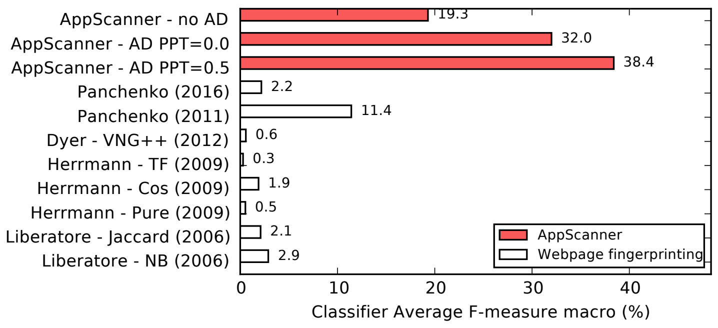
In general, all approaches performed better in the `TIME` test (Figure 3.10a) than in the `DV-110` test (Figure 3.10b). Panchenko et al. [156] performed best with F-measures of 36.9% and 11.4% respectively. AppScanner with no ambiguity detection outperformed this work with 42.6% and 19.2% respectively. Adding ambiguity detection and classification validation further improved the performance of AppScanner over related work.

3.6.2 Limitations

App identification is frustrated by a number of factors such as “flow coverage”, changing app behavior and ambiguous flows. Flow coverage refers to the fraction of the number flows that are actually triggered during UI fuzzing to the total number of flows that can be made by an app. Indeed, UI fuzzing may not elicit all flows from an app. Getting complete code coverage is a challenging task and even human participants were seen to only obtain 60% code coverage [125] in apps through manual interaction.



(a) Performance for the `TIME` test.



(b) Performance for the `DV-110` test.

Figure 3.10: Comparison with related work for the `TIME` and `DV-110` tests. AD = Ambiguity Detection and PPT = Prediction Probability Threshold.

Apps may also have different behavior if they are run at a different time. This may be because the apps themselves have been updated and now have a change in logic or the apps download (dynamic) configuration parameters from a server at runtime. Either of these possibilities may cause apps to have different behavior between training and testing. To mitigate this, repeated and continuous profiling of apps is necessary. Fortunately, profiling can be automated using virtual devices and UI fuzzing, obviating the need for physical hardware or manual intervention.

Ambiguous traffic also poses a problem for app identification. It degrades classifier performance since classifiers may be trained with conflicting data. Additionally, since there are finitely many flows that an app can generate, it is conceivable that more than one app will generate similar flows. Thus, an ambiguity detection scheme is critical to identify the non-ambiguous, distinctive flows coming from apps. It may also be the case that some apps simply do not generate non-ambiguous flows. In this case, other approaches will need to be taken for identifying those apps as this is a fundamental limitation of app classification using network traffic analysis.

3.6.3 Countermeasures

Mitigating app identification through traffic analysis is a complicated task. AppScanner uses coarse side-channel data from network traffic flows for feature generation. Thus, feasible countermeasures will likely involve padding network flows sufficiently so that one app is no longer distinguishable from another. In theory, this approach can be useful for frustrating app fingerprinting. In practice however, as argued by Dyer et al. [62], it is unlikely that bandwidth-efficient, general purpose mitigation strategies can provide the requisite protection. Moreover, this problem is complicated on smartphones (and other mobile devices) where battery power, bandwidth, and data usage are bottlenecks. Indeed, these added bottlenecks on smartphones makes efficient traffic analysis countermeasures an open research problem.

3.7 Summary

In this chapter, we presented AppScanner, a robust and scalable framework for the identification of smartphone apps from their network traffic. We thoroughly evaluated the feasibility of fingerprinting smartphone apps along several dimensions. We collected several datasets of app-generated traffic at different times (six months apart) using different devices (and versions of Android OS) and different app versions. We demonstrated that the passing of time is the variable that affects app fingerprinting the least. We also showed that app fingerprints are not significantly more affected by the device that the app is installed on. Our results show that updates to apps will reduce the accuracy of fingerprints. This is unsurprising since new app versions supposedly will have additional features, which can affect the fingerprint recognition process. We showed that even if app fingerprints are generated on a particular device, they can be identified six months later on a different device running different versions of the same apps with a baseline accuracy that is 20 times better than random guessing. Using the techniques of ambiguity detection and classification validation, we obtained noteworthy increases in system performance. We were able to fingerprint and later re-identify apps with up to 96% accuracy in the best case, and up to 73% accuracy in the worst case. These results suggest that app fingerprinting and identification is indeed feasible in the real-world. App fingerprinting unlocks a variety of new challenges as it relates to user security and privacy.

Part II

Energy Consumption Analysis

Chapter 4

Laptop Users Identification

The usage of electrical devices, ranging from appliances to digital systems, have constantly increased year after year. In order to cope with both rapidly growing demand and waste of electric power, future smart buildings will employ the Internet of Things (IoT) paradigm to overcome many of the current facilities problems. An example is an intelligent optimization of the heating, ventilating and air conditioning (HVAC). To this aim, offices and homes will be equipped with several types of sensors (e.g., user proximity, light, air, energy meter) that will acquire data from the surroundings. Such data can thus be sent to a remote entity that can elaborate it, in order to help addressing facilities issues such as energy saving. Moreover, the communication between the elements in IoT environments (i.e., sensors and actuators) can be performed by both wire and wireless.

Researchers on smart building and smart grid have dramatically advanced the technology and the reading capabilities of smart energy meters (or simply smart meters). Indeed, such sensors are now able to collect not only the consumed *Active Power*, but also several electrical quantities such as *Phase Angle* and *Voltage* [69], with a sampling period of one second. Smart meters have been deployed in several testbeds and outcomes encourages their usage in future Smart Buildings and IoT environments. Moreover, smart meters are commercially available and their cost is decreasing, thus a widespread diffusion in the near future is very likely to happen. Smart meters are mainly of two types, based on the scope of the observed electrical network:

- *household level sensors* are usually embedded into the meter provided by the supply company and they are designed to monitor the entire building.
- *wall-socket level sensors* are deployed on individual sockets and they are distributed all over the building.

The former type has already been considered by research on smart grids, also with respect to the possible privacy issues; public opinion seems aware of the topic as many news headlines report security and privacy issues. In this chapter, we focus on privacy issues for the latter category of smart meters (i.e., wall-socket level sensors) and, supported by our results, we argue that it deserves more attention by the privacy research community. To complicate things further (even for privacy aspects), some off-the-shelf smart meters integrate wireless connectivity in order to form a Wireless Sensors Network (WSN), where intermediate nodes act as gateways to the rest of the intelligent architecture of the building [142, 145].

Some investigations worked to identify appliances [171] and their states [119] from energy traces, as well as to protect users, obfuscating their energy usage [98]. These works showed that energy usage data are privacy sensitive. Moreover, some appliances can be modeled as Finite State Machines (FSM) [92], where the complexity increases with the number of transitions and possible states. Based on this fact, we observe that some devices consume energy depending on the specific configuration and user interaction. For example, a subset of the observable energy-states of a laptop is related with the user's actions.

A widespread diffusion of smart meters is expected in homes and offices, as well as public places such as bars and restaurants. Moreover, the demand of recharging batteries of portable devices in public places is increasing. Indeed, many public places (such as airport waiting areas) attract people as they supply both free energy and Internet connection. Therefore, in order to gather information about their customers, they could deploy our system within a wall-socket monitoring system. We point out that some energy grid architectures have been specifically designed to enable the monitoring at the individual outlet level [99, 117]. For those reasons, in our study we consider laptops as personal mobile devices that are not replaced very often. Laptops consume electricity depending on many factors, mainly the way the user uses her laptop, running applications, its manufacturer and the operative system. Although all the aforementioned factors contributes to the observable energy consumption, defining a relationship between them is out of the scope of this chapter. Thus, we consider the ensemble of laptop model, user activity and the set of applications installed and running, as a single entity (from now on referred as *laptop-user*) to be identified.

Such intelligent environments are able to study the behavior of the users and to address common issues and optimization (e.g., anomalies detection, energy saving). In order to provide the system enough information about the surroundings, IoT sensors produce huge amount of information which is difficult to elaborate locally. For this reason, sensors data is commonly moved and analyzed remotely by remote systems that often is not under the control of users. By employing advanced machine learning techniques is possible to extract knowledge for such big data. Literature shows the feasibility of studying users, their behavior and even extract privacy sensitive insights.

Identifying a specific laptop-user from its energy traces could carry both significant benefits on smart building automation, as well as represents a serious threat to the privacy of users. Indeed, consumption data can be leveraged in several benevolent ways within a smart building. Such knowledge about users can bring several benefits to the Smart Buildings automation:

- Context-aware environments can automatically adjust themselves based on specific users and trigger predefined actions or services.
- The system can automatically intervene on the surrounding of the user, adjusting the temperature and the light in the room according to the her preferences.
- The system can locate and make a user reachable by colleagues, even in the case she frequently moves in other rooms, for example by automatically forwarding phone calls to her closest landline phone.
- In case of smart meters that integrate also the wired network cable socket, an IoT environment could authenticate the user and give her the access according with the network policies. This may act as an additional authentication over the traditional check on the MAC address of network card, that can be easily spoofed. Indeed, the system could confirm the identity of a user.
- The system could detect not authorized users in a access restricted room (e.g., server room) or the presence of intruders in a building.

All of these benefits could be achieved without relying on much more intrusive systems that should be in place otherwise, such as video or audio surveillance. On the other hand, an adversary able to retrieve energy traces from wherever they are stored (i.e., directly from sensors, or from the remote aggregation system such as a cloud platform) could represent a serious threat for the user privacy. Indeed, such adversary could infer the same knowledge about a user as a legitimate system could do. Moreover, after the creation of the user energy profile, the recognition can be extended to contexts like other buildings equipped with smart meters. This is the case in which data are collected by a unique meters manufacturer. Therefore, big smart meters firms might have a very wide scope within which to track laptop-users. For instance, a user profiled in her office can be tracked down when she plugs her laptop in public wall-sockets. This information could be then sold, beyond any still not present regulations, for marketing purposes. In addition to that, an adversary could, for example, pursue the following threats:

- Big smart meters manufacturers that collect data of thousand meters would have a very wide scope within track laptop users. For instance, a user profiled in her office can be tracked down when recharges and uses her laptop at an airport or in other public places. This information could be then sold, beyond any still not present regulations, for marketing purposes.
- The system can remotely monitor where and how often users move within the smart building. Knowing that some users meet in the same room or share the office can highlight social relationships.
- An employer can monitor the productivity of employees by tracking their position within a smart building. Also, the productivity could be assessed by monitoring the expected consumption.

The main risk of the preceding threats consists in turning a system that should help users to save energy in a powerful mass surveillance. While users appreciate the fancy consumption overviews provided by means of web pages or smartphone apps, their privacy is actually at risk and far from their direct control. The ubiquity of power sockets that can be possibly turned in energy consumption monitors suggests this technology can easily enter in our daily life. Moreover, we highlight that in countries with strict privacy regulations, the collection of such sensitive data would require users to accept appropriate Terms of Services.

Furthermore, plugging more than one appliance in the same monitored wall-socket (for example using a power strip) might not be effective in order to hide the single appliance usage. Indeed literature shows that it is feasible to disaggregate the energy consumption per-appliance [114]. Lastly, countermeasures are non-trivial and not yet commercially available, we discuss them in Section 4.5.

Contributions: In this chapter we introduce MTPlug, our proposed framework for the identification of a user solely relying on her laptop energy traces. We have built and tested MTPlug with the energy traces produced by the laptops of 27 users. In what follow, we list the main contributions this chapter to the state-of-the-art:

- We demonstrate that it is possible to build a laptop-user specific energy fingerprint using a common low-frequency smart meter at the wall-socket level.
- We propose and fully implemented MTPlug framework, and evaluate its performance in terms of precision and recall with a thorough set of experiments.
- We study the influence of energy traces collection time for building a reliable laptop-user electrical fingerprint. Such fingerprint can be used

afterwards to rapidly re-identify a laptop-user with an accuracy above 80%.

This work makes a new contribution to the state of the art, since it highlights the existence, and the significance, of a new serious privacy threat in smart metering, and more generally, in anonymity of energy consumption data.

Organization: The rest of this chapter is organized as follows. We review the state-of-the-art related to our research topic in Section 4.1. In Section 4.2, we outline required background knowledge. In Section 4.3, we present our framework MTPlug, describing its components. We evaluate the performance of our solution in Section 4.4, and we discuss possible countermeasures in Section 4.5 Finally, in Section 4.6 we summarize the work presented in this chapter.

4.1 Related work

In this section, we survey the main researches in smart metering, considering different scopes and categories: smart metering privacy issues, appliance identification and user presence detection.

Smart metering privacy issues – Quinn et al. [164] review existing laws and regulations in smart metering and raise significant privacy questions. Furthermore, existing laws which protect user data suffer of weaknesses and possible exceptions about its usage and transfer to third party entities. However, the literature has different proposals for privacy-preserving metering data transmission [7,65]. On the user perspective, protection can be achieved using a rechargeable battery and a power routing algorithm [101]. In particular, it is possible to shape the home load signature to hide traces of appliances usage. Genkin et al. [82] show a side-channel attack able to reconstruct cryptographic keys based on fluctuations of the "ground" electric potential. Although the state of the art raises question about wall-socket level privacy, to the best of our knowledge our work is the first on users recognition through their laptop energy consumption.

Appliance identification – Literature shows that it is possible to perform *appliance identification* observing the energy consumption from different network levels. Non-intrusive load monitoring (NILM) approaches use a centralized sensor, usually located at the main house hold circuit-level. After collecting energy usage data, they perform information disaggregation to isolate single appliances traces [216]. Reinhardt et al. [169] use a more intrusive approach, considering single appliances and sampling the energy consumption from the power plug. Using a period of one second, the authors achieved good performances in recognizing, beside others, a laptop. Anyway, commercial services that implement NILM are already publicly available[1].

User presence detection – Building occupancy is studied by users detection leveraging on data provided by sensors deployed in smart buildings [106]. For example, Heating, ventilating, and air conditioning (HVAC) systems usage can reveal user presence with a true positive rate above 88% [63] and highlight possible energy wasting [4]. While HVAC activity is observed from a centralized sensor, low-power appliances monitoring requires a more invasive approach. Their state recognition is performed in [217] which scores a F-measure ($\beta = 1$) of 0.906 and 0.804 for binary and multi-state appliances. However, since combinations of capacitive and inductive loads monitoring are more difficult to monitor, [216] warns to carefully evaluate such cases.

Going towards user profiling, our survey considers tools that gather, on different levels, energy data about appliance usage, focusing on computers. It is possible to inspect via software the power impact of single processes with negligible overhead [130]. Do et al. [61] use energy consumption features of operating systems to study the amount of energy consumed by each running application. Rashidi et al. [167] propose a semi-supervised approach to build behavioral patterns of users during an extended period (i.e., two weeks). Despite this work seems very similar to ours, the authors consider the aggregated consumption of multiple appliances used by individuals. Although it is a user level profiling, it considers consumption of un-labeled devices. Furthermore, Procaccianti et al. in [162] collected energy data from a computer system performing common usage scenarios (e.g., idle, web navigation, Skype call). Results show that software use cases impact significantly on energy consumption. Similarly, the authors of [48] show the feasibility of identifying specific web-browsing activities, using as a side channel the alternate current (AC) gathered at the power outlet level. The consumption analysis achieves very high precision and recall on determining which page is loaded.

Although related work achieve good results in user level analysis, the focus is mainly on behavioral patterns or other general inspections. In this chapter we present a comprehensive laptop-users profiling and recognition based on energy consumption. Strong of our ground truth with respect to users and monitored laptops characteristics, we push the user monitoring toward a fine-grained level that seems missing in the literature.

4.2 Background knowledge

In this section, we recall background aspects that we use in this chapter. Firstly, we first introduce the electrical quantities considered. Secondly, we

[1] `https://www.bidgely.com/technology`

explain the time series segmentation. Then, we briefly describe some machine learning and data mining concepts. Finally, we illustrate the filtering of noisy signals. We point interested reader to references cited in this section for a more in depth study of specific concepts.

Electrical quantities: A wall-socket smart meter is able to measure the energy consumption of the plugged appliance. Such appliances drain AC from the electrical system of the household. In particular, such device measures several electrical quantities. The electrical quantities listed below are the ones used in this chapter:

- *Active Power* (P), also referred with real power, is expressed in watt (W).
- *Reactive Power* (Q) is often measured in reactive volt-amperes (var).
- *RMS Current* (I) is the root mean square of the alternate current, measured in amperes (A).
- *Phase Angle* (ϕ) between the current and the voltage in the AC domain.

In this chapter, we consider wall-socket smart meters able to measure (at least) the values of these four electrical quantities for each sampling period. Hence, an energy trace consists of a multivariate time series of electrical values of sequential samples.

Time series segmentation: We consider a time series $T = \{x_k = [x_1, x_2, ..., x_n] | 1 \leqslant k \leqslant N\}$ as a finite set of N samples indexed by time points $t_1, ..., t_N$ [3], and a segment as a set of consecutive time points $S(a, b) = a \leqslant k \leqslant b, x_a, x_{a+1}, ..., x_b$. Hence, the segmentation of the time series T into c non-overlapping intervals can be formulated as $S_T^c = \{S_e(a_e, b_e) \,|\, 1 \leqslant e \leqslant c\}$, were $a_1 = 1, b_c = N$ and $a_e = b_{e-1} + 1$. In order to use a time series as a pattern, the series can be represented by Piecewise Linear Representation (PLR), which consists in segmenting a series with K straight segments. Since PLR results in an approximation of T, its fidelity can be expressed by error metrics (e.g., *max_err*, *total_error*) which can consider multiple units of the PLR: from individual segments to the entire segmented series. In general, researchers propose three segmentation approaches [103] that produce a representation of a time series T given: (i) the number K of segments; (ii) a *max_err* threshold, which stands for the error bound a single segment cannot exceed; and (iii) a *total_error* threshold among all segments of T. In this chapter, we applied the second approach to segment the time series obtained by laptop energy traces.

The bottom-up segmentation is a batch approach based on a maximum error per each segment. This approach starts from approximating with a linear interpolation the n-length time series by $n/2$ segments. The algorithm then calculates the cost of merging each pair of adjacent segments until a stopping criteria is met. Approximation is given by a straight line that can be computed by linear interpolation or by regression. The linear interpolation consists in connecting each point present in the subsequence $T[a : b]$, while the regression takes into account the best fitting line in the least square sense. Although the regression can generally achieve higher approximation quality, as Euclidean distance, it takes linear time in the length of the segment. The linear takes a constant time so it is preferable in contexts where computing power is critical (e.g., computer graphic). In order to evaluate the fitting quality of a candidate segment, the algorithm needs a method function that we formalize as $cost(S(a, b))$. It represents the distance (e.g., sum of squares) between a simple function (linear or polynomial) fitted to the actual values of each segments [3]. Possible distance metric can be sum of squares, distance between the best fit line and data point furthest away in the vertical direction, or any other measure.

Supervised learning: Supervised machine learning algorithms acquire knowledge about a specific context through examples. After the training phase, where such algorithms make up their knowledge from past experience, they produce an inferred model able to classify new un-labeled instances. In an optimal test scenario, the algorithm determines properly the class labels for unseen instances. In this chapter, we employ three classification methods: Random Forest (RF) ensemble classifier, k-Nearest Neighbors (KNN) and Support Vector Machine (SVM). The KNN classifier computes the distances between the test example and each member of the training set. Then, the trained KNN classifier predicts the class label as the same of the nearest k training set members. Differently, SVM classifier uses weighted vectors for defining decision boundaries between classes. SVM classifies examples by summing the outputs of a similarity function, taking into account the weights of the vectors, between the support vectors and the unseen instance. Then, the trained SVM classifier predicts the class with the largest sum, relative to a bias. A comprehensive reading about these classifiers is available in [31].

Savitzky-Golay filter: In the sampling of an analog signal, sensors may produce noisy readings. With high frequency fluctuations, this error is more likely to grow and the segmentation could end in meaningless chunks. For this reason, a sampled signal could be smoothed using a low-pass filter without harming the original signal structure. Savitzky-Golay [174] is a low-pass filter very effective in smoothing out highly noisy signals characterized by a wide frequency spectrum. It considers the *frame size* as a parameter, and it must be tuned considering the degree of variance and noise [182]. Savitzky-Golay filters are optimal as low-pass because they preserve the

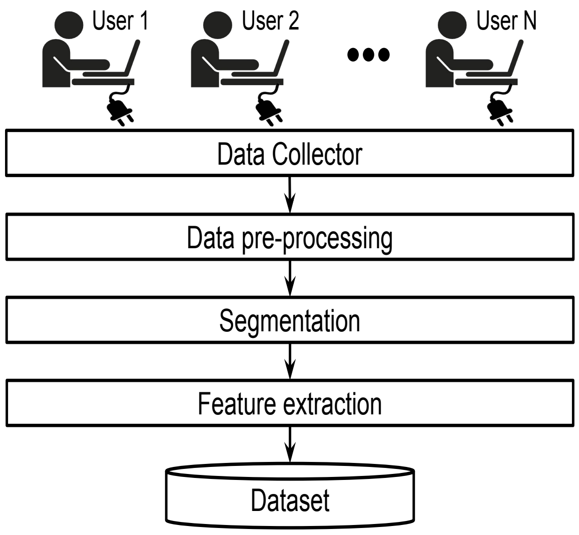


Figure 4.1: MTPlug framework overview. Each user plugs her laptop into the corresponding labeled smart meter.

high-frequency content of the signal and they minimize the amount of noise reduction intended as the fitting least-squares error.

4.3 Our framework: MTPlug

In this section, we describe the design of MTPlug, our framework to identify laptop-users from their energy traces. We divide the overview of the proposed system MTPlug in two blocks: *dataset creation* (in Section 4.3.1) and *classification* (in Section 4.3.6).

4.3.1 Dataset creation

In the first block, the system acquires the raw data and prepares it in order to be handled by a supervised learning method. This block is composed of four steps, summarized in Figure 4.1. The first step, named *Data Collector*, consists of acquiring the raw energy traces of the laptop-users plugged into wall-socket smart meters (Section 4.3.2). The second step is the pre-process of the raw data (Section 4.3.3) and it is in charge to prepare a time series that could be divided in segments in the third step (Section 4.3.4). Then, as last step of the first block (*dataset creation*), the system extracts an array of statistical features from each segment (Section 4.3.5).

4.3.2 Data Collector

In this step, MTPlug system relies on wall-socket smart meters to collect the energy traces produced by laptop-users. These smart devices are able to build an energy trace composed of several electrical quantities such as *Active Power*, *RMS Current*, *Reactive Power* and *Phase Angle.* Since the sampling period is fixed for each electrical quantity (e.g., one second), an energy trace can be handled as a multivariate time series. To build a reliable ground truth, we assigned a wall-socket to a single user, thus the system is able to relate that user with her energy trace.

4.3.3 Pre-processing

The pre-processing phase consists of filtering out the samples that are not relevant or that could hinder the segmentation of the time series. First, the system filters the samples that are meaningful to perform the user identification. Indeed, the system takes into account only the samples likely reporting actual user activity. In order to do so, MTPlug considers only readings with an *Active Power* value above a certain α threshold. Doing this, the system drops the samples of when no useful load is plugged (e.g., only the power supplier, laptop in stand-by mode). In Section 4.4.1, we report how we empirically determine α threshold. Secondly, we filter the samples that contain reading errors, thus they could hinder the time series segmentation. To do so, we smooth the time series using a Savitzky-Golay low-pass filter (previously described in Section 4.2), to preserve the original signal structure. At the end of this pre-processing phase, the time series are ready to be segmented.

4.3.4 Segmentation

After selecting only useful samples from the raw data, MTPlug applies the segmentation to the multivariate time series. In the time series classification, the choice of a proper segmentation approach for a specific domain is fundamental to split the series in meaningful patterns. Many approaches are possible, for instance change-point detection, sliding window, top-down and bottom-up [25, 103]. We employ the latter which starts from a fine series approximation and iteratively merges the lowest cost pair of segments until a stopping criteria is met [103]. We aim to segment the consumption data such that each segment represents a particular pattern in the laptop usage. Differently from the PLR, we just detect segments endpoints considering the *Active Power*, keeping the original data points of the series. Since the segmentation is based on a straight lines approximation, we find them by linear interpolation, which consists in approximating the subsequence $T[a : b]$ by connecting t_a and t_b [103]. Hence, to evaluate the quality of the approximation, we use the Residual Sum of Squares (RSS) as distance metric. RSS is the sum of all the vertical squared differences between the best-fit line and the actual data, a small value means that the model fits tightly. We merge two adjacent segments if the RSS is below a given maximum error, in the following referred as *max_err*. This parameter controls the outcome segments length so we empirically tune it to achieve an optimal representation of our domain, we give further details of the distribution of lengths in Section 4.4.1. We underline that the segmentation is used just to identify the endpoints of segments, thereby we build our pre-processed dataset using the values of the original sampled time series.

4.3.5 Features extraction

This step produces a statistical data set which a classifier algorithm can handle. At the end of the segmentation phase, we obtain segments (i.e., multivariate time series) with variable lengths. To build a dataset suitable for a classifier, we extract a fixed number of features from each segment. Since a segment is a set of four time series (i.e., one for each electrical quantity), we extract the following statistical features from each serie: *mean*, *minimum* and *maximum* values; *sum* of the values; *length* of the segment; *variance*, *standard deviation*, *mean absolute deviation*, *skewness*, *kurtosis* and *variance*. Moreover, we also calculate the value that corresponds to the n^{th} percentile of a series. For each segment, we concatenate the resulting four arrays of statistical features, obtaining a single array of data. In order to simplify the classification execution, we normalize from 0 to 1 the values for each statistical feature. Finally, we label each segment with the identifier of the laptop-user which has produced that segment (e.g., user0, user1). At the end of this step, we obtain a labeled dataset of examples.

4.3.6 Laptop-users classification

In the second block (*classification*), the system uses the dataset created in the previous block to first train a classifier and then to evaluate its performance. As we previously introduced in Section 4.2, in a supervised learning a classifier needs a labeled set of examples to be trained upon (i.e., training set) and another one to test its performance (i.e., test set). Training set and test sets must be disjointed, i.e., they do not share any element. In fact, the test set does not take part in the training of the classifier. In our case, an example is composed of the array of statistical features and an identifier (i.e., class) of an laptop-user (e.g., user0, user1).

Before proceeding with the training, MTPlug runs first a features selection and then a hyper parameters optimization. Since our feature space consists of 109 statistical features, the classification could be affected by a phenomenon known as curse of dimensionality [31]. In order to avoid this phenomenon, the system evaluates the significance of each statistical feature running Random Forest classifier on the training set. Thus, it selects the feature with a significance score higher than 1% (see Section 4.4.2). Afterward, the system selects the optimum set of hyper parameters of the classifier. To do so, it runs an exhaustive search on a wide set of hyper parameters applying a grid-search algorithm with 5-fold cross-validation (CV). Such algorithm iteratively evaluates the performance of a classifier for each possible instance of hyper parameters. For each instance, it runs a cross validation on five disjoint and equipotent sets of the training set (i.e., 5-fold). Finally, MTPlug trains a classifier using the whole training set and the set of hyper parameters that achieved the best results in the cross validation. In Section 4.4.2, we evaluate classification performance on the test set of three classifiers: Support Vector Machine (SVM), k-Nearest Neighbors (KNN) and Random Forest (RF).

A classifier predicts the class to which a segment belongs and the probability (confidence) for that prediction. In order to increase the overall performance, we consider only segment predictions above a threshold *omega*. Note that this approach does not assume any class membership on the stream flow basis. However, since we employ supervised learning, the test phase assures the reliability of high-confidence predictions. We refer to this process as the classification confidence post-process function. By this, we aim to profile a laptop-user when plugged in a specific wall-socket for a reasonable time so we rely on more than a single segment. Thus, we consider only the segments to which the classifier has a confidence above a certain threshold (i.e., *omega*), and simply discarding the others. Unfortunately, a side effect of this approach is that the increasing of *omega*, the number of segments considered as reliable reduces accordingly.

4.4 Experimental results

In this section, we present our testbed configuration for data collection and the performance evaluation of MTPlug. In particular, we detail our testbed in Section 4.4.1, while we report the outcomes of the classification in Section 4.4.2.

4.4.1 Test-bed configuration and data collection

In our settings, we monitored the energy consumption of laptop-users using wall-socket smart meters with a sampling period of one second. Each sample is composed of the user label and an array of real-valued readings of *Active* and *Reactive Power* as well as *RMS Current* and *Phase Angle*. The data collection environment is an office room with 230 Volts and 50 Hz monitored power sockets.

Participants: Our data collection involved a total of 27 participants with their personal laptops. All the participants were volunteers, without any promise of reward, and they signed an informed agreement. In particular, we asked seven participants to work in a restricted office for a period of two weeks. Henceforth, we refer to those seven user as *main users*. The *intruders* are represented by a disjoint set of the remaining 20 users which were not supposed to plug their laptops within the restricted area. Their participation lasted just a few hours, since we aim to simulate several unauthorized accesses. Considered laptops are of seven famous brands, all commercially available.

Energy trace collection: We deploy off-the-shelf meters *Plogg.zbg v2.0* (1 Hz sampling rate) that integrate a *TelosB*[2] for remote management. These devices are designed to arrange hierarchical WSNs where IoT nodes (i.e., smart meters) sample data and gateways collect and forward it to the upper levels, for processing purposes. This approach has been successfully adopted in bigger testbeds [145]. We set up this environment in order to gather the energy consumption related to different laptop-users. Before segmenting the time series, the pre-processing step includes the dropping of those samples with *Active Power* value below a threshold α (see Section 4.3.3). We empirically determine the threshold $\alpha = 12$ Watt by observing that under this value, all monitored laptops were plugged in but no significant user activity was observable (e.g., stand-by or low-energy mode). Lastly, we perform the time series smoothing by a quadratic polynomial Savitzky-Golay low-pass filter. A preliminary analysis shows that a frame size of 30 points is a reasonable value to avoid noisy and scattered readings.

Segments lengths: The time series bottom-up segmentation algorithm produces segments with different lengths. Given n data points and K segments, the average segment length is $L = n/K$ [103]. The lengths of segments vary with the parameters of the segmentation algorithm (i.e., *max_err*, RSS, interpolation function). In this paragraph we focus on *max_err*, which determines the merging of two adjacent segments, hence influencing the output segments lengths. Thus, we tune the *max_err* parameter in order to obtain reasonable segments lengths. For this reason, we set two additional bounds on the length of the segments produced by the segmentation algorithm. On one hand, we fixed the lower bound of segments length to three samples (i.e., three seconds), hence, we simply ignore segments that last less than three samples. A preliminary analysis showed that segments with length smaller than this value appear to be very heterogeneous, and therefore more likely to decrease the classification accuracy. On the other hand, we set an upper limit in order to avoid extremely long segments. Indeed,

[2] `http://www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf`

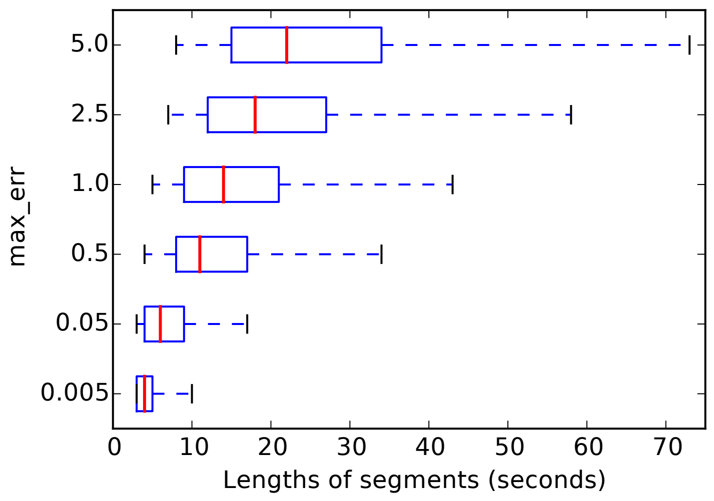


Figure 4.2: Distribution of lengths of segmented time series. First and third quartile are represented as the left and right side of the box. The band inside the box represents the median value. Lines that extend horizontally from the boxes indicate the 2nd percentile (left) and the 98th percentile (right).

long segments are due to a stable (almost-constant) energy consumption for many sequential samples. In our analysis, the aforementioned case is meaningless since it does not show any substantial activity, which might helps the classification. For this reason, we filter out the segments longer than the 90-percentile. Figure 4.2 shows the distribution of the segments lengths varying the *max_err*, considering all 27 users. The plot shows that as the *max_err* segmentation parameter increases, the segments lengths grow as well.

4.4.2 Classification Performance

The examples produced by data collection are divided in two disjoint sets: training set and test set. We use the training set to train the classifier, while we use the test set to evaluate its accuracy. They consists of the 75% and the 25% of the whole data set [31], respectively. In our analysis, we aim to train a classifier that is able to recognize the *main users* from the segmented energy traces (i.e., time series) produced by their laptops. Moreover, this classifier has to be able to discriminate each *main user* from the *intruders*. As motivated in Section-4.4.1, all the examples collected from *intruders* are considered as a single class. In the following, we firstly investigate the performance of the classifiers considering different sets of electrical quantities and different values of *max_err*. We discuss the results obtained with the best scoring setting for each considered classifier. In particular, we discuss the average accuracy reached in recognizing laptop-users, and give

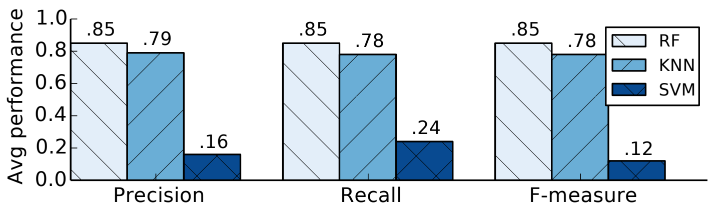


Figure 4.3: Classifiers performance comparison considering the *main users* and the *intruders*.

detailed results for precision, recall and F-measure ($\beta = 1$) metrics. Later, we investigate how to improve classification performance relying on confidence threshold *omega*. Finally, we investigate the performance of the best classifier as the dimension of the training set changes, thus the number of segments necessary to build the energy fingerprint of a laptop-user.

First, we investigated the contribution given by each electrical quantity to the classification of laptop-users. In order to do so, we ran the MTPlug framework on different combinations of the collected electrical quantities and for a set of *max_err* values ranging from 0.005 to 5. We obtained the highest classification performance considering all the electrical quantities and *max_err*=0.005. Hence, we use these settings for our classification, achieving the 85% of accuracy, and the mean length of a segment is just four seconds. Indeed, increasing the *max_err* increases also the mean length of segments, as reported in Figure 4.2.

In the following, we focus our attention on the evaluation of different classifiers: Random Forest (RF), k-Nearest Neighbors (KNN) and Support Vector Machine (SVM). In our analysis, we use a popular machine learning library [37] for implementation of such classifiers. We run our MTPlug (i.e., feature selection, grid-search and training) using the best *max_err* and electrical quantities combination obtained from the previous analysis. In Figure 4.3, we compare the performance of three classifiers in this experiment. Random Forest outperforms k-Nearest Neighbors and Support Vector Machine on all the considered metrics. Hyper-parameters for the RF obtained from the grid-search are 150 estimators, $maxfeatures = sqrt$ and the Gini impurity measure as split criterion.

As we previously explained in Section 4.3.6, considering all the statistical features extracted from a time series segments could affect the classification performance with the curse of dimensionality. In fact, from each segment, we extract its length as well as the features detailed in Section 4.3.5, for a total of 109 features. In order to consider only the most relevant features, we perform a feature selection relying on the feature importance score calculated by a RF classifier over the training set. We selected only the statistical features

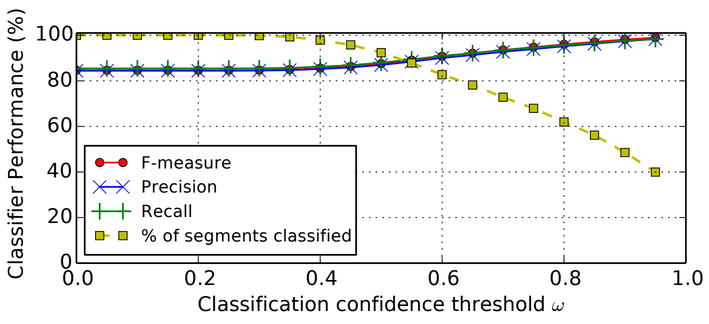


Figure 4.4: Impact of ω threshold on Random Forest classifier performance considering *max_err* = 0.005 and *main users* and *intruders*.

with an importance score higher than 1%. Considering the best settings, we obtained a total of 36 features of the original 109.

Classification confidence analysis: In order to increase the classification performances, we used the post-processing function that relies on classification confidence, introduced in Section 4.3.6. This function considers the classifier confidence of each prediction. Figure 4.4 depicts the obtained performance using different ω. In this analysis, we consider our framework using the RF classifier, all the electrical quantities, *max_err*=0.005, and the training and test sets with all the seven *main users* and the twenty *intruders*. The results obtained with the RF classifier highlight that the classification confidence mean in the test set is $\mu = 0.82$ with standard deviation $\sigma = 0.19$. The classifier achieves an F-measure equal to 0.98 in predicting 49.6% of the whole test set, with a confidence $\omega = 0.9$. Unfortunately, this post-processing function decreases the amount of classified segments by a ≈ 0.5 factor. In order to deal with this loss, we point out that the segmentation (see Section 4.3.4) should use a *max_err* as small as possible, since this would significantly increase the total number of examples. Experiments in this section use *max_err* = 0.005.

Sufficient monitoring time: In order to evaluate the feasibility of producing an energy fingerprint in a limited time, we investigate on the sufficient monitoring time for training our MTPlug framework. The time evaluation is based on the statistical length distribution of the considered segments showed in Figure 4.2. In this analysis, we consider the *max_err* = 0.005, which has mean segments length $\mu = 4.6$, standard deviation $\sigma = 2$ and 75-percentile = 5 seconds. Here we run multiple trainings, varying the number of examples per user $n = [100, 250, ..., 10000]$ with incremental steps of 150 segments. The training time of the i^{th} run is $tr_i = n_i \times \mu$. Each of the n experiments is ran 25 times with different training sets (shuffled subsets

Classifier	**Values of *omega***		
Random Forest	**0.3**	**0.6**	**0.9**
Precision	0.85	0.91	0.98
Recall	0.85	0.91	0.98
F-measure	0.85	0.91	0.98
% Segments classified	99.9	83.3	49.6
K-Nearest Neighbours	**0.3**	**0.6**	**0.9**
Precision	0.79	0.86	0.96
Recall	0.78	0.86	0.96
F-measure	0.78	0.86	0.96
% Segments classified	99.8	79.0	44.3
Support Vector Machine	**0.2**	**0.3**	**0.4**
Precision	0.15	0.12	–
Recall	0.24	0.24	–
F-measure	0.12	0.10	–
% Segments classified	0.99	0.94	0.03

Table 4.1: Classification confidence over some probability thresholds, classifying *main users* and *intruders*.

of the entire training set) and evaluated on the same test set used in previous analysis. Figure 4.5 shows the average scores for the Random Forest classifier, the best classifier so far. From this, we show that the sufficient monitoring time to recognize our *main users* and *intruders* with F-measure = 70% is on average around $tr_0 = 7.5$ minutes, considering the 75-percentile we can compute an a training upper bound as 8.3 minutes. On one hand, we achieve an F-measure above 80% starting from 2650 segments, in this case the upper bound of the training time is around 3.5 hours. On the other hand, for the same amount of training time, we achieve an F-measure of 88% and 97%, with $\omega = 0.6$ and $\omega = 0.9$, respectively. We recall that these results are obtained by solely observing the consumption of the ensemble of user activity and their machines. Although it is speculative to attribute samples to specific events within the laptop-user entity, we stress the fact that each monitored configuration is, as a whole, rather different.

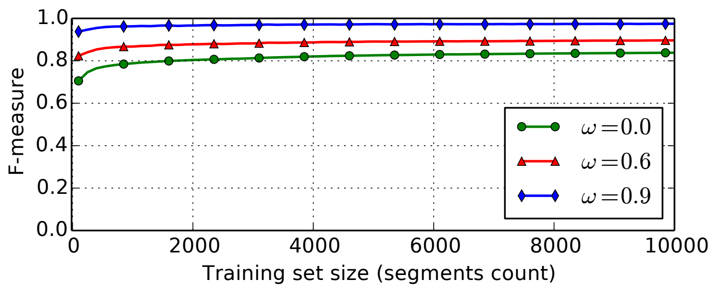


Figure 4.5: Random Forest classifier performances with different training set sizes, considering the classification confidence threshold.

4.5 Possible Countermeasures

As possible countermeasures, we suppose that both hardware and software systems can be employed in order to hide a laptop-user recognition through her energy traces analysis. Researchers propose configurations based on rechargeable batteries connected to the house electricity meter able to reduce the sensitive information leak by obfuscating the actual power usage [101]. Moreover, a software countermeasure might try to shape the energy drain from the AC by controlling the battery usage in order to alter the observable consumption pattern. In [180], authors propose a cloud based battery control client. Although this solution might be effective against MTPlug, we argue that is not directly generalizable to the whole laptop domain because it is based on specific brand's system utilities. We recall that our identification method works only when laptops are plugged to a power socket.

4.6 Summary

In this chapter, we show that the analysis of energy consumption data might be a serious threat for the privacy of users. We perform and evaluate such analysis in a controlled environment using off-the-shelf smart meters and supervised machine learning. In particular, we design and implement MTPlug, a framework to fingerprint and recognize a user based on her laptop energy consumption. We design an automatic statistical feature extraction and selection procedure which considers multiple electrical quantities. In performing this task, the Random Forest classifier outperformed the k-Nearest Neighbors and the Support Vector Classifier. We carried out an analysis which shows the feasibility to identify a pair laptop-user with an accuracy of 86%. In addition to that, the classification accuracy raises to 98% applying a classification confidence post-processing function, with a properly selected threshold (i.e., $\omega = 0.9$) that filters out uncertainly predicted examples (around one every two). Given these facts, we strongly believe that, with ω properly tuned, the performance of MTPlug will remain stable when the number of laptop-users increases, against an increasing number of uncertainly predicted examples filtered out. We also investigated the impact of a single and combined electrical quantities in laptop-users classification. We point out that it is relatively easy to profile a laptop-user. In fact, even collecting not contiguous energy traces for a total amount of less than a 3.5 hours are sufficient to build an effective laptop-user energy fingerprint. Our findings show laptop energy traces have to be considered as sensitive information because they expose the user to threats to her privacy such as identification or position tracking.

Chapter 5

Data Exfiltration

Following the trend of offering a better user experience, mobile apps are becoming more and more energy-draining (e.g., Pokémon Go, Snapchat, Netflix). As a direct consequence, users would eventually look for a source of electric power to recharge the battery of their mobile devices. For this reason, the demand for public charging stations has increased significantly in the last years. Such stations can be seen in public areas such as airports, shopping malls, gyms and museums, where users can recharge their devices for free. In fact, this trend is also giving rise to a special type of business[1], which allows shop owners to install charging stations in their stores so as to boost their sales by providing free phone recharge to shoppers.

As the phone recharging is usually for free, however, at the same time one cannot be sure that the public charging stations are not maliciously controlled by an adversary. The Snowden revelations gave us proof that civilians are constantly under surveillance and nations are competing against each other by deploying smart technologies for collecting sensitive information en mass. In our work, we consider an adversary (e.g., manufacturers of public charging stations, Government agencies) whose aim is to take control over the public charging station and whose motive is to exfiltrate data from the user's smartphone once the device is plugged into the station.

In this chapter, we demonstrate the feasibility of using power consumption (in the form of power bursts) to send out data over a Universal Serial Bus (USB) charging cable, which acts as a covert channel, to the public charging station. We implemented a proof-of-concept app, PowerSnitch, that can send out bits of data in the form of power bursts by manipulating the power consumption of the device's CPU. Interestingly, PowerSnitch does not require any special permission from the user at install-time (nor at run-time) to exfiltrate data out of the smartphone over the USB cable. On the adversary's side, we designed and implemented a decoder to retrieve the bits

[1] `chargeitspot.com,chargetech.com`

that have been transmitted via power bursts. Our empirical results show that we can successfully decode a payload of 512 bits with a 0% Bit Error Ratio (BER). In addition, we stress that the goal of this work is to assess for the first time the feasibility of data transmission on such a covert channel and not to optimize its performance, which we will tackle as future work.

We focus primarily on Android, as it is currently the leading platform and has a large user base. However, we believe that this attack can be deployed on any other smartphone operating systems, as long as the device is connected to a power source at the public charging station.

Our contributions are as follows:

1. To our knowledge, we are the first to demonstrate the practicality of using the power feature of a USB charging cable as a covert channel to exfiltrate data, in the form of power bursts, from a device while it is connected to a power supplier. The attack works in Airplane mode as well.

2. We implemented a prototype of the attack, i.e., we designed and implemented its two components: (i) We built a proof-of-concept app, PowerSnitch, which does not require any permission granted by the user to communicate bits of information in the form of power bursts back to the adversary; (ii) The decoder is deployed on the adversary side, i.e., public charging station to retrieve the binary information embedded in the power bursts.

3. We are able with our prototype to actually send out data using power bursts. Our prototype demonstrates the practical feasibility of the attack.

The rest of the chapter is organized as follows. In Section 5.1, we present a brief literature overview of covert channel and data exfiltration techniques on smartphones. Section 5.2 includes some background knowledge on Android operating system, and signal transmission and processing. In Section 5.3, we provide a description of our covert channel and decoder design, followed by the experimental results in Section 5.4 and discussion in Section 5.5. We conclude the chapter in Section 5.6.

5.1 Related Work

In this section, we survey the existing work in the area of covert channels on mobile devices. We also present other non-conventional attack vectors, such as side channel information leakage via embedded sensors which can be used for data exfiltration.

Covert Channels: A covert channel can be considered as a secret channel used to exfiltrate information from a secured environment in an undetected manner. Chandra et al. [43] investigated the existence of different covert channels that can be used to communicate between two malicious applications. They examined the common resources (such as battery) shared between two malicious applications and how they could be exploited for covert communication. Similar studies presented in [109, 127, 150, 175] exploited unknown covert channels in malicious and clean applications to leak out private information.

As demonstrated by Aloraini et al. [6], the adversary is further empowered as smartphones continue to have more computational power and extensive functionalities. The authors empirically showed that speech-like data can be sent over a cellular voice channel. The attack was successfully carried out with the help of a custom-built rootkit installed on Android devices. In [60], Do et al. demonstrated the feasibility of covertly exfiltrating data via SMS and inaudible audio transmission, without the user's knowledge, to other mobile devices including laptops.

In our work, we present a novel covert channel which exploits the USB charging cable by leaking information from a smartphone via power bursts. Our proposed method is non-invasive and can be deployed on non-rooted Android devices. We explain the attack in more detail in Section 5.3.1.

Power Consumption by Smartphones: In order to prolong the longevity of the smartphone's battery, it is crucial to understand how apps consume energy during execution and how to optimize such consumption. To this end, several works [23, 40, 157, 211] have been proposed. Furthermore, the authors from [105,122] studied apps' power consumption to detect anomalous behavior on smartphones, thus leading to detection of malware.

Since existing work focus on energy consumption on the device, our attack would therefore go undetected as the smartphone's CPU sends small chunks of encoded data, which are translated into power bursts, back to the public charging station. Additionally, state-of-the-art attacks that have been performed while the smartphone is charging [112,132] exploit vulnerabilities of USB interface rather than actual energy consumption.

Attack Vectors using Side Channel Leaks: Modern smartphones are embedded with a plethora of sensors that allow users to interact seamlessly with the apps on their smartphones. However, these sensors have access to an abundance of information stored on the device that can get exfiltrated. These data leaks can be used as a side channel to infer, otherwise undisclosed, sensitive information about the user or device [21, 120, 208].

The authors from [22,153] demonstrated how accelerometer readings can be used to infer tap-, gesture- and keyboard-based input from users to unlock their smartphones. Similarly, Spreitzer [186] showed that the ambient-light sensor can be exploited to infer users' PIN input. Moreover, considering network traffic as a side-channel, it is possible to identify the set of apps installed on a victim's mobile device [190, 196], and even infer the actions the victim is performing with a specific app [51].

As pointed out in the aforementioned existing work, the adversarial model did not require any special privileges to exploit side channel leaks to recover data exfiltrated via sensors. In this chapter, we show that our custom app, PowerSnitch, does not require any special permissions to be granted by the user in order to communicate information (in terms of power bursts) to the adversary. Furthermore, we stress that while the `INTERNET` permission is one approach of data exfiltration, our proposed work is different as we show the feasibility and practicability of using a USB cable to exfiltrate data. In particular, our attack still works even when the phone is switched to Airplane mode and defeats existing USB charging protection dongles, as in [42], since we only require the USB power pins to exfiltrate data.

5.2 Background Knowledge

In this section, we briefly recall several concepts that we use in our work about Android operating system in Section 5.2.1, and signal transmission and processing in Section 5.2.2.

5.2.1 Android System and Permissions

In the Android Operating System (OS), apps are distributed as `APK` files. These files are simple archives which contain bytecode, resources and metadata. A user can install or uninstall an app (thus the APK file) by directly interacting with the smartphone.

Android apps can be of two kinds:

- *GUI apps*, which prompt users with a graphical user interface that they can interact with.
- *Services* that run in the background, independently from user interactions, and provide a service to the user or to other apps.

When an Android app is running, its code is executed in a sandbox. In practice, an app runs isolated from the rest of the system, and it cannot directly access other apps' memory. The only way an app could gain memory access is via the mediation of inter-process communication techniques made available by Android. These measures are in place to prevent the access of malicious apps to other apps' data, which could potentially be privacy-sensitive.

Since Android apps run in a sandbox, they not only have restriction in shared memory usage, but also to most system resources. Instead, the

Android OS provides an extensive set of Accessible Programming Interfaces (APIs), which allows access to system resources and services. In particular, the APIs that give access to potentially privacy-violating services (e.g., camera, microphone) or sensitive data (e.g., contacts) are protected by the Android Permission System [71]. An app that wants access to protected data or service must declare in the form of permission (identified by a string) in its manifest file. The list of permissions needed by an app is shown to the user when installing the app, and cannot be changed while an app is installed on the device. With the introduction of Android M (i.e., 6.0), permissions can be dynamically granted (by users) during an app's execution.

The permission system has also the goal of reducing the damage in case of a successful attack that manages to take control of an app, by limiting the resources that app's process has access to. Unfortunately, permission over-provisioning is a common malpractice, so much so that research efforts have been spent in trying to detect this problem [24]. Moreover, an app asking for permissions not related to its purpose (or functionality) can hide malicious behaviors (i.e., spyware or malware apps) [137].

5.2.2 Signal Transmission and Processing

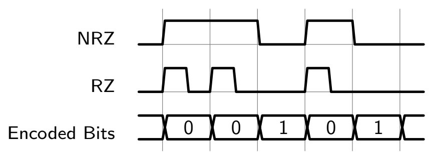
In this section, we provide some background information on bit transmission, and signal processing and decoding used in our proposed decoder (see Section 5.3.4).

Bit Transmission: To enable bit transmission over our channel, an understanding of basic digital communication systems is essential. For proof-of-concept purposes, the design of our bit transmission system was inspired by amplitude-based modulation in the digital communication literature.

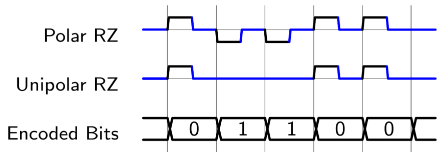
Amplitude-Shift Keying (ASK) is a form of digital modulation where digital bits are represented by variations in the amplitude of a carrier signal. To send bits over our channel, we used On-Off Signaling (OOS), which is the simplest form of ASK where digital data is represented by the presence and absence of some pulse *p(t)* for a specific period of time. Figure 5.1a shows the difference between a Return-to-Zero (RZ) and a Non-Return-to-Zero (NRZ) on-off encoding. In NRZ encoding, bits are represented by a sufficient condition (a pulse) that occupies the entire bit period T_b while RZ encoding represents bits as pulses for a duration of $T_b/2$ before it returns to zero for the following $T_b/2$ period.

On the other hand, Figure 5.1b shows the difference between a unipolar and a polar RZ on-off signaling. In a polar RZ encoding, two different conditions, different-sign pulses are used to encode different bits(zeros/ones) while the presence and absence of a single pulse, a positive one in our case, are used to encode different bits.

For the sake of our channel design, it is safe to assume that we can only increase the power consumption of a phone at certain times and hence, are able to generate only positive (high) bursts. Thus, a unipolar encoding seems more relative and applicable for our channel. Moreover, successive peaks, such as the first two zeros in Figure 5.1a, are easier to identify, and thus decode, in the RZ-encoded signal than in the NRZ one. This advantage of RZ over NRZ becomes especially apparent in cases where the bit period is expected not to be restrictively fixed in the received signal whether it is due to expected high channel noises or lack of full control of the phone's CPU. Therefore, unipolar RZ on-off signaling was used to encode leaked bits over our covert channel.



(a) Return-to-Zero (RZ) and Non- Return-to-Zero (NRZ) On-Off Encoding.



(b) A Polar and a Unipolar encoding of an RZ On-Off Signal.

Figure 5.1: A comparison between bit encoding methods

Signal Processing and Decoding: After choosing the appropriate encoding method to transmit bits, it is also essential to think about the optimal receiver design and how to process the received signal and decode bits with minimum error probability at the receiver side of the channel. As known in the digital communication literature, matched filters are the optimal receivers for Additive White Gaussian Noise (AWGN) channels. We refer the reader to Section 4.2 of [161] for a detailed proof.

Matched Filters are obtained by correlating the received signal $R(t)$ with the known pulse that was first used to encode a transmitted bit, in this case P(t) with period T_b. After correlation, the resulted signal is then sampled at time T_b, which means that the sampling rate equals to $1/T_b$ samples/seconds. This way, each bit is guaranteed to be represented by only one sample. The decoding decision will then be made based on that one sample value; if the sample value is more than a given threshold, this indicates the presence of P(t); and hence a zero in our case, while a sample value below the threshold indicates the absence of P(t) and hence a one is decoded.

However and most importantly, for matched filters to work as expected, it is essential to have fixed bit period T_b throughout the entire received signal. If the periods of the received bits were varying, the matched filter samples taken with the $1/T_b$ sampling rate will not be as optimal and representative of the bit data as expected and synchronization will be lost.

Since there exist infrequent phone-specific, OS-enforced conditions that can affect the power consumption of a phone, the noises on our channel are expected to be more complex to fit in an AWGN model. Hence, a matched filter receiver is most likely not the optimal receiver for our channel. More creative decoder design decisions are needed to maximize the throughput of our channel and minimize the error probability.

5.3 Covert Channel using Mobile Device Energy Consumption

In this section, we elaborate on the components that make up our covert channel attack. We begin by giving an overview of the attack in Section 5.3.1. We then define the terms and parameters for transmission in Section 5.3.2, followed by a description of each component of the attack: PowerSnitch app in Section 5.3.3 and the energy traces decoder in Section 5.3.4.

5.3.1 Overview of Attack

As illustrated in Figure 5.2, the attack scenario considers two components: the victim's Android mobile device (sender) and an accomplice's power supplier (receiver). Victim's mobile device is connected to a power supplier (controlled by the adversary) through a USB cable.

The left side of Figure 5.2 depicts what happens after the victim has installed our proof-of-concept app, *PowerSnitch*. The app is able to exfiltrate victim's private information, which gets encoded as CPU bursts with a specific timing. Indeed, as the CPU is one of the most energy consuming resources in a device, a CPU burst can be directly measured as a "peak" based on the amount of energy absorbed by a mobile device. The right side of Figure 5.2 illustrates how the energy supplier is able to measure (with a given sampling rate) the electric current provided to the mobile device connected to the public charging station. Then, such electric measurement, which is considered as a signal, is given as input to a decoder. It should be noted that the adversary, i.e., the public charging station, has control of the power supplier, and thus is able to control the amount of current provided to the device – even if it has the "fast charge" capability.

In our proposed covert channel attack, we consider situations in which users connect their mobile devices for more than 20 minutes. There are several scenarios that fulfill such time requirements. Examples are: (i) recharging a device overnight in a hotel room; (ii) making use of locked boxes in shopping malls for charging mobile phones; (iii) recharging devices on planes, in trains and cars.

In addition, we argue that those time requirements are more than reasonable since generally, 72% of users leave their phones on charging for more than 30 minutes, with an average time of 3 hours and 54 minutes, as reported in [74]. This means that: (i) the mobile device is in stand-by mode; (ii) CPU and the use of other energy consuming resources (e.g., WiFi or 3/4g data connection) usage is limited only to the OS and background apps. Moreover, since there is no user interaction, it is reasonable to assume that the phone screen, which has a relevant impact on energy consumption, will stay off for the aforementioned period of time.

Moreover, it is also worth noting that the attack is still feasible if there is no data connection between the victim's device and the power supplier, such as Media Transfer Protocol (MTP), Photo Transfer Protocol (PTP), Musical Instrument Digital Interface (MIDI). This is possible as our methodology only requires power consumption to send out the power bursts. Moreover, from Android version 6.0, when a device is connected via USB, it is set by default to "Charging" mode (i.e., just charge the device), thus no data connection is allowed unless the user switches on data connection manually. This improvement in security feature does not impact our proposed attack as we do not make use of data connection to transfer the power bursts.

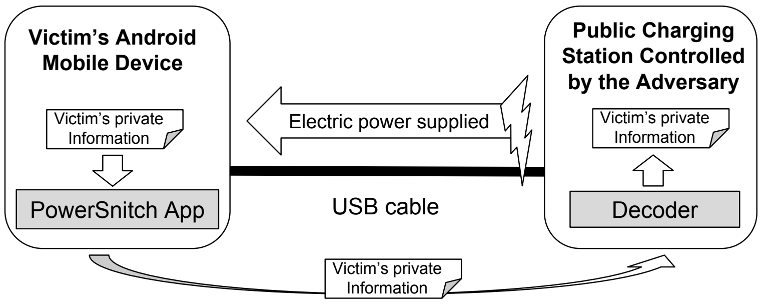

Victim's Android Mobile Device
Victim's private Information
PowerSnitch App
Electric power supplied
USB cable
Public Charging Station Controlled by the Adversary
Victim's private Information
Decoder
Victim's private Information

Figure 5.2: The schema of the components involved in the attack.

5.3.2 Terminology and Transmission Parameters

In this section, we define the necessary terminology to identify concepts used in the rest of the chapter:

- *Payload* is the information that has to be sent from the device to the receiver.
- *Transmission* is the whole sequence of bits transmitted in which the payload is encoded.

In order to obtain a successful communication, the sender and the receiver need to agree on the parameters of the transmission.

- *Period* is the time interval during which a bit is transmitted.
- *Duty cycle* is the ratio between burst and rest time in a period T_b. For example: if a burst lasts for $T_b/2$, the duty cycle will be 50%.
- *Preamble* is the sequence of bit used to synchronize the transmission. Usually a preamble is used at the beginning of a transmission, but it can also be used within a transmission in order to recover the synchronization in case of error. In our case, we used a preamble composed of 8 bits.

5.3.3 PowerSnitch app: Implementing the Attack on Android

The first component of our covert channel we discuss is the proof-of-concept which we called PowerSnitch app. This app, used for the covert channel exploit, has been designed as a service in order to be installed as a standalone app or a library in a repackaged app. Henceforth, we refer to both these variants simply with the term "app".

PowerSnitch app requires only the `WAKE_LOCK` permission and does not need root access to work. Such permission allows PowerSnitch app to wake and force execute the CPU while the device is in sleep mode, so that it can start to transmit the payload. We stress that since it is running as a background service, PowerSnitch app still works even when user authentication mechanisms (e.g., PIN, password) are in place. Moreover, since it does not use any conventional communication technology (e.g., WiFi, Bluetooth, NFC), PowerSnitch app can exfiltrate information even if the device is in airplane mode. It is worth mentioning that Android M (i.e., 6.0) introduced the Doze mode [19], a battery power-saving optimization which reduces the apps activity when the device is inactive and running on battery for extended periods of time. When it is in place, Doze mode stops background CPU and network activity (ignoring wakelocks, job scheduler, WiFi scan, etc.). Then on periodic time intervals (i.e., maintenance windows), the system runs all pending jobs, synchronization and alarms. However, such optimization is not active when a device is connected to a power source or when the screen is on. This means that Doze does not affect our proposal since we need the wakelock function but also the device to be plugged to a power source. Moreover, since our proposed attack needs also the status of the battery, it does not need any permission in order to obtain such information: in fact, it is sufficient to only register at run-time (not even in the manifest) a specific broadcast receiver (i.e., `ACTION_BATTERY_CHANGED`).

In Figure 5.3, we illustrate the modules of PowerSnitch app. It is composed of three modules: *Payload encoder*, *Transmission controller* and *Bursts generator*. *Payload encoder* takes the payload as input and outputs an array of bits. The payload can be any element that can be serialized into an array of bits. We use strings as payloads, they are first decomposed into an array of characters and then, using the ASCII code of each character, into an array of bits. *Payload Encoder* can also add to its output array synchronization bits (e.g., the preamble), and error checking codes (e.g., CRC).

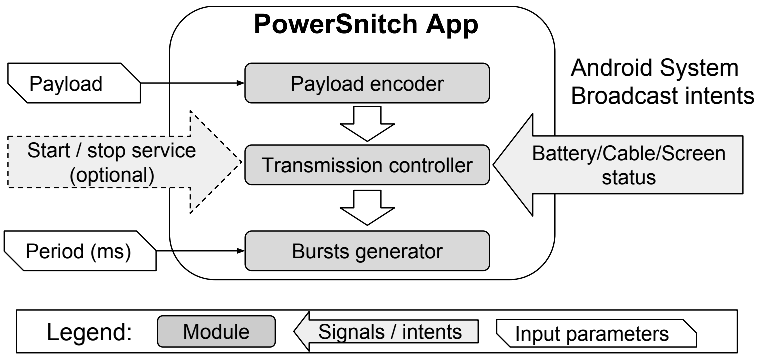


Figure 5.3: The modules involved in the PowerSnitch app.

Transmission controller is in charge of monitoring the status of the device with the purpose of understanding when it is feasible to transmit through the covert channel. Indeed, in order to not be detected by the user, it checks whether all the following conditions are satisfied: (i) the USB cable is connected; (ii) the screen is off; and (iii) the battery is sufficiently charged (see Section 5.5). If our app receives a broadcast intent from the Android OS that invalidates one of the aforementioned conditions, *Transmission controller* module will interrupt the transmission. It is worth noticing that to obtain all this information, PowerSnitch app does not need any additional permission. From the GUI app used in our experiments, we are also able to start or stop PowerSnitch app (represented in Figure 5.3 with a dotted arrow).

The last component of PowerSnitch app is *Bursts generator*. The task of this component is to convert the encoded payload into bursts of energy consumption. These bursts will generate a signal that can be measured at the other end of the USB cable (i.e., the power supplier). In order to obtain these bursts of energy consumption, *Bursts generator* module can use a power consuming resource of the mobile device such as CPU, screen or flashlight. Our proof-of-concept, *Bursts generator* uses the CPU: a CPU burst is generated from a simple floating point operation repeated in a loop for a precise amount of time (given by transmission parameters).

5.3.4 Analysis of Energy Traces

To make better decoder design decisions, several channel traces were observed, collected and then used to calculate channel estimations and implement different simulations of the channel performance and behavior. A standard on-off signaling decoder needs to know the exact period of bits in the received signal in order to be able to decode them. However, a channel built based on a phone's power consumption is expected to have hard-to-model noises that, after examining the collected channel data traces, are actually affecting not only the peak periods but also the peak amplitudes. The amount of external power consumed by a phone can be largely affected by dominant OS-enforced, manufacturer-specific factors. For instance, different sudden drop patterns in power consumption especially when the phone is almost or completely charged, lack of control over the OS scheduler; when, how often and for how long do some heavy power-consuming OS background services run, as well as the precision and sampling rate of the power monitor on the receiver side of the channel.

Figure 5.4 shows a portion of the channel data captured after a transmission of ten successive bits (ten Zeros, therefore ten peaks) was initiated by our app on a Nexus 6 phone. It should be noted that the data was passed through a low-pass filter to get rid of harsh, high frequency noises in order to make the signal looks smoother. As a result, based on a threshold of $100mA$, ten peaks are successfully detected. Moreover, the width of each peak, and hence the period of each bit, is varying sufficiently. The first bit, for example, has a period of $300ms$ while the eighth one has a period of only $195ms$. Although the intended bit period generated and transmitted by the app was $500ms$, the average period of the received bits was actually $311ms$, which the receiver has no way to predict in advance. Such variations in the received signal are expected to affect the performance of any decoder. An ideal matched filter receiver will have hard time decoding such inconsistent signal and synchronization will be lost very quickly. We elaborate further on this issue in the remaining sections.

Decoder Design

In this section, we provide additional explanation about the different processing stages that our decoder is taking the received signal through in order to overcome the channel inconsistencies and decode the sent bits with the minimum Bit Error Ratio (BER). In signal processing, the quality of a communication channel can be measured in terms of BER (represented as a percentage), which is the number of bit errors divided by the total number of transmitted bits over the channel. Channels affected by interference, distortion, noise, or synchronization errors have a high BER.

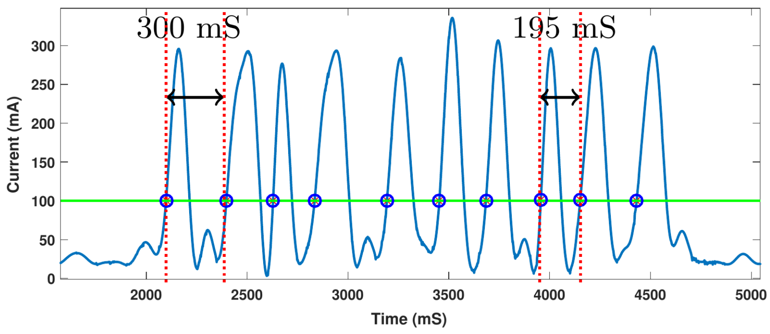


Figure 5.4: A portion of a received signal showing the variations in peak widths and amplitudes.

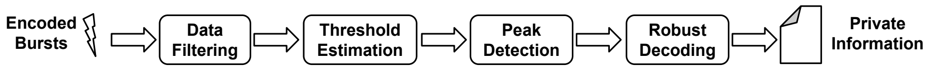


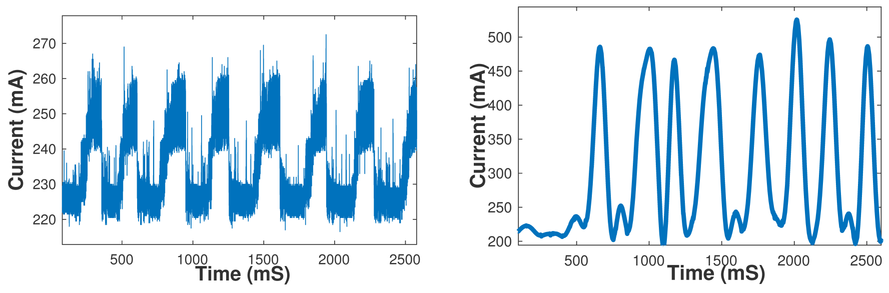
Figure 5.5: Different phases of our decoder.

Figure 5.5 summarizes the different processing stages which will be discussed in the order they take place in, along with some background information and algorithm justifications, where applicable.

Data Filtering First, the received signal is passed through a low-pass filter to get rid of the harsh high-frequency noises. For instance, Figure 5.6 shows the same portion of a received signal before and after applying the low-pass filter. The low-pass filter helps not only to make the signal looks smoother, but also to make the threshold-based detection of real peaks easier by eliminating narrow-peak noises that can be falsely identified as real peaks or bits. Additionally, the low-pass filter used in our decoder adjusts its pass and stop frequencies based on the intended bit period generated by the phone in order to make sure that we do not over-filter or over-attenuate the signal.

Threshold Estimation The decoder detects peaks by decoding unipolar RZ on-off encoded bits. The presence or absence of a peak (a 0 or a 1 in our case, respectively) at a certain time and for a specific period is then translated to the corresponding bit. Peak detection is usually done by setting an appropriate threshold; anything above the threshold is a peak and anything below is just noise. However, deciding which threshold to use is not a trivial process especially with the unpredictable noise in our channel and the variations in width and amplitude of the received peaks.

The threshold value used by the decoder is highly critical to peaks detection, the resulted width of detected peaks and the decoder performance. Hence, we primarily use a known preamble data sent prior to the actual



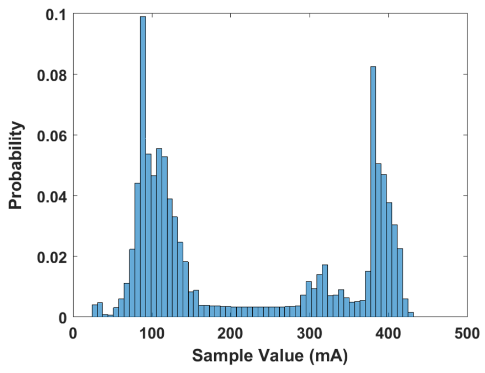
(a) Raw received signal. (b) Low-pass filtered received signal.

Figure 5.6: A portion of a received signal before and after applying the low-pass filter.

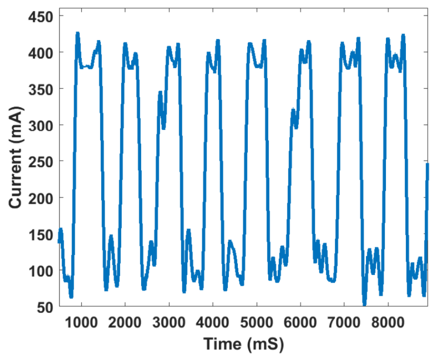
packet to estimate the threshold. The preamble consists of eight known bits (eight zeros in our case) at the start of the transmission, which means that the decoder is expecting eight peaks at the start. Since a unipolar RZ on-off encoded zero has a pulse for half of the bit period, the preamble is expected to have roughly the same number of peak and no-peak samples. Therefore, a histogram of the preamble samples is expected to split into two portions; peak and no-peak portions. Figure 5.7a shows a histogram of the preamble samples shown in Figure 5.7b. As observed, the histogram has two distinguishable densities; each of them look like the probability density function of a Gaussian distribution.

Estimating the parameters (mean and variance) of two Gaussians that are believed to exist in one overall distribution is a complicated statistical problem. However, the Gaussian Mixture Model (GMM), introduced and explained in [170], is a probabilistic model commonly used to address this type of problem and to statistically estimate the parameters of existing Gaussian populations. To estimate the threshold, as shown in Figure 5.8, the decoder uses the GMM to fit two Gaussians to the two histogram portions, find the mean of each one of them and then compute the threshold as the middle point between the two means. As a result, our decoder is able to estimate the threshold independently and without any previous knowledge of the expected amplitudes of the received bits. After that, each sample is converted to either a peak sample or no-peak sample based on whether the sample value is above or below the estimated threshold.

Robust Decoding Generally, the way a decoder translates the peak and no-peak samples to zeros and ones is highly time-sensitive. For instance, if the bit period is fixed and equals to T_b, the decoder simply checks the presence or absence of the peak in each T_b period. Since this decoding decision is made based on a very strict timing manner, the slightest error



(a) A histogram of the preamble samples.



(b) A received preamble signal.

Figure 5.7: A histogram of the preamble samples shows a mixture of two Gaussian-like densities.

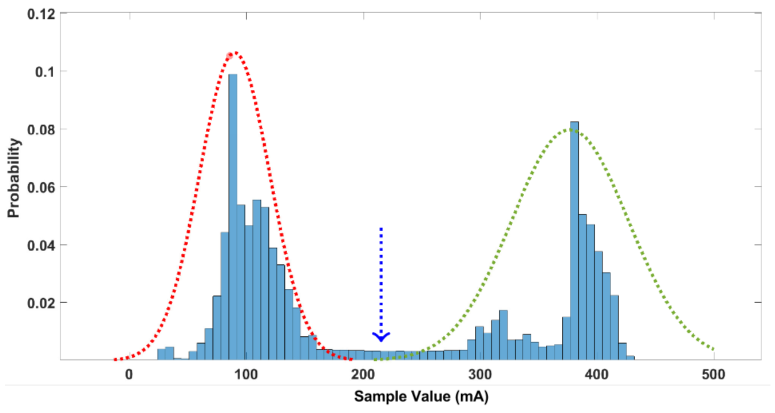


Figure 5.8: Using the Gaussian Mixture Model to estimate the threshold.

in the received bit periods will cause a quick loss of synchronization. As mentioned in the previous section, the received peak widths (and hence bit periods) over our channel are changing with a high variation around their mean. Therefore, our decoding decision cannot rely on an accurate notion of time. Instead, our decoder needs to assume a sufficient amount of error in the period of each received bit and to search for the peaks in a wider range instead of a strict period of time.

To address this level of time-insensitivity and achieve robustness to synchronization errors, our decoding decision was made based on the time difference between each two successive peaks. As an example, assume that two successive zeros were sent and hence two peaks were received. The difference between the start time of each peak should be rounded to the average bit period. It should be noted that the decoder computes the average bit period based on the received preamble data. However, if a zero-one-zero transmission was made, the time difference of the start of the two received peaks should be rounded to double of the average bit period. If a zero-one-one-zero transmission was made, the difference should be rounded to triple the average period and so on.

Eventually, synchronization is regained with every detected peak and based only on the time difference between peaks, the decoder makes a decision on how many no-peak bits (ones in our case) are transmitted between the zeros. The time difference does not have to be exactly equal to a multiple of the average bit period. Instead, a range of values can be rounded up to the same value and thus more flexible time-insensitive decoding decision is made.

5.4 Experimental Evaluation

In this section, we first describe the devices used in our experiments and the values for transmission parameters. We then report the results of the transmission evaluation.

5.4.1 Experiment Settings

In our experiments, we programmed the PowerSnitch app using Android Studio with API. The device used to measure the energy provided to the device via USB cable is Monsoon Power Monitor[2] in USB mode with $4.55V$ in output. The decoder used to process signal was implemented in MATLAB. In order to evaluate the performance of the transmission, we send out a payload comprised of letters and numbers of ASCII code for a total of 512 bits. The values of period used range from $500ms$ to $1000ms$ with increments of $100ms$. It is worth mentioning that bits sent over our channel were not packeted and no error detection or correction techniques were used. For each phone and bit period, BER was computed after sending 512 bits at once and then number of bits that were incorrectly decoded was calculated.

We evaluate the performance of our proposal on the following devices running Android OS: Nexus 4 with Android 5.1.1 (API 22), Nexus 5 with Android 6.0 (API 23), Nexus 6 with Android 6.0 (API 23) and Samsung S5 with Android 5.1.1 (API 22). We underline that the devices used in our experiments are actual personal devices, kindly lent by some users without any money reward. In order to replicate an actual real world scenario, we did not uninstall any app, nor stopped any app running in background. The only intervention we made on those devices is the installation of our PowerSnitch app.

[2] `www.msoon.com/LabEquipment/PowerMonitor`

Device		**Period (milliseconds)**					
Model	Operating system	1000	900	800	700	600	500
Samsung S5	Android 5.1.1 (API 22)	12.5	13.5	13.31	16.33	17.9	21.42
Nexus 4	Android 5.1.1 (API 22)	13.5	0.78	0.0	0.0	13.33	16.21
Nexus 5	Android 6.0 (API 23)	21.0	0.0	0.95	36.82	40.35	13.4
Nexus 6	Android 6.0 (API 23)	1.07	0.0	0.21	0.0	4.05	7.42

Table 5.1: Results in terms of Bit Error Ratio (BER) as percentage.

5.4.2 Results

In Table 5.1, we report the performance of the decoder for processing the received power bursts on different mobile devices. The results presented in the table are in terms of BER in the transmission of the payload; the lower the BER, the better is the quality of the transmission. For Nexus devices (i.e., Nexus 4, 5 and 6), we achieve a zero or low BER of periods of 800ms and 900ms (i.e., 1.25 and 1.11 bits per seconds, respectively). While for Nexus 4 and 6, the BER remains under 20% and, for Nexus 5, it increases to 37% and 40% with periods 700ms and 600ms, respectively. For Samsung S5, the transmission BER is at 12.5% with a period of 1 second, and it slowly increases to around 21% with a period of half a second.

The higher BER for Nexus 5 (i.e., periods 700ms and 600ms in Table 5.1) are due to de-synchronization of the signal that the decoder was not able to recover. To cope with this problem, we can divide the payload into packets, where a packet header will be the preamble in order to recover the synchronization. A quick overview of the communication literature can show how a BER of 30% can be recovered using a simple Forward Error Correction (FEC) technique where the transmitter encodes the data using an Error Correction Code (ECC) prior to transmission; for example bits redundancy or parity checks.

5.5 Discussion and Optimizations

In this section, we elaborate further on the results obtained in the experimental evaluation of our proposed attack (Section 5.4). In particular, we discuss on interesting observation made during our experiments. We also present the optimizations that were implemented in the framework in order to make our proposed attack more robust.

An interesting phenomenon to notice is that, as observed in our experiments, the level of battery affects the quality of the transmission signal. In Figure 5.9, we present the amount of electric current provided by the power supplier to a Nexus 6 during recharge (i.e., the first 35 minutes) and full battery states (i.e., after 35 minutes). Indeed, when the level for the battery is low (i.e., 0% to around 40%) the device consumes a high amount of energy, and almost all of it is used to recharge the battery.

When attempting to transmit data in the aforementioned conditions, we discover that the bursts were not easily distinguishable. In fact, the difference in terms of energy consumption between burst and rest was so small that it cannot be distinguished from noise; thus, they can be filtered out during the signal processing. Additionally, when the level of the battery is increased, the amount of energy consumed to recharge the battery gradually decreases. We observed that when the battery level is higher than 50%, the power bursts become more and more distinguishable. However the best condition under which the bursts are clear is when the battery is fully charged. Indeed, as we can notice from Figure 5.9, the current drops down after the battery level reaches 100%, because there is no need to provide energy to the battery anymore - except to keep the device running.

The percentages mentioned above also depends from the power supplier used to provide energy to the device. In our experiments, we used Monsoon power monitor which provides as output at most $4.55V$. Due to the limitation of such power monitor, during the recharge of devices with fast charge technology (e.g., Samsung S5, Nexus 6 and 6P), which are able to work with $5.3V$ and $2mA$, the energy consumed is almost constant until the battery is almost fully charged. Thus, we cannot decode any signal from the energy consumption.

In order to avoid to transmit when the receiver is not able to decode the signal, PowerSnitch checks whether the battery level is among a certain threshold ω. Such threshold ω can be obtained by PowerSnitch itself, simply knowing the model in which it is running. This information can be easily obtained without any permission (`android.os.Build.MODEL` and `MANUFACTURER`).

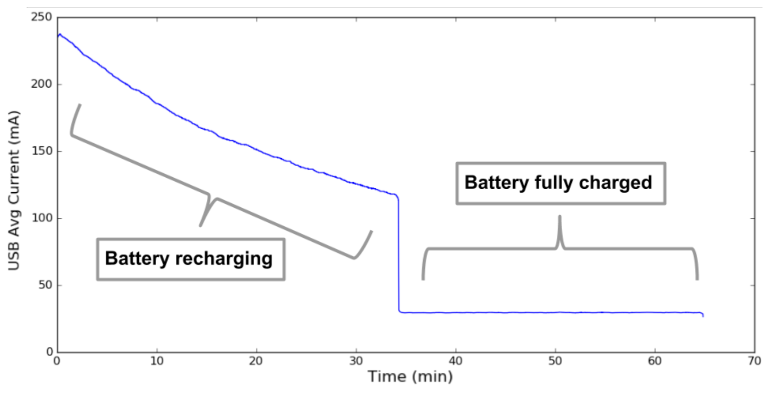


Figure 5.9: Electric current provided to a Nexus 6 during recharge phase and battery fully charged.

Optimizations In what follows, we elaborate on the optimizations that were implemented in order to not be detected or make the victim suspicious. The first optimization is to keep a duty cycle (i.e., the time of burst in a period) under 50%. During an attack, if such optimization is not taken into account (i.e., a duty cycle greater than 75%), the victim may be alerted by two possible effects:

- the temperature of the device could increase significantly, in a way that could be perceived by touching it.
- if the attack takes place during the battery charge phase, the battery will take more time to recharge due to the high amount of energy used by CPU.

However, as previously explained in Section 5.2.2, the duty cycle should be 50% of period (i.e., $T_b/2$) in order to achieve a RZ. Thus, the above effects are already taken care of in our proposed attack.

Another optimization involves the Android Debug Bridge (ADB) tool. It is possible to monitor CPU consumption of an Android device via ADB. Hence, one may use such debug tool to detect that something strange is happening on the device (i.e., a transmission on the covert channel using CPU bursts). Fortunately, PowerSnitch app could easily detect whether ADB setting is active through `Settings.Global.ADB_ENABLED`, once again provided by an Android API.

Another optimization to PowerSnitch app would be the ability to detect if the power supplier is an accomplice of the attack. The accomplice has to let PowerSnitch app know that it is listening to the covert channel by communicating something equivalent to a "hello message". In order to do so, we can rely on the information about the amount of electric current provided to recharge the battery. Such information is made available through `BatteryManager` object, provided by Android API. In particular, `BATTERY_PROPERTY_CURRENT_NOW` data field (available from API 21 and on devices with power gauge, such as Nexus series) of `BatteryManager` records an integer that represents the current entering the battery in terms of mA.

On one hand, the power supplier can then variate the current in output above and below a certain threshold θ with a precise timing. As a practical and non-limiting example, at a point in time during the recharging, the power supplier can output current with the following behavior: (i) below θ for t seconds, (ii) above θ for t seconds, (iii) again below θ for t seconds and finally (iv) above θ for good. On the other hand, since PowerSnitch app monitors `BATTERY_PROPERTY_CURRENT_NOW` and knows the aforementioned behavior (along with both θ and t), it will be able to understand that at the other end of the USB cable there is an accomplice power supplier ready to receive a transmission. This optimization is significant for reducing the chance to remain undetected, since PowerSnitch app will transmit data if and only if it is sure that an accomplice power supplier is listening. With such optimization, we will obtain a half-duplex communication channel, since the communication is bidirectional but only one participant (i.e., the device or the power source) is allowed to transmit at a time. This optimization is not currently implemented and will be considered as future work.

To summarize, the conditions under which the transmission of data is optimal and the chance of being detected is lowest are as follows: the mobile device has to be charged more than 50%, the screen has to be off, ADB tool should be switched off (which is true by default) and the phone must to be plugged with a USB charging cable to a public charging station which is controlled by the adversary.

5.6 Summary

In this chapter, we demonstrate for the first time the practicality of using a (power-only) USB charging cable as a covert channel to exfiltrate data from a smartphone, which is connected to a charging station. Since there are no visible signs of the existence of a covert channel while the battery is recharging, the user is oblivious that data is being leaked from the device. Moreover, our proposed covert channel defeats existing USB charging protection dongles, as described in [42] because it requires only the USB power pins to exfiltrate data in the form of CPU power bursts.

To show the feasibility and practicality of our proposed covert channel, we implemented an app, *PowerSnitch*, which does not require the user to grant access to permissions at install-time (nor at run-time) on a non-rooted Android phone. Once the device is plugged in a compromised public charging station, the app encodes sensitive information and transmits it via power bursts back to the station. Our empirical results show that we are able to exfiltrate a payload encoded in power bursts at 1.25 bits per seconds with a BER under 1% on the Nexus 4-6 devices and a BER of around 13% for Samsung S5.

Part III

Built-in Sensors Analysis

Chapter 6

Logging and Data Extraction Tool

Along with mobile operating systems, mobile devices are equipped with a wide range of sensors, which collect data from the surrounding environment. Built-in sensors such as gyroscopes, accelerometers, GPS receivers and digital compasses allow a mobile device to know its orientation, speed, and position. As another example, built-in microphones and cameras collect audio and visual feedback. Nowadays, all the aforementioned components (and more) are commonly included in modern mobile devices, usually even in low and mid-range ones.

The above mentioned hardware and software features make modern mobile devices excellent data gathering devices for research purposes. In fact, they allow us to explore whole new areas of research, spanning multiple fields. For example, in the field of smartphone security, sensor and usage data allows the development of new authentication techniques [54, 75, 85], user profiling studies [192], and new attacks on user privacy that exploit side-channel information [51, 70, 196]. Another interesting avenue of research is using smartphones as portable monitoring stations, able to perform a variety of background monitoring tasks. Examples include collecting readings from personal health sensors [166], recording ambient data like air and sound pressure, monitoring and tracing people movements and habits [96]. Software development can also benefit from such data from simply collecting logs to monitor apps performance and crashes [47].

In this chapter, we categorized the types of data we can extract from mobile devices into three main categories:

- **Sensor data** is data we can gather directly by querying the many sensors embedded in modern mobile devices. This includes a wide variety of information about the device itself and its surrounding environment (e.g., the device's position, orientation and relative speed).

- **Device/OS context data** is the state of the device itself and its operating system (e.g., battery level, list of running processes, traffic statistics, and file system activity).

- **User interaction data** is related to the device's user and her actions and habits, such as how she interacts with the touchscreen, with the keyboard and with elements of the User Interface.

Given the high value of this data for research, a powerful and flexible multi-purpose logging tool would be of extreme value, as it would enables researchers to make data gathering for their projects easier, effective and efficient.

The contribution of our work is two fold:

1. We present a survey on the existing logging tools for mobile devices. In particular, we select the more prominent tools present in the literature and we thoroughly compare their data collection features and architecture, highlighting both their selling points and limitations.

2. We present DELTA - Data Extraction and Logging Tool for Android[1], our own logging solution for the Android platform. We designed and implemented DELTA to cover the shortcomings most commonly found in other tools. Our tool logs as many information sources as possible, while at the same time allowing flexibility in what data is logged and at which frequency. Moreover, DELTA's architecture is designed to be modular and as non-invasive as possible in regards to user privacy and system security.

While DELTA does not introduce novel techniques in the way data is collected from the Android operating system, we believe that the real value of our tool is the way it makes easy for the user to collect and organize logs from many sources without having to write her own implementation.

We make the DELTA source code and toolset available to the research community and practitioners, so that interested people can leverage it to streamline the process of logging data for their experiments.

Organization: The rest of this chapter is organized as follows. In Section 6.1, we present a thorough survey on the existing data collection tools for mobile devices. In Section 6.2, we go in-depth about the design and inner workings of the DELTA system. Finally, in Section 6.3, we summarize the work in this chapter.

[1]DELTA is open source and it is available at `http://spritz.math.unipd.it/projects/deltaloggingtool/`

6.1 Mobile Data Collection Tools

In this section, we open with a survey on existing mobile logging tools and other related work in the literature (sections 6.1.1 and 6.1.2). We then sum up the main limitations common to existing tools (Section 6.1.3). Finally, we explain how our proposal outperforms existing solutions (Section 6.1.4).

6.1.1 Existing tools

While several logging tools exist that are aimed at a mainstream public (e.g., mSpy[2], MobiStealth[3]), they are generally unsuitable for research purposes. Indeed, such tools are designed and marketed as "spy apps" and they do not provide the precision and customization required for a research project. In addition to that, these apps are typically designed to send data to third-party remote servers, compromising user privacy and blocking access to the collected data behind a pay-wall. For these reasons, we did not consider any "spy apps" in this survey. Our solution is explicitly aimed at researchers, and we want it to be as open and easily accessible as possible.

Data collection is also an important aspect in the field of forensics. However, in forensic analysis the aim is to extract relevant information from a device at a certain point in time, usually in the context of a law enforcement operation. This is why forensic tools for mobile devices (e.g., ADEL [77]) are designed to perform a one-time data extraction, rather than to continuously monitor a device. This approach is very limiting for research purposes, where researchers typically want to collect usage data over time in order to find correlations and make predictions. In fact, a one-time extraction cannot provide a history of sensor readings and system events because it is usually more focused at making a snapshot of the current state of a device.

In our comparison, we focus on tools that can do continuous and/or periodic logging of data from more than a single source or sensor: SystemSens [68], DroidWatch [90], MobileSens [91], LiveLab (iOS) [179], PhoneLab [144] and DeviceAnalyzer [202]. Tables 6.1 and 6.2 summarize a feature comparison between the tools we tested and our solution. We grouped the logging features in tables 6.1 and 6.2 to highlight the various contexts from which we can gather data from (e.g., sensor readings, screen interactions, network features). In the following paragraphs, we analyze each of these tools in detail.

SystemSens is an open source logging application presented by Falaki et al. in [68] that collects data from sensors and OS context, and it is designed to offer some extensibility options. However, *SystemSens* has some shortcomings that limit its usefulness for research purposes. Firstly, it has

[2]mSpy - `https://www.mspy.com`
[3]MobiStealth - `http://www.mobistealth.com`

Features	**Tool**	SystemSens [68]	DroidWatch [90]	MobileSens [91]	LiveLab (iOS) [179]	PhoneLab [144]	DeviceAnalyzer [202]	**DELTA (our proposal)**
Sensors	Gravity sensor	✗	✗	✗	✗	✗	✓	✔
	Accelerometer sensor	✗	✗	✗	✓	✗	✓	✔
	Magnetic field sensor	✗	✗	✗	✗	✗	✓	✔
	Proximity sensor	✗	✗	✗	✗	✗	✗	✔
	Pressure sensor	✗	✗	✗	✗	✗	✗	✔
	Light sensor	✗	✗	✗	✗	✗	✗	✔
	Humidity sensor	✗	✗	✗	✗	✗	✗	✔
	Log noise level around device	✗	✗	✗	✗	✗	✗	✔
	Record from microphone	✗	✗	✗	✗	✗	✗	✔
Screen	Screen state (off/on/unlock)	✗	✓	✓	✗	✓	✓	✔
	Touch events logging	✗	✗	✗	✗	✗	✗	✔
	Keyboard state (open/close)	✗	✗	✗	✗	✗	✗	✔
	Keylogging	✗	✗	✗	✗	✗	✗	✔
System & Power	CPU statistics	✓	✗	✗	✗	✗	✗	✔
	Battery statistics	✓	✗	✗	✓	✗	✓	✔
	Battery charging status	✓	✗	✓	✗	✓	✓	✔
	Memory statistics	✓	✗	✗	✗	✗	✗	✔
	System volume change	✗	✗	✗	✗	✗	✗	✔
	Date / Time / Timezone changes	✗	✗	✓	✗	✗	✓	✔
	Device turning on/off	✗	✗	✓	✗	✗	✓	✔
	Storage space monitoring	✗	✗	✗	✗	✗	✓	✔
	File system activity monitoring	✗	✗	✗	✗	✗	✗	✔
	Alarms ringing	✗	✗	✗	✗	✗	✓	✔
Telephony	Phone calls	✓	✓	✓	✓	✓	✓	✔
	Incoming SMS messages	✓	✓	✓	✓	✗	✓	✔
	Outgoing SMS messages	✗	✓	✗	✓	✗	✓	✔
	Address book changes	✗	✗	✓	✓	✗	✗	✘

Table 6.1: Feature comparison table – Part 1.

	Features \ **Tool**	SystemSens [68]	DroidWatch [90]	MobileSens [91]	LiveLab (iOS) [179]	PhoneLab [144]	DeviceAnalyzer [202]	**DELTA (our proposal)**
Networking	Airplane mode on/off	✗	✗	✗	✗	✗	✓	✔
	Cell tower ID	✓	✗	✗	✓	✓	✓	✔
	Cell signal strength	✗	✗	✗	✗	✓	✓	✔
	WiFi connection info	✓	✗	✓	✗	✓	✓	✔
	Scan of nearby WiFi HotSpots	✓	✗	✗	✓	✓	✗	✔
	Network status (3G/WiFi/none)	✓	✗	✗	✓	✓	✓	✔
	Network traffic statistics	✓	✗	✓	✓	✗	✓	✔
	Network packet sniffing	✗	✗	✗	✓	✗	✗	✔
	Opened URL logging	✗	✓	✓	✓	✗	✗	✘
	Bluetooth state changes	✗	✗	✗	✓	✗	✓	✔
	Bluetooth packet sniffing	✗	✗	✗	✗	✗	✗	✘
	NFC device scanning	✗	✗	✗	✗	✗	✗	✘
	NFC packet sniffing	✗	✗	✗	✗	✗	✗	✘
Apps & events	Broadcast intents logging	✗	✗	✗	✗	✗	✗	✔
	Running services	✓	✗	✓	✓	✗	✓	✔
	Running applications and activities	✓	✗	✓	✓	✓	✓	✔
	Foreground activity detection	✗	✗	✗	✗	✓	✗	✔
	In-app UI interactions and changes	✗	✓	✓	✗	✓	✗	✔
	App installs/uninstalls	✗	✓	✓	✓	✓	✓	✔
Geoloc.	Location services status	✓	✓	✗	✗	✓	✗	✔
	Coarse location	✓	✓	✓	✓	✓	✓	✔
	Precise location	✗	✓	✓	✓	✓	✗	✔
	Total logged features	15	10	16	18	15	17	**44**

Table 6.2: Feature comparison table – Part 2.

a monolithic design, with extensibility provided via Android Interface Definition Language (AIDL) [8]. as an optional feature. Secondly, *SystemSens* uses a fixed global polling interval, set at a frequency of two minutes, which is often not sufficient for fine grained data analysis (more on these issues in Section 6.1.3).

Another similar tool is **DroidWatch** [90], an enterprise monitoring system for Android mobile devices. *DroidWatch* focuses on data sources such as phone call logs, visited websites, text messages and the user location. On the other hand, it ignores system information and sensor readings. *DroidWatch* is also not natively extensible, and suffers from the same limitations as *SystemSens*: no fine-tuning or advanced customization of logging behavior and a monolithic design.

MobileSens [91] is another app for logging user behavior in Android. The main aim of this tool is to profile user actions in order to study their behavior. Thus, logging is geared toward user actions and how they affect the state of the device (e.g., when the screen turns on/off, when the user sends messages). Like *DroidWatch*, *MobileSens* is not extensible and no source code is provided.

Developed by Wagner et al. at the University of Cambridge, **DeviceAnalyzer** [202] is a comprehensive logging tool for Android. Among the Android tools we analyzed, *DeviceAnalyzer* is the one that logs the largest number of data sources. This tool is monolithic and not natively extensible, nor is it open-source, making it unsuitable for researchers that want to add their own custom logging plug-ins to the experiment. Moreover, *DeviceAnalyzer* collects all data and merges it into a global data set, controlled by its authors. Although the authors provide access to the data set (on request), this makes it impossible for a researcher to deploy a specific, customized experiment to a specific group of users.

PhoneLab [144] is a full Android OS distribution that integrates logging features into the original Android source code. The authors of PhoneLab review and accept third party change requests to the codebase and integrate them into the distribution, which is later deployed on a number of devices (their testbed) under their control. This is an interesting approach, as it allows to integrate logging at an OS level, bypassing API restrictions and overheads. However, we also feel that such approach has some shortcomings. Firstly, creating an experiment with PhoneLab requires non-trivial development knowledge, as the contributors are required to branch, modify, rebuild and merge the Android source code. On the other hand, DELTA is designed to be usable for people that are not necessarily skilled in programming. Secondly, similarly to *DeviceAnalyzer* [202], *PhoneLab* is managed entirely by the PhoneLab team, and contributors have no control over deployment of their changes and over the data collection. Indeed, the PhoneLab team has to evaluate and approve every change made to the codebase, with no guarantees on how long the experiment will be kept running on the testbed.

Finally, all gathered data is granted at discretion of the PhoneLab team. We believe our approach with DELTA is more flexible, as experiments can be distributed directly to participants and left running as long as the author requires. Moreover the gathered data using DELTA can be collected and accessed directly without intermediaries.

LiveLab [179] is a tool built for the iOS operating system. This tool allows logging of various sensor readings and context data, with support for uploading it to a remote server. Similarly to other tools we examined, *LiveLab* does not provide fine-grained tuning of polling intervals. In addition to this, since apps not approved by the manufacturer cannot be installed on iOS devices, it requires an unlocked ("jailbroken") copy of iOS. This requirement strongly limits the number of devices on which it can be deployed. For our implementation, we decided to target the Android operating system, to achieve maximum flexibility and guarantee a large potential user base. Contrarily to *LiveLab*, our tool is designed to require an unlocked ("rooted") device only for certain advanced logging features, not obtainable through the standard operating system API.

6.1.2 Additional Related Work

In this section, we briefly describe some additional tools and frameworks that perform logging operations on mobile devices. We did not include these solutions in our feature comparison, either because these solutions have a limited scope, or because they are not straightforward logging tools, but rather tools that log data to achieve a different primary goal.

One of the first works on device usage logging in the literature is **MyExperience**, targeted at the Windows Mobile platform and presented in [79]. It is a logging tool capable of collecting context data and performing actions in response to triggers configured by the experiment designer. **DroidTracer** [108] is a Linux kernel module that hooks into the Android system core at a low level. This tool aims to capture app interactions, logging data such as remote method calls between apps and Android API calls (e.g., access to disk or telephony services). In our tool, we followed the more traditional approach of leveraging (when possible) the Android API to collect our data. Implementing an approach similar to *MyExperience*, **Ohmage** [166] is an Android tool designed to present the user with interactive surveys and self-monitoring tasks depending on triggers such as time and location. This tool is also capable of automatically collecting sensor data from the device, such as accelerometer, GPS, WiFi, microphone audio recordings and cell towers logs.

A novel solution is presented by Brouwers et al. with **Pogo** [36], a middleware for Android mobile phone sensing. *Pogo* provides a JavaScript API, which exposes a limited subset of the Android API. This feature allows researchers to design experiments without being familiar with Java or the

Android development platform. While the idea of using a simpler programming language to design experiments is an interesting one, it also makes *Pogo* very limited in its logging capabilities. In fact, a lot of sensor and advanced contextual data can only be captured through calls to the native Android API.

Finally, **Dynamix** [39] is an open-source extensible context-sensing framework for Android. This tool is a plugin-based framework that collects sensor data, processes it to build a “context”, and then makes this context accessible to other applications via a dedicated API. While not strictly similar to our tool its basic principles are similar to the ones present in DELTA.

6.1.3 Limitations

We encountered three limitations common to most existing tools. These limitations greatly reduce the usefulness of such tools in a research context. In fact, we found that researchers often opted for a custom solution, developed specifically for their project, instead of relying on existing logging tools [51, 54, 85, 187].

The first problem is finding a tool that single-handedly satisfies all the data collection requirements for a particular research project. Most existing tools concentrate on one particular area (e.g., logging data from the device's sensors, collecting network packets). While a combination of different tools can be used to cover all the logging requirements of an experiment, this approach introduces some drawbacks:

- *Time consistency* - different tools will operate independently, without synchronization, and will timestamp data based on their internal timings. This can cause inconsistencies in the timestamps, which in turn can render collected data difficult to correlate precisely and thus useless.

- *Sampling rate consistency* - different tools will most likely poll the sensors/APIs of the device at different frequencies. This causes potentially undesirable differences in the granularity of collected data.

- *Data format consistency* - different tools will use different formats to save the logged data. This means that a researcher would have to perform additional (and possibly non-trivial) post-processing on the data. This post-processing can cause inconsistencies when trying to correlate data together and is generally inefficient.

- *Data collection* - if different tools are used, there is no centralized mechanism to collect gathered data and send it back to the researchers. This means that an additional ad-hoc implementation must be developed and deployed for this purpose.

The second common problem that existing tools have is limited support for fine tuning the sampling rates, in all those scenarios that require periodic polling of data (e.g., sensors reading). Existing tools tend to set a predetermined (and often relatively long) polling interval. While this limitation is usually implemented with the intent of reducing energy consumption, it can negatively impact on usefulness of collected data. In our approach, we want the experiment designers to be in control of the sampling rates of each logging operation, so that usefulness of collected data is maximized.

Finally, the tools we examined were not modular, instead using a monolithic design which did not provide an easy way to extend their logging capabilities. Even the ones that were extensible (e.g., SystemSens [68]) still had a monolithic "core" that aggregated the basic logging features provided by the tool, with extensibility being an added extra. This approach leads to a lack of customizability and often violates the "principle of least privilege", i.e., the app will often require more permissions than are actually needed to gather data for a certain experiment.

6.1.4 Comparison of DELTA with Other Solutions

Given the above premises and limitations of the existing solutions, we decided to design a tool that would adhere to the following principles:

- *Feature-richness* - our tool aims to provide a large variety of logging features out of the box, focusing on features that are not implemented (or not found together) in other logging tools.
- *Modularity* - we are aware that it would be impossible to provide support for every single logging need a researcher might have. Consequently, our tool is designed to be easy to extend through a dedicated plug-in system. This modular approach allows a developer to implement data collection features that are not available out of the box.
- *Fine-tuning* - our tool can be fine-tuned so that every single source of data can be polled at a configurable interval (or not polled at all). This approach allows the experiment designer to decide exactly which data sources she wants to monitor and at which frequency. This brings three advantages: (i) it maximizes usefulness of collected data; (ii) it reduces overhead by only tracking data that the experiment needs at the required intervals; (iii) the logger only requires the minimum privileges necessary to run each experiment. This is important to encourage user participation, especially if experiments are deployed on a larger scale.
- *Data distribution* - our tool can be configured to send the collected data back to a central server to make it accessible to the researchers that are running the experiment. This feature is important, as it lets

the researchers access the data as it is collected, allowing them to monitor the logging process while it progresses. Automatic uploading also removes the need to physically retrieve collected data from the devices (although this is still an option, which might be more suitable when dealing with large amounts of data).

Thus, we designed and implemented **DELTA - Data Extraction and Logging Tool for Android**, an extensible Android logging framework that aims to satisfy the aforementioned needs.

6.1.5 The "plugin problem" in Android

In this section, we integrate the background knowledge about Android operating system and permissions previously introduced in Section 5.2.1: In particular, we go more in-depth about the difficulties of developing an extensible application for Android. As stated in Section 6.1.4, we wanted our system to be modular, so that researchers could easily implement additional logging capabilities besides the ones we provide out of the box. This also means that we wanted our logging app (from now on referred to as the *logger*) to only include the bare minimum code to fulfill the needs of each experiment. In essence, this consists of using techniques that allow the logger to dynamically load additional code (i.e., a plugin) at runtime. A typical technique is using a class loader, like the one provided by the Java Virtual Machine [115]. However, even though Android programming is done in Java, and the class loader is still usable to load code dynamically from an external file, the Android permission system can severely hinder its functionality. In practice, even if the logger can technically load additional code at runtime, this code will still run in the same process as the logger, and thus with its permission set. Consequently, if the *logger* does not hold the required permissions to run the dynamically loaded code, it will not be able to execute it correctly. Note that this problem is not mitigated by the new on-demand permission system introduced in Android 6.0 [10]: even if apps can now request permissions at runtime only when needed, these still need to be declared in the manifest, and thus known at compilation time. This means that creating a plugin system for applications such as DELTA is not straightforward. Our tool is designed to log a wide spectrum of data, and access to this data is often mediated by a specific permission. There are two ways to overcome this problem at runtime, but they both come with specific drawbacks:

- **Preemptive permission over-provisioning** consists in having the logger greedily declare the use of all existing permissions and be able to execute any dynamically-loaded code that is permission-protected. The advantage is that this way, the logger will be able to load any plugin from an external file. However, this solution violates the principle of least privilege, which as we have seen can be a security risk. Moreover, it is not an effective solution: it needs to be constantly updated to include new permissions and it does not cover other Manifest extensions like services or GUI components that a plugin might require.

- **Full decoupling** consists of having the plugins required for an experiment installed as distinct, separate apps on the device, alongside the logger. This way, the logger can be a minimal skeleton that delegates logging operations to the plugins. Then, the logger will communicate with them using one of Android's built-in inter-process communication mechanisms. This solution seems more reasonable than the former one, but it has flaws of its own. The first one is that plugins have to be installed on the device as separate apps, which can be extremely annoying to the user. Secondly, inter-process communication in Android relies on data serialization and message routing, which result in overhead and consequently in a higher impact on system performance and battery life.

6.2 Our proposal: DELTA

In this section, we present the system architecture of DELTA, and explain how our implementation deals with the plugin problem. To better understand the specific needs and concerns of researchers and users, we set up two focus groups to help us define DELTA's architecture and user interface. The first group included researchers that had worked on projects that used smartphone data as input, as well as Android developers. This group focused on system architecture, and helped us develop a design that would be easy to use for researchers and easy to extend for developers. The second group focused on usability, and included people not necessarily familiar with Android, such as researchers, students and other department staff. Input from this group helped us with suggestions about the design and functionality of the smartphone-side user interface of DELTA.

From the input received from these two focus groups, we set the following goals that the DELTA system design should fulfill:

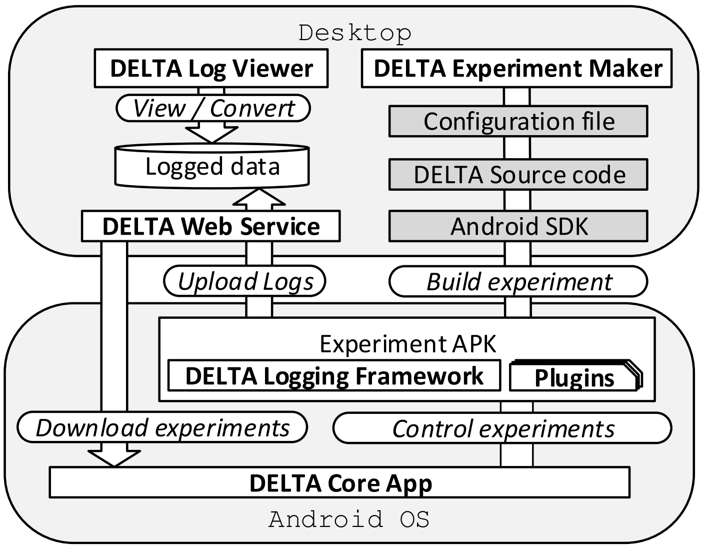
- *Minimal knowledge required by experiment designers* - we believe our tool should be accessible to everybody, even if they are not knowledgeable about the Android platform. DELTA provides dedicated graphical tools and an automated build system, which make creating new experiments straightforward.

- *Ease of extension for plugin developers* - DELTA uses a fully modular plugin-based architecture, allowing developers to easily extend its logging capabilities.

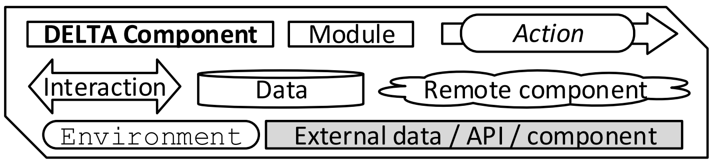
- *Simplicity and security for users* - DELTA was designed to be as easy-to-use and non-invasive of user privacy as possible, thus encouraging voluntary user participation.

6.2.1 System model

In this section, we illustrate the architecture of DELTA and show how each of the above mentioned goals (Section 6.2) is achieved. Figure 6.1a shows a high-level view of the DELTA system architecture, highlighting the various components. Figure 6.1b explains the symbology of all diagrams in this section.



(a) Architecture of DELTA, highlighting its components and their interactions.



(b) Legend for all the system architecture diagrams.

Figure 6.1: DELTA's architecture and legend.

DELTA allows users to configure, create, deploy and manage data logging experiments, in the form of standalone Android apps. From now on we will refer to these apps as *DELTA Experiments*, or simply *experiments*. Each DELTA Experiment is a specialized instance of DELTA Logging Framework component, plus a variable set of additional libraries, the DELTA plugins. Each plugin library contains one or more specialized modules that implement the actual data gathering operations. The DELTA Logging Framework is a generic framework that is able to instantiate and use the aforementioned plugins to schedule and perform the data logging and storage operations required by the author of the experiment.

Researchers who want to create new experiments can do so with the DELTA Experiment Maker tool. DELTA Experiment Maker is a desktop application that presents the users with a dedicated graphical interface, through which they are able to configure and build new DELTA Experiments. Depending on the chosen configuration, DELTA Experiment Maker is able to build a custom DELTA Experiment that will include the minimal set of plugins to log the required data. On the other hand, users that want to participate in experiments can use DELTA Core App. DELTA Core App is an Android app that is able to install, run, stop and remove DELTA Experiments on the user device. Optionally, DELTA Core App also allows users to download new experiments from the web or send the logged data back to the researchers. These features are made possible by DELTA Web Service, a simple self hosting web server that researchers can run to allow remote collection of the logged data and remote deployment of new experiments. Once the logged data has been collected, DELTA Log Viewer desktop application can help researchers to merge it and convert it into different formats for better analysis.

6.2.2 Logging Framework

The DELTA Logging Framework is the core engine that runs DELTA's data gathering and storage operations. It is an Android app that implements a background service [11], the *Logging Service*, which is in charge of running the experiment. When a researcher creates a new experiment a copy of the DELTA Logging Framework, together with a configuration file and all the necessary plugins, is compiled into a single `APK` file. This file is then deployed to the user's device, meaning the user only has to install a single package to run an experiment. This architecture allows DELTA to reap the benefits of a plugin system, while at the same time being as noninvasive as possible. In particular, only the strictly required plugins are included with the Logging Framework when building an experiment. Consequently, the generated `APK` will only require the bare minimum permissions. This minimizes overhead and complies with the principle of least privileges. The anatomy and sub-components of a packaged DELTA Experiment are shown in Figure 6.2a.

The *Logging Service* component is in charge of governing the experiment's life cycle. The framework itself does not implement any logging operations directly, delegating them to the various plugins instead. The service autonomously maintains wakelocks and manages the timers that trigger periodic logging operations. Its *Storage Manager* module implements facilities for timestamping, formatting and storing the logged data. This relieves plugin authors from having to manage such implementation details, so they only have to implement the routines that actually perform the logging of data.

Since the framework continuously runs in the background (if its experiment is running) it was important to minimize its impact on system resources, in order to save battery and not slow down the device. We designed the DELTA Logging Framework so it minimizes the time it keeps the device CPU awake. In particular, depending on the configuration of the plugins, we can have three cases:

- *Plugins that do not require periodic polling.* These are plugins that log data reactively. In this case, no wakelocks are used.
- *Plugins that need to be polled periodically, but at long intervals.* In case of plugins that do require polling, but are set to a polling frequency ≥ 10 seconds (i.e., the average time after which the CPU goes to sleep in Android devices), we use the energy-efficient alarms subsystem [9]. This facility allows us to schedule a periodic CPU wake-up without keeping the device constantly awake.
- *Plugins that need to be polled at fast intervals.* In these cases, it is necessary to resort to wakelocks to keep the device awake, as the Alarms subsystem does not guarantee sufficient precision to schedule low-latency events.

Even when using wakelocks, we aim to minimize DELTA Logging Framework's overhead, by limiting the number of timers and threads that perform polling. In particular, the *Polling Scheduler* module groups all plugins configured with an harmonic polling frequency into shared polling cycles, triggered by a single timer. This minimizes context switches and computational overhead. Nonetheless, the creator of an experiment can also decide to forgo wakelocks altogether, and only let the experiment run when the device is awake for other reasons. This is known as "piggyback sensing" [110] and is useful, for example, for experiments that only need to be running when the device screen is on.

Finally, the *Storage Manager* module implements an internal cache, invisible to the plugins, in order to minimize I/O operations. On disk, data is stored using lossless `ZIP` compression, to reduce occupation of the user's storage. Once logged and stored, data can be either uploaded remotely, via the *Data Upload* module, or dumped to a public portion of the user storage for easy manual extraction via the *Data Dump* module. For more information on data storage and format, see Section 6.2.6.

We also want to notice that our plugin architecture is designed in a way that makes it easy for developers to modify and extend an existing plugin. All of our plugin's source code is freely available and our interfaces provide entry points at various moments in an experiment life cycle (e.g., when the experiment starts, when the plugin posts data back to storage). This allows developers to quickly add functionality to an existing plugin with minimal effort, e.g: adding additional checks at the start of an experiment, changing the format in which data is stored or performing filtering/post processing before data is sent to storage

6.2.3 Experiment Maker

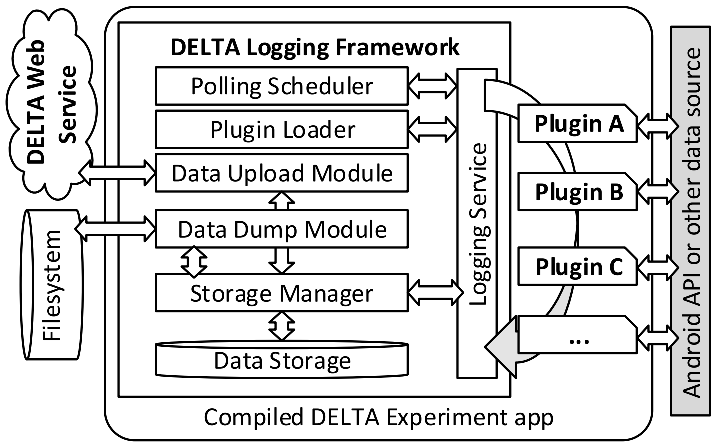
The main components of DELTA Experiment Maker, and their interactions are shown in Figure 6.2b. DELTA Experiment Maker is a graphical cross-platform desktop tool, written in Java, that allows a researcher to easily create and configure a new DELTA Experiment. The Experiment Maker is designed so that the user can create a customized experiment with minimal instructions, without any knowledge of Android or Java programming.

The *Plugin Parser* module leverages the `Javaparser` library to parse the DELTA source tree and detect all existing plugins. Using this information, the user interface presents the researcher with an automatically-populated list of available plugins. The researcher can then choose which plugins to include in the experiment, and is able to fine-tune the logging frequency of each one, plus any other advanced logging option. The Experiment Maker is also able to sign experiments after building, using a user-provided certificate. This protects experiments against tampering.

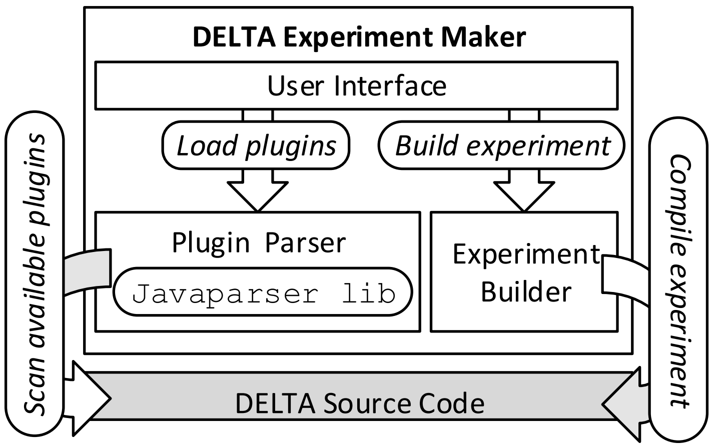
Once the configuration has been decided, the *Experiment Builder* module will invoke a series of custom build scripts to compile the experiment in a self-contained Android `APK` package. This package includes the main logging routines (the DELTA Logging Framework) and all (and only) the selected plugins. The generated experiment packages are lightweight (usually around 1MB) and thus can be easily distributed to participants without concern for download times.

6.2.4 Core App

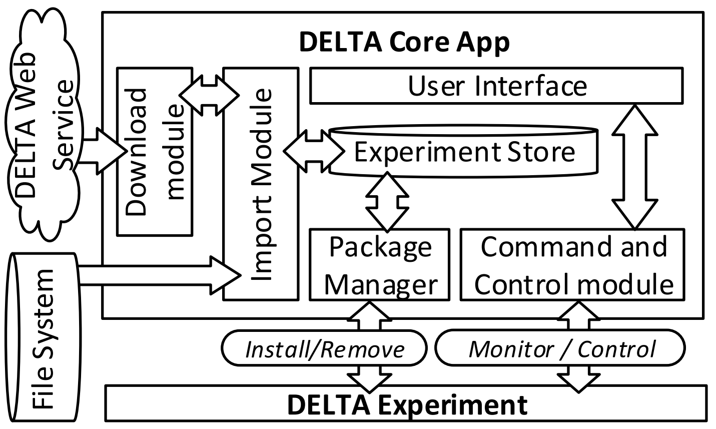
The **DELTA Core App** is an Android app that manages DELTA Experiments, aimed at end users that want to participate in experiments. Its architecture and main modules are shown in Figure 6.2c.



(a) Components of a DELTA experiment package.

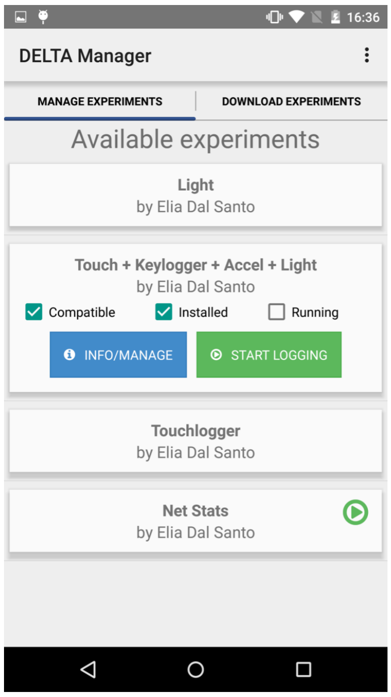


(b) Architecture of the DELTA Experiment Maker and its main interactions.



(c) Architecture of the DELTA Core App and its modules.

Figure 6.2: DELTA's components.


16:36
DELTA Manager
MANAGE EXPERIMENTS | DOWNLOAD EXPERIMENTS

Available experiments

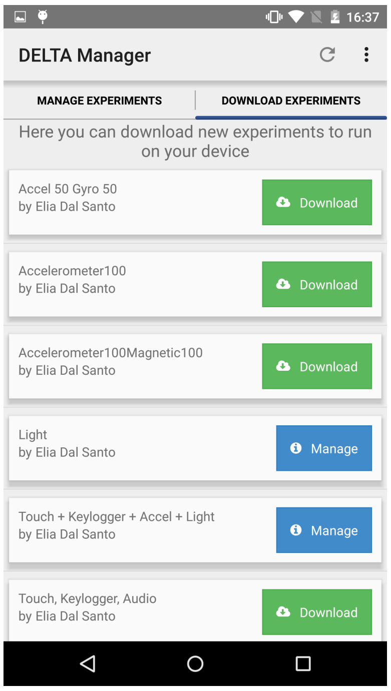
Light
by Elia Dal Santo

Touch + Keylogger + Accel + Light
by Elia Dal Santo
✓ Compatible ✓ Installed ☐ Running
INFO/MANAGE START LOGGING

Touchlogger
by Elia Dal Santo

Net Stats
by Elia Dal Santo

(a) Browsing experiments.


16:37
DELTA Manager
MANAGE EXPERIMENTS | DOWNLOAD EXPERIMENTS

Here you can download new experiments to run on your device

Accel 50 Gyro 50
by Elia Dal Santo — Download

Accelerometer100
by Elia Dal Santo — Download

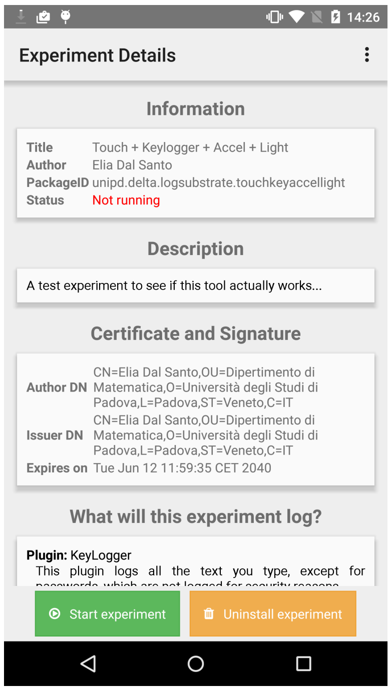
Accelerometer100Magnetic100
by Elia Dal Santo — Download

Light
by Elia Dal Santo — Manage

Touch + Keylogger + Accel + Light
by Elia Dal Santo — Manage

Touch, Keylogger, Audio
by Elia Dal Santo — Download

(b) Downloading experiments


14:26
Experiment Details

Information

Title	Touch + Keylogger + Accel + Light
Author	Elia Dal Santo
PackageID	unipd.delta.logsubstrate.touchkeyaccellight
Status	Not running

Description

A test experiment to see if this tool actually works...

Certificate and Signature

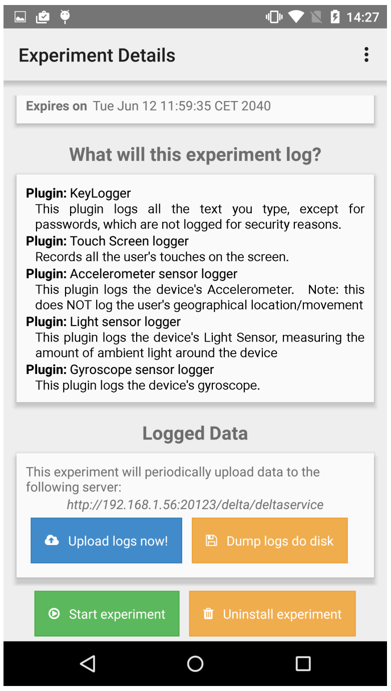
Author DN	CN=Elia Dal Santo,OU=Dipertimento di Matematica,O=Università degli Studi di Padova,L=Padova,ST=Veneto,C=IT
Issuer DN	CN=Elia Dal Santo,OU=Dipertimento di Matematica,O=Università degli Studi di Padova,L=Padova,ST=Veneto,C=IT
Expires on	Tue Jun 12 11:59:35 CET 2040

What will this experiment log?

Plugin: KeyLogger
This plugin logs all the text you type, except for

Start experiment | Uninstall experiment

(c) Managing an experiment (1/2).


14:27
Experiment Details

Expires on Tue Jun 12 11:59:35 CET 2040

What will this experiment log?

Plugin: KeyLogger
This plugin logs all the text you type, except for passwords, which are not logged for security reasons.
Plugin: Touch Screen logger
Records all the user's touches on the screen.
Plugin: Accelerometer sensor logger
This plugin logs the device's Accelerometer. Note: this does NOT log the user's geographical location/movement
Plugin: Light sensor logger
This plugin logs the device's Light Sensor, measuring the amount of ambient light around the device
Plugin: Gyroscope sensor logger
This plugin logs the device's gyroscope.

Logged Data

This experiment will periodically upload data to the following server:
http://192.168.1.56:20123/delta/deltaservice

Upload logs now! | Dump logs do disk

Start experiment | Uninstall experiment

(d) Managing an experiment (2/2).

Figure 6.3: Screenshots depicting the DELTA Core App UI and functionality.

The DELTA Core App provides an intuitive graphical interface, from which users can manage all the experiments available on their device. In particular, the app allows them to:

- Import new experiments, either directly from a file or by acquiring them through the DELTA Web Service (see Section 6.2.5 for details).
- Browse, install, remove and view details about stored experiments through the *Experiment Browser* and *Package Manager* modules. A detailed report is shown for each experiment, including information such as what the experiment logs, who is the author and the certificate used to sign it
- Monitor the status of an experiment and send commands to it, using the *Command and Control* module. These commands include the ability to start or stop the experiment at any time, plus additional commands to extract logged data or upload it to a remote server

The DELTA Core App maintains a list of running experiments, and is able to automatically restart them after the device is rebooted. This way, long-running experiments do not have to be manually restarted by the user in case they switch their phone off. The app is also able to automatically schedule periodic uploads of the logged data to a remote server, if the author of the experiment enabled this functionality when configuring the experiment. To avoid depleting the user's mobile data plan, this function only runs if a WiFi connection is available.

Figure 6.3 shows the DELTA Core App's user interface. Figure 6.3a shows a screenshot of the main activity of our app, which shows the list of installed experiments. In the main activity, the user can see at a glance what experiments she has available and whether they are compatible, installed and running. For the sake of usability, we included two quick access buttons: (i) *INFO/MANAGE* to view the details of an experiment and manage it (see figures 6.3c and 6.3d); and (ii) *START/STOP LOGGING* to start/stop an experiment. Moreover, a green icon (in "Net Stats" experiment in Figure 6.3a) tells the user whether the experiment is running. Figure 6.3b shows the list of experiments available for download from a DELTA Web Service (see Section 6.2.5) and allows a user to download them. The screenshots in figures 6.3c and 6.3d show the experiment manager activity. In this activity, the user can manage and read all the information about an experiment. In particular, information about the author is shown (including information on the digital certificate used to sign the experiment), together with a description of what the experiment does and a detailed list of what the experiment will log. This activity also allows users to easily manage every aspect of an experiment in a single place (i.e., start/stop, download-/remove and install/uninstall it) and its logs (i.e., request a data dump to the local storage or trigger a data upload to the DELTA Web Service).

6.2.5 Web Service

The **DELTA Web Service** is a standalone self hosting web server, written in Java, that adds two key features to the system:

- It allows users to **download DELTA Experiments** directly from within the DELTA Core App. This greatly facilitates the distribution of experiments, as researchers can deploy them without having to manually distribute the required files to all users.
- It allows experiments to **send the logged data back to the researchers**. This means that researchers can collect the logged data automatically, without needing access to the user's device to perform a data dump. This also makes it possible for researchers to start analyzing incoming data while the experiment is still running, thus allowing them to speed up their analysis. It also has the added bonus of not clogging the user device with old data, since segments that are uploaded to the server are deleted from the local device cache.

DELTA Web Service is provided alongside the main components, so any experiment author can run a copy of it independently. This ensures that the experiment author has total control over the collected data, which is not sent to a third-party server.
Note that using the DELTA Web Service is entirely optional. Experiment creators can still distribute experiment packages through any other mean (e.g., email, USB side-loading), the DELTA Core App is able to import them directly from the file system. Similarly, logged data does not have to be uploaded to the web service: it can simply be dumped to the device's public storage to then be extracted manually.

6.2.6 Data format and the Log Viewer

One of the advantages of having a multi-purpose logging tool like DELTA is that logged data has a consistent format, independent from the data source. In our implementation, we use `JSON` as our format for storing data. `JSON` is lightweight but at the same time, unlike the popular `CSV` format, it supports nested objects, so it can be used to represent complex data.

For added flexibility, our data-reporting interface also supports logging of raw binary data. This comes in handy, as some plugins may log data that is not suited to be represented efficiently through strings (e.g., the audio recording plugin).

To avoid potential data corruption or loss, the DELTA Logging Framework stores data in separate chunks, which are then uploaded to DELTA Web Service or otherwise retrieved by the interested party. To store data on disk, we use the `DEFLATE` algorithm. This gives good compression for textual data, without being too heavy on the device's CPU. This is important, since complex computations greatly affect the performance and battery life of mobile devices. Our tests have shown that the output from most of our plugins achieves an extremely high compression ratio, with compressed data typically ranging from 2% to 6% of the size of the original. All data is timestamped in milliseconds since Linux Epoch[4].

DELTA Log Viewer is a utility that can preview, merge and convert the logs collected by a DELTA Experiment. In particular, the user can choose to export just a specific data chunk, all the data logged from a specific device or all the data collected from all devices involved in the experiment. The DELTA Log Viewer can also convert data, if requested, from its native JSON format to a more user-readable CSV file format. DELTA Log Viewer can also be used to preview the collected logs, and to merge together any binary (non-textual) data collected during the logging process.

6.2.7 Plugins and extensibility model

A **DELTA Plugin** is a specialized Java class that logs data on behalf of the DELTA Logging Framework. DELTA Plugins are contained in standard Android library packages (`AAR`), where each library package can contain one or more plugins. Contrary to what most other logging tools do, we designed DELTA to be completely modular, meaning that all the logging features we implemented are implemented as standard, optional DELTA plugins. No logging functionality is hardcoded in the DELTA Logging Framework which is, in fact, oblivious of the plugins. In our source code, we put plugins that require the same set of permissions together in the same library packages. This way we comply with the principle of the least privilege, as the final `APK` will only contain the strictly necessary libraries and thus will only require the minimum set of permissions to run the experiment.

From the point of view of a plugin developer, creating new plugins is a straightforward process. The implementation footprint consists of only a couple of standard interfaces and a Java annotation containing metadata (e.g., the author of the plugin, description of what it does, developer notes). This data is shown to experiment creators during the configuration phase. We also employ the Java annotation system to allow developers to easily define advanced options that experiment authors can modify at configuration time, so that plugins can be flexible in how they log data.

There are two distinct kinds of DELTA plugins that can be created, depending on how the data is gathered:

- **Event plugins** log data reactively, i.e., they do not need to be polled periodically, and are useful to subscribe to system events or other data sources that can actively notify a plugin of content changes.

[4]00:00:00 UTC, 1 January 1970

- **Polling plugins** are polled periodically at a certain frequency, and are typically used to log data at a fixed rate. Examples include logging sensor readings or periodically collecting statistics about the system or the apps running on it.

While the DELTA codebase already includes several plugins, as discussed, we also implemented support for using plugins as standard precompiled Android Library `AAR` packages. This way, the DELTA Experiment Maker can load the plugins even if they are not merged into the official DELTA source tree. Plugin developers can thus easily share new plugins with experiment creators, without necessarily having to share the source code. Users of the DELTA Experiment Maker can add precompile plugins to their project by simply copying them to a specific folder, without any additional configuration needed.

6.3 Summary

In this chapter, we presented DELTA - Data Extraction and Logging Tool for Android, our implementation of a multi-purpose logging tool for Android. We started by comparing similar pre-existing tools, highlighting their target audience, features and common shortcomings. Then, we showed how DELTA improves on existing solutions in terms of flexibility, customization, extensibility and logging scope. Of the solutions we examined, our solution is the only one that is built from the ground-up to be fully modular. DELTA is also the only one to achieve this without either violating the principle of least privilege or relying on inter-process communication. DELTA is also, by far, the most complete of the examined tools, logging more than forty different data sources.

We believe that DELTA's feature-richness and simple extensibility model can make it a precious tool for researchers. Writing a custom logging tool for an experiment is often a non-trivial endeavor, requiring time and knowledge about Android development. DELTA's feature-richness and high level of customization is often enough to create an experiment suited to a lot of data logging needs. When the basic set of plugins does not suffice, DELTA's modular design makes extension easy, abstracting away most of the complexities involved in developing a full-fledged custom logging tool.

Chapter 7

Robust Sandbox Against Malware Analysis Evasion

Among mobile operating systems, Android is the leading platform, with a market share of 86% in 2016 [81], and it is growing as a new target for malware and attacks against users' privacy. The Android operating system uses a modified version of the Linux kernel, where each app runs individually in a secured environment, which isolates its data and code execution from other apps. The operating system mediates apps' access requests to sensitive user data and input devices (i.e., enforcing a Mandatory Access Control). Without any permission, an app can only access few system resources (e.g., sensors, device model and manufacturer) [17].

Although malware could escalate privileges by exploiting vulnerabilities in the operating system, new threats arise also from apps that run unprivileged. Malware for Android often harms users by abusing the permissions granted to it. For example, malware can cause financial loss by leveraging features such as telephony, SMS and MMS, while with access to camera, microphone, and GPS it can turn a smartphone into an advanced covert listening device. Moreover, the leak of confidential data, such as photos, emails and contacts, threatens users privacy as never before [205]. Attackers usually spread malware infections by repackaging an app to contain malicious code, and by uploading it to Google Play (i.e., the official marketplace) or alternative marketplaces [215]. A possible approach to reveal malicious Android apps consists of analyzing them one at a time. However, this may be fighting a losing battle: Google Play counts more than 2.2 million apps today [189]. Thus, in recent years, researchers attention moved to the study of batch (i.e., non-interactive) analysis systems [121, 193, 197].

Malware analysts can examine suspicious apps through static analysis and dynamic analysis. On one hand, static analysis consists of inspecting the resources in the packaged app (e.g., manifest, bytecode) without executing it.

Unfortunately, an adversary can hinder static analysis by using techniques such as obfuscation, encryption, and by updating code at runtime. On the other hand, dynamic analysis consists in monitoring the execution of an app in a test system. During such analysis, the sample (i.e., an app submitted for the analysis) runs in a sandbox. A sandbox is an isolated environment where malware analysts can execute and examine untrusted apps, without risking harm to the host system.

Academic and enterprise researchers independently developed many malware analysis systems for Android. For example, Google introduced Bouncer, a dynamic analysis system that automatically scans apps uploaded to Google Play [123]. Such analysis systems run long queues of batch analyses in parallel, and typically do not rely on real devices but on Android emulators. Unfortunately, emulators present some hardware and software differences (i.e., artifacts) with respect to real devices, which can also be recognized at runtime by apps: By detecting these artifacts, an app can easily recognize whether it is running or not on a real device. A malicious app can exploit emulator detection to evade dynamic analysis and show a benign behavior, instead of the malicious payload. Relying on such mechanism, malware authors might spread a new generation of malicious apps, which they would be hardly detectable with current dynamic analysis systems. While researchers keep improving dynamic analysis techniques, they are overlooking the accuracy of virtualization. In current malware analysis services for Android, the coarseness of the underlying emulator hinders researchers efforts.

The contribution of this work is a step towards the development of a stealthier malware analysis sandbox for Android, which reproduces as much as possible the characteristics of real devices. Our goal is to show malware the characteristics of an execution environment that appear to be real but are not actually there[1]. In this chapter, we make the following contributions:

- We define six requirements to design a sandbox that can cope with current evasion attacks, and is easy to evolve in response to novel detection techniques.
- We propose Mirage, an architecture that fulfills all these requirements. Researchers can use Mirage to implement more effective malware analysis sandboxes for Android.
- We describe our proof of concept implementation of Mirage.
- We evaluate the effectiveness and the modularity of Mirage by tackling a specific and representative case: address sandbox detection techniques that exploit sensors capabilities and events.

[1]Like a mirage in a sand(box) desert, and this motivates the name of our proposed solution.

- We show that Mirage, with our sensors module, can cope with most evasion attacks based on sensors that affect current dynamic analysis systems.

Organization: The rest of the chapter is organized as follows. We start by presenting related work in Section 7.1. In Section 7.2, we define six requirements that, based on the characteristics of evasion attacks, we believe are essential to develop a malware analysis sandbox for Android. In Section 7.3, we present the components of Mirage. As a representative case study, in Section 7.4, we describe our proof of concept implementation of Mirage which addresses evasion attacks based on sensors. In Section 7.5, we compare our system with state of the art malware analysis services, with respect to sensors-based detection techniques. and we discuss its effectiveness in Section 7.6. Finally, Section 7.7 concludes the chapter.

7.1 Related Work

Security researchers put a lot of effort in detecting PC virtualization [154, 165]. However, in the era of cloud computing, a desktop or server operating system running inside a virtual machine is no longer a sign that dynamic analysis is taking place. Regarding mobile devices, nowadays malware analysts mainly rely on emulators, so malware can use emulator detection to evade dynamic analysis. Therefore, we strongly believe that evasion attacks on mobile emulators will be a hot topic for researchers in the years to come.

In what follows, we report the work related to the domain of sandbox detection. In [199], Vidas et al. described four classes of techniques to evade dynamic analysis systems for Android. The authors categorize such techniques with respect to differences in behavior (e.g., Android API artifacts, emulated networking), in CPU and graphical performances, in hardware and software components (e.g., CPU bugs, sensors, emulated battery), and in system design. Similarly, Petsas et al. in [159] presented evasion attacks against Android virtual devices. The authors divide the attacks in three categories based on static properties, dynamic sensors information and Android emulator (modified QEMU emulator) artifacts. In the evaluation, the authors successfully evaded several dynamic analysis software and online services. Jing et al. in [100] introduced Morpheus, a software that automatically extracts and rank heuristics to detect Android emulators. Morpheus retrieves artifacts from real and virtual devices, and it compares the retrieved artifacts to generate heuristics. The results that the authors achieved during their experiments are surprising: Morpheus derived 10,632 heuristics from three out of thirty-three sources of artifacts. Maier et al. in [126] presented a tool for Android called Sand-Finger, which is able to collect information from sandboxes that malware can use to evade dynamic analysis. The authors tested ten sandboxes and antivirus services and all of them exposed some artifacts to Sand-Finger. As a result, the authors developed a split-personality version of the AndroRAT malware relying on their findings.

Some industry presentations examined the sandbox detection problem as well. Strazzere, in [191], proposed detection techniques based on system properties, QEMU pipes and content in the device, which he embedded in an app for Android. Oberheide et al., in [151], and Percoco et al., in [158], showed that Google Bouncer is not resilient against evasion attacks, and an attacker can bypass it to distribute malware via Google Play marketplace. In addition to fingerprinting Bouncer, the former managed to launch a remote connect-back shell in its infrastructure.

Researchers proposed many dynamic malware analysis systems for Android that rely on an emulator. Few examples of such systems are CopperDroid [193], CuckooDroid [44], DroidBox [111] and DroidScope [209]. Other systems such as AASandbox [32], Andrubis [121], SandDroid [173] and TraceDroid [197] perform dynamic analysis on an emulator as well, but they also use static analysis to improve their performances. In addition to performing both static and dynamic analysis, authors in [200] proposed to analyze samples using an emulator that they enhanced to tackle some evasion attacks. Although authors in [200] focus on how to perform malware analysis, it presents some interesting ideas against sandbox detection techniques. An interesting idea is to use a mixed infrastructure composed of real and virtual devices. Mutti et al. in [143] presented BareDroid, a malware analysis system based on real devices, instead of emulators, which consequently is more robust to evasion attacks. The authors estimated that a BareDroid infrastructure would cost almost two times the cost of a system based on emulators with the same capabilities. However, a virtual infrastructure is more elastic when compared to a cluster composed only of physical devices, which may suffer from under or over-provisioning. The authors estimated that a BareDroid infrastructure would cost less than twice the investment needed to deploy a system based on emulators with the same capabilities. This thanks to their fast approach to restore devices to a clean snapshot. However, cloud providers offer virtual machines on demand. Hence, a virtual infrastructure is more elastic when compared to a cluster composed only of smartphones. Indeed, a cluster of smartphones may suffer from under or over-provisioning.

To the best of our knowledge, the work by Gajrani et al. [80] is the most similar to our proposal. After giving a general overview on emulator detection methods, the authors present DroidAnalyst, a dynamic analysis system that is resilient against some of them. Their system can hinder evasion attacks based on device properties, network, sensors, files, API methods and software components. We share a common goal with the authors of [80]: the development of a malware sandbox for Android resilient against evasion attacks. However, we identified in [80] the following limitations (that we instead overcome with our proposal):

- Their analysis about artifacts in Android sensors API is not exhaustive. For example, they do not take some of our findings (see Section 7.4.2) into consideration.
- They propose a solution that consists of a set of patches to their analysis system based on QEMU. Therefore, it is not a general architecture like our proposal.
- DroidAnalyst uses an approach based on emulator binary and system image refinement, which does not allow the same emulator to impersonate two different real devices, unless they are modified again and restarted. Conversely, the requirements of Mirage (presented in Section 7.2) discourage any modification to the emulator, since the sandbox would be less flexible and hard to maintain.

To evaluate the effectiveness of our sandbox detection heuristics based on sensors, we tried to submit to DroidAnalyst our sample, i.e., the *SandboxStorm app* (see Section 7.5). Unfortunately, the DroidAnalyst dynamic analysis subsystem was under maintenance, and is still not available at the time of writing.

7.2 Sandbox Requirements

After studying state of the art sandbox detection techniques [100,126,159,199], we define six key requirements that we believe are essential to develop a malware analysis sandbox for Android. Our goal is to derive the design of an architecture from the requirements, which can consist of one or more parts (i.e., components). We formulate the first three requirements on the basis of desired features to cope with the evasion attacks described in the aforementioned work (see Section 7.1). Moreover, we formulate three additional requirements taking into account that the sandbox should be flexible. The requirements are:

- **Stealthiness of sandbox components**: The components of the sandbox shall be unnoticeable by malware. Otherwise, an adversary could recognize a component of the sandbox, and evade dynamic analysis. This may seem a trivial requirement, but it serves as a cornerstone for our work. Nowadays, virtualized environments are not realistic and easily detectable [128,154,159,165]. Unfortunately, adding new countermeasures in such environments to achieve stealthiness produces new artifacts (e.g., processes and files). Such artifacts allow malware authors to fingerprint the whole system, causing the ineffectiveness of the countermeasures in place to make the virtualized environment stealthy. If the sandbox is not fully undetectable, it should be able to hide its imperfections, hiding them to the samples.

- **Consistency of bogus data**: The sandbox shall provide realistic and consistent information to the sample throughout the analysis. Otherwise, an adversary could detect the sandbox by exploiting the discrepancies in information that comes from different sources. To hide the artifacts in the emulator, the sandbox must produce a large amount of fake data. In this case, random generation is not an option, since it is prone to introduce discrepancies in data. For example, telephone numbers in contacts shall be composed of a country calling code plus a fixed number of digits [199]. A possible solution could be to use data collected from real mobile devices. In addition to that, the modules in the sandbox that inject such data must coordinate with each other to mimic a realistic environment.

- **Monitor known evasion attempts**: The sandbox should be able to notice whenever a sample is likely exploiting known detection techniques. Even if some artifacts are obvious but not fixable with nowadays technologies, it is worth to log all the suspicious attempts and act in an alternative way. However, when an app looks for artifacts, it does not strictly means that that app is trying to evade the analysis.

- **Modularity of sandbox components**: The components of the sandbox shall be modular with respect to detection techniques that malware exploits. We believe this is a key requirement, since researchers keep reporting cutting-edge [100, 126, 159, 199] evasion attacks every year. Researchers shall have the opportunity to develop, customize and publish new parts in a modular fashion, to keep up with the state of the art. A system designed to be open to new contributions makes it also improvable, in order to cope with emerging threats. Furthermore, since the Android operating system and its SDK change rapidly, sometimes new features break the compatibility with old ones that were available in previous versions. Hence, it is necessary to divide the components internally into modules. This allows to redesign and implement again just the modules that the changes affect.

- **No modifications to the Android source code**: The sandbox should not require any change of the Android source code. Although it would possible to alter APIs by modifying the operating system, compiling Android requires a significant amount of computational resources. In fact, a single build of an Android version newer than Froyo (2.2.x) requires more than two hours on a 64-bit consumer PC, plus at least 250GB (including 100GB for a checkout) of free disk space [12].

Even with the necessary resources and a semi-automated workflow, maintaining several versions simultaneously would be an overwhelming task.

- **No modifications to the Android emulator**: The sandbox should not require significant modifications to the emulator. Researchers are using different hypervisors and virtual machines for dynamic malware analysis, therefore we cannot focus on a specific technology. For example, systems like CopperDroid [193] use virtual machine introspection to reconstruct the behaviors of malware, hence such systems are potentially adaptable to any emulator. Forcing the scientific community to port the existing software to meet a modified emulator would likely lead to failure in the adoption.

7.3 Mirage: Our System Architecture

In this section we present Mirage, our architecture for a malware analysis sandbox robust against evasion attacks. One of the key feature of Mirage is that it is composed of processes that execute inside the operating system, and software that runs outside the emulator. This feature allows Mirage to be not tied to a specific analysis system. Although in this chapter we refer to a single instance of the sandbox, we argue that it is possible to replicate an instance multiple times to scale out and serve more requests concurrently. In addition to that, we could deploy a small bare metal infrastructure (i.e., composed of real devices), like BareDroid [143], in conjunction with our virtualized environment. The analysis through a real device is useful whenever we suspect that a sample might use sophisticated emulator detection techniques. We assume that most of the requests are addressed in our sandbox, and we consider the forwarding of the samples to a real device as a last chance. It is worthy of note that, although we focused on the Android operating system, Mirage architecture is not operating system specific. Indeed, Mirage could be implemented in other mobile platforms as well.

In Figure 7.1, we illustrate the four main components of Mirage which are the *Methods Hooking Layer* (Section 7.3.1), the *Events Player* (Section 7.3.2), the *Coordinator and Logger* (Section 7.3.3), and the *Data Collection App* (Section 7.3.4).

7.3.1 Methods Hooking Layer

The first component of Mirage architecture is the *Methods Hooking Layer*. This component executes as a process in the Android operating system. The main function of *Methods Hooking Layer* is to intercept calls to methods of Android API and manipulate their return value. Such manipulation occurs just whenever the original returned value may reveal

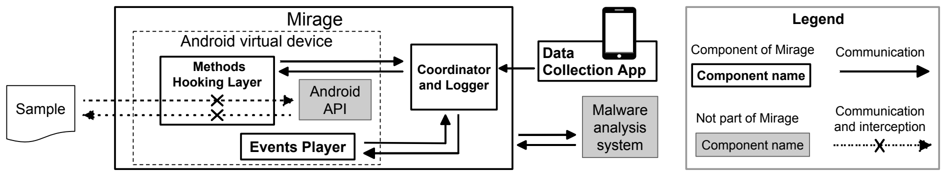


Figure 7.1: Mirage architecture, highlighting its components and their interactions.

the presence of the underlying emulator. Relying on this component, we can address the majority of behavioral differences. As an example, we can return a well-formed telephone number when a sample asks for `TelephonyManager.getLine1Number()`, instead of the default one (which in an emulator always begin with 155552155, followed by two random digits). Since it is possible to predict which artifacts the *Methods Hooking Layer* introduces, we can use such component to hide them as well. Moreover, hooked methods should perform minimal computation to reduce the risk of detection via computational timing attacks.

The *Methods Hooking Layer* do not require a particular Android emulator or modifications to the emulator. Therefore, this Mirage component is simple to develop and to maintain across different Android versions and emulators. The code of the *Methods Hooking Layer* executes directly on a compiled operating system image. Hence, such code is debuggable without modifying and compiling every time the Android source code. In compliance with the modularity of sandbox components requirement (see Section 7.2), the modular sub-architecture of the *Methods Hooking Layer* makes it flexible with respect to changes. The *Methods Hooking Layer* divide hooks by target artifacts, thus they are editable without touching the other hooks. Moreover, such sub-architecture allows researchers to share their proof of concepts or mature modules in a common framework. However, system constants expose some artifacts as well (e.g., the ones contained in `android.os.Build`). As a worst case scenario, the implementation of the *Methods Hooking Layer* might not be able to intercept accesses to system constants. Nevertheless, in this case we may patch the app and redirect the accesses to system constants toward a class that we redefined ad-hoc.

7.3.2 Events Player

Real mobile devices generate many events in response to external stimuli, hence hooking methods calls and manipulation their return value is not enough to simulate such asynchronous behavior. Listeners catch most of the events in Android, and they perform an associated action afterwards. In order to make our runtime environment as realistic as possible, we need the *Events Player* replay recorded or generated streams of events in the emulator. Besides the touch screen, the main sources of events are sensors (e.g., accelerometer, thermometer) and multimedia interfaces (e.g., camera, microphone). The *Events Player* shall stimulate other hardware conditions that change over time, such as battery drain.

The *Events Player* replays tidily the streams of events, respecting their order. The accuracy of values domain is crucial to build a stealthy sandbox. Indeed, the sandbox would be vulnerable to detection and fingerprinting, whether the injected events do not resemble the ones that come from a real sensor (e.g., they are out of range). Like the *Methods Hooking Layer*, the *Events Player* runs along with the operating system. This means that the *Events Player* does not require to rebuild Android from scratch or to modify its source code. Similarly to the *Methods Hooking Layer*, the *Events Player* uses only tools from Android, Android SDK and emulators, without requiring any modification. Finally, the *Events Player* is also able to feed the camera and the microphone with pre-recorded footage.

7.3.3 Coordinator and Logger

The *Coordinator and Logger*, as it results clear from its name, has two roles: to coordinate and to log. Its first role as coordinator consists in ensuring consistency of bogus data, which the other components inject into the emulator. Whenever the *Methods Hooking Layer* loads a new module, or when the *Events Player* opens an events stream, we have to instruct the coordinator on how to manage such hooks or events stream in accordance with the other modules. A deep study of the interaction between Android features lead to a set of rules, which the coordinator feature is able to interpret. For example, data that sensors acquire is interdependent (e.g., accelerometer and GPS). Moreover, actuators on the device (i.e., the screen, the notification LED, the flash, speakers and the vibrator) can also influence data that sensors record (e.g., speakers may influence the microphone).

The second role of this component consists in logging what happens inside the sandbox. This logging feature of the *Coordinator and Logger* is useful to have an insight on which detection techniques the samples are probably exploiting. In addition to that, the logging feature is even more useful to signal whenever a sample attempts to use a known technique which the sandbox is not able to cope with yet. In this way, Mirage is able to monitor all possible evasion attempts. The *Methods Hooking Layer* reports to the *Coordinator and Logger* every suspect or evidence about the analyzed sample. The *Coordinator and Logger* could manage the analysis process entirely. As an example, this component could handle tasks such as sample submission or the presentation of results.

7.3.4 Data Collection App

The task of the *Data Collection App* is to collect information from real mobile devices. Then, the *Coordinator and Logger* will inject such information into the *Methods Hooking Layer* and into the *Events Player*. The goal of this process is to hide artifacts in the emulator. Indeed, acquiring data from different smartphones and tablets models allows to create emulator instances with different characteristics. At the same time, this approach also reduces the risk that malware authors detect a particular image. The app is also responsible of capturing events streams on the real device, and store them in a compact and easy to replay representation.

The *Data Collection App* can retrieve information from real mobile devices available in a laboratory, but a real advantage would be to collect data with crowd-sourcing. On one hand, in a laboratory scenario researchers could ask their colleagues or students to kindly give their help by installing the app and uploading data. The DELTA framework (See Chapter 6) is an example of logger for Android that may be used to collect the data from real mobile devices. On the other hand, in a crowd-sourcing scenario companies could include the *Data Collection App* in their mobile app. Adopting a freemium pricing strategy, companies can freely distribute their software for free in exchange for data collected from the device. With an app with a wide user base, it is also possible to acquire "disposable" data on demand. As an example, an antivirus app may offer to the user an extension of the license or a month of premium features, if she agrees to share with the company her sensors events for the next ten minutes. In both scenarios, we highlight that data collection must be respectful of the privacy of the participants, e.g., applying perturbation on collected data. Such perturbation is meant to alter information in such a way that avoids to expose the contributing user's identity (e.g., biometrics, habits) and, at the same time, preserves the characteristics of the device.

7.4 A Representative Case Study: Tackling Evasion Attacks Based on Sensors with Mirage

In this section, we present the development process of a sensors module for Mirage, i.e., a collection of modules that emulates sensors in one or more Mirage components. Designing an effective countermeasure against evasion attacks requires a deep understanding of the problem. In this case study, we analyzed the differences in sensors characteristics between real devices and emulators. This case study has two purposes: (i) to briefly describe how we implemented Mirage, and (ii) to show that Mirage is effective against the proposed detection heuristics based on sensors. With a proof of concept implementation, we propose also an approach to carry out an investigation on evasion attacks. The final goal of such investigation is the development of a module for Mirage. In this way, researchers can extend Mirage to tackle novel evasion attacks, by following the workflow we present in this section.

In what follow, we discuss some choices about the components of our Mirage implementation. First, the *Methods Hooking Layer* rely on the Xposed framework as a methods hooking facility [201]. Xposed is an open source tool that allows to inject code before and after a method call. It is worthy of note that other hooking tools, such as Cydia Substrate [76], adbi [141] serve the same purpose. In particular, we preferred Xposed because Cydia Substrate is not open source and adbi supports only the instruction sets of ARM processors. Xposed by its nature is detectable, since it introduces some artifacts. However, subverting methods hooking detection techniques is not difficult, as pointed out in [28]. Secondly, the *Events Player* relies on a Telnet console in QEMU, which allows to remotely inject sensors events into the emulator. During our preliminary studies, we considered multiple alternative approaches. Unfortunately, most of the alternative approaches we investigated are not viable due to our requirements in Section 7.2 (e.g., modifications to the emulator) or because they are not compatible with recent Android versions (e.g., RERAN [87]). Although this is a QEMU-specific feature, other emulators (e.g., Genymotion, Andy) offer a similar events injection mechanism. Finally, we develop a custom *Data Collection App* and we implement the remaining components as a set of scripts.

The case study we report in this chapter is focused on sensors artifacts. We chose detection techniques based on sensors for three reasons:

1. Researchers pointed the feasibility of such detection techniques [159, 199] without providing any effective countermeasure.

2. A possible countermeasure against such detection techniques involves multiple components in our system (i.e., the *Methods Hooking Layer*, the *Events Player*, the *Coordinator and Logger*, the *Data Collection App*).

3. Accessing motion, position and environmental sensors do not require any permission. This means that the sensors-based detection techniques are stealthier than the ones that do not rely on sensors. In fact, a popular app can be repackaged to include a sensors-based detection technique, without altering the original permission list in its manifest.

Modern mobile devices embed motion sensors that measure acceleration and rotational forces along three axes. Motion sensors are usually hardware-based (e.g., accelerometer, gyroscope). However, some sensors are software-based (e.g., gravity sensor), since they derive their data from hardware-based sensors. Mobile devices embed also hardware-based sensors that measure the position of the device, such as orientation and other magnetometer sensors (e.g., compass). In addition to that, some other hardware-based sensors measure various environmental parameters (e.g., ambient temperature, pressure, light level, and humidity). Android sensors API provides classes and interfaces that allow to retrieve sensors characteristics and to acquire raw data. Sensors availability varies among mobile device models. Indeed, most of mobile devices embed an accelerometer and a magnetometer but fewer have a barometer or a thermometer. High-end smartphones and tablets even embed more than a single hardware-based sensor per type. In addition to the model of the mobile device, the availability of a sensor also depends on the Android version installed. For example, only recent platform releases (API level greater than or equal to 14) support the humidity sensor [18].

Our workflow starts with threat modeling (described in Section 7.4.1), continues with artifacts discovery and analysis (Section 7.4.2), and ends with the implementation of the module (Section 7.4.3). By following the above steps, researchers can progressively improve Mirage, toward an ideally undetectable sandbox.

7.4.1 Threat Model

In our threat model, we assume an attacker that is running a malicious app on a mobile device, with full access to the Android sensors API. The sensors API is composed of `SensorManager`, `Sensor`, and `SensorEvent` classes, plus the `SensorEventListener` interface. An instance of `SensorManager` corresponds to the sensor service, which allows to access to the set of sensors available on the device. An instance of `Sensor` is related to a specific sensor, which can be hardware or software-based. The methods of the `Sensor` object permit to identify sensor capabilities. To maximize the pool of potential victims, the attacker may prefer to include in the malicious app a detection technique based on a low API level for compatibility reasons. The `SensorEvent` class represents a single sensor event, that contains: the sensor type, the sensor state (i.e., value and accuracy), and the event timestamp. The `SensorEventListener` is a Java interface to implement in order to receive notifications whenever a sensor state changes. In our threat model, we also assume that the malicious app has a limited timespan before deciding whether to execute the payload or to remain dormant. In that time interval, the malicious app can monitor some sensors events.

We argue that sensors should only measure environmental properties, but they expose much more information. In fact, researchers demonstrated that it is possible to extract a reliable hardware fingerprint of a mobile device from accelerometer calibration errors [33,58]. Moreover, the Google API for Android provides methods that rely on sensors data to recognize the state of the user, i.e., staying still or moving [89]. If the user is moving, such API can even recognize her mean of transport (e.g., if she is in a vehicle, on a bicycle or if she is walking). Although a malware author may want to target a specific user or device model, for the purpose of this work we assume that malware authors' main goal is to detect the sandbox, in order to evade dynamic analysis. In the case we do not apply perturbation on sensor data, the *Events Player* can replay the exact events stream to reproduce the device-specific fingerprint. The *Coordinator and Logger* could also take charge of linking sensors events to GPS, network and battery data in order to deceive activity recognition.

7.4.2 Artifacts Analysis

Artifacts are imperfections that make a sandbox distinguishable from a real device. To put ourselves in attacker's shoes, we studied the Android sensors API in order to find out which sensors artifacts malware could leverage to evade dynamic analysis. First, we analyzed real smartphones such as LG/Google Nexus 5 and 5X, Samsung Galaxy S5 and S6, Galaxy Ace Plus, and Asus ZenFone 2. These real devices were running different operating system versions, ranging from Android 2.3 (API level 9) to Android 7 (API level 24), which is the most recent release at the time of writing. Then, we analyzed how emulators supports sensors. In this analysis, we considered Android SDK's emulator and Genymotion (free plan), given their popularity among developers. On one hand, the Android SDK provides a mobile device emulator based on QEMU (QEMU from now on). Such emulator uses Android Virtual Device (AVD) configurations to customize the emulated hardware platform. Moreover, QEMU includes a log console and it can simulate interrupts and latency on the network [16]. On the other hand, Genymotion is a third party emulator, but it is compatible with Android SDK tools. Genymotion allows developers to control features like the camera, the GPS and battery charge levels. Most of the features of Genymotion are also manageable through a Java API [83].

The first discrepancy we noticed is that both emulators support a limited set of sensors. In Table 7.1, we show that QEMU simulates nearly as many sensors as an entry-level smartphone embeds (e.g., Galaxy Ace), but Genymotion only simulates the accelerometer. Despite in Table 7.1 we report our findings only focusing on three real mobile devices (due to lack of space), we verified that the other devices at our disposal exhibit similar characteristics. The developers of Android defined some types of sensors (i.e., the ones whose names begin with `android.sensor.*`). For such sensors, the `getType` method returns an integer number less than or equal to 100. Moreover, vendors can introduce custom sensors, i.e., the sensors for which `getStringType` returns a string that begins with `com.google.sensor.*` in the Nexus 5X. Given this fact, we can argue that a malware author who wants to target as much users as possible will not rely on device-specific sensors. In addition to that, malware authors have to focus on sensors available in API level 9 in or-

getStringType		getType	API Lv.	Nexus 5X	Galaxy S5	Galaxy Ace	QEMU	Geny-motion
`android.sensor.`	`accelerometer`	1	3	✓	✓	✓	✓	✓
	`magnetic_field`	2	3	✓	✓	✓	✓	✗
	`orientation`	3	3	✓	✓	✓	✓	✗
	`gyroscope`	4	3	✓	✓	✗	✗	✗
	`light`	5	3	✓	✓	✗	✓	✗
	`pressure`	6	3	✓	✓	✗	✓	✗
	`temperature`	7	3	✗	✗	✗	✓	✗
	`proximity`	8	3	✓	✓	✓	✓	✗
	`gravity`	9	9	✓	✓	✓	✗	✗
	`linear_acceleration`	10	9	✓	✓	✓	✗	✗
	`rotation_vector`	11	9	✓	✓	✓	✗	✗
	`relative_humidity`	12	14	✗	✗	✗	✓	✗
	`magnetic_field_uncalibrated`	14	18	✓	✓	✗	✗	✗
	`game_rotation_vector`	15	18	✓	✓	✗	✗	✗
	`gyroscope_uncalibrated`	16	18	✓	✓	✗	✗	✗
	`significant_motion`	17	18	✓	✓	✗	✗	✗
	`step_detector`	18	19	✓	✓	✗	✗	✗
	`step_counter`	19	19	✓	✓	✗	✗	✗
	`geomagnetic_rotation_vector`	20	19	✓	✗	✗	✗	✗
	`tilt_detector`	22	n/a	✓	✗	✗	✗	✗
	`pick_up_gesture`	25	n/a	✓	✗	✗	✗	✗
`com.google.`	`sensor.internal_temperature`	65536	n/a	✓	✗	✗	✗	✗
	`sensor.sync`	65537	n/a	✓	✗	✗	✗	✗
	`sensor.double_twist`	65538	n/a	✓	✗	✗	✗	✗
	`sensor.double_tap`	65539	n/a	✓	✗	✗	✗	✗
	`sensor.window_orientation`	65540	n/a	✓	✗	✗	✗	✗
Empty String		65558	n/a	✗	✓	✗	✗	✗
Total				**24**	**17**	**7**	**8**	**1**

Table 7.1: Sensors availability in some of the tested devices.

der to target most of the devices (approximately 99.9% of the active devices according to Google Play [13]). As an example, API levels prior to 9 do not support the relative humidity sensor (`TYPE_RELATIVE_HUMIDITY`) [18].

In our analysis, we considered the sensors embedded in real devices and the ones simulated by virtual devices. For each sensor, we called all methods available in the `Sensor` class. As an example, in Table 7.2 we show the discrepancies in terms of return values for accelerometer methods on real and emulated Nexus 5X. In Table 7.2, we also include the return values for our proposal, which we discuss in details in Section 7.5. Malware authors can rely on those discrepancies to develop simple detection techniques (a single conditional statement is enough). We refer to these techniques as *static heuristics*, since they exploit an artifact due to the Android API, which is not related to events streams. Based on such value, the app decides whether to execute the malicious payload or to remain dormant. Another example is the proximity sensor, which is called `Goldfish Proximity sensor` in QEMU and `RPR0521 proximity` in the real Nexus 5X. The accelerometer, thanks to its wide availability, is particularly well suited for broad-spectrum heuristics.

A malware can leverage several methods for the same purpose, e.g., `getFifoMaxEventCount` in Algorithm 1. This second static heuristic checks the number of events that could be batched for the accelerometer. The number of events is not likely to be zero for real accelerometers, since they usually support batch mode (i.e., the ability of storing events in a hardware FIFO). Please notice that `getMaxDelay` may return zero on older real devices, hence a zero delay does not necessarily guarantee that the execution is taking place in an emulator [14]. In conclusion, an adversary could exploit most of the return values from the `Sensor` class for sandbox detection.

```
if accelerometer.getFifoMaxEventCount() == 0 then
    benign behavior
else
    malicious behavior
end
```

Algorithm 1: Static heuristic based on `getFifoMaxEventCount` method.

In the literature, researchers already pointed out the feasibility of *dynamic heuristics*, in which they exploited sensors events that emulators generate [159, 199]. We investigated further: for each real mobile device at our disposal, we registered callback methods to receive changes in sensors state. By applying the option `SENSOR_DELAY_FASTEST`, we got those states as fast as possible. In our experiments, we observed that collecting an incoming stream of events for ten seconds is enough for our purpose. We collected sensors data from real mobile devices in three different scenarios: lying on a table, while typing and leaving them in a pocket while walking. Then, we repeated the data collection task on QEMU and Genymotion emulators. Such emulators allow only two modes of screen rotation: portrait and landscape.

During our experiments, we were able to observe some differences between real and emulated motion sensors. In real mobile devices, we noticed that motion sensors (e.g., the accelerometer) quickly oscillate among a small range of values, even when the device is lying on a flat surface. In emulators, we noticed that it is possible to stimulate the accelerometer by changing from landscape to portrait mode. In contrast, without rotating the screen, each motion sensor in emulators produce the same value. As an example, the accelerometer in QEMU constantly returns `(0, 9.77622, 0.813417)`. Again in QEMU, rotating the screen has the sole effect of toggling the accelerometer values between `(0, 9.77622, 0.813417)` and `(9.77622, 0, 0.813417)` in portrait position and landscape mode, respectively. Table 7.3 records the constant values that each sensor in QEMU produces. It is worthy of note that some sensors in QEMU produce values only along one axis, so in Table 7.3 we mark the cells related to the other two axes as n/a. A similar logic applies to the accelerometer in Genymotion, which toggles between `(-0, 9.81, 0)` and `(9.81, 6.0068924E-16, 0)` in portrait and landscape mode, respectively. Regarding the events that sensors produce

Device	**getName**	**getVendor**	**getFifoMax-EventCount**
Real	`BMI160 accelerometer`	`Bosch`	5736
QEMU	`Goldfish 3-axis Accelerometer`	`The Android Open Source Project`	0
Genymotion	`Genymotion Accelerometer`	`Genymobile`	0
QEMU + Mirage	`BMI160 accelerometer`	`Bosch`	5736

Table 7.2: Example of return values for Nexus 5X accelerometer in real devices, vanilla emulators (i.e., QEMU and Genymotion) and QEMU enriched with Mirage.

		Portrait			**Landscape**		
getStringType		**values[0]**	**values[1]**	**values[2]**	**values[0]**	**values[1]**	**values[2]**
`android.sensor.`	`accelerometer`	0	9.77622	0.813417	9.77622	0	0.813417
	`magnetic_field`	0	0	0	0	0	0
	`light`	0	n/a	n/a	0	n/a	n/a
	`pressure`	0	n/a	n/a	0	n/a	n/a
	`proximity`	1	n/a	n/a	1	n/a	n/a
	`relative_humidity`	0	n/a	n/a	0	n/a	n/a

Table 7.3: Constant values produced by sensors in QEMU, grouped by screen orientation.

whenever their accuracy changes, we observed no significant differences between QEMU and a real mobile device lying on a table. However, during our experiments, Genymotion did not report any accuracy change, hence malware may exploit such imperfection to detect the emulator.

To show the detectability of the analyzed emulators, we implemented a fast dynamic heuristic that observes the variance of accelerometer values. This heuristic consists in observing the variance of values along one accelerometer axis. Since by default such emulators are able to produce at most two different accelerometer values along one axis, if the accelerometer produces at least three different values it is likely to be on a real device. In general, static heuristics are faster than dynamic ones, because static heuristics do not require looping or waiting. Hence, the execution time of our dynamic heuristics depends on how fast sensors generate events, since it needs to retrieve at least three values in order to decide. For example, a malicious app that monitors proximity values may have to wait a long time before the user covers the proximity sensor. Until the user does not bring the smartphone to her ear, or until she covers the proximity sensor with an object, the malicious app cannot be sure that dynamic analysis is not taking place. For this reason, motion sensors are the best choice since they oscillate quickly even if the device is lying on a table. Unfortunately, dynamic heuristics that rely on sensors are harder to tackle than static ones. Indeed, an ideal countermeasure against such dynamic heuristics consist in simulating or replaying events.

7.4.3 Module Implementation

In order to tackle the evasion attacks in Section 7.4.2 with Mirage:

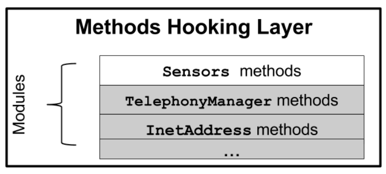
- We included in our *Data Collection App* the code we used for artifacts analysis.

- We patched the discrepancies in return values using information we obtained from the *Data Collection App*.

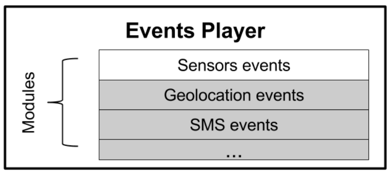
To address static heuristics, we added to the *Methods Hooking Layer* our knowledge about the characteristics of real sensors. In fact, the *Methods Hooking Layer* can intercept methods calls directed to the `Sensor` class, returning values that we collected from sensors of a real device. Xposed executes a method before (pre-method) and after (post-method) each method hooked [201]. The pre-method can evaluate and alter the arguments, or it can return a custom result. In our implementation, we used only post-methods. In fact, first we allow the original methods to execute, then we inspect the sensor type, and finally we alter its return value accordingly. After defining an hook for each method of `Sensor` class, for every available sensor type, Mirage is able to mimic the behavior of a real device. In Figure 7.2a, we show the internal structure of the *Methods Hooking Layer*.

From revision 16 of Android SDK tools, QEMU supports "realistic" sensor emulation [15], although it is still an experimental feature. This feature consists of connecting the emulator to a real mobile device, which runs a special controller app. Such app retrieves sensors values from the real mobile device and transmits them back to the emulator, in real time. Despite this experimental feature apparently sounds promising, it has many disadvantages. Firstly, maintaining such connection active consumes both bandwidth and battery power in the real device. Secondly, port forwarding between the host and the real device can be both detectable and fragile at the same time. Lastly, realizing an automatic batch analysis system using this technique is not feasible since it requires continuous human interaction. The premium plans of Genymotion offer a similar feature, which allow to stream multi-touch, accelerometer and gyroscope events from a connected device. Unfortunately, also this premium feature by Genymotion suffers the same limitation above. To cope with limitations given by live events streaming, our approach is based upon a record and replay mechanism: sensors values are first recorded from a real device and then they are replayed offline in the emulator.

In our proof of concept implementation, we leveraged QEMU to develop the replay mechanism of *Events Player*. This is because QEMU exposes a console via Telnet and it supports more sensors than Genymotion. Such console allows to control the virtualized environment, including sensors. The syntax of a Telnet command is `telnet <host>`



(a) Modules in the Methods Hooking Layer.



(b) Modules in the Events Player.

Figure 7.2: Modular internal structure of the Methods Hooking Layer (a) and of the Events Player (b). We mark in white and in gray the modules implemented and not implemented yet in Mirage, respectively.

`<console-port>`, where the default port is 5554. Once connected, we can set the values for a given sensor using the command `set <sensorname> <value-a>[:<value-b>[:<value-c>]]`. We implemented a prototype that reads a stream of values from a file and injects such stream (i.e., replay) into a running emulator. Under these settings, the *Coordinator and Logger* ensures that the *Events Player* replays for each sensor a sequence that is part of the same stream. This solution is adaptable to all Android emulators that expose a similar injection mechanism (including the premium releases of Genymotion), and it does not require any modification to the emulator. We underline that our solution meets the requirements in Section 7.2 since it does not require any modification to the emulator.

We also designed the *Events Player* in a modular way, and Figure 7.2b shows an example of its internal structure.

7.5 Evaluation

For the evaluation of our proposal, we developed the *SandboxStorm app*. Such app includes the static and dynamic heuristics in Section 7.4.2, thus it easily detected both QEMU and Genymotion emulators. To show that similar artifacts are also present in state of the art systems, we submitted our *SandboxStorm app* both to offline and online malware analysis services. We picked CuckooDroid and DroidBox as offline dynamic analysis software, mainly because they are open source. CuckooDroid adds to the Cuckoo Sandbox a QEMU-based virtual machine to execute and analyze Android apps [44]. Unfortunately, we were not able to determine which techniques this package uses, because it was not open source and it lacked of documentation. DroidBox relies on QEMU and it tries to understand the sample's behavior by repackaging the app with monitoring code [111]. Then, we picked some state of the art online malware analysis services from [147]. Among them SandDroid [173] and TraceDroid [197] were in working order. Moreover, we had the opportunity of testing the *SandboxStorm app* also

Analysis System	Static Heuristic	Dynamic Heuristic
CuckooDroid	16 ms	70361 ms
DroidBox	18 ms	69581 ms
Andrubis	16 ms	n/a
SandDroid	15 ms	73964 ms
Tracedroid	17 ms	n/a

Table 7.4: Detection time by analysis system and heuristic type.

on Andrubis [121] before its shutdown. Unfortunately, CopperDroid [193] was stuck on a long queue of unaccomplished analysis at the time of our evaluation.

In Table 7.4, we summarize the results we obtained by running our *SandboxStorm app* in the aforementioned malware analysis systems. The results show that both static and dynamic heuristics of our *SandboxStorm app* successfully detected the presence of an underlying emulator. In the worst case, our dynamic heuristic took about 74 seconds to detect that the app is running on a virtual device. However, we believe that such amount of time is still negligible in this scenario, since a malicious app can delay the start of its malicious behavior by 74 seconds. It is worthy of note that Andrubis and TraceDroid did not made available any sensor. However, the absence of sensors is a clear evidence that the sample is not running on a real device.

In order to evaluate our sensors module, we executed the *SandboxStorm app* in QEMU enhanced with our proof of concept implementation of Mirage. Once distributed, our *Data Collection App* can retrieve more information from real smartphones and tablets, in order to build several profiles. The static heuristics in *SandboxStorm app* failed to detect Mirage, because its *Methods Hooking Layer* manipulates the return values of methods on the fly. In Table 7.2, we compare some artifacts in vanilla emulators (see Section 7.4.2) to the corresponding values we collected during the execution inside Mirage. These values are identical to the ones provided by a real Nexus 5X. Moreover, the methods in `Sensor` class return realistic values for each sensor in the emulator. Even our dynamic heuristic in *SandboxStorm app* failed to detect the sandbox. In fact, the *Events Player* injects previously recorded sensors values, that are naturally different from the ones that we described in Table 7.3. Overall, *SandboxStorm app* is not able to notice that it is running in Mirage.

We also tested the stealthiness of our Mirage implementation against DroidBench (version 3.0-develop) [78], an open source benchmark suite, which includes several test cases (i.e., apps) for emulator detection. In particular, we tested our sensors module against "Sensors1" test case. The "Sensors1" test successfully recognized our real devices and both vanilla QEMU and Genymotion, but it misclassified our QEMU enhanced with Mirage as a real device. Therefore, the sensors module in Mirage was able to confuse the emulator detection heuristic in DroidBench.

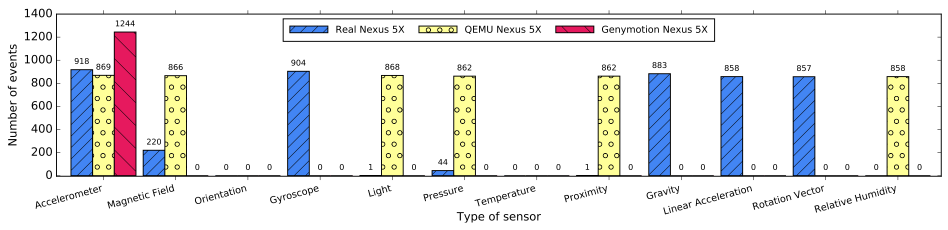


Figure 7.3: Number of events retrieved in ten seconds for several sensors and devices.

7.6 Discussion

The core contribution of our work is Mirage, a framework that allows researchers to build malware analysis systems for Android that are robust against evasion attacks. Mirage is potentially adaptable to work with any emulator or hypervisor. Our implementation of Mirage, is able to tackle complex heuristics based on sensors of our *SandboxStorm app* (see Section 7.5). In the rest of this section, we first discuss the effectiveness of our sensors module and possible limitations of Mirage. Then, we present some alternative implementations of the *Events Player*. Finally, we remark the importance of modularity in the design of Mirage.

Effectiveness of the Sensors Module: We verified that the sensors module is effective against our static and dynamic heuristics based on sensors (see Section 7.5). We further investigated the reasons underneath the success of our implementation of Mirage against the "Sensor1" heuristic of DroidBench. We noticed that such heuristic checks the presence of 13 distinct types of sensors. If DroidBench detects a number of sensors that is less than or equal to a given threshold (the authors set this threshold to 7), it reports the presence of an emulator. Hence, this test case consists in a static heuristic. In Mirage, the *Methods Hooking Layer* is responsible to cope with detection techniques that are purely static. Since our sensors module for the *Methods Hooking Layer* can fake the presence of sensors that are not actually there, "Sensor1" test case fails to count the number of real sensors.

Overall, the development process of the sensors module for Mirage helped us to show that our proposal can be a useful tool to tackle evasive malware on Android. Unfortunately, such module has some shortcomings. Given a specific real device simulated by Mirage, the *Methods Hooking Layer* is able to mimic static characteristics of sensors available in such device, even if these sensors are not present in the underlying emulator. Similarly, Mirage can also hide the sensors that are available in the emulator whenever they are not present in the real device. Nevertheless, for sandbox detection techniques that monitor the events stream (like the dynamic heuristic in

SandboxStorm app), our *Events Player* implementation is limited to the set of sensors supported by the underlying emulator (i.e., QEMU, in our current implementation). If an advanced malware wants to target specific devices, it may check the actual presence of a sensor by registering a callback method to receive values of that sensor. In addition, researchers found that the sensors in QEMU generate events on a regular time basis [159], which is less frequently than we observed on real devices. Fortunately, since Mirage is not tightly coupled to QEMU, we could easily adapt Mirage to work with an emulator that does not have this limitation.

Pre-filter NDK-based Applications: The Native Development Kit (NDK) allows embedding native code into Android apps. NDK can be useful for developers that need reduced latency to run computationally intensive apps (e.g., games) or to reuse code libraries written in C and C++. Unfortunately, allowing developers to code using NDK enables mobile malware authors to develop kernel-level exploits and sophisticated detection techniques [159,199]. Malware that is able to measure performances at low level (e.g., that measure the duration of time-consuming computation) can evade analysis systems based on virtualization by performing computational timing attacks. This is because such systems insert additional layers between the Android operating system and the CPU, with respect to real devices. Even though all these kind of artifacts are hard to patch, we can easily detect the usage of native code. Since Mirage cannot handle NDK-based malware properly, it could forward these samples to a real device or to a small bare metal infrastructure for the analysis. We assume that most of the requests are addressed in our sandbox, and we consider the forwarding of the samples to a real device as a last chance.

Alternative Implementations of the Events Player: Before deciding to rely on the Telnet console in QEMU in order to implement the sensors module in the *Events Player*, we considered different approaches. In order to simulate sensor events in real time, researchers in [159] suggested to use external software simulators, like OpenIntents Sensor Simulator (OISS) [152], or to adopt or a record-and-replay approach, like RERAN [87]. On one hand, OISS is an app that transmits simulated or recorded sensors streams to an emulator. Unfortunately, to receive the generated sensors events, OISS forces apps developers to use its own API instead of Android sensors API. This constraint is unsuitable for malware analysis, because the source code of the sample usually is not available. Given this, we are not able to rewrite the code of the sample using OISS sensors API and then rebuild it from scratch. Although it would be possible to replace methods calls by patching the compiled app, we believe that such intrusive approach may hinder the behavior of the sample, and it may cause a misclassification of it. On the other hand, RERAN is a tool that first captures an events stream from a real device and then injects the stream in another device. Input events are recorded from `/dev/input/event*` in the source device and stored in a trace using *getevent* tool of Android SDK. A custom replay agent reads the trace and writes events to `/dev/input/event*` in the destination device. Unfortunately, in recent smartphones (e.g., Nexus 5X, Galaxy S5) *getevent* tool is able to get the touchscreen and buttons events, but not sensors ones. Authors tested RERAN on smartphones released in 2011 [87] that were running Android version 2.3.4, which newer devices and Android versions superseded.

Modularity of Mirage Components: One of the most important lesson we learned during our experiments is that the Android platform is rapidly and unpredictably changing. To give a significant example, while we were testing our heuristics, the developers of Android released an improved version of QEMU (along with Android Studio 2.0 release). This new version handles many more simulated events than the previous ones, actually resembling a real device. To show that, in Figure 7.3 we compare the number of events retrieved in ten seconds from real and virtual Nexus 5X. In this experiment, we used the last releases of QEMU and Genymotion. Each sensor that the two emulators support is able to generate a number of events approximately equal or greater than the sensors on the real Nexus 5X. Unfortunately, the developers of Android arbitrarily decided to remove the opportunity to set sensors values via Telnet in the improved version of QEMU, which we exploited in our implementation of the *Events Player*.

Although we still do not know if the developers will reintroduce such feature in the future, this change highlights that the modularity in Mirage components is fundamental. Now QEMU is able to produce a significant number of sensors events on its own. Hence, it is possible to hook also methods of `SensorEventListener` class and manipulate the returned sensors values directly (without injecting sensors events from the *Events Player*). The isolation between the modules of the *Events Player* and the *Methods Hooking Layer* allows to relocate the simulation of sensors events from the former to the latter, without modifying the other modules. Nonetheless, to give a more comprehensive proof of concept of Mirage, we preferred to use the previous release of QEMU (prior to Android Studio 2.0), keeping the simulation of sensors events in the *Events Player*.

7.7 Summary

In this chapter, we take a step towards the stealthiness of malware analysis sandboxes for Android. After carefully reviewing the state of the art, we enlisted six essential requirements that an analysis system have to fulfill to tackle evasion attacks. Hence, we proposed Mirage, a framework that fulfills all these requirements. We also presented a representative case study, which shows how Mirage can cope with sandbox detection techniques that exploit artifacts in emulators due to sensors API. To evaluate our proposal, we developed a proof of concept implementation of Mirage, enabled with our sensors module. To compare our sandbox to state of the art dynamic analysis services for Android, we also developed the *SandboxStorm app.* This app contains some static and dynamic heuristics to detect emulators, based on our findings about sensors API artifacts. Our thorough evaluation shows that all dynamic analysis systems that we tested are detectable by our *SandboxStorm app.* Conversely, Mirage resembled a real device and, consequently, sensors-based heuristics in *SandboxStorm app* and in DroidBench were not able to detect Mirage as a sandbox.

Chapter 8

Conclusions

In the last decade, we face the rising of mobile devices as a fundamental tool in our everyday life. Thanks to an operating system, a wide selection of multi-purpose application and portability, such devices can accompany users everywhere they go and allow them to perform operations that were only possible through a PC a few years ago. Besides, users heavily rely on them for storing even the most sensitive information from the privacy point of view. This aspect leads mobile them under the target of who aims to gain financial benefits from obtaining the access to such information, such as malware designers. If an attacker is not able to obtain a local or remote access to the device, she can still rely on side-channel analysis to harm users privacy. Fortunately, side-channels can be also relied on for enforcing mobile devices' security. In fact, benign applications of side-channel analysis range from user behavioral authentication, to resources optimization and intrusion detection.

In this dissertation, we focused on possible attacks and benefits of the analysis of three side-channels: network traffic in Part I; energy consumption in Part II; and built-in sensors in Part III. In what follows, we first summarize our contributions to the state of the art in Section 8.1, and we then outline some possible future work in Section 8.2.

8.1 Summary of Contributions

In this section, we summarize the contributions given by the work we presented in this dissertation.

8.1.1 Network Analysis

In Part I, we presented the side-channel that can be obtained from network traffic. We assumed the attacker performed a Man-In-The-Middle (MITM) attack, thus she was able to observe TCP/IP traffic generated and from the victim's mobile device. Since such traffic has been encrypted with SSL/TLS, hence the attacker could only observe the network traffic flows, without being able to perform a Deep Packet Inspection (DPI). Under these setting, we demonstrated that encryption alone is not enough to stop an attacker from inferring private information about the victim. In particular, we focused on identifying the action a user performs with an app and on retrieving the list of apps installed on her mobile device.

- *User Actions Recognition*: In Chapter 2, we aimed to identify a set of sensitive user actions carried out with seven popular apps only analyzing encrypted network traffic. We designed and fully implemented a machine learning-based framework to carry out the analysis. To build our dataset, our framework uses a feature extraction procedure that is composed of two steps: (i) we clustered network flows using an unsupervised machine learning method with DTW as a similarity metric; and (ii) for each user action, we build a vector of features according to the presence or not of a specific type of flow in the time interval of such action (i.e., we consider a feature for each cluster obtained at the previous step). Subsequently, the dataset is used to train a classifier that is then able to recognize which action has been performed from unseen network traffic. With our framework, an adversary may collect private insight about multiple victims and aggregate them for market or intelligence surveys. Moreover, a strong adversary (e.g., Government, ISP) may profile a user's behavior according to her actions and habits in order to de-anonymize or track her, even if she changes mobile device.

- *Mobile Apps Fingerprinting*: In Chapter 3, we presented AppScanner, a framework for automatic fingerprinting and real-time identification of mobile apps from the encrypted network traffic they generated. In our evaluation, we did not only show that apps can indeed be identified with a high accuracy relying on a multi-label classifier, but we also investigate the robustness of app fingerprints across a different period of time, operating system versions, and mobile devices. On one hand, our results showed that the passage of time reduces app fingerprints' accuracy. On the other hand, we assessed that an app fingerprint is not affected by the device that the app is installed on. In order to cope with this reduction of accuracy, we proposed two methods: ambiguity detection (i.e., flows in common among more than one app) and classification validation. With these measures in place, we notice a significant increase in the robustness of app fingerprints by sacrificing the ambiguous or least representative samples. In particular, we were able to classify apps with an accuracy of 96% and 73% in the best and worst case, respectively.

8.1.2 Energy Consumption Analysis

As second side-channel, we investigated the energy consumption of mobile devices while connected to a charging cable in Part II. We proposed a scenario in which the adversary was able to control the power source by measuring the electric current provided. It is worth noticing that we extended the definition of mobile devices also to laptops since they are actually portable devices and nowadays they are more similar to smartphone and tablets than traditional PCs. In this part, we showed two different application of energy consumption analysis.

- *Laptop Users Identification*: We investigated the feasibility of recognizing a specific laptop user by its energy consumption in Chapter 4. We recall that in this work we considered as a single entity (i.e., laptop user) the ensemble composed of laptop model, user behavior, and software installed. Despite our smart meters' low sampling rate (i.e., 1 Hz), we were able to first profile and later recognize a laptop user. In particular, we considered a scenario of an office in which there were several authorized users and potential intruders. We developed MTPlug, a framework that was able to extract features from segmented power traces and train a multilabel classifier. Our results showed that MTPlug were not only able to recognize with a high accuracy which laptop user was connected to a specific socket, but also discriminate between authorized users and intruders.

- *Data Exfiltration*: In Chapter 5, we presented a novel covert-channel that exploits the electrical current absorbed by victim's mobile device while it is recharging its battery through a USB cable. Such covert-channel was established between a malicious app, named PowerSnitch, installed on victim's mobile device and a power supplier controlled by the attacker. PowerSnitch app encoded the target information into CPU bursts that could be measured from the power source. It is worth noticing that PowerSnitch app did not require any dangerous static or run-time permission considered as dangerous. At the other side of the USB cable, our decoder interpreted the signal carried by energy consumption and re-constructed the target information. We designed and fully implemented both the PowerSnitch app and the decoder, and we ran a thorough set of tests to prove the feasibility of our attack.

8.1.3 Built-in Sensors Analysis

In Part III, we considered the analysis of built-in sensors data, a side-channel that can be measured from inside the device. As far as concerns Android operating system, any running app can access to sensors data without asking any permission. At the same time, such data can be valuable for opposite reasons to researchers and malicious users. On one hand, the former ones use sensors data to improve mobile devices' usability or enhance security mechanisms (e.g., behavioral and two-factor user authentication). On the other hand, the latter ones may use such data to track user's movements and habits or evading malware analysis using sensors-based sandbox detection heuristics. In this part, we first presented a logging tool designed for researchers, and we then showed possible malware analysis evasion techniques and countermeasures.

- *Logging and Data Extraction Tool*: After a thorough study of the literature, we realized that researchers had to design their own app to collect data from Android mobile devices. This because the state of the art lacked a customizable and usable logging and data collection tool for Android. In Chapter 6, we present DELTA, a new tool that fills that gap. We designed DELTA to be modular and easily customizable, also taking into account typical researchers' needs. Moreover, we showed how researchers can design their own experiment, distribute it to the experiment's participants, and collect fresh data directly via the Internet. In the comparison with other proposals, we observed that DELTA overcame all of them in terms of richness of logged features.

- *Robust Sandbox Against Malware Analysis Evasion*: Dynamic analysis for detecting malware for Android usually relies on emulator-based sandboxes. Unfortunately, emulators are not specifically designed for that usage, thus they present several artifacts. Artifacts are discrepancies in terms of values or behavior between real mobile devices and emulators. Malware may exploit such artifacts using heuristics to detect whether is running on an emulator, thus it would nullify the efforts of researchers on dynamic analysis. In Chapter 7, we presented MIRAGE, a novel framework that enhances sandboxes for malware analysis to be resilient against emulator detection techniques. MIRAGE was designed to be modular because it can be enriched with additional modules to fix new artifacts as soon as they are discovered. In particular, we presented a module that fixed artifacts generated by built-in sensors as a representative case study. In our comparison, we showed that both static and dynamic heuristics identified MIRAGE as a real device, while they identified our competitors as emulators.

8.2 Future Work

In this section, we discuss possible future research directions that follow the research contributions reported in this dissertation.

8.2.1 Network Traffic Analysis

The literature offers lots of work related to the network traffic analysis of mobile devices, but there is still room for significant improvements. As highlighted by Conti et al. in [50], many research directions have barely been explored, such as user identification, indoor device positioning, or sociological information inference. On one hand, we are working to overcome the shortcoming of time needed for user actions dataset collection for the work presented in Chapter 2. In fact, we are relying on DELTA logging tool (see Chapter 6) for collecting network traffic synchronized with user actions from real users. We are also evaluating to use emulators to make scalable the collection of network traffic generated by apps (to further extend the work in Chapter 3) and by simulate user actions. This in order to investigate the similitudes and differences in terms of network traffic between simulated and real devices. On the other hand, we also intend to investigate on possible efficient countermeasures that can also be adopted on mobile devices. These countermeasures may require a trade-off between power efficiency and privacy level required. In the future, we intend to explore other applications of network traffic analysis in order to infer additional information about the user. As an example, we will investigate whether it is feasible to extract user habits or behavioral patterns by aggregating sequences of user actions.

8.2.2 Energy Consumption Analysis

As far as concern the work presented in Chapter 4, as a future work we intend further investigate the impact on energy consumption traces of laptop model, set of the applications installed and user behavior. As the first step in this direction, our preliminary studies suggest that the energy trace is more related to user activity rather than her laptop model. We also intend to investigate the feasibility of re-using a user energy fingerprint even when she changes the model of her laptop. The case is when a user adopts a new machine, different from the one associated with her profile. This new research will aim to evaluate if a laptop user energy fingerprint persists over a long period, for example when a user buys a new laptop. Another possible future work could consist of inferring the actions performed by a user with her laptop (e.g., watching a movie, surfing the web), similarly to what has been recently done for smartphone apps relying on network traffic analysis presented in Chapter 2.

Regarding the work in Chapter 5, we already extended our covert-channel attack to malicious power banks, reducing the tool to measure the energy consumption both in terms of price and size. We will also work on the transmitter and decoder by extending the framework to include error correction algorithms and synchronization recover mechanisms to lower down the Bit Error Ratio of data transmission.

As a possible future work for energy consumption side-channel, we intend to investigate the feasibility to identify apps installed on mobile devices relying on their background activities (e.g., push notifications). Similarly to our work on network traffic analysis in Chapter 3, we believe that an app also produce specific energy consumption fingerprint. Moreover, we aim to investigate on possible countermeasures to such attacks that could be applied both on hardware and software sides.

8.2.3 Built-in Sensors Analysis

Although, as we showed in Chapter 6, our implementation of DELTA is stable and fully working, there are ample opportunities for future extensions. In a possible extension, we consider very interesting is adding contextual awareness to DELTA. This idea consists of extending the DELTA Logging Framework's architecture so that plugins can be dynamically started and stopped, depending on the context provided by other plugins (e.g., running the touch events monitor plugin only when a certain app is in the foreground, as reported by the foreground app logger plugin). We also plan to expand the functionality of the DELTA Web Service, in order to allow researchers to configure and build an experiment directly from a web interface, rather than relying on a desktop application.

As a future work to our proposal in Chapter 7, we are looking for novel artifacts that could be exploited in heuristics to uncover sandboxes for malware analysis. After that, we will develop additional modules for MIRAGE that aim to patch such discrepancies. For both DELTA and MIRAGE, we are also evaluating to create two start-ups.

In the future, we plan to rely on DELTA's extraordinary data collection capabilities to further investigate on security and usability aspects of Human-Computer Interaction (HCI). We believe that continuously monitoring users behavior and habits from multiple data sources could reveal novel insights about HCI.

As other possible side-channels to be analyzed, we aim to exploit other source of information that range from visual to audio feedbacks. For instance, we could be able to infer user actions (e.g., taking a picture, typing on the soft-keyboard) applying computer vision techniques on video footages of mobile users while they interact with their mobile devices. Moreover, we could uncover which app a user is using only relying on rely on specific sounds produced by mobile devices (e.g., ringtone, audio or vibration feedbacks).

Bibliography

[1] Androidrank. `http://www.androidrank.org/`.

[2] Top 15 most popular social networking sites. `http://www.ebizmba.com/articles/social-networking-websites`, Jan. 2014.

[3] J. Abonyi, B. Feil, S. Nemeth, and P. Arva. Principal component analysis based time series - segmentation application to hierarchical clustering for multivariate process data. In Proceedings of IEEE Int. Conf. on Computational Cybernetics, 2003.

[4] A. Akbar, M. Nati, F. Carrez, and K. Moessner. Contextual Occupancy Detection for Smart Office by Pattern Recognition of Electricity Consumption Data. `http://iot-cosmos.eu/node/1566`, 2014.

[5] H. F. Alan and J. Kaur. Can Android Applications Be Identified Using Only TCP/IP Headers of Their Launch Time Traffic? In Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec '16, pages 61–66, New York, NY, USA, 2016. ACM.

[6] B. Aloraini, D. Johnson, B. Stackpole, and S. Mishra. A New Covert Channel over Cellular Voice Channel in Smartphones. Technical report, 2015. arXiv preprint arXiv:1504.05647.

[7] M. Ambrosin, H. Hosseini, K. Mandal, M. Conti, and R. Poovendran. Verifiable and privacy-preserving fine-grained data-collection for smart metering. In Proceedings of the IEEE Conference on Communications and Network Security (CNS), pages 655–658. IEEE, 2015.

[8] Android. AIDL documentation. `http://goo.gl/LdyUA1`.

[9] Android. Alarms. `https://goo.gl/782nTv`.

[10] Android. Permissions at runtime. `https://goo.gl/9FTnEL`.

[11] Android. Services. `http://goo.gl/UnjvA3`.

[12] Android. Building requirements. `https://goo.gl/7rLNfX`, 2016.

[13] Android. Dashboards. `https://goo.gl/7ygJx`, 2016.

[14] Android. Developer's guide. `goo.gl/lvtCmr`, 2016.

[15] Android. Hardware emulation. `https://goo.gl/ZDJstE`, 2016.

[16] Android. Run apps on the Android emulator. `https://goo.gl/2wNXcW`, 2016.

[17] Android. Security. `https://goo.gl/lZlkRN`, 2016.

[18] Android. Sensors overview. `https://goo.gl/VCSsWf`, 2016.

[19] Android Developers. Optimizing for Doze and App Standby. `http://tinyurl.com/zvphw46`.

[20] R. Atterer, M. Wnuk, and A. Schmidt. Knowing the user's every move: User activity tracking for website usability evaluation and implicit interaction. In Proceedings of the ACM international conference on World Wide Web, 2006.

[21] A. J. Aviv, K. Gibson, E. Mossop, M. Blaze, and J. M. Smith. Smudge attacks on smartphone touch screens. In Proceedings of USENIX Workshop on Offensive Technologies, 2010.

[22] A. J. Aviv, B. Sapp, M. Blaze, and J. M. Smith. Practicality of accelerometer side channels on smartphones. In Proceedings of USENIX Annual Computer Security Applications Conference, 2012.

[23] S. Baghel, K. Keshav, and V. Manepalli. An investigation into traffic analysis for diverse data applications on smartphones. In Proceedings in Conference on Communications (NCC), 2012.

[24] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to android. In Proceedings of IEEE/ACM International Conference on Automated Software Engineering, 2012.

[25] M. Basseville, I. V. Nikiforov, et al. Detection of abrupt changes: theory and application, volume 104. Prentice Hall Englewood Cliffs, 1993.

[26] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida. Characterizing user navigation and interactions in online social networks. Information Sciences, July 2012.

[27] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: Trading privacy for application functionality on smartphones. In Proceedings of ACM Workshop on Mobile Computing Systems and Applications (HotMobile), 2011.

[28] N. Bergman. Android anti-hooking techniques in Java. `goo.gl/vN1iDU`, 2015.

[29] L. Bernaille, R. Teixeira, and K. Salamatian. Early application identification. In Proceedings of ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2006.

[30] O. Berthold, H. Federrath, and M. Köhntopp. Project anonymity and unobservability in the internet. In Proceedings of the ACM conference on Computers, freedom and privacy: challenging the assumptions, 2000.

[31] C. M. Bishop et al. Pattern recognition and machine learning, volume 4. springer New York, 2006.

[32] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak. An Android application sandbox system for suspicious software detection. In Proceedings of IEEE International Conference on Malicious and Unwanted Software (MALWARE), 2010.

[33] H. Bojinov, Y. Michalevsky, G. Nakibly, and D. Boneh. Mobile device identification via sensor fingerprinting. arXiv preprint arXiv:1408.1416, 2014.

[34] L. Bordoni, M. Conti, and R. Spolaor. Mirage: Toward a stealthier and modular malware analysis sandbox for android. In Proceedings of European Symposium on Research in Computer Security (ESORICS). Springer, 2017.

[35] L. Breiman. Random forests. Machine Learning, 45, 2001.

[36] N. Brouwers and K. Langendoen. Pogo, a middleware for mobile phone sensing. In Proceedings of ACM Middleware, 2012.

[37] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In ECML PKDD Workshop: Languages for Data Mining and Machine Learning, pages 108–122, 2013.

[38] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In Proceedings

of the ACM conference on Computer and Communications Security (CCS), pages 605–616. ACM, 2012.

[39] D. Carlson and A. Schrader. Dynamix: An open plug-and-play context framework for android. In Proceedings of IEEE International Conference on the Internet Of Things, 2012.

[40] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In Proceedings of USENIX Annual Technical Conference (ATC), 2010.

[41] P. Casas, J. Mazel, and P. Owezarski. Minetrac: Mining flows for unsupervised analysis & semi-supervised classification. In Proceedings of the International Teletraffic Congress, 2011.

[42] B. Chacos. Usb condom promises to protect your dongle from infected ports. PC World, August 2014. `http://tinyurl.com/hvlqkrt`.

[43] S. Chandra, Z. Lin, A. Kundu, and L. Khan. Towards a systematic study of the covert channel attacks in smartphones. In Proceedings of international conference on Security and privacy in communication networks (SecureComm), 2014.

[44] Check Point Software Technologies LTD. Automated Android malware analysis with Cuckoo Sandbox. `https://goo.gl/pDokqw`, 2016.

[45] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In Proceedings of IEEE Symposium on Security & Privacy (SP), 2010.

[46] E. Chin, A. P. Felt, V. Sekar, and D. Wagner. Measuring user confidence in smartphone security and privacy. In Proceedings of USENIX Symposium on Usable Privacy and Security (SOUPS), 2012.

[47] M. Cinque, D. Cotroneo, and A. Testa. A logging framework for the on-line failure analysis of android smart phones. In Proceedings of the ACM European Workshop on AppRoaches to MObiquiTous Resilience (ARMOR), 2012.

[48] S. S. Clark, H. Mustafa, B. Ransford, J. Sorber, K. Fu, and W. Xu. Current events: Identifying webpages by tapping the electrical outlet. In Proceedings of European Symposium on Research in Computer Security (ESORICS), pages 700–717. Springer, 2013.

[49] M. Conti, N. Dragoni, and S. Gottardo. Mithys: Mind the hand you shake-protecting mobile devices from ssl usage vulnerabilities. In Security and Trust Management. Springer, 2013.

[50] M. Conti, Q. Li, A. Maragno, and R. Spolaor. The dark side (-channel) of mobile devices: A survey on network traffic analysis. arXiv preprint arXiv:1708.03766, 2017.

[51] M. Conti, L. Mancini, R. Spolaor, and N. V. Verde. Analyzing android encrypted network traffic to identify user actions. IEEE Transactions on Information Forensics and Security, 2016.

[52] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde. Can't you hear me knocking: Identification of user actions on android apps via traffic analysis. In Proceedings of ACM Conference on Data and Application Security and Privacy (CODASPY).

[53] M. Conti, M. Nati, E. Rotundo, and R. Spolaor. Mind the plug! laptop-user recognition through power consumption. In Proceedings of ACM International Workshop on IoT Privacy, Trust, and Security (IoTPTS), pages 37–44. ACM, 2016.

[54] M. Conti, I. Zachia-Zlatea, and B. Crispo. Mind how you answer me!: transparently authenticating the user of a smartphone when answering or placing a call. In Proceedings in ACM Asia Conference on Computer and Communications Security (ASIACCS), 2011.

[55] S. E. Coull and K. P. Dyer. Traffic analysis of encrypted messaging services: Apple imessage and beyond. ACM SIGCOMM Comput. Commun. Rev., 2014.

[56] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli. Traffic classification through simple statistical fingerprinting. ACM SIGCOMM Computer Communication Review, 37(1):5–16, 2007.

[57] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. Networkprofiler: Towards automatic fingerprinting of android apps. In Proceedings of IEEE International Conference on Computer Communications (INFOCOM), 2013.

[58] S. Dey, N. Roy, W. Xu, R. R. Choudhury, and S. Nelakuditi. AccelPrint: Imperfections of accelerometers make smartphones trackable. In Proceedings on Network and Distributed System Security Symposium (NDSS), 2014.

[59] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

[60] Q. Do, B. Martini, and K.-K. R. Choo. Exfiltrating data from android devices. Comput. Secur., 48(C):74–91, Feb. 2015.

[61] T. Do, S. Rawshdeh, and W. Shi. ptop: A process-level power profiling tool. In Proceedings of the 2nd Workshop on Power Aware Computing and Systems (HotPower), 2009.

[62] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. In Proceedings of IEEE Symposium on Security and Privacy, pages 332–346, May 2012.

[63] A. Ebadat, G. Bottegal, D. Varagnolo, B. Wahlberg, and K. H. Johansson. Estimation of building occupancy levels through environmental signals deconvolution. In Proceedings of ACM International Conference on Systems for Energy-Efficient Built Environments (BuildSys), 2013.

[64] T. Eder, M. Rodler, D. Vymazal, and M. Zeilinger. Ananas - a framework for analyzing android applications. In Procedings of International Conference on Availability, Reliability and Security (ARES), 2013.

[65] C. Efthymiou and G. Kalogridis. Smart grid privacy via anonymization of smart metering data. In Proceedings of IEEE Smart Grid Communications, 2010.

[66] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2010.

[67] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A first look at traffic on smartphones. In Proceedings of the ACM SIGCOMM conference on Internet measurement (IMC), pages 281–287. ACM, 2010.

[68] H. Falaki, R. Mahajan, and D. Estrin. Systemsens: a tool for monitoring usage in smartphone research deployments. In Proceedings of ACM Workshop on Mobility in the Evolving Internet Architecture (MobiArch), 2011.

[69] Z. Fan, G. Kalogridis, C. Efthymiou, M. Sooriyabandara, M. Serizawa, and J. McGeehan. The new frontier of communications research: Smart grid and smart metering. In Proceedings of ACM International Conference on Future Energy Systems (e-Energy), 2010.

[70] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. Gaur, M. Conti, and R. Muttukrishnan. Android security: A survey of issues, malware penetration and defenses. IEEE Communications Surveys & Tutorials, 2015.

[71] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In Proceedings of the ACM conference on Computer and Communications Security (CCS), 2011.

[72] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In Proceedings of ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), 2011.

[73] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In Proceedings of USENIX Symposium on Usable Privacy and Security (SOUPS), 2012.

[74] D. Ferreira, A. K. Dey, and V. Kostakos. Understanding human-smartphone concerns: a study of battery life. In Proceedings of Pervasive computing, 2011.

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

[76] J. Freeman. Instrument Java methods using native code. `https://goo.gl/1yqeFj`, 2016.

[77] F. Freiling, M. Spreitzenbarth, and S. Schmitt. Forensic analysis of smartphones: The android data extractor lite (adel). In Proceedings of Conference on Digital Forensics, Security and Law (ADFSL), 2011.

[78] C. Fritz, S. Arzt, and S. Rasthofer. DroidBench. `goo.gl/MEPCsD`, 2016.

[79] J. Froehlich, M. Y. Chen, S. Consolvo, B. Harrison, and J. A. Landay. Myexperience: a system for in situ tracing and capturing of user feedback on mobile phones. In Proceedings of ACM International Conference on Mobile Systems, Applications, and Services (MobiSys), 2007.

[80] J. Gajrani, J. Sarswat, M. Tripathi, V. Laxmi, M. Gaur, and M. Conti. A robust dynamic analysis system preventing sandbox detection by Android malware. In Proceedings of ACM International Conference on Security of Information and Networks (SIN), 2015.

[81] Gartner. Top 10 worldwide mobile phone vendors increased sales in second quarter of 2016. `https://goo.gl/XOArDi`, 2016.

[82] D. Genkin, I. Pipman, and E. Tromer. Get your hands off my laptop: Physical side-channel key-extraction attacks on pcs. Journal of Cryptographic Engineering, 5(2):95–112, 2015.

[83] Genymotion. Using Genymotion Java API. `https://goo.gl/zCTuDl`, 2016.

[84] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating ssl certificates in non-browser software. In Proceedings of the ACM conference on Computer and Communications Security (CCS), 2012.

[85] C. Giuffrida, K. Majdanik, M. Conti, and H. Bos. I sensed it was you: authenticating mobile users with sensor-enhanced keystroke dynamics. In Proceedings of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA). Springer, 2014.

[86] Y. Go, D. F. Kune, S. Woo, K. Park, and Y. Kim. Towards accurate accounting of cellular data for tcp retransmission. In Proceedings of ACM Workshop on Mobile Computing Systems and Applications (HotMobile), 2013.

[87] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing-and touch-sensitive record and replay for android. In Proceedings of IEEE International Conference on Software Engineering (ICSE), 2013.

[88] G. Gómez Sena and P. Belzarena. Early traffic classification using support vector machines. In Proceedings of ACM International Latin American Networking Conference (LANC), 2009.

[89] Google. ActivityRecognitionApi. `goo.gl/NIFHRv`, 2016.

[90] J. Grover. Android forensics: Automated data collection and reporting from a mobile device. Digital Investigation, 2013.

[91] R. Guo, T. Zhu, Y. Wang, and X. Xu. Mobilesens: A framework of behavior logger on andriod mobile device. In Proceedings of IEEE International Conference on Pervasive Computing and Applications (ICPCA), 2011.

[92] G. W. Hart. Non-intrusive appliance load monitoring. Proceedings of the IEEE, 1992.

[93] T. Hastie, R. Tibshirani, and J. Friedman. The Elements of Statistical Learning (2nd ed.). Springer, 2009.

[94] D. Herrmann, R. Wendolsky, and H. Federrath. Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the

Multinomial Naive-bayes Classifier. In Proceedings of ACM Cloud Computing Security Workshop (CCSW), 2009.

[95] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In Proceedings of the ACM conference on Computer and Communications Security (CCS), 2011.

[96] C.-K. Hsieh, H. Tangmunarunkit, F. Alquaddoomi, J. Jenkins, J. Kang, C. Ketcham, B. Longstaff, J. Selsky, B. Dawson, D. Swendeman, D. Estrin, and N. Ramanathan. Lifestreams: A modular sense-making toolset for identifying important patterns from everyday life. In Proceedings of ACM Conference on Embedded Networked Sensor Systems (SenSys), 2013.

[97] B. Insider. The Smartphone Market Is Now Bigger Than The PC Market, 2011. `http://goo.gl/XkM8XM`.

[98] M. Jawurek, M. Johns, and K. Rieck. Smart metering de-pseudonymization. In Proceedings of ACM Annual Computer Security Applications Conference (ACSAC), 2011.

[99] X. Jiang, S. Dawson-Haggerty, P. Dutta, and D. Culler. Design and implementation of a high-fidelity ac metering network. In Proceedings of IEEE International Conference on Information Processing in Sensor Networks (IPSN), 2009.

[100] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu. Morpheus: automatically generating heuristics to detect Android emulators. In Proceedings of ACM Annual Computer Security Applications Conference (ACSAC), 2014.

[101] G. Kalogridis, C. Efthymiou, S. Z. Denic, T. A. Lewis, and R. Cepeda. Privacy for smart meters: Towards undetectable appliance load signatures. In Proceedings of IEEE Smart Grid Communications, 2010.

[102] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. Blinc: Multilevel traffic classification in the dark. In Proceedings of ACM SIGCOMM, 2005.

[103] E. Keogh, S. Chu, D. Hart, and M. Pazzani. An online algorithm for segmenting time series. In Proceedings of IEEE International Conference on Data Mining (ICDM), 2001.

[104] H. Kim, K. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee. Internet traffic classification demystified: Myths, caveats, and the best practices. In Proceedings of ACM International Conference

on emerging Networking EXperiments and Technologies (CoNEXT), 2008.

[105] H. Kim, J. Smith, and K. G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In Proceedings of ACM International Conference on Mobile Systems, Applications, and Services (MobiSys), 2008.

[106] W. Kleiminger, C. Beckel, T. Staake, and S. Santini. Occupancy detection from electricity consumption data. In Proceedings of ACM International Conference on Systems for Energy-Efficient Built Environments (BuildSys), 2013.

[107] B. Krishnamurthy. Privacy and online social networks: Can colorless green ideas sleep furiously? IEEE Security and Privacy, 2013.

[108] J.-C. Kuster and A. Bauer. Platform-centric android monitoring - modular and efficient. CoRR, abs/1406.2041, 2014.

[109] J.-F. Lalande and S. Wendzel. Hiding privacy leaks in android applications using low-attention raising covert channels. In Procedings of International Conference on Availability, Reliability and Security (ARES), 2013.

[110] N. D. Lane, Y. Chon, L. Zhou, Y. Zhang, F. Li, D. Kim, G. Ding, F. Zhao, and H. Cha. Piggyback crowdsensing (pcs): Energy efficient crowdsourcing of mobile sensor data by exploiting smartphone app opportunities. In Proceedings of ACM Conference on Embedded Networked Sensor Systems (SenSys), 2013.

[111] P. Lantz. Dynamic analysis of Android apps. `https://goo.gl/bFvjWS`, 2015.

[112] B. Lau, Y. Jang, C. Song, T. Wang, P. H. Chung, and P. Royal. Mactans: Injecting malware into ios devices via malicious chargers. Black Hat USA, 2013.

[113] A. Le, J. Varmarken, S. Langhoff, A. Shuba, M. Gjoka, and A. Markopoulou. AntMonitor: A System for Monitoring from Mobile Devices. In Proceedings of ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big (Internet) Data, C2B(1)D '15, pages 15–20, New York, NY, USA, 2015. ACM.

[114] J. Liang, S. K. Ng, G. Kendall, and J. W. Cheng. Load signature study - part ii: Disaggregation framework, simulation, and applications. IEEE Transactions on Power Delivery, 2010.

[115] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. ACM SIGPLAN Notices, 1998.

[116] M. Liberatore and B. N. Levine. Inferring the Source of Encrypted HTTP Connections. In Proceedings of the ACM conference on Computer and Communications Security (CCS), 2006.

[117] J. Lifton, M. Feldmeier, Y. Ono, C. Lewis, J. Paradiso, et al. A platform for ubiquitous sensor deployment in occupational and domestic environments. In Proceedings of IEEE International Conference on Information Processing in Sensor Networks (IPSN), 2007.

[118] Y.-s. Lim, H.-c. Kim, J. Jeong, C.-k. Kim, T. T. Kwon, and Y. Choi. Internet traffic classification demystified: On the sources of the discriminative power. In Proceedings of ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2010.

[119] G.-y. Lin, S.-c. Lee, and J. Y.-j. Hsu. Sensing from the panel: Applying the power meters for appliance recognition. In Proceedings of the Conference on Artificial Intelligence and Applications. Springer, 2009.

[120] L. Lin, T. Kasper, M.and Güneysu, C. Paar, and W. Burleson. Trojan side-channels: lightweight hardware trojans through side-channel engineering. In Proceedings of CHES. Springer, 2009.

[121] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer. Andrubis–1,000,000 apps later: A view on current Android malware behaviors. In Proceedings of IEEE International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), 2014.

[122] L. Liu, G. Yan, X. Zhang, and S. Chen. VirusMeter: Preventing Your Cellphone from Spies. In Proceedings of International Workshop on Recent Advances in Intrusion Detection (RAID), 2009.

[123] H. Lockheimer. Android and security. `https://goo.gl/fFFQcC`, 2012.

[124] X. Luo, P. Zhou, E. W. W. Chan, W. Lee, R. K. C. Chang, and R. Perdisci. Httpos: Sealing information leaks with browser-side obfuscation of encrypted flows. In Proceedings on Network and Distributed System Security Symposium (NDSS), 2011.

[125] A. MacHiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In Proceedings of ACM Joint Meeting on Foundations of Software Engineering, pages 224–234, 2013.

[126] D. Maier, M. Protsenko, and T. Müller. A game of droid and mouse: The threat of split-personality malware on Android. Computers & Security, 2015.

[127] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun. Analysis of the communication between colluding applications on modern smartphones. In Proceedings of ACM Annual Computer Security Applications Conference (ACSAC), 2012.

[128] F. Matenaar and P. Schulz. Detecting Android sandboxes. `https://goo.gl/0fp4bB`, 2012.

[129] A. Mcgregor, M. Hall, P. Lorier, and J. Brunskill. Flow clustering using machine learning techniques. In Proceedings of Passive and Active Network Measurement (PAM). Springer, 2004.

[130] D. McIntire, T. Stathopoulos, and W. Kaiser. etop-sensor network application energy profiling on the leap2 platform. In Proceedings of IEEE International Conference on Information Processing in Sensor Networks (IPSN), 2007.

[131] W. McKinney. Data Structures for Statistical Computing in Python. In S. van der Walt and J. Millman, editors, Proceedings of the 9th Python in Science Conference, pages 51 – 56, 2010.

[132] W. Meng, W. H. Lee, S. R. Murali, and S. P. T. Krishnan. Charging me and i know your secrets!: towards juice filming attacks on smartphones. In Proceedings of ACM Workshop on Cyber-Physical System Security (CPSS), 2015.

[133] B. Miller, L. Huang, A. D. Joseph, and J. D. Tygar. I Know Why You Went to the Clinic: Risks and Realization of HTTPS Traffic Analysis, pages 143–163. Springer International Publishing, Cham, 2014.

[134] T. M. Mitchell. Machine learning. 1997.

[135] S. Mongkolluksamee, V. Visoottiviseth, and K. Fukuda. Enhancing the Performance of Mobile Traffic Identification with Communication Patterns. In Proceedings of Annual Computer Software and Applications Conference, volume 2, pages 336–345, 2015.

[136] S. Mongkolluksamee, V. Visoottiviseth, and K. Fukuda. Combining Communication Patterns &Traffic Patterns to Enhance Mobile Traffic Identification Performance. Journal of Information Processing, 24(2):247–254, 2016.

[137] V. Moonsamy, J. Rong, and S. Liu. Mining Permission Patterns for Contrasting Clean and Malicious Android Applications. Journal of Future Generation Computer Systems, 36:122–132, 2013.

[138] A. W. Moore and D. Zuev. Internet traffic classification using bayesian analysis techniques. SIGMETRICS Perform. Eval. Rev., 2005.

[139] J. Muehlstein, Y. Zion, M. Bahumi, I. Kirshenboim, R. Dubin, A. Dvir, and O. Pele. Analyzing HTTPS Encrypted Traffic to Identify User Operating System, Browser and Application. arXiv.org, 2016.

[140] M. Müller. Information Retrieval for Music and Motion. Springer, 2007.

[141] C. Mulliner. The Android dynamic binary instrumentation toolkit. `https://github.com/crmulliner/adbi`, 2016.

[142] N. Murtagh, M. Nati, W. R. Headley, B. Gatersleben, A. Gluhak, M. A. Imran, and D. Uzzell. Individual energy use and feedback in an office setting: a field trial. Energy Policy, 2013.

[143] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna. BareDroid: Large-scale analysis of Android apps on real devices. In Proceedings of ACM Annual Computer Security Applications Conference (ACSAC), 2015.

[144] A. Nandugudi, A. Maiti, T. Ki, F. Bulut, M. Demirbas, T. Kosar, C. Qiao, S. Y. Ko, and G. Challen. Phonelab: A large programmable smartphone testbed. In Proceedings of ACM International Workshop on Sensing and Big Data Mining (SenseMine), 2013.

[145] M. Nati, A. Gluhak, H. Abangar, and W. Headley. Smartcampus: A user-centric testbed for internet of things experimentation. In Proceedings of IEEE International Symposium on Wireless Personal Multimedia Communications (WPMC), 2013.

[146] C. Neasbitt, R. Perdisci, K. Li, and T. Nelms. Clickminer: Towards forensic reconstruction of user-browser interactions from network traces. In Proceedings of the ACM conference on Computer and Communications Security (CCS), 2014.

[147] S. Neuner, V. Van der Veen, M. Lindorfer, M. Huber, G. Merzdovnik, M. Mulazzani, and E. Weippl. Enter sandbox: Android sandbox comparison. arXiv preprint arXiv:1410.7749, 2014.

[148] T. T. T. Nguyen and G. Armitage. A survey of techniques for internet traffic classification using machine learning. IEEE Communications Surveys & Tutorials, 10(4):56–76, 2008.

[149] T. T. T. Nguyen, G. Armitage, P. Branch, and S. Zander. Timely and continuous machine-learning-based classification for interactive ip traffic. IEEE/ACM Transactions on Networking (TON), 20(6), Dec. 2012.

[150] E. Novak, Y. Tang, Z. Hao, Q. Li, and Y. Zhang. Physical media covert channels on smart mobile devices. In Proceedings of ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp), 2015.

[151] J. Oberheide and C. Miller. Dissecting the Android Bouncer. SummerCon, 2012.

[152] OpenIntents. Sensor Simulator. `https://goo.gl/n1a9XD`, 2014.

[153] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang. ACCessory: Password Inference using Accelerometers on Smartphones. In Proceedings of ACM Workshop on Mobile Computing Systems and Applications (HotMobile), 2012.

[154] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In Proceedings of USENIX Workshop on Offensive Technologies (WOOT), 2009.

[155] A. Panchenko, F. Lanze, J. Pennekamp, T. Engel, A. Zinnen, M. Henze, and K. Wehrle. Website Fingerprinting at Internet Scale. In Proceedings on Network and Distributed System Security Symposium (NDSS), 2016.

[156] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website Fingerprinting in Onion Routing Based Anonymization Networks. In Proceedings of ACM Workshop on Privacy in the Electronic Society (WPES), 2011.

[157] A. Pathak, Y. Charlie Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with Eprof. In Proceedings of ACM European conference on Computer Systems (EuroSys), 2012.

[158] N. J. Percoco and S. Schulte. Adventures in BouncerLand. Black Hat USA, 2012.

[159] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of Android malware. In Proceedings of ACM European Workshop on System Security (EUROSEC), 2014.

[160] Pragmatic Software. Network Log, April 2014.

[161] J. G. Proakis. Intersymbol interference in digital communication systems. Wiley Online Library, 2003.

[162] G. Procaccianti, L. Ardito, M. Morisio, et al. Profiling power consumption on desktop computer systems. In Proceedings of International Conference on Information and communication on technology for the fight against global warming (ICT-GLOW). Springer, 2011.

[163] Z. A. Qazi, J. Lee, T. Jin, G. Bellala, M. Arndt, and G. Noubir. Application-awareness in SDN. In Proceedings of ACM SIGCOMM, 2013.

[164] E. L. Quinn. Smart metering and privacy: Existing laws and competing policies. Available at SSRN 1462285, 2009.

[165] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators. In Proceedings of International Conference on Information Security (ISC). Springer, 2007.

[166] N. Ramanathan, F. Alquaddoomi, H. Falaki, D. George, C. Hsieh, J. Jenkins, C. Ketcham, B. Longstaff, J. Ooms, J. Selsky, et al. Ohmage: an open mobile system for activity and experience sampling. In Proceedings of IEEE International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2012.

[167] L. Rashidi, S. Rajasegarar, C. Leckie, M. Nati, A. Gluhak, M. A. Imran, and M. Palaniswami. Profiling spatial and temporal behaviour in sensor networks: A case study in energy monitoring. In Proceedings of IEEE International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014.

[168] J.-F. Raymond. Traffic analysis: Protocols, attacks, design issues, and open problems. In Designing Privacy Enhancing Technologies. Springer, 2001.

[169] A. Reinhardt, P. Baumann, D. Burgstahler, M. Hollick, H. Chonov, M. Werner, and R. Steinmetz. On the accuracy of appliance identification based on distributed load metering data. In Proceedings of IEEE Conference on Sustainable Internet and ICT for Sustainability (SustainIT), 2012.

[170] D. Reynolds. Gaussian mixture models. Encyclopedia of biometrics, 2015.

[171] A. Ridi and J. Hennebert. Hidden markov models for ilm appliance identification. Procedia Computer Science, 2014.

[172] S. Rosen, Z. Qian, and Z. M. Mao. Appprofiler: A flexible method of exposing privacy-related behavior in android applications to end users. In Proceedings of ACM Conference on Data and Application Security and Privacy (CODASPY), 2013.

[173] SandDroid. An automatic Android application analysis system. http://sanddroid.xjtu.edu.cn/, 2014.

[174] A. Savitzky and M. J. Golay. Smoothing and differentiation of data by simplified least squares procedures. Analytical chemistry.

[175] R. Schlegel, K. Zhang, X. Y. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In Proceedings or Network and Distributed System Security Symposium (NDSS), 2011.

[176] F. Schneider, A. Feldmann, B. Krishnamurthy, and W. Willinger. Understanding online social network usage from a network perspective. In Proceedings of ACM Internet Measurement Conference (IMC), 2009.

[177] G. G. Sena and P. Belzarena. Statistical traffic classification by boosting support vector machines. In Proceedings of ACM International Latin American Networking Conference (LANC), 2012.

[178] S. Seneviratne, A. Seneviratne, P. Mohapatra, and A. Mahanti. Predicting user traits from a snapshot of apps installed on a smartphone. ACM SIGMOBILE Mobile Computing and Communications Review, 18(2):1–8, 2014.

[179] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum. Livelab: measuring wireless networks and smartphone users in the field. ACM SIGMETRICS Performance Evaluation Review, 2011.

[180] M. Shinohara, M. Murakami, H. Iwane, S. Takahashi, S. Yamane, H. Anai, T. Sonoda, and N. Yugami. Development of an integrated control system using notebook pc batteries for reducing peak power demand. International Journal of Informatics Society, 5(3):109–118, 2013.

[181] D. X. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on ssh. In Proceedings of USENIX Security, 2001.

[182] S. Spiegel, J. Gaebler, A. Lommatzsch, E. De Luca, and S. Albayrak. Pattern recognition and classification for multivariate time series. In Proceedings of ACM Sensor-KDD, 2011.

[183] R. Spolaor, L. Abudahi, V. Moonsamy, M. Conti, and R. Poovendran. No free charge theorem: A covert channel via usb charging cable on mobile devices. In Proceedings of International Conference on Applied Cryptography and Network Security (ACNS), pages 83–102. Springer, Cham, 2017.

[184] R. Spolaor, M. Monaro, P. Capuozzo, M. Baesso, M. Conti, L. Gamberini, and G. Sartori. You are how you play: Authenticating mobile users via game playing. In Proceedings of International Worskhop on Communication Security. Springer, 2017.

[185] R. Spolaor, E. D. Santo, and M. Conti. Delta: Data extraction and logging tool for android. IEEE Transactions on Mobile Computing, 2017.

[186] R. Spreitzer. Pin skimming: Exploiting the ambient-light sensor in mobile devices. In Proceedings of ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM), 2014.

[187] V.-D. Stanciu, R. Spolaor, M. Conti, and C. Giuffrida. On the effectiveness of sensor-enhanced keystroke dynamics against statistical attacks. In Proceedings of ACM Conference on Data and Application Security and Privacy (CODASPY), pages 105–112, 2016.

[188] Statista. Shipment forecast of laptops and desktop pcs. `https://www.statista.com/statistics/272595/global-shipments-forecast-for-tablets-laptops-and-desktop-pcs/`.

[189] Statista. Number of apps available in leading app stores as of June 2016. `https://goo.gl/tCnPXW`, 2016.

[190] T. Stöber, M. Frank, J. Schmitt, and I. Martinovic. Who do you sync you are? In Proceedings of ACM conference on Security and privacy in wireless and mobile networks (WiSec), 2013.

[191] T. Strazzere. Dex education 201 - anti-emulation. `https://goo.gl/jrqaaJ`, 2013.

[192] G. Suarez-Tangil, M. Conti, J. E. Tapiador, and P. Peris-Lopez. Detecting targeted smartphone malware with behavior-triggering stochastic models. In Proceedings of European Symposium on Research in Computer Security (ESORICS). Springer, 2014.

[193] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic reconstruction of Android malware behaviors. In Proceedings on Network and Distributed System Security Symposium (NDSS), 2015.

[194] V. F. Taylor and I. Martinovic. To Update or Not to Update: Insights From a Two-Year Study of Android App Evolution. In Proceedings of ACM Asia Conference on Computer and Communications Security (ASIACCS), pages 45–57, 2017.

[195] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic. AppScanner: Automatic Fingerprinting of Smartphone Apps from Encrypted Network Traffic. In IEEE European Symposium on Security and Privacy (Euro S&P), pages 439–454, 2016.

[196] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic. Robust smartphone app identification via encrypted network traffic analysis. IEEE Transactions on Information Forensics and Security, 2017.

[197] V. Van Der Veen, H. Bos, and C. Rossow. Dynamic analysis of Android malware. Internet & Web Technology Master thesis, VU University Amsterdam, 2013.

[198] N. V. Verde, G. Ateniese, E. Gabrielli, L. V. Mancini, and A. Spognardi. No nat'd user left behind: Fingerprinting users behind nat from netflow records alone. In Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS), 2014.

[199] T. Vidas and N. Christin. Evading Android runtime analysis via sandbox detection. In Proceedings in ACM Asia Conference on Computer and Communications Security (ASIACCS), 2014.

[200] T. Vidas, J. Tan, J. Nahata, C. L. Tan, N. Christin, and P. Tague. A5: Automated analysis of adversarial Android applications. In Proceedings of ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM), 2014.

[201] R. Vollmer. XposedBridge development tutorial. `https://goo.gl/P0piK`, 2016.

[202] D. T. Wagner, A. Rice, and A. R. Beresford. Device analyzer: Understanding smartphone usage. In Proceedings of International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services, pages 195–208. 2014.

[203] Q. Wang, A. Yahyavi, M. Kemme, and W. He. I Know What You Did On Your Smartphone: Inferring App Usage Over Encrypted Data Traffic. In 2015 IEEE Conference on Communications and Network Security (CNS), 2015.

[204] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: Multi-layer profiling of android applications. In Proceedings of ACM Annual International Conference on Mobile Computing and Networking (Mobicom), 2012.

[205] R. Wheatstone. Pippa Middleton's iCloud hacked as 3,000 private photos of her and fiance James Matthews stolen. `https://goo.gl/xnNQ5u`, 2016.

[206] C. V. Wright, L. Ballard, S. E. Coull, F. Monrose, and G. M. Masson. Spot me if you can: Uncovering spoken phrases in encrypted voip conversations. In Proceedings of IEEE Symposium on Security & Privacy (SP), 2008.

[207] C. V. Wright, S. E. Coull, and F. Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In Proceedings on Network and Distributed System Security Symposium (NDSS), 2009.

[208] L. Yan, Y. Guo, X. Chen, and H. Mei. A study on power side channels on mobile devices. In Proceedings of ACM Internetware, 2015.

[209] L. K. Yan and H. Yin. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In Proceedings of USENIX Security, 2012.

[210] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In Proceedings of the ACM conference on Computer and Communications Security (CCS), 2013.

[211] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. AppScope: Application Energy Metering Framework for Android Smartphone Using Kernel Activity Monitoring. In Proceedings of USENIX Annual Technical Conference, 2012.

[212] F. Zhang, W. He, X. Liu, and P. G. Bridges. Inferring users' online activities through traffic analysis. In Proceedings of ACM conference on Security and privacy in wireless and mobile networks (WiSec), 2011.

[213] J. Zhang, C. Chen, Y. Xiang, and W. Zhou. Robust network traffic identification with unknown applications. In Proceedings in ACM Asia Conference on Computer and Communications Security (ASIACCS), 2013.

[214] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In Proceedings of the ACM conference on Computer and Communications Security (CCS), 2013.

[215] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In Proceedings of IEEE Symposium on Security and Privacy (SP), 2012.

[216] A. Zoha, A. Gluhak, M. A. Imran, and S. Rajasegarar. Non-intrusive load monitoring approaches for disaggregated energy sensing: A survey. Sensors, 2012.

[217] A. Zoha, A. Gluhak, M. Nati, and M. A. Imran. Low-power appliance monitoring using factorial hidden markov models. In Proceedings of IEEE International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2013.