KTH Information and Communication Technology

MobiLeak: A System for Detecting and Preventing Security and Privacy Violations in Mobile Applications

Pasquale Stirparo
pasquale@kth.se

Licentiate Thesis in
Information and Communication Technology
Stockholm, Sweden 2013

School of Information and
Communication Technologies
Royal Institute of Technology
Forum 100, SE-16440
Kista, Sweden

TRITA-ICT/ECS AVH 13:22
ISSN 1653-6363
ISRN KTH/ICT/ECS/AVH-13/22-SE
ISBN 978-91-7501-927-7

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till of- fentlig granskning för avläggande av teknologie licentiatexamen i radiosystemteknik tors- dagen den 28 november 2013 klockan 15.00 i Forum, Isafjordsgatan 39, 164 40, Kista, Stockholm

Tryck: Universitetsservice US AB

Table of Contents

List of Figures

List of Tables

1. Introduction and Motivation

Our society is becoming more digitalized. Citizens rely on their mobile devices more and more and the market penetration of smart phones and tablets is growing at extremely high pace [1]. Through smartphones, citizens perform sensitive operations like mobile banking or online shopping: they also access their private documents and emails. Managers and executives use their corporate tablets for important projects and at meetings with customers. Moreover smartphones and tablets, due to their nature, easily get stolen or lost. According to Lookout [2], in 2011 a total of 9 million phones were lost, which is equivalent to having a phone lost every several seconds. Therefore, it is extremely important that data and information stored in these devices are properly handled and not easily accessible to an eventually untrusted third party; at least those data that can be considered as personal or private.

Sales of mobile devices reached 428.7 million units in the second quarter of 2011 [3], and in the very near future the vast majority of mobile phones will be NFC-enabled. According to iSupply [4] there will be 544.7 million NFC-enabled mobile phones by 2015. The organization "in charge" of developing standards-based NFC specification is the NFC Forum [5], which was founded in 2004 by Nokia, Philips and Sony, and now has about 150 members. Currently, NFC technology is almost exclusively used for mobile payments [6] and ticketing. Therefore, it is necessary to develop tools and techniques that assess and improve the security of NFC-devices and services.

The need of focusing on security aspects is underlined in the Lookout Mobile Threat Report [7], where the following emerging patterns, among others, are listed for the mobile malware:

- Malware that acts as a botnet, exposing an array of remotely controlled device capabilities.
- Abuse of premium-rate text messages.
- Targeted attacks aimed at gathering sensitive data for commercial or political purposes.
- Financial fraud as more mobile finance and payment applications emerge.
- Application-based phishing attacks (e.g. fake login/sign up screens).

Considering all of the above, along with the fact that only in the official Android Market [8] there are already hundreds of NFC related applications, the need of solutions to effectively test the robustness of such standards and applications rises exponentially.

This thesis is structured in three main areas. The **first part**, which comprises of Chapters 1 and 2, introduces and describes motivations behind this work. After that it presents **MobiLeak**, the framework for the methodologies that are used or can be used to identify privacy threats, vulnerabilities and information leaks from mobile applications. For this reason, an important preliminary study on the status of data is presented in this part, which is fundamental prerequisite to understand privacy. When talking about privacy, we talk about information and data. There are several aspects that have to be considered when aiming to assess privacy level of an application. To identify privacy leakages, it is important to know all the different states in which data can exist and can be found in a mobile environment. Those states, which have been investigated within ***MobiLeak Project***, are ***Data-at-Rest***, ***Data-in-Use*** and ***Data-in-Transit***. After this study, the methodology and techniques are introduced from where development is derived. Particularly the notion of Mobile Forensics [9][10], mainly referring to Android Forensics [11], Memory Forensics and Analysis [12][13], and Fuzz Testing [14][15][16] principles.

The **second part**, which comprises Chapters 3, 4 and 5, presents an investigation of the problems related to each of the different states described above. Particularly, this part presents practical developments for detection of privacy leakages in two states, notably ***data-at-rest*** and ***data-in-use***, whereas for the ***data-in-transit*** state it is proposed a design of a framework to test the security and privacy of NFC protocol implementation on Android mobile devices. More specifically:

- In relation to data-at-rest state, the structure of Android file system and of Android's applications has been analyzed. From here, a methodology to find privacy leaks have been proposed, based on effective experimental results;
- In relation to data-in-use state, the memory management of the Android operating system, as well as of its applications, has been analyzed. From here, a methodology to find and extract sensitive data from volatile memory has been proposed, based on effective experiment results;
- In relation to data-in-transit state, Bluetooth and NFC protocols have been analyzed. From the Bluetooth analysis of the current state of the art, it came out as conclusion that Bluetooth standard became pretty much stable and most of the security and privacy issues are due to improper implementations. For this reason, our focus turned towards Near Field Communication (NFC) protocol. As a new standard, from the studies the results show that proper tools to effectively test NFC implementation on mobile devices are missing. Therefore, the design of a fuzzing framework for security evaluation of NDEF Message Format is proposed.

In the **third and last part**, which comprises Chapters 6 and 7, the conclusions and proposals of a series of effective countermeasures are presented based on the results of the experiments presented in the second part. Finally, the design of a global solution to identify and counter privacy leaks, based on such results, is described and proposed. Such solution is based on the development of the Critical State Analysis [17] concept implemented for a mobile environment. This design is described in the Future Direction chapter, as it is intended as the final solution and the goal to be achieved at the conclusion of the Ph.D. studies.

2. MobiLeak: Detecting Privacy Leaks in Mobile Applications

This chapter introduces the ***MobiLeak Project***. The idea behind is to investigate and develop methodologies that are used or can be used to perform privacy assessments of mobile applications. As first step, it has been performed a study to identify and characterize all the possible states at which data can exist in a mobile environment. Lastly, the methodologies and techniques to use in order to identify privacy leaks have been selected based on this characterization.

2.1. Setting the scene: nature and life cycle of mobile data

When talking about privacy, we talk about information, about data. There are several aspects that have to be considered when aiming to assess the privacy level of an application. To perform a proper privacy assessment, it is important to know all the different states in which data can exist and can be found in a mobile environment. Those states, which are being investigated in the MobiLeak, are a) Data-at-Rest, b) Data-in-Use and c) Data-in-Transit:

a) ***Data at Rest*** is data stored in storage media, in this case in the mobile device's internal memory or SD card. This data can be regarded as secure, if it is protected by strong encryption, if the key is not present on the media itself, and if the key is of sufficient length and randomness to be functionally immune to a dictionary attack.

b) ***Data in Use*** is all data in the process of being created, retrieved, updated, deleted, or manipulated. Data-in-use is data not in an at-rest state and that resides on only one particular mobile device. This data can be regarded as secure, if access to the memory is rigorously controlled, and if regardless of how the process terminates, data cannot be retrieved from any location other than the original at rest state, requiring re-authorization.

c) ***Data in Transit*** is all data being transferred between two nodes in a network. In this case, one node would be mobile device and the other one could be the GSM network, an access point Wi-Fi providing Internet connection, or another mobile device for a peer-to-peer communication (e.g. via Bluetooth, NFC, etc.). This data can be regarded as secure if the communication between the two nodes is authenticated, authorized, and private, meaning no unauthorized third party can eavesdrop the communication between the two nodes.

2.2. Analysis Methodologies to detect Privacy Leakages

In this section are listed and explained the methodologies identified so far to conduct the research activity exposed in this thesis work. These include techniques of Mobile Forensics, Memory Forensics and Fuzz Testing. Other methodologies and techniques exist, and they will be subject of further research and future activities of this work (see Chapter 7 "Future Direction and Work").

2.2.1. Mobile Forensics

As defined at the first Digital Forensics Research Workshop (DFRWS) in 2001, digital forensics is "*the use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation,* *documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations"* [18]. Mobile Forensics is the digital forensics field of study focusing on mobile devices.

Although the scope of this research is not the forensics field itself, as we are dealing with privacy and not cybercrime, it is anyway clear that the ultimate meaning of digital evidence is data and information. This is indeed the goal of this research, to find (personal/private) information not properly handled by mobile applications, which in turn may lead to privacy leaks. As mentioned in Chapter 1, mobile devices are easily stolen or lost. Therefore, it is important that, even if the device ends up in someone else's hands, our private data will not be accessible even though this potentially untrusted/unauthorized person will have physical access to the device. For this reason it is important that applications handling sensitive data store them in a secure manner, so that our privacy is protected.

Since the full forensics process itself is not the scope of this research, only some of the steps of the forensics methodology have been followed, leaving those mainly related to the investigation part and the information intended as evidence to present in court. The steps considered are:

- Collection, usually intended for both the device and the data. In this case it means following the forensics methodology to properly collect data from the mobile device;
- Identification of the areas and data of possible interest for the investigation. In this case the investigation is represented by the research conducted;
- Analysis of the data identified in the previous step;
- Interpretation of the findings in the context of the application analyzed.

Mobile forensics methodology described above has been mainly used to carry out research related to the data-at-rest state (see Chapter 3).

2.2.2. Memory Forensics

Forensics investigations targeting volatile data that can be found in a systems main memory (RAM) is also known as live forensics [19] or RAM forensics [20]. In this context, live forensics tries to express that the focus lies on a systems' current state that can be obtained by freezing the scene at a specific point in time. This needs to be performed while the system is live and operational. If that is not the case (e.g. because the main power has been removed) the volatile data is lost. The main memory contains the complete state of an operating system, including running and historical processes, open network connections, management data, and personal data.

Memory forensics methodology and techniques have been used to perform research related to the data-in-use state by analyzing the memory of running Android smartphones (see Chapter 4).

2.2.3. Fuzzing

The origin of the term ***Fuzzing*** can be traced back to 1988, when Professor Barton Miller [21] at the University of Wisconsin gave a class project titled *"Operating System Utility Program Reliability - The Fuzz Generator"*. It was the first form of fuzzing and it included sending a stream of random bits to UNIX programs by the use of a command line fuzzer.

The final goal of Fuzz Testing is to verify the robustness, which is defined as "*the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions"* [22], and therefore to find vulnerabilities in any type of system, e.g. software, network protocols, web applications, etc. This is accomplished by creating malformed or semi-malformed data to send to an application in an attempt to cause faults [23]. There are basically two main categories of fuzzers: ***generation-based*** and ***mutation-based*** fuzzers. A generation based fuzzer generates its own semi-valid sessions (or semi-invalid, has the same meaning), while a mutation fuzzer takes a known good network session, file, and so on and mutates the sample (before replay) to create many semi-valid sessions. Mutation fuzzers are typically general purpose fuzzers, while generation fuzzers tend to be more specific [16]. Independently from the goal of the testing or from the type of fuzzer used, the same basic phases always apply [15]:

1. Identify the target
2. Identify inputs
3. Generate fuzzed data
4. Execute fuzzed data
5. Monitor for exceptions
6. Determine exploitability

Fuzzing technique has been used to design a framework for testing the NDEF Message Format of the NFC protocol in the context of analyzing the data-in-transit state (see Chapter 5). As a basis for this part of the research the operational methodology of the Sulley Fuzzing Framework has been adopted, briefly discussed here.

Sulley Fuzzing Framework

There are several fuzzers available, both free and commercial. One of the most known and widely used is Sulley [24][25]. Sulley is a fuzzer development and fuzz-testing framework consisting of multiple extensible components. According to its author, it "exceeds the capabilities of most previously published fuzzing technologies, in a commercial or in a public domain."

Sulley uses a block-based approach [26] to protocol representation, which allows to represent most of the protocols. This approach has many advantages compared to others, because it lets the user shaping the details the protocol or format to fuzz. Remaining in the context of protocol testing, it is possible to describe, for each packet field, its format, if to fuzz it or not and to which extent. Just to give a short example, the following Sulley code describes an email address:

```
s_string("pstirparo")
s_delim("@",fuzzable=False)
s_string("gmail")
s_delim(".",fuzzable=False)
s_string("com")
```

Some of the outputs produced by the framework, when deciding to fuzz such a field, will be the following:

```
     pstirparo@gmail.comAAAAAAAAAAAAAAAAAAAAAAAA
     pstirparo@\%n\%n\%n\%n\%n\%n.com
     \%d\%d\%d@gmail.comAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

3. Development of a Data-at-Rest Analysis

This Chapter presents the results of the first phase of the MobiLeak project, dealing with the analysis of the ***"Data at Rest"*** state. Data in this state is data stored in the phone, either in the internal storage memory or in the external SD card.

3.1. Related work

Most of research works with similar intent have been conducted mainly analyzing data in transit and most of them focusing in particular on the security issues rather than privacy, although sometimes the two aspects are not easily separable.

In [27], Schrittwieser et al. evaluate the security of popular smart phones messaging applications. Although they focus on the security flaws, they also expose several privacy concerns related to the data in transit state.

In [28] Kruegel et al. developed PiOS, a tool that uses a static analysis system to detect leakage of private data by applications for Apple's iOS operating system. PiOS constructs a call graph from the binary and looks at all paths from a privacy source to a *sink* and checks whether private data (such as address book or GPS location) are transmitted to a sink without notifying the user.

In contrast to PiOS static analysis, TaintDroid [29], a real-time information flow analysis tool for Android, uses dynamic taint analysis in order to label privately declared data with a taint mark, audit on track tainted data as it propagates through the system and alerts the user if tainted data aims to leave the system at a taint sink (e.g. network interface).

Focusing more on forensics analysis, in [30] Hoog analyzes Google Wallet [6] Android application. In order to evaluate the security of the application, he first runs a network test attempting a Man-in-The-Middle (MITM) attack; secondly he forensically analyzes data stored on the device and lastly examines system logs.

3.2. Android Data Storage and Structure

In this research, the focus has been put on Android device, since it is the most popular mobile operating system in the market [31]. In [11], Hoog describes in detail the applications data storage directory structure, as well as how data are stored. Basically, applications can store data in any location they wish in the external storage space, which can be either the real SD card or the emulated one. On the other hand, internal data storage is controlled by the Android APIs. Therefore when an application is installed, an internal data storage is saved in a subdirectory of */data/data/<packagename>* (e.g. */data/data/com.android.browser*). While applications are not required to store data files, most do so. Hence, inside the applications' directory other than the standard ones, there are as well those that the developers control. The most common standard directories are:

- *shared prefs*: Shared preferences in XML format;
- *lib*: Custom library files an application requires;
- *files*: Files the developer saves to internal storage;
- *cache*: Files cached by the application; cache files from the web browser or other apps that use the WebKit engine;

- *databases*: SQLite databases and journal files.

Android provides developers with five methods for storing data to a device, to the NAND flash, to the SD card, or to the network. Specifically, the five methods are:

1) Shared preferences. It allows a developer to store key-value pairs of primitive data types in a lightweight XML format;
2) Internal storage. More complicated data structures can be stored in files and saved on the file internal storage. The files are stored in the applications subdirectory and the developer has control over the file type, name, and location.
3) External storage. While files stored on the internal devices storage have strict security and location parameters, files on external storage have far fewer constraints (e.g. no chances to enforce permissions on FAT32 file system). The main external location is */mnt/sdcard/*.
4) SQLite. It is the database format used for structured data storage. The SQLite files are generally stored on the internal storage under */data/data/<package name>/databases*. However, there are no restrictions on creating databases elsewhere.
5) Network. Last but not least, data can be stored remotely sending them out over the network connection. However, this aspect is not of interest for this report.

Although the analysis of Android system logs is an important part in a forensics analysis, for the purpose of this research we will not go into those details. Thus, we will focus on the analysis of data ”produced and consumed” within the realm of the applications themselves.

3.3. Experiment Setup and Methodology Used

For the evaluation of our tests, we used the following methodology steps:

- **Identification of the target application to be tested**. Among several different categories (e.g. Business, Communication, Finance, Shopping, Social, etc.), we identified those that were more likely to deal with personal information among the top 50 free for each category in the Google Play Market [32].
- **Data population.** We have created accounts for some applications or used already existing ones for others and we used these applications for about two months in order to get enough data to cover and fill as much databases as we could.
- **Data acquisition.** Having a rooted Android device, data acquisition can easily be done via Android Device Bridge (adb) [33] as follows:

```
$ adb pull /data/data/ data/
$ adb pull /mnt/sdcard/ sdcard/
```

- **Analysis**. Once the data has been acquired, we proceeded with the analysis. Most of the files that we found inside an application package are of three types: xml files, sqlite databases, binary files. The system used to perform the analysis was a MacBook Pro running Mac OS X 10.7.4, and to analyze those files we used mainly the following free Unix command line tools:

- `sqlite3` is a terminal-based front-end to the SQLite library that can evaluate queries interactively and display the results in multiple formats;
- `hexdump` is a filter which displays the specified files in a user specified format (ASCII, decimal, hexadecimal, octal);
- `tree`, is a recursive directory listing program that produces a depth indented listing of files in a tree- like format
- `strings` looks for ASCII strings in a binary file or standard input. A string is any sequence of four (the default) or more printing characters ending with a newline or a null.

3.4. Experiment Results in form of detected Privacy Leaks

In this section will not provide details about the complete content of each application. Instead, it will present the relevant results for each one of them. The summary of the results is presented in Table 1. The applications tested are listed in alphabetical order.

Box: com.box.android

Box is an online file sharing and Cloud content management service. A mobile version of the service is available for Android, BlackBerry, iPhone, iPad, WebOS, and Windows Phone devices. Notable results for this application are the following:

- In the folder */shared prefs/*, the files *folderIdToFolderName.xml* and *folderIdToParentFolderId.xml* contain the name of the internal folders, their id (0 is the root) and the relation parent-child between them. In this way it is possible to rebuild the entire internal structure of the user's Box account.
- Still in the folder */shared prefs/*, the files *myPreference.xml* and *myPreference<user account>.xml* reveals respectively the user's email address (used as username for the Box account) and the path to where the files have been exported from the Box account to local storage.
- The folder */files/* contains several JSON [34] files, one for each folder in the user's Box account. The file names are in the form of JSON static model <username> <folder *id*>. Such file reveals information like creation and update timestamps, file/folder name, files contained in case of a folder, but the most relevant information is probably the *mSharedLink* parameter present in the JSON file related to the root folder. For each folder in the root folder, there is one instance of the *mSharedLink* parameter. Box account default setting is "Link Access: Open", which means that anyone with the link can have access to it and no login is required. Therefore with the information inside this JSON file we have been able to browse through the whole Box account and to download a copy of every file.
- The folder */sdcard/Android/data/com.box.android/cache/working/* contains folders with the files inside (in clear text) that have been made available offline. Such folders are named in the form of <10-digit-id> <download *timestamp*>. The 10-digit-id is also possible to retrieve in the file */shared prefs/DOWNLOAD SALTS.xml*.

Dropbox: com.dropbox.android

Dropbox is one of the most popular files hosting service, operated by Dropbox Inc., which offers cloud storage, file synchronization, and client software. Notable results for this application are the following:

- Inside the database */databases/db.db*
    - Table *thumbnail_info*: it contains the path and name of all the image files.
    - Table *dropbox*: the field *data* contains the path of the file inside the device, the field *path* contains the path related to the internal dropbox structure, the field *display name* is the file name and the field *local modified* refers to when the file has been modified on the device.
    - The table *DropboxAccountPrefs* related to the database */databases/prefs.db* contains full name and email address of the dropbox account owner.
    - Finally, all the documents accessed are stored in clear under */sdcard/Android/data/com.dropbox.android/files/scratch/,* where the complete structure of the dropbox drive is recreated.

eBay: com.ebay.mobile

eBay Inc. is a multinational Internet consumer-to-consumer corporation that manages eBay.com, an online auction and shopping website where people and businesses buy and sell a variety of goods and services worldwide. Notable results for this application are the following:

- The file */shared_prefs/com.ebay.dataservice.prefs.xml* reveals the usernames related to all accounts that logged via the application, as well as information about their shipping address.
- Finally, the table *suggestions* from the database */databases/suggestions.db* contains the item searched (with the keywords) through the application and the related timestamp.

Evernote: com.evernote

Evernote is a suite of software and services designed for note taking and archiving. A "note" can be a piece of formatted text, a full webpage or webpage excerpt, a photograph, a voice memo, or a handwritten "ink" note. Notes can also have file attachments. Notable results for this application are the following:

- The file */shared_pref/com.evernote_preferences.xml* contains many interesting information such as *userid, username, last user refresh time, email* address, *last launch time, SYNC STATUS MSG* in the format of "Last sync: Month Day Time", *evernote email* address, *LAST DB FILEPATH* that shows the path where the actual notes are, which is not inside the application folder, but in **/mnt/sdcard/Evernote/.external-<last *db changed timestamp>-Evernote.db*** , *SIM OPERATOR, LAST SYNC COMPLETED, LAST SYNC STARTED.* All time and date information are expressed in Unix Epoch Time format[1]. These values have been verified to correspond to our activities, by converting them with the following shell command:

```
$ date -r 1342608184
Wed Jul 18 12:43:04 CEST 2012
```

The file contains also the field *encrypted_password* with a 20 char/digit long encrypted password value. The verification of the encryption algorithm used has not been taken into account, since this would fall into security issues, which are out of the scope of this work.

- The other folder created by the Evernote application is */mnt/sdcard/Evernote*. As data of interest it contains a folder *notes* and, as already listed above, the hidden database file.
- By accessing the database */mnt/sdcard/Evernote/.external-<last db changed timestamp>-Evernote.db* we can get plenty of information:
    - Table *notes*: it contains *title* of the note, timestamp values about when it has been *created,* last *updated* and *deleted,* if it is still *active* (boolean value 0|1). These last two parameters mean that it is possible to have access even to notes that have been deleted by the user, but actually still stored in the phone. Other interesting fields are *source_url, guid* and *notebook_guid.* These last two fields are useful to correlate with information present into other tables and to retrieve the content of each note from the *notes* folder (see later).
    - Table *note_tag*: it contains a *tag_guid* for each note *guid,* so it is possible to know which tag has been assigned to each note.
    - Table *resources*: Evernote allows users to add external resources, such as images, to their notes. This table contains information about this file such as *note_guid, latitude, longitude* and *altitude,* which in case of picture with EXIF (Exchangeable Image File format) [35] metadata allows to retrieve the place where the picture has been taken.
    - Table *search_index_content*: the field *c0note_guid* contains the guid of the related note; the field *c1content_id* can have as value either "meta" or "enml". The next field *c3keywords* will contain the keywords/tags of the note if the content id value is "meta" or else it will contain the actual content of the note, if the content id value is "enml".
    - Table *notebooks*: it contains the *guid* and the *name* of each notebook.
    - Table *tags_table*: it contains the id (*guid* field) and the *name* of every tag defined by the user.

[1] The Unix epoch (or Unix time) is the number of seconds that have elapsed since January 1, 1970 (midnight UTC/GMT), not counting leap seconds (in ISO 8601: 1970-01- 01T00:00:00Z).

Groupon: com.groupon

Groupon is a deal-of-the-day website that features discounted gift certificates usable at local or national companies. Notable results for this application are the following:

- The file */shared_prefs/com.groupon.preferences.xml* contains location information such as
    - *LFT SP KEY LAST LOCATION UPDATE LAT (38.298866);*
    - *LFT SP KEY LAST LOCATION UPDATE LNG (16.318663);*
    - *LFT SP KEY LAST LOCATION UPDATE TIME (1342892851430)*, expressed in Unix epoch time format.

  It is therefore possible to find the exact location of the user at that time, by simply querying Google Maps *https:// maps.google.com/maps?q=38.298866,16.318663.*
- In the database file */databases/groupon.db*, the table *blob provider data* stores (using JSON data format) several personal information such as, name, surname, email address, last four digits of the credit card and its expiration date, billing address, etc.

```
{"user": {"id":"8-digits-field",
"firstName":"Pasquale", "lastName":"Stirparo",
"primaryEmailAddress":"pstirparo@gmail.com",
"emailAddresses":[{"address":"pstirparo@gmail.com"}],
"referralCode":"OjXCtk", "rewardPointsAvailable":0,
"creditsAvailable": {"formattedAmount":"0,00
?","amount":0,"currencyCode": "EUR"},
"billingRecords": [{"billingRecordId":"16-digits-field",
"paymentType":"creditcard", "firstName":"Pasquale",
"lastName":"Stirparo", "streetAddress1":"Sreet_Address",
"streetNumber":"Street_Number", "postalCode":"Code",
"city":"Town_Name", "country":"ITALY",
"cardNumber":"************XXXX", "expirationMonth":"M",
"expirationYear":"YYYY"}],
"merchants":[],"birthday":"","gender":"m"}}
```

Kiwi Local: com.kiwirobotics.kiwi.android

Kiwi Local is a new social mobile application. It is a local digital board where people in the same place share profiles and messages. It allows the user to check if, in a close distance, he can meet friends or someone with his interests using the function called *People Nearby*. Being a new application, it was difficult to find people around and therefore populate it properly.

```
$ tree com.kiwirobotics.kiwi.android/
com.kiwirobotics.kiwi.android/
|---- shared_prefs
    |----- activation.xml
    |----- c2dm.xml
    |----- chatting_with.xml
    |----- credentials.xml
    |----- eula.xml
    |----- first_time_view.xml
    |----- profile.xml
    |----- session.xml
    |---- slidingdrawer.xml
1 directory, 9 files
```

However, as it may look clear from the structure and file names listed above, some files may contain stored in clear text a lot of personal information. This seems even more plausible looking at the only file we have been able to populate just by registering and logging in on the application, *credentials.xml:*

```
shared_prefs $ cat credentials.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<boolean name="remember" value="true" />
<string name="userName">EMAIL ADDRESS</string> <
string name="password">PASSWORD CLEAR TEXT</string>
</map>
```

LinkedIn: com.linkedin.android

LinkedIn is a social networking website for people in professional occupations and it is mainly used for professional networking. This package is basically empty from a content point of view: one xml file with no personal information in it, and empty databases. This can be explained with the fact that the application has been entirely rewritten in HTML5, so it is not a native application like the others. This case would require a sort of browser-forensics analysis, to check whether information is left in logs and temporary files, instead of using the approach adopted for this paper.

MyMonster: com.monster.android.Views

Monster is one of the largest employment websites in the world, owned and operated by Monster Worldwide, Inc. Monster is primarily used to help those seeking work to find job openings that match their skills and location. Notable results for this application are the following:

- The file */shared_prefs/com.monster.android/View_preferences.xml* contains stored in clear text *Password, UUID, Email* address, which is used also as username, *lastUpdatedNotification* in the form of MM/DD/YY hh:mm AM|PM.
- The database */databases/Monster* has the following interesting tables:
  - Table *Resume*: it contains CVs and saved jobs, it could reveal private information and jobs interests even for those where *visibilitystatus* is set as "Private".
  - Table *savedjobs*: it contains details about all saved jobs, such as location, job title, company name, creation date, expiration date, etc.
  - Table *users*: it contains name, surname, email address and userid of all the application's users.
  - Table *CoverLetter*: it contains userid, letter title and letter body in clear text.

PayPal: com.paypal.android.p2pmobile

PayPal is a global e-commerce business allowing payments and money transfers to be made through the Internet. The analysis of this package did not reveal much private information. The following is the output of the command `tree`:

```
$ tree com.paypal.android.p2pmobile/
com.paypal.android.p2pmobile/
|----- files
| |----- DeviceReferenceToken
|----- shared_prefs
    |----- com.paypal.android.p2pmobile.xml
2 directories, 2 files
```

The only file to analyze, for the purpose of this work, is *com.paypal.android.p2pmobile.xml,* which contains name, surname and email address of the account owner.

Skype: com.skype.raider

Skype is a proprietary voice-over-Internet Protocol service and software application. It allows users to communicate with peers by voice, video, and instant messaging over the Internet. Phone calls may also be placed to recipients on the traditional telephone networks. Notable results for this application are the following:

- The following command reveals the Skype account username

  ```
  $ cat /files/config.properties
  lname=<username>
  ```

- For each account it creates two files, *<username>.ser* and *<username>.prop*, under the *file* folder. In the same folder it also creates a folder with the account name, where all the databases files related to that account will be placed.
- The following command gives as results usernames and real names of people the user has been in contact with

  ```
  $ hexdump -C /files/<username>.ser | less
  ```

- The following command reveals the timestamp of the first login, in Unix Epoch time format

  ```
  $ cat /files/<username>.prop
  firstLoginTimestamp=1342461507175
  ```

- The following command reveals the field *<Sync-Set><u>,* which contains a list with usernames of all Skype contacts

  ```
  $ cat /files/<username>/config.xml
  ```

- With the following command it is possible to retrieve contact names and pieces of conversations

  ```
  $ strings /files/<username>/main.db-journal
  ```

  When a database file with *db-journal* extension is present with a non-zero size, this indicates that an operation in the database (e.g. update, delete, insert, etc.) didn't complete successfully (it indicates also that the database uses

Rollback Journal mechanism). Therefore, it can be possible to retrieve some old data entry in the journal file, which would not be present in the updated current version of the database (the file with *.db* extension).

- Finally, analyzing the database */files/<username>/msn.db* with sqlite3, plenty of information in clear text has been found, such as:
    - Table *Contacts*: reveals user's contacts personal information, like skypename, fullname, birthday, last time online, and other profile personal info.
    - Table *CallMembers*: reveals the caller or the callee, call duration (in seconds), starting time (Unix timestamp).
    - Table *Calls*: contains begin timestamp, duration of calls.
    - Table *Accounts*: contains all the personal information from Skype User Profile.
    - Table *Transfers*: reveals which files have been sent/received, timestamp, and their location.
    - Table *Messages*: contains the text messages sent and received, and their timestamp.
- All the files received via Skype are stored under */sdcard/Download/* in clear text.

TweetDeck: com.thedeck.android.app

TweetDeck is a social media dashboard application for management of Twitter and Facebook accounts. In May 2011 Twitter acquired TweetDeck. Notable results for this application are the following:

- The file */shared_prefs/account0.xml* reveals the user id, twitter username, link to the avatar/picture for the account zero. In fact, a file named accountX.xml is created for each TweetDeck account logged in the application, where X is an increasing digit value.
- The file */shared_prefs/accounts.xml* contains information

```
$ cat accounts.xml
<?xml version='1.0' encoding='utf-8'
standalone='yes' ?>
<map>
  <int name="twitter" value="0" />
  <string name="td_pass">PASSWORD_CLEAR</string>
  <int name="next_id" value="1" />
  <string name="td_user">USERNAME@PROVIDER.com
  </string>
</map>
```

- Inside the database */databases/tweetdeck.db*, the table contacts contains the list of all twitter contacts for the related account.

Twitter: com.twitter.android

Twitter is an online social networking and micro-blogging service, which enables its users to send and read text-based messages of up to 140 characters, known as "tweets". Notable results for this application are the following:

- Inside the folder */shared_prefs/*, the files *ConnectActivity.xml, HomeActivity.xml* and *HomeTabActivity.xml* all leak the username.
- The file */shared_prefs/c2dm.xml*, which contains information related to the "Cloud to Device Messaging" push notification service, reveals the last refresh time and date (expressed in Unix Epoch Time format) at *last_refresh.<username>,* as well as the registration id *reg_id* that could have security implications. We will not get into details of this last point, since it is out of the scope of this report.
- The file */shared_prefs/com.twitter.android_preferences.xml* reveals the *last_sync* date and time, and the *client_uuid.* The file */shared_prefs/profile_prefs.xml* contains the last refresh time and date (expressed in Unix Epoch Time format).
- The */database/* folder, other than the general databases, contains one database for each account name that has as file name the account id. Therefore in the file *<user id>.db,* we found:
    - Table *users*: it contains user id, username, real name, description and other info about the profile owner and hundreds (not all though) of his contacts.

```
databases $ sqlite3 110177863.db
SQLite version 3.7.10 2012-01-16 13:28:40
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .mode line
sqlite> SELECT * FROM users WHERE _id=1;
_id = 1
user_id = <USER_ID>
username = pstirparo
name = Pasquale Stirparo
description = Digital Forensics & Mobile Security
researcher
web_url = http://it.linkedin.com/in/pasqualestirparo
bg_color = -4137235
location =
protected = 0
verified = 0
followers = 219
friends = 563
statuses = 1307
geo_enabled = 0
profile_created = 1264955811000
image_url =
https://si0.twimg.com/profile_images/951092759/foto_tesser
a_normal.jpg
hash = 1864837700
updated = 1342889001000
friendship = 0
friendship_time =
favorites = 82
image = ?PNG
```

    - Table *statuses*: it contains tweets and related author_id. In case of private twitter account, tweets are of course revealed anyway.
    - Table *messages*: It contains all the private messages. The field *Type 0|1* discriminates if it was sent or received, the field *content* has the actual text of the message, the field *created* has the timestamp in Unix Epoch format of when the message has been sent or received, the fields *sender_id* and *recipient_id* contain the users_id. Therefore by looking up with the "users" table is it possible to determine who is who.
- In the database */databases/global.db*, the table *user_values* contains the account information related to the users of the app.

3.5. Summary and Analysis

The results of this first MobiLeak campaign, summarized below in Table 1, are quite clear and scary at the same time considering that all the companies behind the applications analyzed invest a lot in the security and privacy of their services.

The analysis shows that 25% of the applications reveals password, while all of them save the username in clear text although this could be considered as minor issue. The results become more interesting in terms of privacy leaks when it comes it personal information. In fact 83% of applications reveals personal information of the users, like home address, phone number, credit card number, etc., while 67% leaks activity information, such as last time logged in, location coordinates of last time the user logged in the applications, calls duration, etc. Of all the applications analyzed, only 58% was dealing with users documents (managing of file of any kind, text, picture, etc.) but 86% of this subgroup (50% of all applications analyzed) store documents in unprotected or shared areas, such as the sdcard.

The goal of this phase of the research was twofold: prove that still too many applications store personal information in clear text and demonstrate that principles of an already known methodology (the forensics one) can be used to develop a methodology for the purpose of privacy assessment, therefore increasing the option for assessing the privacy level of an application.

APP	VERSION	CATEGORY	USER NAME	PASSWORD	PERSONAL INFO	ACTIVITY INFO	DOCS
Box	1.8.1	Productivity	Yes	No	Yes	Yes	Yes
Dropbox	2.1.6	Productivity	Yes	No	Yes	Yes	Yes
eBay	1.8.1.5	Shopping	Yes	No	Yes	Yes	n/a
Evernote	4.0.4	Productivity	Yes	No	Yes	Yes	Yes
Groupon	2.0.22.65	Shopping	Yes	No	Yes	Yes	Yes
Kiwi Local	1.11.3	Social	Yes	Yes	n/a	n/a	n/a

Linkedin	2.4.3	Social	No	No	No	No	n/a
MyMonster	1.6	Business	Yes	Yes	Yes	Yes	Yes
PayPal	3.3.0.0	Finance	Yes	No	Yes	No	No
TweetDeck	1.0.7.4	Social	Yes	Yes	Yes	No	n/a
Twitter	3.2.2	Social	Yes	No	Yes	Yes	n/a
Skype	2.8.0.920	Communication	Yes	No	Yes	Yes	Yes

Table 1: Overview of the selected applications and the overall results based on the type of data leaked

4. Development of Data-in-Use Analysis

This chapter presents the results of the second phase of the MobiLeak project, dealing with the analysis of the *"Data in Use"* state. Data in this state is data found in the memory of the phone, i.e. data that is currently or has recently been manipulated. Sensitive information found in the *data in use* state can be exploited by malware in various ways. For example, the most direct exploration would be to impersonate user's identity by finding his/her account's credentials, i.e. username and password.

For this part of the research, Android has again been targeted as the underlying operating system. The main motivation behind this decision was to exploit the open nature of Android and have direct access to the kernel and the source files that were needed for our research. Moreover, we wanted to target the most used mobile operating system in order for our research to be applicable to the largest number of mobile users and Android is at the moment the most sold operating system, gaining a 75% market share in the third quarter of 2012 [36]. Finally, recent numbers show that Android is increasingly targeted by malware [37], making it even more important to identify potential misuse of personal and sensitive information that malware may have access to.

The analysis focused mainly on two different groups of applications: mobile banking applications and a subgroup of some of the most popular Android applications. The first category is a very sensitive one: being able to recover the credential for online banking represents a high risk for the user. Potential leakage of information from a bank transaction will have a direct economical impact on the user that could be much higher than having a credit card lost or stolen. The reasons behind the choice of the second category were mainly two: on one hand we wanted to give a continuity to the first phase of MobiLeak [38], testing also the same applications analyzed in Phase 1 against Data in Use threats and, on the other hand, this group of applications represents some among the most popular mobile applications. Therefore privacy issues affecting them will automatically affect millions of users.

4.1. Related Work

As Android is based on Linux, most of the relevant work for this research has been performed for both Android and Linux memory analysis areas.

Linux Memory Analysis

Traditionally it was possible to perform memory captures on Linux by accessing the `/dev/mem` device. Such device contained a map of the first gigabyte of RAM and allowed acquisition only of the first 896 MB of physical memory, without the need to load code into the kernel. However, due to security concerns, the `/dev/mem` device has recently been disabled on all major Linux distributions, as it allowed reading and writing of kernel memory. To overcome these problems, Kollar created *fmem* [39], a loadable kernel module that creates a `/dev/fmem` device supporting memory capture. However, this solution appears to be unsuitable for Android, since it makes use of some kernel functions that are not available on ARM [12].

In [20] Urrea describes the case of a specific Linux distribution by outlining kernel structures relevant for memory management that can be used to retrieve corresponding data. In his solution he uses the `dd` tool to read the physical memory from `/proc/mem`.

Andrew Case has done an extensive amount of work in the field of memory forensics, making a lot of effort to extract forensically relevant information from memory captures [40] and to perform deep analysis of Linux kernel data structures as well as userland information [41][42]. Although all these works have been able to gather numerous objects and data structures from memory, a shortcoming is their inability to deal with the vast number of Linux kernel versions and the large number of widely used Linux distributions.

Android Memory Analysis

In [13] Thing, Ng and Chang focus on capturing a specific, running processes, using the `ptrace()` functionality of the kernel to dump specific memory regions of a process, instead of capturing the whole physical memory of Android. Virtual memory captures are then analyzed to discover evidence. This approach requires, obviously, memory to be extracted separately for each process of interest.

In [12] Sylve, Case, Marziale and Richard present methods that obtain complete captures of volatile memory from Android devices, along with subsequent analysis of that data in both userland and the kernel. They developed kernel module, DMD, to perform full dump of the device memory, as well as kernel analysis support for the Volatility framework [43][44], implementing ARM-specific support. However, this solution doesn't solve one common problem all modules have, which is security mechanism of the kernel called module verification. The purpose of this mechanism is to prevent the kernel from loading incompatible or possibly malicious code into the operating system. Therefore, since it is not possible to load a module in a kernel-agnostic way, an alternative solution is to create a pool of precompiled modules against a specific kernel, which basically would mean one module for each device and Android version. This is feasible only for those devices for which the corresponding vendor releases the kernel source code together with its build configuration. As explained in more details later, our acquisition of the memory dump is based on this work.

In [45] Leppert discusses several methods to generate heap dumps from the first Android version till 2.2 (Froyo), and from Android 2.3 (Gingerbread) till version 4.0 (Ice Cream Sandwich). These methods are based on old Android versions and rely on tools no longer available. Moreover, his solution is not applicable to commercial applications bought or downloaded from an application market and therefore not scalable. In fact acquiring a heap dump is only possible for applications prepared for debugging. When developing Android applications there is a flag called `android:debuggable` in the applications configuration file, the `AndroidManifest.xml`. If that option is set to *true*, it causes the application to open a debug port that can be used by DDMS [46] to acquire a heap dump from the application running on a device, which has to be physically connected to a computer. This usually means that the application is still in testing phase, since from the moment the application is available online the `android:debuggable` flag is supposed to be set to *false*.

In [47] Macht takes Android Live Memory Forensics to the next level by performing a deep analysis of the Linux kernel, the Dalvik Virtual Machine and a chosen set of applications. The result is a set of plugins that extend the Volatility Framework, to read data such as user names, passwords, chat messages, and emails chosen from a set of target applications.

4.2. Android Memory Management

We can generalize memory organization as a structure divided into three main areas, also known as segments: the text segment, the stack segment, and the heap segment. The **text segment,** or code segment, is where the compiled code of the program itself resides. The **stack segment** is where local variables are stored and it is also used for passing arguments to functions along with the return address of the instruction which is going to be executed after the function call is over. When a new function is called and a new stack frame needs to be added, the stack grows downward. Finally the **heap segment,** or data segment, is the area of memory that gets allocated at runtime and therefore contains the variables that are defined during the program execution and their value. When more memory needs to be allocated, the heap grows upward.

Android is an open source, Linux-based operating system primarily designed for mobile devices such as smartphones and tablets. Previously based on the Linux kernel version 2.6, it uses version 3.x starting from Android 4.0 Ice Cream Sandwich onwards. Android uses the Dalvik virtual machine with just-in-time compilation to run Dalvik ‘dex-code’ (Dalvik Executable), which is usually translated from Java bytecode. DalvikVM is an interpreter for the Java programming language. It is similar to the Java Virtual Machine (JVM), but it has been specifically designed to operate in embedded environments.

In Android every application runs as a separate process, which has its own instance of the DalvikVM. As Android is a multi-user Linux system, in which each application is a different user, the system by default assigns to each application a unique Linux user ID (the ID is used only by the system and is unknown to the application). The system sets permissions for all files in an application, so that only the user ID assigned to that application could access them. Android has at its core the “Zygote” process, which starts up at *init*. When you start an application, the Zygote is forked, and the Dalvik heap is preloaded with classes and data by Zygote. Dalvik, like virtual machines for many other languages, does garbage collection (GC) on the heap [48]. Garbage collection is a form of automatic memory management that attempts to reclaim garbage or memory occupied by objects that are no longer in use by the process.

Android memory management involves freeing objects from memory when they are no longer needed and assigning memory to processes that require it. As stated in the Android Developers portal [49], in order to determine which processes to keep and which to kill, the system places each process into an “importance hierarchy" based on the components running in the process and their state. Processes with the lowest importance are eliminated first, then those with the next lowest importance, and so on, as necessary to recover system resources. There are five levels in the importance hierarchy. The following list presents the different types of processes in order of importance (the first process is the most important and is killed last):

1. Foreground process. A process that is required for what the user is currently doing. Generally, only a few foreground processes exist at any given time. They are killed only as a last resort, if memory is so low that they cannot all continue to run.
2. Visible process. A process that doesn't have any foreground components, but still can affect what the user sees on screen. A visible process is considered extremely important and will not be killed unless doing so is required to keep all foreground processes running.
3. Service process. This process started some non-active services that do not interact directly with the user.
4. Background process. A process holding an activity that is not currently visible to the user (the activity's `onStop()` method has been called). These processes have no direct impact on the user experience and the system can kill them at any time to reclaim memory for a foreground, visible, or service process. Usually there are many background processes running.
5. Empty process. A process that does not hold any active application components. The only reason to keep this kind of process alive is for caching purposes, to improve startup time the next time a component needs to run in it.

As one could imagine, the management of the memory plays a key role under a privacy perspective, as all the user's data needed by an application to fulfill its functions will be stored somewhere in the smart-phone memory.

4.3. Experiment Setup and Methodology Used

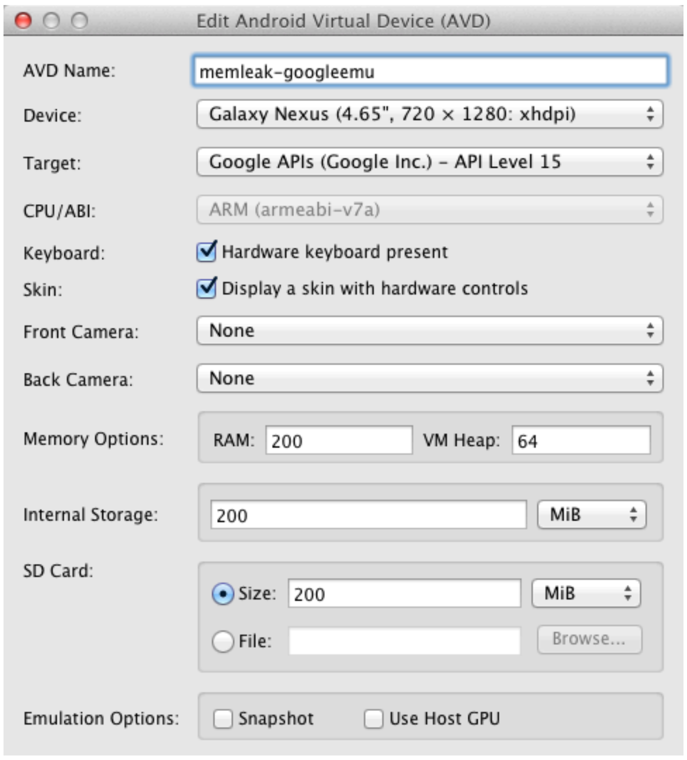
The scope of the experimental campaign presented in this chapter was to analyze the memory content of a smartphone (a) while using a certain application and (b) just after closing the target application. The focus of the analysis was on two different groups of applications: mobile banking applications and a subgroup of some of the most popular Android applications. For each application, two types of tests have been performed. In the first one we took a dump of the memory immediately after hitting the "login" button, while the application was still running. In the second test, the memory dump was taken right after terminating the application. While it might be relatively "normal" to find sensitive data in memory during the application execution, still being able to retrieve the same data after the application has been terminated would highlight a bad memory management procedure either by the application developer or by the Android team, or both. Table 2 summarizes the steps taken for each of the two types of tests.

STEP	TEST 1	TEST 2
1	Reboot phone	Reboot phone
2	Launch the app	Launch the app
3	Login	Login
4	Memory dump	App is running
5	Finish	Quit app
6	//	Memory dump
7	//	Finish

Table 2: Step-by-step procedure for the two types of tests

As testing environment we used the Android emulator [50] version 21.1.0, which requires Android SDK [51] and Android NDK [52] to be properly installed and initialized. We then used the LiME kernel module cross-compiled against the device kernel to dump the memory of the emulated device and the Volatility Framework to analyze the dumps. We will explain later what LiME and Volatility exactly are.

After setting up the environment, in order to create the virtual device we used Android Virtual Device Manager (AVD), choosing as device to be emulated a Galaxy Nexus (see Figure 1).


Edit Android Virtual Device (AVD)

AVD Name:	memleak-googleemu
Device:	Galaxy Nexus (4.65", 720 × 1280: xhdpi)
Target:	Google APIs (Google Inc.) – API Level 15
CPU/ABI:	ARM (armeabi-v7a)
Keyboard:	✓ Hardware keyboard present
Skin:	✓ Display a skin with hardware controls
Front Camera:	None
Back Camera:	None
Memory Options:	RAM: 200 VM Heap: 64
Internal Storage:	200 MiB
SD Card:	● Size: 200 MiB; ○ File:
Emulation Options:	☐ Snapshot ☐ Use Host GPU

Figure 1: Parameter used to generate Android Virtual Device

As target platform we chose the Google API version 15, corresponding to Android 4.0.3 *Ice Cream Sandwich*, because it is the most used of the Android versions based on the new kernel and the second most deployed version on the market in general. Once the virtual device has been defined, we need to get the kernel source code from the device manufacturer's website. In our case, since we are using the virtual device, we can use the Android emulator source code, code name *Goldfish*. We selected the kernel version 2.6.29 with the following steps:

```
$ git clone https://android.googlesource.com/kernel/
      goldfish.git ~/android-goldfish
$ cd ~/android-goldfish/
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/android-goldfish-2.6.29
  remotes/origin/android-goldfish-3.4
  remotes/origin/linux-goldfish-3.0-wip
  remotes/origin/master
$ git checkout -t remotes/origin/android-goldfish-2.6.29 -b goldfish
```

As last step of this preliminary setup phase, we need to cross compile the kernel source code for our system, which runs on an ARM architecture. To do so we first need to set the following environment variables:

```
$ export ARCH=arm
$ export SUBARCH=arm
$ export CROSS_COMPILE=arm-eabi-
```

and then we can complete the compilation with the following commands:

```
$ make goldfish_armv7_defconfig
$ make
```

If everything went well, we should be able to run the virtual device created earlier with the kernel we just compiled.

```
$ cd <path-to-android-sdk>/sdk/tools/
$ ./emulator -avd <virtual device name> -kernel ~/android-
        goldfish/arch/arm/boot/zImage -show-kernel -verbose
```

It has to be noticed that an original *.config* file is needed in case we want to compile the kernel for a real Android device. In such case we first need to export the *config.gz* file from the device, which unfortunately is not always possible, and then decompress it and place the *.config* file exported in the kernel source folder. The *config.gz* file can be exported using the following command:

```
$ adb pull /proc/config.gz
```

4.4. Identification of Target Memory Range and Data Collection

As mentioned in [45], until Android version 2.2 (Froyo) one of the methods to obtain the heap dump was to send a SIGUSR1-signal to the process. In the DavlikVM the SIGUSR1-signal is defined to force the Garbage Collector and `hprof` heap dumping. The easiest and most common way to do that was to send the SIGUSR1-signal to the process using the `kill` command:

```
# kill -10 <PID>
```

This `kill` command will trigger the system to force GC and dump the process heap. Accessing the system log via `logcat`, it would show something like the following:

```
I/dalvikvm(\emph{PID}): SIGUSR1 forcing GC and HPROF dump
I/dalvikvm(\emph{PID}): hprof: dumping VM heap to
 "/data/misc/heap-dump-tmYYYYYYYYYY-pidXXX.hprof-hptemp".
I/dalvikvm(\emph{PID}): hprof: dumping heap strings to
 "/data/misc/heap-dump-tmYYYYYYYYYY-pidXXX.hprof".
```

The above described action was possible until Android version 2.2 [53]. As discussed earlier in the report, in order to acquire the memory dumps for further analysis we used Linux Memory Extractor, a.k.a. LiME [54][55]. As explained by Sylve in the project's documentation, LiME (formerly DMD [12], see Section 4.1) is a Loadable Kernel Module (LKM), which allows the acquisition of volatile memory from Linux and Linux-based devices, therefore Android systems too. To the best of our knowledge, at the time of writing, LiME is the first and currently the only tool that allows full memory captures from Android devices. Thanks to its nature of Loadable Kernel Module, LiME also minimizes its interaction between user and kernel space processes during acquisition. In order to acquire the physical memory from the operating system, the LiME module a) parses the kernel structure called *iomem_resource* to get the physical memory address ranges, b) performs virtual to physical address translation for each memory area, and c) reads all pages in each range from RAM. To complete the LiME module preparation for our dump, we need to point the *makefile* variables *KDIR* and *CCPATH* to the kernel directory and Android cross compiler for ARM in the NDK directory respectively. Once the module has been compiled, we first load it into the device:

```
$ adb push lime.ko /sdcard/lime.ko
$ adb forward tcp:4444 tcp:4444
$ adb shell
$ su
# insmod /sdcard/lime.ko "path=tcp:4444 format=lime"
```

and then we setup `netcat` on listening mode in the host computer:

```
$ nc localhost 4444 > ram-dump.lime
```

LiME requires two compulsory parameters that are *path*, which takes either a filename to write on the local system (SD Card) or tcp:<port> and *format*, which can have one of the following values:

- *raw*, simply concatenates all System RAM ranges;
- *padded*, pads all non-System RAM ranges with 0s, starting from physical address 0;
- *lime*, each range is prepended with a fixed-sized header which contains address space information.

We have chosen *lime* as dump format since Volatility address space has been developed to support this format. To analyze the memory dumps, we used the Volatility Framework [43][44], an open source collection of tools implemented in Python, for the extraction of digital artifacts from volatile memory (RAM) samples. The first step in order to use Volatility is to create a device profile, as explained in the project's webpage documentation. Basically, a device profile is a zip file with information on the kernel's data structures and debug symbols, which is used by Volatility in order to locate and parse critical information. To create kernel's data structures (vtypes) we need to compile *'module.c'*, present in the volatility source code under *'tools/linux'*, against the kernel we want to analyze. We first customize the *Makefile* in the folder related to the *'module.c'* file:

```
obj-m += module.o
KDIR := /Volumes/android-fs/android-sources/goldfish/
CCPATH := ~/android-ndk-r8d/toolchains/arm-linux-
     androideabi-4.7/prebuilt/darwin-x86/bin
-include version.mk
all: dwarf
dwarf: module.c
        $(MAKE) ARCH=arm CROSS_COMPILE=$(CCPATH)/arm-linux-
        androideabi- -C $(KDIR) CONFIG_DEBUG_INFO=y M=$(PWD)
        modules /Tools/dwarf-20130207/dwarfdump/dwarfdump -di
        module.ko > module.dwarf
```

Then we run the `make` command, which will produce the file *module.dwarf*:

```
$ cd volatility/tools/linux
$ make
$ head module.dwarf

.debug_info

<0><0x0+0xb><DW_TAG_compile_unit> DW_AT_producer<GNU C 4.7>
DW_AT_language<DW_LANG_C89> DW_AT_name </volatility/tools/
linux/module.c> DW_AT_comp_dir</Volumes/android-fs/android-
sources/goldfish> DW_AT_stmt_list<0x00000000>
<1><0x1d><DW_TAG_base_type> DW_AT_byte_size<0x00000004>
DW_AT_encoding<DW_ATE_unsigned> DW_AT_name<long unsigned int>
<1><0x24><DW_TAG_pointer_type> DW_AT_byte_size<0x00000004>
DW_AT_type<<0x0000002a>>
...
```

After that, in order to get the symbol list we need to grab the *System.map* file for the kernel we want to analyze. This can be found in the main directory of the compiled kernel source. Regarding this file, while for Linux systems it is also possible to find the *System.map* file under the */boot* directory, for mobile device this is kind of equivalent to the */proc/kallsyms* file. However, based on our experience, it is strongly recommended to use the *System.map* file generated by compiling the kernel source code, since the */proc/kallsyms* file is actually missing many symbols and may drive Volatility to raise errors (i.e. ValueError). Only if everything else fails, */proc/kallsyms* should be used. Finally, to create the profile we need to place both the *module.dwarf* and the *System.ma*p file into a zip file, and then move this zip file under *volatility/plugins/overlays/linux/*:

```
$ zip /volatility/volatility-read-only/volatility/plugins/
        overlays/linux/Golfish-2.6.29.zip module.dwarf
        /Volumes/android-fs/android-sources/goldfish/
        System.map
  adding: module.dwarf (deflated 90\%)
  adding: Volumes/android-fs/android-sources/goldfish/
          System.map (deflated 73\%)
```

Since Android is based on Linux, we can use most of the Linux Commands / plugins to analyze the memory dump, once the correct profile has been created. In order to find the data related to the user input information (i.e. usernames, passwords, etc.) for a specific application, we first need to identify the Process ID (PID) of the target application, then we map the process in the memory, in order to find the offsets of the *heap*, which is the area of the memory we are interested in, and lastly we dump the *heap*. To do that, we used the following three Volatility plugins:

- *linux_pslist*, which gathers active tasks by traversing the task_struct->task list,
- *linux_proc_maps*, which gathers process maps for Linux,
- *linux_dump_map*, which writes selected memory mappings to disk.

Table 3 presents the case of the *eBay* application, as one example out of the 26 total applications analyzed for this research. As described above, once we find the PID of the eBay process we use Volatility to identify the offsets and then dump the process' heap and the DalvikVM's heap.

```
$ python vol.py --profile=LinuxGolfish-2_6_29x86 -f ebay.lime linux_pslist
Offset     Name             Pid  Uid    Gid   DTB        Start Time
---------- ---------------- ---  ----- -----  ---------- ----------------------------
0xca969400 com.ebay.mobile  379  10067 10067  0x0aec8000 2013-03-29 09:22:08 UTC+0000



$ python vol.py --profile=LinuxGolfish-2_6_29x86 -f ebay.lime linux_proc_maps -p 379
Pid Start      End        Flags Pgoff      Major Minor Inode File Path
--- ---------- ---------- ----- ---------  ----- ----- ----- -----------------------
379 0x0000b000 0x003d1000 rw-   0x0        0     0     0     [heap]
379 0x409b2000 0x42124000 rw-   0x0        0     7     353   /dev/ashmem/dalvik-heap
379 0x42124000 0x449b2000 ---   0x1772000  0     7     353   /dev/ashmem/dalvik-heap
[...]


$ python vol.py --profile=LinuxGolfish-2_6_29x86 -f ebay.lime linux_dump_map -s
0x0000b000 --dump-dir ~/memdump/ebay-heap/
Task VM Start   VM End     Length    Path
---- ---------- ---------- --------  ---------------------------------------
379  0x0000b000 0x003d1000 0x3c6000  /memdump/ebay-heap/task.379.0xb000.vma



$ python vol.py --profile=LinuxGolfish-2_6_29x86 -f ebay.lime linux_dump_map -s
0x409b2000 --dump-dir ~/memdump/ebay-heap/
Task VM Start   VM End     Length     Path
---- ---------- ---------- ---------  ---------------------------------------
379  0x409b2000 0x42124000 0x1772000  /memdump/ebay-heap/task.379.0x409b2000.vma
```

Table 3: Output and step-by-step of the volatility commands used for the analysis

The reason why we find two heaps is because, as explained in Section 4.2, when an application is launched, it runs as its own process, which contains its own instance of the DalvikVM. Therefore, one heap is the general heap of the process itself and the other is the internal heap of the virtual machine. Since the application runs inside the DalvikVM, we expect to find input data inserted by the user (i.e. usernames, passwords, PIN codes, etc.) eventually in the Dalvik-heap if not in both.

Memory dumps are binary files, so in order to look into them we used the command line utility `string` and a hex-editor. In the analysis we looked, for the same value, for two different types of encoding: ASCII and Unicode. For example, if the password used in the test was *my-password* we looked for both:

- *my-password*
- 6D 00 79 00 2D 00 70 00 61 00 73 00 73 00 77 00 6F 00 72 00 64 00, which is the hexadecimal equivalent of the ASCII value *"m.y.-.p.a.s.s.w.o.r.d."* and of the Unicode value *"my-password"*

We also looked for the eventual presence of identification keywords, such as "username", "password", "pin", etc., because the potential danger of the evidence found increases if preceded by something that uniquely identifies it. In fact, even if leaving a password in cleartext is wrong, if this password is randomly placed with other information, it may not be easily identifiable as the password, if at all. Therefore, if a malicious user/application does not know the password itself, placing the password value in cleartext that follows its identification label in a form like *'password=my-password'* makes it easily recoverable for anyone who looks for it.

4.5. Experiment Results

As claimed at the beginning of this chapter, we took under consideration for our test 26 different applications. We looked for user credentials, namely username, password, pin code for the bank, etc., either in the process heap and the Dalvik-heap. The results confirmed that the heap of reference is mainly the Dalvik-heap, which is the container where the application is executed in. In the following we describe the results of the testing for both the banking applications and general purposes applications.

Banking Apps Results

Banking applications are a quite sensitive field of analysis especially considering the direct impact of their vulnerabilities on the citizen life. For this analysis campaign we considered 15 different mobile banking applications. As the scope of this work is not to perform dedicated "penetration tests" of the banking environment, but rather to raise the attention of the security and development communities of possible implications of un-careful development of these applications, we will not disclose here the name of the banking applications analyzed.

Table 4 summarizes the results of our investigation with regards to the banking applications. More in details, the table reports our findings in the normal heap and in the Dalvik-heap during the application execution and after its termination. In an average of 75% of the applications we found username and password following their respective identification label, in both types of dumps (i.e. during and after the application execution). In 16% of the applications analyzed we found username and password in clear (ASCII and/or Unicode), but without any identification that could help identify them. Only in 7% of the cases we did not find any trace of the user credentials in the memory.

DUMP	TYPE	HEAP		DALVIK-HEAP	
		USERNAME	PASSWORD	USERNAME	PASSWORD
during execution	with id	26.7	20.0	80.0	73.3
	without id	6.7	6.7	20.0	13.3
	nothing found	66.7	73.3	0.0	13.3
after execution	with id	26.7	20.0	73.3	80.0
	without id	6.7	6.7	20.0	13.3
	nothing found	66.7	73.3	6.7	6.7

Table 4: Statistics related to the Bank applications group

General Purposes Apps Results

In the second group of applications, where we analyzed 11 of the most popular Android applications already studied in the first part of our research related to MobiLeak [38] namely *Box, Dropbox, eBay, Evernote, Facebook, Groupon, Linkedin, Monster, Skype, Tweetdeck* and *Twitter*, the situation reported is slightly better. Against what expected in the process heap, we found the username in an average of 73% of the applications, 55% were following their identification label. On the other hand, in 90% of the cases we did not find the password at all while the application was still running, reaching 100% if we consider the dumps taken after the application has been terminated. The improvement compared to the previous group regards the Dalvik-heap. Also here there is a significant difference between usernames and passwords occurrences found, while in the first group the situation was a bit more homogeneous. In 90% of the cases we found the username while the application was running, 45% of which following their identification label. In both types of dumps, while the application was running and after its execution, only in the 9% of the cases we did not find any occurrences of username. We found passwords occurrences in about 80% of the cases while the application was running, just 20% of which followed their identification label. Instead, from the dumps taken after the application has been terminated we found passwords occurrences in about 45% of the cases, 0% of them linked to an identification label, and 55% of the time we did not find any occurrence. Table 5 summarizes the described results.

DUMP	**TYPE**	**HEAP**		**DALVIK-HEAP**	
		USERNAME	**PASSWORD**	**USERNAME**	**PASSWORD**
during execution	with id	63.6	9.1	45.5	18.2
	without id	18.2	0.0	45.5	63.6
	nothing found	18.2	90.9	9.1	18.2
after execution	with id	54.5	0.0	63.6	0.0
	without id	18.2	0.0	27.3	45.5
	nothing found	27.3	100.0	9.1	54.5

Table 5: Statistics related to the general purposes applications group

5. Design of Data-in-Transit Analysis

This chapter presents the third phase of the MobiLeak project, dealing with the analysis of the *"Data in Transit"* state. Data in this state is data sent over a communication channel, transferred from the device to a remote server or provider. Sensitive information can be in the *data in transit* state in different ways, for example by eavesdropping wireless communication channels or by triggering the device to connect to a rogue/fake device in order to expose sensitive data.

5.1. Description of mobile (wireless) communication protocols

Due to their "nomad" nature, mobile devices communicate mainly (if not solely) over wireless channels. The most widespread mobile communication protocols are the Global System for Mobile Communications (GSM), the Wi-Fi, the Bluetooth, and lastly fast growing adoption is the Near Field Communication (NFC).

This third phase of our research has not been completed yet and it will be subject to further studies (see Chapter 7). However the results of the first preliminary study, which has been conducted with two communication protocols, Bluetooth and NFC, and is presented in this chapter, have resulted very important in order to understand the state of the art and identify which protocol presents higher risks for security and privacy violation.

5.2. Bluetooth Analysis

The Bluetooth wireless technology is a short-range communication system (see Table 6), intended to replace the cable(s) connecting portable and/or fixed electronic devices. The key features of Bluetooth wireless technology are robustness, low cost and device discovery support. Many features of the core specifications are optional, allowing product differentiation [56].

Created by telecom vendor Ericsson in 1994, it was originally conceived as a wireless alternative to RS-232 data cables. In 1998 Ericsson, IBM, Intel, Nokia, and Toshiba formed a trade association known as Bluetooth SIG (Special Interest Group) to publish and promote the Bluetooth standard. From the first Bluetooth enabled device in 1999, to 2008 more than 2 billion devices were using the Bluetooth technology (according to a press release from Bluetooth SIG dated May 2008). It is therefore clear the high level of pervasiveness and ubiquity of this technology, which justifies the need for a deep analysis related to the State of The Art of its security and privacy features as well as possible threats and vulnerabilities. Still according to Bluetooth SIG [56] listed below, there are numbers of Bluetooth products worldwide that give a clearer picture of the dimension of this technology:

- 906 million mobile phones sold in 2010, almost 100 percent with Bluetooth technology.
- 171 million laptops shipped in 2010, including 77 percent with Bluetooth technology.
- More than 50 million game consoles shipped in 2010, including 62 percent with Bluetooth technology.

- More than 40 million Bluetooth enabled health and medical devices were already in the market in early 2011.
- One third of all new vehicles produced worldwide in 2011 include Bluetooth technology, growing to 70 percent by 2016, according to Strategy Analytics.

Class	Power (mW)	Power (dbM)	Distance (m)	Sample Devices
1	100	20	~ 100	BT Access Point, dongles
2	2.5	4	~ 10	Keyboards, mice
3	1	0	~ 1	Mobile phone headset

Table 6: Bluetooth Classes

Having stated that, it is immediately clear the high level of pervasiveness and ubiquity of Bluetooth technology, which justify the need of a deep analysis related to the State of The Art of its security and privacy features as well as possible threats and vulnerabilities.

5.2.1. Bluetooth Protocol

The Bluetooth technology operates in the frequency band 2400-2800 MHz, called ISM (Industrial Scientific Medical) license free of any use. According to the standard, information is sent using a technology called FHSS radios (Frequency-hopping spread spectrum), which allows sending pieces of information using 79 different bands (1 MHz, 2402-2480 MHz in the range) included in frequency band used.

The Bluetooth protocol uses a packet-based paradigm with a Master/Slave structure (different from client-server protocols used by others). A device in master mode can communicate with up to seven devices in slave mode thus forming a *piconet*, a network of computers connected in ad-hoc mode. Each device connected to a *piconet* is synchronized with the master clock, which determines how packets are exchanged between devices of the *piconet*. Figure 2 shows an example of Bluetooth *piconet* topology.

There are two forms of Bluetooth wireless technology systems: *Basic Rate (BR)* and *Low Energy (LE)*. Both systems include device discovery, connection establishment and connection mechanisms. The Basic Rate system includes optional Enhanced Data Rate (EDR), alternate Media Access Control (MAC) and Physical layers extensions (PHY). The *LE* system includes features designed to enable products that require lower current consumption, lower complexity and lower cost than BR/EDR. LE is primarily designed to bring Bluetooth technology to coin cell battery-powered devices such as medical devices and sensors.

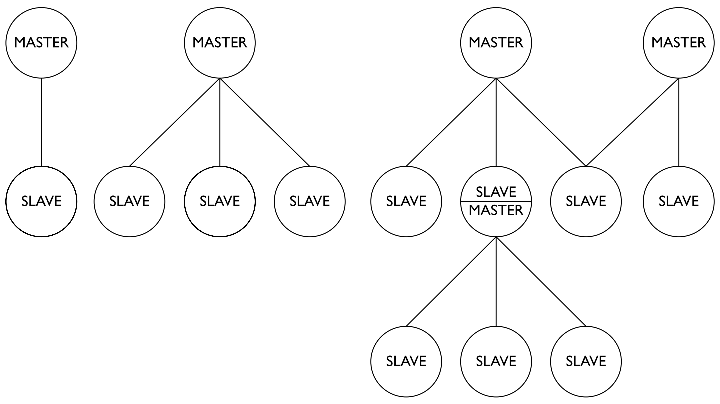


Figure 2: Example of Bluetooth piconet topology [57]

The key technology goals of Bluetooth LE (compared with Bluetooth BR/EDR) include lower power consumption, reduced memory requirements, efficient discovery and connection procedures, short packet lengths, and simple protocols and services. Four main versions of the Bluetooth protocol have been released until now [58] [59] [60] [61].

5.2.2. The Bluetooth Stack

Bluetooth is defined as a layered protocol architecture consisting of core protocols, cable replacement protocols, telephony control protocols, and adopted protocols [62]. Mandatory protocols for all Bluetooth stacks are: LMP, L2CAP and SDP (see Figure 3**Error! Reference source not found.**). Additionally, other two protocols are almost universally supported: HCI and RFCOMM. The lower layer is the physical layer and it handles the radio signal. The second layer is the Baseband, which is in charge of formatting the packets before they are sent out; specifically it builds the header, computes the checksum, data encryption and decryption, etc. The Link Controller manages the implementation of the Baseband protocol, while the Link Manager manages the Bluetooth connections via Link Manager Protocol.

Bluetooth uses a 48-bit identifier for device identification. This identifier is referred to as the Bluetooth device address (BD_ADDR). The first three bytes of the BD_ADDR are specific to the manufacturer of the Bluetooth radio, with identification assignments controlled by the IEEE Registration Authority [57].

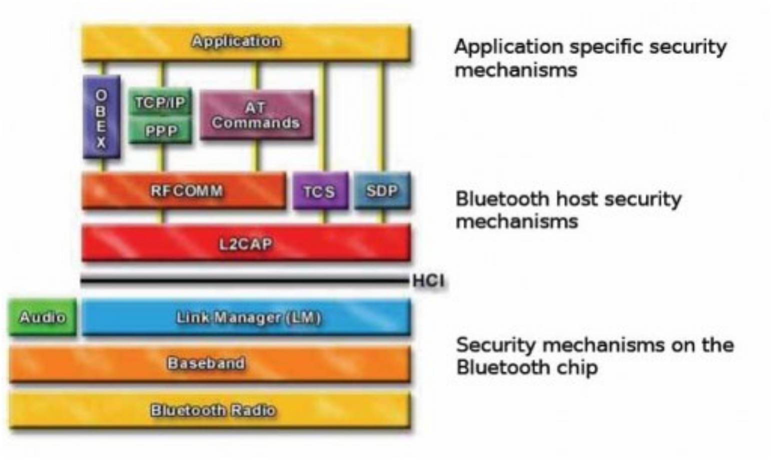

Application
OBEX
TCP/IP
PPP
AT Commands
RFCOMM
TCS
SDP
L2CAP
HCI
Audio
Link Manager (LM)
Baseband
Bluetooth Radio
Application specific security mechanisms
Bluetooth host security mechanisms
Security mechanisms on the Bluetooth chip

Figure 3: The Bluetooth Stack

- **Link Manager Protocol.** The Link Manager Protocol (LMP) is used to control and negotiate all aspects of the Bluetooth connection between two devices. This includes the set-up and control of logical transports and logical links, and for control of physical links.
- **Logical Link Control & Adaptation Protocol.** The Logical Link Control & Adaptation Protocol (L2CAP) is used to multiplex multiple logical connections between two devices using different higher-level protocols. It provides segmentation and reassembly of packets, as well as quality of service (QoS) related features.
- **Service Discovery Protocol.** Service Discovery Protocol (SDP) allows a device to discover services supported by other devices, and their associated parameters. A Universally Unique Identifier (UUID) identifies each services, with official services (Bluetooth profiles) assigned a short form UUID (16 bits rather than the full 128). There are two different ways to perform service discovery:
    - Searching: it refers to a specific service and it can be performed only knowing one or more attributes of the service;
    - Browsing: it is performed by sending a request for the root browse group UUID. The inquired device reply with the list of all UUID related to the services available. At this point the inquiring device can perform the searching as described before, one for each service/UUID.
- **Serial Port Emulation.** Radio frequency communications (RFCOMM) is a cable replacement protocol used to create a virtual serial data stream. RFCOMM provides a simple reliable data stream to the user, similar to TCP. It is used directly by many telephony related profiles as a carrier for AT commands, as well as being a transport layer for OBEX (Object Exchange) over Bluetooth.
- **Profiles.** Profiles have been developed in order to offer interoperability and to provide support for specific applications. A profile defines an unambiguous description of the communication interface between two units for one particular service. A new profile can be built using existing ones, allowing efficient reuse of existing protocols and procedures. This gives raise to a hierarchical profiles structure as outlined in Figure 4. The most fundamental definitions, recommendations, and requirements related to modes of operation and connection and channel setup are given in the generic access profile (GAP). Profiles are linked to the services a given device offers/supports. Therefore, from a security point of view, since Bluetooth enabled devices broadcast the list of supported services list upon request, each profile that is "advertised" could be seen as another potential door opened, more or less like tcp/upd ports for PCs.

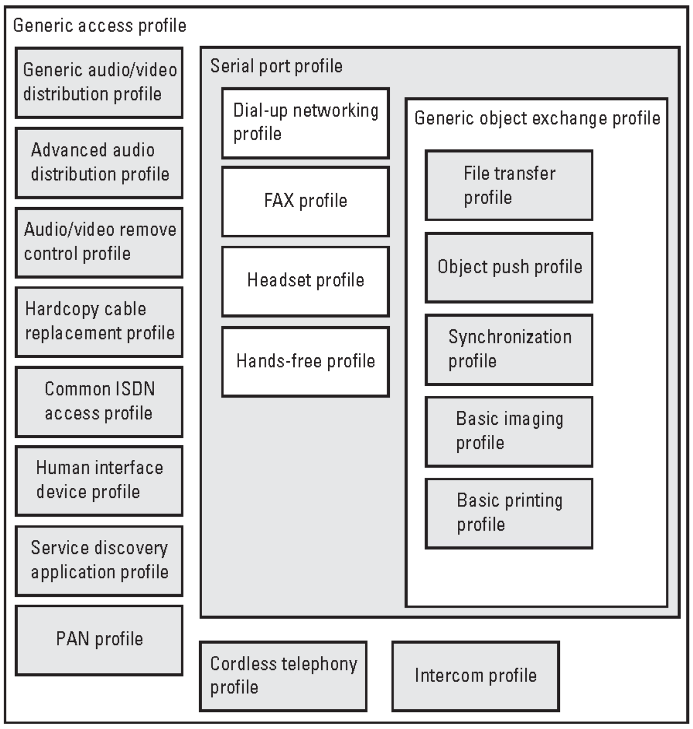

Generic access profile
Generic audio/video distribution profile
Advanced audio distribution profile
Audio/video remove control profile
Hardcopy cable replacement profile
Common ISDN access profile
Human interface device profile
Service discovery application profile
PAN profile
Serial port profile
Dial-up networking profile
FAX profile
Headset profile
Hands-free profile
Generic object exchange profile
File transfer profile
Object push profile
Synchronization profile
Basic imaging profile
Basic printing profile
Cordless telephony profile
Intercom profile

Figure 4: Bluetooth Profiles

5.2.3. Bluetooth Threats and Vulnerabilities

Due to its wireless nature, the Bluetooth communication channel is already subject to several threats like eavesdropping, impersonation, denial of service and man-in-the-middle attack. Other than the general wireless protocols' issues, there are the following threats specific to the Bluetooth enabled devices:

- ***Location tracking:*** Bluetooth devices broadcast their unique address, being therefore subject to location-tracking threats [63].
- ***Key management:*** Like many technologies that use cryptography for features such as authentication and encryption, Bluetooth devices are subject to threats related to key management, including key disclosure or tampering.
- ***Bluejacking:*** It involves the sending of unsolicited messages to a victim's Bluetooth device. This can be leveraged as a social-engineering attack that is enabled by susceptible Bluetooth devices. It can be also exploited for malware propagation, as demonstrated in [64].
- ***Incorrect protocol implementation:*** The quality of security on Bluetooth devices is determined to some degree by product-specific implementations. When a product manufacturer incorrectly implements the Bluetooth specification on its device, it makes the device or communications subject to security issues that would not exist if the specifications were implemented correctly. Implementation flaws have been at the root of many well-known Bluetooth security attacks (see Section 5.2.4).

Here follows a summary of well-known security vulnerabilities associated with Bluetooth. Some of them are version specific, while others are common to all versions. For a more comprehensive list refer to [65]:

- Bluetooth Versions Prior to v1.2
    - The unit key is reusable and becomes public when used. The unit key is a type of link key generated during device pairing, and has been deprecated since Bluetooth v1.2. This issue allows arbitrary eavesdropping by devices that have access to the unit key.
- Bluetooth Versions Prior to v2.1
    - Short PINs are permitted. Because PINs are used to generate encryption keys and users may tend to select short PINs, this issue can lower security assurances provided by Bluetooth's encryption mechanisms.
    - The encryption keystream repeats. In Bluetooth versions prior to v2.1, the keystream repeats after 23.3 hours of use. Therefore, a keystream is generated identical to that used earlier in the communication.
- Common to all Bluetooth versions:
    - Unknown random number generator (RNG) strength for challenge-response. The strength of the RNG used to create challenge-response values for Bluetooth authentication is unknown. Weaknesses in this RNG could compromise the effectiveness of Bluetooth authentication and overall security.
    - Negotiable encryption key length. The Bluetooth specification allows the negotiation of the encryption key down to a size as small as one byte.
    - Shared master key. The encryption key used to key encrypted broadcast communications in a Bluetooth piconet is shared among all piconet members.
    - Weak E0 stream cipher. A theoretical known-plaintext attack has been discovered that may allow recovery of an encryption key much faster than a brute-force attack.

5.2.4. Known Attacks

This section contains a list of few of the well-known attacks successfully carried against Bluetooth devices. The Trifinite Group published [66] detailed descriptions of Bluetooth attacks along with downloadable audit and demonstration software.

Blueprinting

Blueprinting is a method to remotely find out details about Bluetooth-enabled devices. Blueprinting can be used for generating statistics about manufacturers and models and to find out whether there are devices in range that have issues with Bluetooth security [67].

BlueBug

BlueBug is a security loophole that is present in some Bluetooth-enabled cell phones. Exploiting this loophole allows unauthorized downloading of the phone books and the calls list, sending and reading of SMS messages from the attacked phone, and many other problems.

BT Audit

BT Audit is a scanner for L2CAP and RFCOMM in order to find open ports and possible vulnerable applications linked to them.

BlueSmack

BlueSmack is a Bluetooth attack that disconnects some Bluetooth-enabled devices from the piconet they are connected to. This Denial of Service attack can be conducted using standard tools that are shipped with the official Linux Bluez utility package.

BlueSnarf

This attack allows access to a victim Bluetooth device because of a flaw in device firmware. In order to perform a BlueSnarf attack, the attacker needs to connect to the OBEX Push Profile (OPP), which has been specified for easy exchange of business cards and other objects. In most of the cases, this service does not require authentication. Missing authentication is not a problem for OBEX Push, as long as everything is implemented correctly. The BlueSnarf attack connects to an OBEX Push target and performs an OBEX GET request for known filenames, such as 'telecom/pb.vcf' for the devices phone book or 'telecom/cal.vcs' for the devices calendar file. In case of improper implementation of the device firmware, an attacker is able to retrieve all files where the name is either known or guessed correctly.

Bluesnarf++

BlueSnarf++ gives the attacker full read/write access when connecting to the OBEX Push Profile. Instead of a less functional OBEX Push daemon, these devices run an OBEX FTP server that can be connected as the OBEX Push service without pairing. Here the attacker can see all files in the file system (ls command) and can also delete them (rm command). The file system includes eventual memory extensions like memory sticks or SD cards.

HeloMoto

The HeloMoto attack takes advantage of the incorrect implementation of the 'trusted device' handling on some Motorola devices. The attacker initiates a connection to the unauthenticated OBEX Push Profile pretending to send a vCard. The attacker interrupts the sending process and without interaction the attacker's device is stored in the 'list of trusted devices' on the victim's phone. With an entry in that list, the attacker is able to connect to the headset profile without authentication.

Once connected to this service, the attacker is able to take control of the device by means of AT-commands.

BlueChop

BlueChop is an attack that disrupts any established Bluetooth piconet by means of a device that is not participating in the piconet. A precondition for this attack is that the master of the piconet supports multiple connections (a feature that is necessary for building up scatternets). In order to BlueChop a piconet, a device that is not participating to the targeted piconet spoofs a random slave out of the piconet and contacts the master. This leads to confusion of the master's internal state and disrupts the piconet. This attack is not specific to any device manufacturer and seems to have general validity.

BlueZ Arbitrary Command Execution Vulnerability

hcid utility spawns a helper program to request a PIN from the user when it receives a pairing request from a remote device. One of the arguments for calling the PIN helper application is the name of the remote device. However, when doing this, hcid does not escape shell characters. Thus an attacker can give a device a name containing commands to execute enclosed within ` characters. In addition, it is possible for an attacker to cause the PIN helper application to automatically pair with the remote device by adding ">/dev/null&echo PIN:<PIN code>" to the device name.

Redfang

Redfang is a tool that brute-forces Bluetooth BD addresses in order to communicate with devices in non-discoverable mode. Redfang accomplishes this by iterating through a user supplied range of device addresses and attempting to do a `read_remote_name()` on each one. If an address belongs to a Bluetooth device in the area, then the `read_remote_name()` call will return the device's name. A malicious person can then use this information to attack the device even if it's non-discoverable. To speed up the process, Redfang supports the user of multiple Bluetooth adapters to scan the supplied address range. Each adapter then scans disjoint portions of the address range. This tool is at a proof-of-concept development stage.

Bluetooth Stack Smasher

Bluetooth Stack Smasher (BSS) is a L2CAP protocol fuzzer designed to identify implementation weaknesses in Bluetooth devices. BSS is designed to transmit malformed L2CAP frames with a standard Bluetooth dongle on Linux systems. The malformed frames are designed to trigger and identify vulnerabilities in Bluetooth stack implementations, often resulting in denial of service conditions. Through the use of BSS, several L2CAP implementation weaknesses have been discovered in common devices.

Nokia N70 Malformed L2CAP Frame DoS

The Nokia N70 phone is vulnerable to Denial of Service attack involving malformed L2CAP frames with unknown properties. The Nokia N70 phone contains a vulnerability that causes a DoS condition when a malformed L2CAP frame is received by the device's Bluetooth interface. This can cause the device to become unresponsive and to display a "System Error" message.

5.2.5. Security Features and Architectures

Bluetooth wireless technology provides peer-to-peer communications over short distances. In order to provide usage protection and information confidentiality, the system provides security measures both at the application layer and the link layer. These measures are designed to be appropriate for a peer environment. This means that in each device, the authentication and the encryption routines are implemented in the same way. The encryption key is entirely different from the authentication key. A new encryption key shall be generated each time encryption is activated. Thus, the lifetime of the encryption key does not necessarily correspond to the lifetime of the authentication key. The authentication key will be more static in its nature than the encryption key: once established, the particular application running on the device decides when, or if, to change it. To underline the fundamental importance of the authentication key to a specific link, it is often referred to as the link key. Three basic security services are specified in the Bluetooth standard:

- ***Authentication***: verifying the identity of communicating devices based on their Bluetooth device address. Bluetooth does not provide native user authentication.
- ***Confidentiality***: preventing information compromise caused by eavesdropping by ensuring that only authorized devices can access and view transmitted data.
- ***Authorization***: allowing the control of resources by ensuring that a device is authorized to use a service before permitting it to do so.

The security policies of a device determine when and how to use security mechanisms. The Bluetooth standard provides some basic principles for enforcing link-level security and building more advanced security polices through four defined security modes:

- Security Mode 1: A Bluetooth unit in security mode 1 never initiates any security procedures; that is, it never demands authentication or encryption of the Bluetooth link.
- Security Mode 2: When a Bluetooth unit is operating in security mode 2, it shall not initiate any security procedures, that is, demand authentication or encryption of the Bluetooth link, at link establishment. Instead, security is enforced at channel (L2CAP) or connection (e.g., SDP, RFCOMM, TCS) establishment.
- Security Mode 3: When a Bluetooth unit is in security mode 3, it shall initiate security procedures before the link setup is completed. Two different security policies are possible: always demand authentication or always demand both authentication and encryption.

- Security Mode 4: it was defined in the v2.1 + EDR specification. It requires encryption for all services except Service Discovery, and it's compulsory between v2.1 + EDR devices (essentially making Modes 1 through 3 legacy modes once v2.1 + EDR becomes widespread). Like Security Mode 2, security in Security Mode 4 is implemented after link setup, at service level, and it uses Secure Simple Pairing (SSP), in which Elliptic Curve Diffie-Hellman (ECDH) replaces legacy key agreement for link key generation. However, the device authentication and encryption algorithms are identical to the algorithms in Bluetooth v2.0 + EDR and earlier versions. Under Security Mode 4, service security requirements must be identified as one of the following: a) authenticated link key required, b) unauthenticated link key required or c) no security required.

Table 7 contains a summary of the different security mode options for Master respective Slave, and the resulting security mechanism(s).

Slave Security Mode	**Master Security Mode**		
	1	**2**	**3**
1	No Authentication, no encryption.	If the master application demands authentication (and encryption), then the link will be authenticated (and encrypted).	The link will be authenticated. If the master policy demands it, the link will be encrypted.
2	If the slave application demands it, the link will be authenticated (and encrypted).	If the master or slave application demands it, the link will be authenticated (and encrypted).	The link will be authenticated. If the master policy demands it, or if the slave application demands it, the link will be encrypted.
3	The link will be authenticated. If the slave policy demands it, the link will be encrypted.	The link will be authenticated. If the slave policy demands it, or the master application demands it, the link will be encrypted.	The link will be authenticated. If the slave or the master policy demands it, the link will be encrypted.

Table 7: Security mode options and resulting configurations

5.2.5.1. Pairing

Many services offered over Bluetooth can expose private data or allow the connecting party to control the Bluetooth device. For security reasons it is therefore necessary to control which devices are allowed to connect to a given Bluetooth device. At the same time, it is useful for Bluetooth devices to automatically establish a connection without user intervention as soon as they are in range. To resolve this conflict, Bluetooth uses a process called pairing, which is generally manually started by a device user, making that device's Bluetooth link visible to other devices. Two devices need to be paired to communicate with each other; the pairing process is typically triggered automatically the first time a device receives a connection request from a device with which it is not yet paired. Once a pairing has been established, it is remembered by the devices, which can then connect to each other without user intervention. When desired, the user can later remove the pairing relationship.

During the pairing process, the two devices involved establish a relationship by creating a shared secret known as link key. If both devices store a link key, they are paired or bonded. A device that wants to communicate only with a bonded device can cryptographically authenticate the identity of the other device and so be sure that it is the same device it previously paired with. Once a link key has been generated, an authenticated link (ACL) between the devices may be encrypted, so that the data they exchange over the airwaves is protected against eavesdropping. Link keys can be deleted at any time by either device.

Pairing mechanisms have changed significantly with the introduction of Secure Simple Pairing in Bluetooth v2.1. The following summarizes the pairing mechanisms:

- Legacy pairing: This is the only method available in Bluetooth v2.0 and before. Each device must enter a PIN code; pairing is only successful if both devices enter the same PIN code. Any 16-byte UTF-8 string may be used as a PIN code, however not all devices may be capable of entering all possible PIN codes.
- Limited input devices: The obvious example of this class of devices is a Bluetooth Hands-free headset, which generally has few inputs. These devices usually have a fixed PIN, for example "0000" or "1234", that are hard-coded into the device.
- Numeric input devices: Mobile phones are classic examples of these devices. They allow a user to enter a numeric value up to 16 digits in length.
- Alphanumeric input devices: PCs and smartphones are examples of these devices. They allow a user to enter full UTF-8 text as a PIN code. If pairing with a less capable device the user needs to be aware of the input limitations on the other device, there is no mechanism available for a capable device to determine how it should limit the available input a user may use.

PIN Pairing

In versions prior to Bluetooth v2.1 + EDR pairing between devices is accomplished by the entry of a PIN or passkey with a maximum length of 128 bits. For PIN pairing, two Bluetooth devices simultaneously derive link keys when the user(s) enter an identical secret PIN into one or both devices, depending on the configuration and device type. There are two types of such passkeys: variable passkeys, which can be chosen at the time of pairing via some input mechanism, and fixed passkeys, which are predetermined [57]. The type of passkey used is typically determined by a device's input and display capabilities (for example, a Bluetooth-enabled phone with keyboard input and visual display may use a variable passkey, whereas a Bluetooth-enabled mouse may use a fixed passkey because it has neither input nor display capabilities to enter or verify a passkey).

Secure Simple Pairing (SSP)

This protocol is required from Bluetooth v2.1 + EDR. A Bluetooth v2.1 device may only use legacy pairing to interoperate with a v2.0 or earlier device. Secure Simple Pairing uses a form of public key cryptography, and has the following association models [57] [65] [68]:

- **Numeric comparison**: If both devices have a display and at least one can accept a binary Yes/No user input, they may use Numeric Comparison. This method displays a 6-digit numeric code on each device. The user should compare the numbers to ensure they are identical. If the comparison succeeds, the user(s) should confirm pairing on the device(s) that can accept an input. This method provides protection against MitM attck, assuming the user confirms on both devices and actually performs the comparison properly.
- **Passkey Entry**: This association model may be used between a device with a display and a device with numeric keypad entry (such as a keyboard) or two devices with numeric keypad entry. In the first case, the display is used to show a 6-digits numeric code to the user, who then enters the code on the keypad. In the second case, the user of each device enters the same 6-digits number. Both cases provide MitM protection.
- **Just Works**: It was primarily designed for scenarios where at least one of the devices does not have a display nor does it have a keyboard to enter six decimal digits. A good example of this model is the cell phone/mono headset scenario where most headsets do not have a display. The Just Works association model uses the Numeric Comparison protocol, but the user is never shown a number and the application may simply ask the user to accept the connection (exact implementation is up to the end product manufacturer). When compared against today's experience of a headset with a fixed PIN, the security level of the Just Works association model is considerably higher, since a high degree of protection against passive eavesdropping is realized. This method provides no Man-in-the-Middle (MitM) protection.
- **Out of Band (OOB)**: The Out of Band (OOB) association model was designed for devices that support a common additional wireless/wired technology (e.g., Near Field Communication or NFC) for the purposes of device discovery and cryptographic value exchange. In the case of NFC, the OOB model allows devices to pair by simply “tapping” one device against the other, followed by the user accepting the pairing via a single button push. It is important to note that to keep the pairing process as secure as possible, the OOB technology should be designed and configured to mitigate eavesdropping and MitM attacks. If it is not, security may be compromised during authentication. The user's experience differs a bit depending on the Out of Band mechanism. The OOB association model does not support a solution where the user has activated a Bluetooth connection and would like to use OOB for authentication only.

5.2.5.2. Authorization

Bluetooth allows two different levels of trust related to the devices and three levels of service security. A device is considered trusted if it has previously been paired with the device and will have full access to services on the Bluetooth device. On the other hand, untrusted devices are those that have not previously been paired with the device (or the relationship has been otherwise removed) and will have restricted access to services. The Bluetooth specification specifies also three levels of security for Bluetooth services:

- Service Level 1: These services require device authentication and authorization. Trusted devices will be granted automatic access to these services. Manual authentication and authorization will be required before untrusted devices are granted access to these services.
- Service Level 2: These services require authentication, but do not require authorization.
- Service Level 3: These services have no security and are open to all devices.

5.2.5.3. Confidentiality

Bluetooth uses E0, a stream cipher, as the basis for the encryption associated with these encryption modes. The defined modes include:

- Encryption Mode 1: No encryption. All traffic is unencrypted when this mode is used.
- Encryption Mode 2: Traffic between individual endpoints (non-broadcast) is encrypted with individual link keys. Broadcast traffic is unencrypted.
- Encryption Mode 3: Both broadcast and point-to-point traffic is encrypted with the same encryption key (the master link key). In this mode all traffic is readable by all nodes in the piconet (and remains encrypted to outside observers). Note that the notion of privacy in Encryption Mode 3 is predicated on the idea that all nodes in the piconet are trusted, because all nodes will have access to the encrypted data.

Modes 2 and 3 uses the same encryption mechanism. Of importance to note is that when encryption is used in Bluetooth, not all parts of the Bluetooth packets are encrypted. Because all members of a piconet must be able to determine whether the packet is meant for them, the header of the message must be unencrypted.

5.2.5.4. Security of Bluetooth LE

This is required by Bluetooth v4. Bluetooth LE has some differences in security aspects with respect to BR/EDR security features such as Secure Simple Pairing. The association models are similar to Secure Simple Pairing from the user perspective and have the same names, but have differences in the quality of the protection provided. One difference is that LE pairing results in the generation of a Long-Term Key (LTK) rather than a Link Key, which is determined by one device and sent over to the other device during pairing, instead of both devices generating the same key individually. Due to its very limited resources, the encryption through Elliptic Curves Diffie-Hellman could not be used here. Thus passive eavesdropping protection is not present in LE. Therefore, if an attacker can capture the LE pairing frames, he/she may be able to determine the resulting LTK.

LE uses Advanced Encryption Standard–Counter with CBC-MAC (AES-CCM). LE also introduces new cryptographic keys called the Identity Resolving Key (IRK) and Connection Signature Resolving Key (CSRK). The IRK is used to resolve public to private device address mapping. This allows a trusted device to determine another device's private device address from a public (random) device address. This new feature provides privacy for a particular device meaning that, if the device remains discoverable, an adversary cannot track its location over the time.

The CSRK is used to verify cryptographically signed data frames from a particular device. This allows a Bluetooth connection to use data signing (providing integrity and authentication) to protect the connection instead of data encryption (which, in the case of AES-CCM, provides confidentiality, integrity, and authentication) [65].

There is no separate authentication challenge/response step as with BR/EDR/HS to verify that they both have the same LTK or CSRK. Because the LTK is used as input for the encryption key, successful encryption setup provides implicit authentication. Similarly, successful data signing provides implicit authentication that the remote device holds the correct CSRK, although confidentiality is not provided. Key generation and distribution is summarized in Figure 5.

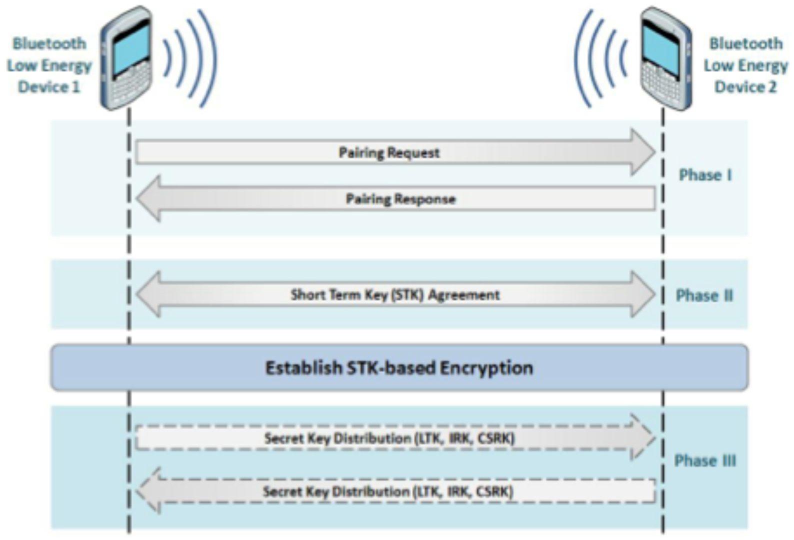

Bluetooth Low Energy Device 1
Bluetooth Low Energy Device 2
Pairing Request
Pairing Response
Phase I
Short Term Key (STK) Agreement
Phase II
Establish STK-based Encryption
Secret Key Distribution (LTK, IRK, CSRK)
Secret Key Distribution (LTK, IRK, CSRK)
Phase III

Figure 5: LE Pairing and Key Distribution scheme [65]

5.3. Near Field Communication (NFC): the three modalities

The Near Field Communication (NFC) is a bidirectional proximity coupling technology, which allows data transfer between devices at a short distance (up to 10cm), and extends several Radio Frequency IDentification (RFID) standards [69][70]. NFC operates at a frequency of 13.56 MHz, it supports data transfer rates of 106, 216 and 424 kbit/s. Other than supporting contactless smartcard systems based on the standards ISO/IEC 14443 and FeliCa, NFC extends the above with peer-to-peer functionality standardized in [71][72]. NFC has three operative modes:

- Reader/Writer Mode (Proximity Coupling Device, PCD), where an NFC device can read data from, and write data to, passive tags;
- Card Emulation (Proximity Inductive Coupling Card, PICC), where an NFC device can emulate a traditional contactless card for different possible usages, such as payment, ticketing, etc.;
- Peer-to-Peer (Near Field Communication, NFC), where one NFC device communicates with another NFC device offering multiple data channels with data potentially flowing in both directions simultaneously. It can be used to exchange contacts, Bluetooth pairing information or any other kind of data.

Therefore, an NFC-enabled mobile phone integrates an NFC-chip, such as the NXP PN544[2] on board of Google Galaxy S II, and possible a smart card (Secure Element) into an ordinary mobile phone. The NFC antenna, if not switched off from the settings menu, is active as long as the mobile phone is active, i.e. the screen is on (even if locked and password protected, it will read the tag). Therefore the NFC-subsystem constantly scans for NFC-tags. If a tag is detected, the phone will notify the NFC- aware application delegated to handle the request. If there is no NFC-aware application delegated to handle such request, or no application is running, the operating system reads the tag. If the tag contains data in a supported format, the data is passed over to the application that is registered for handling it.

5.3.1. Related Work

Research works related to NDEF and NFC Fuzzing mainly refers to two of Collin Mulliner's works [73][74]. Mulliner developed sets of tools to perform fuzz testing on NDEF with interesting results on fuzzing URIs type, as well as successfully demonstrating attacks against NFC-enabled mobile phones and services, including a proof-of-concept NFC-based worm. The rest of relevant research can be structured in two areas.

The first area consists of research on security analysis of NFC/NDEF. Verdult and Kooman [75] demonstrate how, by tricking the user into touching a malicious NDEF tag and invoking the Bluetooth channel, it is possible to install (malicious) applications on the phone without user consent, proving in fact that it is possible to spread malware via NDEF through malicious tags. The other researches, related to NDEF security, mainly focus on the Signature Record Type [76]. In their first work, Roland et al. [77] investigate several methods of signing NDEF messages underlining the most critical fields, while subsequently in [78], as well as Saeed and Walter in

[2] http://www.chipworks.com/en/technical-competitive-analysis/resources/technology-blog/2011/05/nxps-nfc-technology-in-the-pn544-controller/

[79], expose several weaknesses of the signature method demonstrating how it could be exploited.

In the second area of research, which is related to fuzzing mobile phones and mobile services, there are much more works that have been considered for this report. Collin Mulliner and Charlie Miller discuss how to use the Sulley fuzzing framework to fuzz SMS messages for smart phones [14]. They injected the SMSs with a man-in-the-middle attack to the communication channel between the application processor and the modem, causing both iPhone and Android to loose their network connectivity (DoS). Papers [80] and [81] also present research on fuzzing SMS in smart phones and feature phones respectively. Again Miller in [82] shows how to fuzz test Android devices, as well as how to monitor and debug them when fuzzing. Finally, Krishnan et al. in [83] introduce their Multi-target Automated Fuzzing Infrastructure and Arsenal (MAFIA), a composite, distributed client-server fuzz testing infrastructure for software applications and libraries. Particularly, it's a generation-based fuzzer specific for file format fuzz tests.

5.3.2. Possible Threats

As already introduced in Chapter 1, NFC technology, brought to mobile phones, opens new attacks and threats scenarios. According to several studies on the topic [84][85], it is possible to group these threats in the following main categories:

- <u>Eavesdropping</u>: As NFC is a wireless communication protocol, the possibility of sniffing data in transit is very high. Moreover, a part from NFC-SEC [86] that provides security standard for peer-to-peer NFC communication does not include reader/writer and card emulation mode [87], there is no link level security therefore the wireless signal is not encrypted;
- <u>Data Modification, Corruption, Insertion</u>: This kind of attacks is very hard to accomplish during communication, because it requires transmitting valid frequencies of the data spectrum at a correct time. But it becomes much more feasible when it refers to the data contained in the tags;
- <u>Man-in-the-Middle (MITM)</u>: Theoretically possible although practically very difficult in a real-world scenario;
- <u>Denial of Service (DoS)</u>: This is always a threat when it comes to wireless protocols. Due to the nature of the communication itself, it is usually easy to disrupt or jam a radio signal, knowing its frequency. But in the case of NFC and its applications and usages, DoS can be even accomplished on a "tag level", e.g. writing non-valid contents on sticky tags that are placed on top of valid ones. This may result on multiple phone/application crashes, bringing the users to stop using NFC-based services and discrediting the same.
- <u>Phishing</u>: This is probably the easiest attack to accomplish. As mentioned in [74], an attacker could destroy a valid tag [88] on a SmartPoster and place on top of it a sticky one with valid yet malicious content, e.g. to make the user visit a malicious website by replacing the valid URL of the original tag.

5.3.3. NFC Data Exchange Format

The NFC Data Exchange Format (NDEF) specification [89] defines a message encapsulation format to exchange information, e.g. between an NFC Forum Device and another NFC Forum Device or an NFC Forum Tag.

NDEF is a lightweight, binary message format that can be used to encapsulate one or more application-defined payloads of arbitrary type and size into a single message construct. A type, a length, and an optional identifier describe each payload. An NDEF message is composed of one or more NDEF records (see Figure 6).

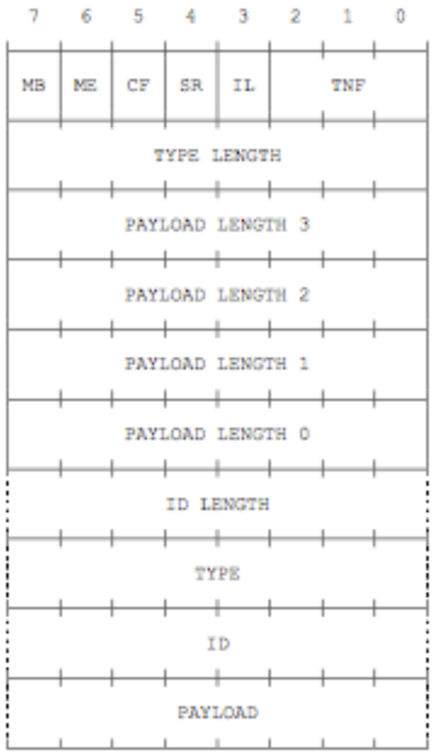


Figure 6: NDEF Record Layout [89]

The first record in a message is marked with the MB (Message Begin) flag set and the last record in the message is marked with the ME (Message End) flag set (see Figure 7). The minimum message length is one record, which is achieved by setting both the MB and the ME flag in the same record. The flag CF (Chunk Flag) indicates that this is either the first record chunk or a middle record chunk of a chunked payload, which means that the payload is continued on the next record. SR (Short Record) indicates that the PAYLOAD LENGTH field is a single byte, instead of the normal 4 bytes payload length. This short record layout (see Figure 8) is intended for compact encapsulation of small payloads, which will fit within PAYLOAD fields of size ranging between 0 to 255 octets. The flag IL determines if the optional ID field and its corresponding length field are present.

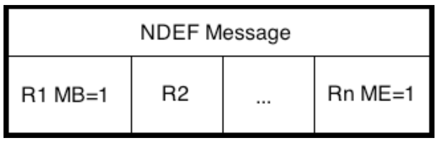


Figure 7: NDEF Message Layout

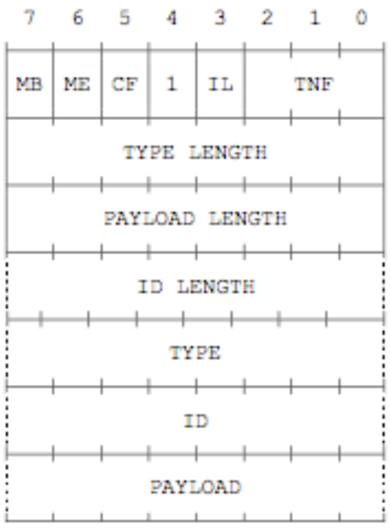


Figure 8: NDEF Short Record Layout [89]

The following TNF field (3 bits), which stands for Type Name Format, determines the structure of the type field:

- [0x00] The record is *Empty*, therefore type, id and payload fields are not present and their respective length fields are zero;
- [0x01] Type is a *NFC Forum well-known type*, e.g. URI [90], SmartPoster [91], MIME, etc., as defined in [92];
- [0x02] Indicates that the type is a MIME *media-type*, as defined in RFC 2046;
- [0x03] Indicates that the type is an *absolute-URI*, as defined in RFC 3986;
- [0x04] Type is a *NFC Forum external type*'s URN (Uniform Resource Identifier), as defined in [92];
- [0x05] The record contains data in an *unknown* format. No type information is present and the length of the Type field is zero;
- [0x06] *Unchanged*, it must be used in all middle record chunks and the terminating record chunk used in chunked payloads. It states that the record is continuing the payload of the preceding one. No type information is present and the length of the Type field is zero.
- [0x07] *Reserved* for future uses.

The ID field is a unique identifier in the form of URI reference (as defined in RFC 3986) and it can be used to cross-reference between different records.

5.4. A Framework for the Security Evaluation of NDEF Message Format

Based on the results of our study the conclusion is that NFC presents higher risks, also due to its foreseen heavy use in the mobile payment sector, and therefore a design of a framework for the security evaluation of NFC is proposed as a starting point for further research.

What stands out from the overview of the related works, introduced in Section 5.3.1, is that there have been few works around NDEF testing for several reasons:

- Lack of proper tools. Although there are several fuzzing frameworks available, frameworks are by nature more generic than, obviously, specific to one particular technology (e.g. network protocol, a specific type of software, etc.). On the other hand, due to the "young age" of NDEF and NFC-enabled mobile phones, there is no dedicated tool yet out for this kind of tests. However, as proved by several works mentioned before, frameworks can still be used to generate test cases, although support on delivery and monitoring will still persist as drawbacks.
- Difficulties in monitoring crashes. Differently from the classical computer environment, in the mobile one is not always easy to trace back system or applications crashes.
- Automation not always possible. While for certain type of mobile services it is possible to automate fuzz testing, when it comes to reading NFC tags the situation becomes a bit more difficult. Taking as example one of the works related to SMS vulnerability, in that case the researcher could easily send one after the other all the fuzzy SMSs. When it comes to NFC tags, the phone needs to be close to the tag, to "touch" it, in order to read it. This means what is needed is a system that generates the NDEF fuzzy messages and emulate the tags, and a person that every time puts the phone close to the tag to read it. If we consider all the possible fuzzy combinations, doing such thing for hundreds thousand times become unfeasible (or at least extremely slow).

The solution proposed here is a generation-based fuzzer in the form of both computer and Android application, a fuzzer specific for NDEF testing. The choice of developing it for Android devices brings several advantages:

- Comprehensive API. Several different libraries/modules have been taken into account for this work, such as libnfc[3], pynfc[4] and nfcpy[5]. Among all mobile OSes Android has the most, for sure one of, comprehensive API [93] regarding NFC and NDEF tags management and manipulation. Moreover, it supports a big variety of tag types such as Mifare Classic, Mifare Ultralight, Mifare DESfire, FeliCa, iCLASS, etc.
- Chip compatibility. Every different library supports a small number of NFC chip, while developing for Android it will work in every NFC-enabled Android device.
- Emulation and Monitoring. Android SDK and AVD Manager [51] provides the developer with a great emulation environment. By combining this with the Android Debug Bridge (adb) [33], the developer can simulate a tag discovery event inside the emulation environment itself (or even inside the smart phone itself) and monitor for eventual misbehavior via adb console.
- Portability. Although it is not a strict need or requirement, another advantage is that the Android device will turn into a portable NDEF fuzzing platform.

The idea is to create three different structures that will act as sort of wrappers to build the fuzzy NDEF messages. The following pseudo-code illustrates the structure of the highest level, which gives a general description of the tag:

[3] Libnfc, http://www.libnfc.org/documentation/introduction

[4] pynfc, http://code.google.com/p/pynfc/

[5] nfcpy, https://launchpad.net/nfcpy

```
      NFC_Tag struct {
         byte [] id;
         TTech tech;
         NDEF_Msg message;
         };
```

The field value *TTech tech* refers to the type of technology represented by the tag, e.g. TypeA, Felica, etc., while *byte [] id* refers to the tag "unique" id. As the name suggests, *NDEF_Msg message* describes the content of the tag.

```
      NDEF_Msg struct {
         NDEF_Rec [] records;
         };

      NDEF_Rec struct {
         Field [] fields;
         };
```

Based on the NFC Forum specifications, the code above describes the NDEF message as container of one or more NDEF records (*NDEF_Rec [] records*), which in turn are made of several fields (as described in Figure 6). Finally, the following structure is the one describing each single field of the NDEF Record, e.g. MB (Message Begin), ME (Message End), etc. Therefore, *String name* will refer to the name of the field (e.g. MB), *Int size* refers to the field size expressed as number of bits, *BitSet content* contains the actual values of the field.

```
      Field struct {
         String name;
         Int size;
         BitSet content;
         String min_value_by_spec;
         String max_value_by_spec;
         boolean comp_by_spec;
         boolean comp_by_usr;
         boolean fuzzable;
         };
```

The last three values will contain information that will help the fuzzer generating the different test cases, as well as giving the researcher chance to customize the output of the fuzzer. Particularly, the value *boolean comp_by_spec* indicates that this value is required by the specification (one more reason to try test cases without it), *boolean comp_by_usr* indicates that the researcher decided to set it as compulsory (therefore the fuzzer will not provide test cases without this field), and *boolean fuzzable* indicates if the researcher wants to fuzz this field or not, leaving it to its initial value. The field values *String min_value_by_spec* and *String max_value_by_spec*, if not NULL, give an indication of the boundary values such field can have.

In order to generate "intelligent test cases", other than describing the structure of the components of a NDEF message, rules that regulate the dependencies between different fields are also needed. On this matter, the best resource to follow is obviously the standard itself. The description of the "interaction" of different fields is well expressed in the NFC Forum documentation, which already tells the researcher how to set the rules simply using an appropriate terminology. In fact it is clearly stated if there have to be such dependencies or not and how. To give an example, at page 16 of NFC Data Exchange Format Definition [89] is written:

If the IL flag is 0, the ID_LENGTH field MUST NOT be present; or also

The TNF field MUST have a value between 0x00 and 0x06.

This information is clear indication on where to start fuzzing. For this purpose the concept of *Rule* is introduced:

```
      Rule struct {
         String [] if_field_name;
         Int [] if_size;
         BitSet [] if_content;
         String [] then_field_name;
         Int [] then_size;
         BitSet [] then_content;
         };
```

A *Rule* structure defines dependencies between fields of the same record. All the fields are represented as arrays to allow multiple dependencies for each rule (e.g. if MB==1, if ME==1, then CF=0). Rephrasing the example above having in mind the structure of a *Rule*, it would be *if IL == 0, then ID_LENGTH = null*. Therefore the rule would look like this:

```
      Rule struct {
         String [] if_field_name = new String [] {"IL"};
         Int [] if_size = new Int [] {1};
         BitSet [] if_content = new BitSet [] {0};
         String [] then_field_name = new String [] {"ID_LENGTH"};
         Int [] then_size = new Int [] {null};
         BitSet [] then_content = new BitSet [] {null};
         };
```

6. Conclusions and Summary of Countermeasures

This research investigated aspects related to security and privacy issues of mobile applications. The main goal was to identify and develop correct methodologies to analyze the security and privacy levels of mobile applications in a more effective way than it is currently done. Understanding how personal and sensitive data are handled was the fundamental requirement needed to properly detect threats that would harm the user.

After the initial characterization of the states at which data can exist in the mobile environment, a series of tests campaigns have been conducted for all these states. The conclusions and countermeasures derived from the results obtained are presented in the following sections, one for each test campaign. Such conclusions will be the bases on which a final global solution will be further researched and developed (see Chapter 7).

6.1. Data-at-Rest

The results of the first phase of the MobiLeak project underline that most developers rely solely on access controls, and security measures in general, defined by the underlying mobile operating system (OS). However, this assurance *could* be valid as long as the device is not rooted or jailbroken. The truth is that many users root/jailbreak their device to enhance and activate extra features, while at the meantime disabling any protection mechanism put in place by the mobile OS. Even when the device is still running the original OS (which is the most common scenario), rooting and jailbreaking procedures are easy and immediate to perform, don't require any deep technical knowledge (there are several packages that allow users to root/jailbreak the device just with couple of clicks) and therefore allow such protections to be easily bypassed, giving immediate access to all data present into the device.

The goal of this phase of the research was to show how many applications still store personal information in cleartext, while at the same time pointing out that an already known methodology (the forensics one) can be used for a different purpose, therefore increasing the option for assessing the privacy level of an application.

In order to reach this point, it has been demonstrated how data could be retrieved using free open source tools. We showed how this could be achieved without the need of a high budget for expensive professional forensics tools, which makes retrieving private information an affordable and easy task for the occasional thief or the one that finds a lost mobile device. Therefore based on the results of this study, there is a serious potential threat for identity or financial theft whenever a lost smartphone falls into the wrong hands. For instance, if a cybercriminal would be able to steal one password, coupled with all of the usernames recovered, this would pose a serious threat for someone who uses the same password on many accounts.

Countermeasures to the issues underlined in Chapter 3 would be, for developers, to identify and protect sensitive data on mobile devices as well as to handle password credentials securely. All this should be applied starting from the design phase and with the use of proper encryption algorithms and key management solutions, as underlined in [94] and [95] by ENISA and the OWASP Mobile Security Project.

6.2. Data-in-Use

The results of the presented testing campaign can be considered in some way unexpected. In fact, the tests clearly show that huge amounts of personal and critical data are stored without the due care in the mobile-device memory. This fact is surprising, considering that the first group of tests was related to mobile banking applications (i.e. an area where cyber-security is taken in high consideration) and the second group of tests was related to companies with large investments in cyber security. From the results of our analysis we can clearly state that there is an issue concerning both privacy and security as, in general, applications keep sensitive data in cleartext in their memory space. However, in order to evaluate correctly the impact of our findings and to propose a set of solutions, it is indeed needed to make a distinction between the results related to the memory analysis while the application is in use and the results related to the memory analysis after terminating the application.

While the application is running, the presence of cleartext data in the memory at a certain point in time is unavoidable. In this case the first and most immediate thing to do would be **to avoid keeping such data coupled with its clear identification label**. If a malicious user/application is looking for a password without knowing it, storing a password which follows the keyword *'password'* would make the research straightforward. It is also true that this would not completely solve the issue of keeping data in clear-text. In some cases a determined attacker could reverse engineer the data structures allocated in the memory by the application, being able to locate the password or other specific data anyway. To mitigate the impact of this attack-approach, the OS manufacturer should provide APIs that would help the developer to **clear that specific sensitive data from the memory after it has been used**, since after a successful login, there is no need to keep the credentials in the memory waiting for someone to grab them.

Another *"bad practice"* identified is that **most of the applications analyzed do not have a 'quit' button** or they have it hidden within a series of menus, making it hard to find. The only way to quit these applications is often by pushing the phone home button, which leaves the process still running in the background for a while and with it all the data allocated in the memory. This also explains why we found the **processes still running after exiting from the applications for 25 out of 26 applications analyzed**. Only one process immediately terminated when quitting the application, and which had a proper "quit" button implemented. Although it may be also due to the underlying OS when the process remains running still for a while after the application has been terminated, **developers should make sure that the quit button implements procedures to clean the memory before shutting down the application**.

Unfortunately, the results of this campaign magnify one more time how the cyber security is a matter of secure-development and secure-design. Cyber security by design is a costly approach in term of development time and resources and the *App* world is by nature based on a completely different paradigm (rapid development and low cost). Nevertheless, as mobile applications are becoming the daily companions of everybody, the *App* model need to be reviewed as soon as possible in order to protect the citizen's privacy and security.

6.3. Data-in-Transit

Bluetooth specifications offer several mechanisms to protect security and privacy to a certain extend. The major issues are caused by erroneous implementation of the protocol stack by vendors/manufacturers. This is proven by the fact that most of the vulnerabilities discovered are “vendor-related” and not general for a certain core version of the Bluetooth standard. Most of the vulnerabilities and attacks presented in Chapter 5 go back to several years ago to the core specification number 2 (which is still the most widespread). Since then the new versions have not introduced new security features (except for v4) and also not many new vulnerabilities have been discovered. In the meantime, vendors have often patched those vulnerabilities that were published.

However, Bluetooth security is critical and as such it should still be considered not strong enough for sensitive and privacy invasive applications. It is important that mobile application developers provide appropriate security controls that offer identity-level security features, such as user authentication and user authorization for applications that require security above and beyond what Bluetooth natively offers.

The NIST document “Guide to Bluetooth Security” [65] provides a comprehensive checklist with additional recommendations concerning Bluetooth security:

- Use complex PINs for Bluetooth devices.
- In sensitive and high-security environments, configure Bluetooth devices to limit the power used by the Bluetooth radio.
- Avoid using the “Just Works” association model for v2.1 + EDR devices.
- Limit the services and profiles available on Bluetooth devices to only those required.
- Configure Bluetooth devices as non-discoverable except during pairing.
- Avoid use of Security Mode 1.
- Enable mutual authentication for all Bluetooth communications.
- Configure the maximum allowable size for encryption keys.
- In sensitive and high-security environments, perform pairing in secure areas to limit the possibility of PIN disclosure.
- Unpair devices that had previously paired with a device if a Bluetooth device is lost or stolen.

It has been therefore concluded that Bluetooth protocol does not leave too much room for further research, since most of the security issues have been patched by vendors, and those left are mostly related to bad implementation practice and not to flaws in the protocol design itself.

Regarding the NFC protocol described in Chapter 5 were presented the motivations, both technical and practical from a security point of view, of the need to conduct security evaluation tests of the NDEF message format. As support to such motivations there is the lack of dedicated tools as well as the difficulty of performing this type of tests on NFC-enabled mobile devices. The solution proposed can, as described in Section 5.4, overcome such difficulties and fill the gap in the area of security tools available.

The preliminary analysis presented in Section 5.3 outlines that further research for the NFC state is needed in the area of the analysis of privacy leaks (see Section 7.1), as well as to the complete development of the proposed solution including comparison tests with alternative tools already on the market, such as Sulley Fuzzing Framework that will be used to generate tests cases (to be delivered outside of the framework).

7. Future Direction and Work

The overall message that can be taken from the conclusions in the previous chapter is that the security and privacy level of mobile applications needs to be increased. As clearly demonstrated by the types of applications analyzed within the MobiLeak project tests, the problem does not rise only with malicious applications, but also with most of the legitimate ones. Moreover, antivirus companies have failed to provide effective solutions for the mobile environment [96].

The next immediate step will be to complete the MobiLeak project tests targeting the *Data-in-Transit* state with the analysis of the NFC protocol. With the completion of that missing piece, this research work will have provided a comprehensive system for detecting privacy violation, with possible security implications, of mobile applications supported by concrete tests results.

Future research will then focus on designing an innovative solution Intrusion Detection like systems, based on the concept of Critical State Analysis and State Proximity.

7.1. Experimental Planning for the Data-in-Transit Analysis

As already mentioned earlier, NFC has been identified as the protocol to be analyzed in order to extend the *MobiLeak* project coverage to the *Data-in-Transit* state. What makes this technology particularly interesting from the research point of view, is that a) it is relatively new and b) it is being deployed and adopted as a way to make payments, using a mobile device to communicate credit card information to an NFC enabled terminal. As with the introduction of any new technology, the question that must be asked is what kind of impact the inclusion of this new functionality will have on the attack surface of mobile devices.

We would like to take into consideration two aspects:

I. The NFC radio signal. No matter if the data transmitted are encrypted or not, a spectral analysis of the NFC signal may reveal information related to the particular chip being used or the kind of operation being performed during a specific NFC communication session. For example, the noise introduced in the signal by the hardware components may be used to uniquely fingerprint the targeted device.

II. The NFC Application layer. In this case the target is the software built on top of the NFC stack. It turns out that through NFC, using technologies like Android Beam or NDEF content sharing, one can force some phones to parse images, videos, contacts, office documents, and even open up web pages in the browser, all without user interaction [97]. Moreover Visa and Mastercard, just to mention the biggest players in the field of payments, have already introduced their offers for mobile payments over NFC, namely PayWay and PayPass respectively [98][99].

In both cases the planned activity would include the analysis of the data being sent over the communication protocols, other than a series of tests at the application level in order to trigger the several NDEF types handlers and verify that a proper implementation is in place, which means that there would not be disclosure of private information and the handler correctly manages improper input without extending the NFC attack surface.

7.2. Critical State Analysis to Enhance Privacy of Mobile Apps

The idea presented in this section recall an approach proposed for Supervisory Control and Data Acquisition (SCADA) systems [17], based on the concept of Critical State Analysis, and explore the feasibility of adopting a similar approach in a mobile environment. To the author's knowledge, at the time of writing, no similar solution has been investigated yet within the research community.

The approach proposed is based on monitoring the evolution of the target application's states, which could be eventually applied also to the whole system. The following elements are therefore required, and will have to be defined, for tracking and analyzing the evolution of an application state:

- An *application/system representation language* in order to describe in a formal way the application/system under analysis.
- An *application/system state language* to describe in a formal way the critical states associated to the application/system under analysis.
- A *state evolution monitor* to follow the evolution of the application/system.
- A *critical state detector* to check whether the state of the application/system is evolving toward a defined critical state.
- A *critical state distance metric* to compute how close any state is with respect to the critical states.

For the formal representation of the application and system state a modeling language will be defined. A rule in the modeling language will have the form like *<condition>* → *<action>*, where *<condition>* is a boolean formula composed of conjunctions of predicates describing what values can be assumed by the different critical components, in this case represented by the sensitive data that may be accessed and disclosed. An example of such language could be defined with the following standard BNF notation:

<rule> ::= <condition> → *<action> : <level>*

<action> ::= <Alert> | *Log*

<condition> ::= <predicate>|*<predicate>, <condition>*

<predicate> ::= <Data_type><ID>.<Access_type><ID>| ...

<Data_type> ::= Contact|*Location*|*eMail*|*SMS*|*File*|...

<Access_type> ::= Read|*Read&Send*|*Write*|...

...

Every possible state of the system is described by the values of the terms representing the components of the system. A system state with n components can be represented by a vector $s \in \mathbb{R}^n$. The set of *critical states* $CS \subseteq \mathbb{R}^n$ is the set of states *satisfying* the *critical conditions,* which will have been described with the formal modeling language previously defined. Let $s(t)$ be the system state at a given time t. We can say that the monitored system is in a critical state $iif\ s(t) \in CS$.

One more thing to define would be the notion of *distance* between states. In fact we may want to raise an alert not only when the system enters a critical state, but also when the system is (rapidly) moving towards it. This may also be an indication of an anomalous behavior for a target application and the notion of distance may help to predict that the system is entering into a critical state. The notion of distance is based on the *Manhattan distance*, which is a form of geometry in which the usual distance function or metric of Euclidean geometry is replaced by a new metric in which the distance between two points is the sum of the absolute differences of their coordinates [100].

$$d_1(s, c) = \sum_{i=1}^{n} |s_i - c_i|$$

$$d_v(s, c) = \#\{i | s_i \neq c_i\}$$

In the two formulas above, the distance $d_1$represents the Manhattan distance between a given state s and a critical state c, while the distance d_v counts the number of application/system components whose values differ between the two states.

The research activity proposed in this section still needs further studies and the concepts exposed here represent part of the theoretical background behind it. The results of the MobiLeak campaigns and the failure of the antivirus companies to address the problem, show that there is an urgent need for a solution for privacy and security issues discussed in this thesis. Therefore the author believes that proving the effectiveness of such solution would be an important milestone for the research community and a step forward towards the protection and security of the final user.

8. Research Contributions

The research work presented in this Licentiate thesis contributed to the following peer-reviewed publications:

- Stirparo, P., Nai Fovino, I., Kounelis, I., "Data-in-Use leakages from Android Memory - Test and Analysis," *Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on*, 7-9 Oct. 2013 [Acceptance rate 28%]

- Baldini, G., Stirparo, P., "A Cognitive access framework for privacy protection in mobile cloud computing", *Mobile Computing over Cloud: Technologies, Services, and Applications*. 2013. IGI Global.

- Stirparo, P., "A Fuzzing Framework for the Security Evaluation of NDEF Message Format," *Computational Intelligence, Communication Systems and Networks, 2013. CICSYN '13. Fifth International Conference on*, June 2013.

- Stirparo, P., Loeschner, J., "Secure Bluetooth for Trusted m-Commerce," *International Journal of Communications, Network and System Sciences*, volume 6, number 6, 2013.

- Stirparo, P., Kounelis, I., "The MobiLeak Project: Forensics Methodology for Mobile Application Privacy Assessment", *Proceedings of the 7th International Conference for Internet Technology and Secured Transactions, ICITST, 2012*, p. 297-303.

- Loeschner, J., Kounelis, P., Mahieu, V., Nordvik, J. P., Striparo, P., Muftic, S., "Towards a better understanding of the impact of emerging ICT on the safety and security of the Citizen", *1st SysSec Workshop, SysSec 2011*, Amsterdam, The Netherlands, 2011.

References

[1] mobiThinking, “Global Mobile Statistics 2012,” Jun-2012. [Online]. Available: http://mobithinking.com/mobile-marketing-tools/latest-mobile-stats/a. [Accessed: 13-Nov-2012].

[2] Lookout, “Mobile Lost and Found,” 2011. [Online]. Available: https://www.mylookout.com/resources/reports/mobile-lost-and-found. [Accessed: 13-Nov-2012].

[3] “Sales of Mobile Devices in Second Quarter of 2011 Grew 16.5 Percent Year-on-Year; Smartphone Sales Grew 74 Percent.” [Online]. Available: http://www.gartner.com/it/page.jsp?id=1764714. [Accessed: 13-Nov-2012].

[4] “NFC Adoption Forecast.” [Online]. Available: http://www.isuppli.com/Mobile-and-Wireless-Communications/News/Pages/US-Wireless-Carriers-Partner-with-Big-Credit-Card-Companies-Boosting-Cell-Phone-NFC-Market.aspx. [Accessed: 17-Jun-2013].

[5] “NFC Forum.” [Online]. Available: http://www.nfc-forum.org/. [Accessed: 17-Jun-2012].

[6] Google, “Google Wallet.” [Online]. Available: https://www.google.com/wallet/. [Accessed: 13-Nov-2012].

[7] Lookout, “Lookout Mobile Threat Report,” Aug-2011. [Online]. Available: https://www.mylookout.com/mobile-threat-report. [Accessed: 16-Jun-2013].

[8] Google, “Android Market.” [Online]. Available: https://play.google.com/store?hl=en. [Accessed: 17-Jun-2013].

[9] F. Marturana, G. Me, R. Berte, and S. Tacconi, “A Quantitative Approach to Triaging in Mobile Forensics,” in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, Nov., pp. 582–588.

[10] P. Thomas, P. Owen, and D. McPhee, “An Analysis of the Digital Forensic Examination of Mobile Phones,” in *Next Generation Mobile Applications, Services and Technologies (NGMAST), 2010 Fourth International Conference on*, July, pp. 25–29.

[11] A. Hoog, *Android Forensics: Investigation, Analysis and Mobile Security for Google Android*. Syngress, 2011.

[12] J. Sylve, A. Case, L. Marziale, and G. G. Richard, “Acquisition and analysis of volatile memory from android devices,” *Digital Investigation*, 2011.

[13] V. L. L. Thing, K. Y. Ng, and E. C. Chang, “Live memory forensics of mobile phones,” *Digital Investigation*, vol. 7, pp. S74–S82, 2010.

[14] C. Miller and C. Mulliner, “Fuzzing the Phone in Your Phone,” presented at the Black Hat USA, 2009.

[15] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[16] A. Takanen, J. DeMott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*, 1st ed. Norwood, MA, USA: Artech House, Inc., 2008.

[17] A. Carcano, A. Coletta, M. Guglielmi, M. Masera, I. N. Fovino, and A. Trombetta, “A Multidimensional Critical State Analysis for Detecting Intrusions in SCADA Systems,” *Industrial Informatics, IEEE Transactions on*, vol. 7, no. 2, pp. 179–186, May.

[18] DFRWS, “A Road Map for Digital Forensic Research,” Report from the 1st Digital Forensic Research Workshop (DFRWS), 2001, p. 16.

[19] F. Adelstein, “Live forensics: diagnosing your system without killing it first,” *Communications of the ACM*, vol. 49, no. 2, pp. 63–66, 2006.

[20] J. M. Urrea, “An analysis of Linux RAM forensics,” Monterey, California. Naval Postgraduate School, 2006.

[21] B. Miller, “Fuzz Testing of Application Reliability.” [Online]. Available: http://pages.cs.wisc.edu/bart/fuzz/. [Accessed: 24-Apr-2013].

[22] IEEE, “IEEE Standard Glossary of Software Engineering Terminology,” *IEEE Std 610.12-1990*, p. 1, 1990.

[23] J. Seitz, *Gray Hat Python: Python Programming for Hackers and Reverse Engineers*. San Francisco, CA, USA: No Starch Press, 2009.

[24] P. Amini, “Sulley Fuzzing Framework.” [Online]. Available: https://github.com/OpenRCE/sulley. [Accessed: 24-Apr-2013].

[25] P. Amini, “Fuzzing Software Collection.” [Online]. Available: http://fuzzing.org/. [Accessed: 24-Apr-2013].

[26] D. Aitel, *The Advantages of Block-Based Protocol Analysis for Security Testing*. 2002.

[27] S. Schrittwieser, P. Fruehwirt, P. Kieseberg, M. Leithner, M. Mulazzani, M. Huber, and E. Weippl, “Guess Who’s Texting You? Evaluating the Security of Smartphone Messaging Applications,” in *Proceedings of the 19th Annual Network Distributed System Security Symposium (NDSS’12)*, San Diego, California, USA, 2012.

[28] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, “PiOS: Detecting Privacy Leaks in iOS Applications,” in *Proceedings of the 18th Annual Network Distributed System Security Symposium (NDSS’11)*, 2011.

[29] 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 the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI’10)*, 2010.

[30] A. Hoog, “Forensics Security Analysis of Google Wallet,” Dec-2011. [Online]. Available: https://viaforensics.com/mobile-security-category/forensics-security-analysis-google-wallet.html. [Accessed: 13-Nov-2012].

[31] Gartner, “Gartner Says Worldwide Sales of Mobile Phones Declined 2.3 Percent in Second Quarter of 2012,” Aug-2012. [Online]. Available: http://www.gartner.com/it/page.jsp?id=2120015%20. [Accessed: 13-Nov-2012].

[32] Google, “Google Play.” [Online]. Available: https://play.google.com. [Accessed: 17-Jun-2013].

[33] "Android Debug Bridge." [Online]. Available: http://developer.android.com/guide/developing/tools/adb.html. [Accessed: 17-Jun-2013].

[34] "JSON Format home page." [Online]. Available: http://www.json.org/. [Accessed: 13-Nov-2012].

[35] CIPA, "Exchangeable Image File format (EXIF)." [Online]. Available: http://www.cipa.jp/english/hyoujunka/kikaku/pdf/DC-008-2010_E.pdf. [Accessed: 13-Nov-2012].

[36] "IDC: Android Market Share Reached 75% Worldwide In Q3 2012 | TechCrunch," *TechCrunch*. [Online]. Available: http://techcrunch.com/2012/11/02/idc-android-market-share-reached-75-worldwide-in-q3-2012/. [Accessed: 26-Feb-2013].

[37] TheNextWeb, "In one year, android malware up 580%, 23 of the top 500 apps on google play deemed high risk." [Online]. Available: http://thenextweb.com/google/2012/10/25/in- one-year-android-malware-up-580-23-of-the-top-500-on-google-play- deemed-high-risk/. [Accessed: 29-Apr-2013].

[38] P. Stirparo and I. Kounelis, "The mobileak project: Forensics methodology for mobile application privacy assessment," in *Internet Technology And Secured Transactions, 2012 International Conferece For*, Dec., pp. 297–303.

[39] I. Kollár, "Forensic RAM dump image analyser," Department of Software Engineering, Charles University, Prague, 2010.

[40] A. Case, A. Cristina, L. Marziale, G. G. Richard, and V. Roussev, "FACE: Automated digital evidence discovery and correlation," *Digital Investigation*, vol. 5, Supplement, no. 0, pp. S65 – S75, 2008.

[41] A. Case, L. Marziale, C. Neckar, and G. G. Richard III, "Treasure and tragedy in kmem_cache mining for live forensics investigation," *Digital Investigation*, vol. 7, Supplement, no. 0, pp. S41 – S47, 2010.

[42] A. Case, L. Marziale, and G. G. Richard III, "Dynamic recreation of kernel data structures for live forensics," *Digital Investigation*, vol. 7, Supplement, no. 0, pp. S32 – S40, 2010.

[43] Volatile System, "The Volatility Framework: Volatile memory artifact extraction utility framework." [Online]. Available: https://www.volatilesystems.com/default/volatility. [Accessed: 29-Apr-2013].

[44] Volatile System, *The Volatility Framework*. [Online]. Available: https://code.google.com/p/volatility/. [Accessed: 29-Apr-2013].

[45] S. Leppert, "Android Memory Dump Analysis," *Student Research Paper, Chair of Computer Science*, vol. 1, 2012.

[46] Google, "Using DDMS." [Online]. Available: http://developer.android.com/tools/debugging/ddms.html. [Accessed: 29-Apr-2013].

[47] H. Macht, "Live Memory Forensics on Android with Volatility," Friedrich-Alexander University Erlangen-Nuremberg, 2013.

[48] EmbeddedLinux, “Android Memory Usage.” [Online]. Available: http://elinux.org/Android_Memory_Usage. [Accessed: 29-Apr-2013].

[49] Google, “Processes and Threads.” [Online]. Available: http://developer.android.com/guide/components/processes-and-threads.html. [Accessed: 29-Apr-2013].

[50] “Android Emulator.” [Online]. Available: http://developer.android.com/tools/help/emulator.html. [Accessed: 08-Apr-2013].

[51] “Android SDK.” [Online]. Available: http://developer.android.com/sdk/index.html. [Accessed: 16-Jun-2013].

[52] “Android NDK.” [Online]. Available: http://developer.android.com/tools/sdk/ndk/index.html. [Accessed: 04-Jun-2013].

[53] A. McFadden, “Don’t do heap dump on SIGUSR1.” [Online]. Available: https://github.com/android/platform_dalvik/commit/b037a464512c0721bdca96 9ae19cce3d4b17b083#vm/SignalCatcher.c. [Accessed: 08-Apr-2013].

[54] J. Sylve, “Android Mind Reading,” in *ShmooCon Security Conference*, 2012.

[55] J. Sylve, “LiME - Linux Memory Extractor.” [Online]. Available: https://code.google.com/p/lime-forensics/. [Accessed: 17-Jun-2013].

[56] Bluetooth SIG, “Bluetooth Specification.” [Online]. Available: https://www.bluetooth.org/Technical/Specifications/adopted.htm. [Accessed: 23-Apr-2013].

[57] H. Dwivedi, C. Clarck, and D. Thiel, *Mobile Application Security*. McGraw Hill, 2010.

[58] Bluetooth SIG, “Bluetooth Specification: Core Versione 2.0 + EDR,” Nov-2004. [Online]. Available: https://www.bluetooth.org/docman/handlers/DownloadDoc.ashx?doc_id=40560 . [Accessed: 23-Apr-2013].

[59] Bluetooth SIG, “Bluetooth Specification: Core Versione 2.1 + EDR,” Jul-2007. [Online]. Available: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=241363 . [Accessed: 23-Apr-2013].

[60] Bluetooth SIG, “Bluetooth Specification: Core Versione 3.0 + HS,” Apr-2009. [Online]. Available: https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=1742 14. [Accessed: 23-Apr-2013].

[61] Bluetooth SIG, “Bluetooth Specification: Core Versione 4.0,” Jun-2010. [Online]. Available: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737 . [Accessed: 23-Apr-2013].

[62] W. Stallings, *Wireless Communications and Networks*, 2nd ed., Prentice Hall, 2004.

[63] S. Hay and R. Harle, “Bluetooth tracking without discoverability,” in *4th International Symposium on Location and Context Awareness*, 2009, pp. 120–137.

[64] L. Carettoni, C. Merloni, and S. Zanero, “Studying bluetooth malware propagation: The bluebag project,” *IEEE Security & Privacy*, vol. 5, no. 2, pp. 17–25, 2007.

[65] NIST, “Guide to Bluetooth Security (Draft), Special Pubblication 800-121, Rev. 1,” NIST, 2011.

[66] “Trifinite Group.” [Online]. Available: http://www.trifinite.org. [Accessed: 23-Apr-2013].

[67] M. Herfurt and C. Mulliner, “Remote Device Identification based on Bluetooth Fingerprinting Techniques,” Trifinite Group, White Paper, 2004.

[68] C. Gehrmann, J. Persson, and B. Smeets, *Bluetooth Security*. Artech House, Inc., 2004.

[69] *ISO/IEC 14443 - Identification cards - Contactless integrated circuit cards - Proximity cards*. 2001.

[70] *ISO/IEC 15693 - Identification cards — Contactless integrated circuit(s) cards — Vicinity cards*. 2000.

[71] *ISO/IEC 18092 / ECMA-340 - Near Field Communication Interface and Protocol (NFCIP-1)*. 2004.

[72] *ISO/IEC 21481 / ECMA-352 - Near Field Communication Interface and Protocol (NFCIP-2)*. 2004.

[73] C. Mulliner, “Vulnerability Analysis and Attacks on NFC-enabled Mobile Phones,” in *Proceedings of the 1st International Workshop on Sensor Security (IWSS) at ARES*, Fukuoka, Japan, 2009.

[74] C. Mulliner, “Hacking NFC and NDEF,” presented at the NinjaCon / B-Sides Vienna, 2011.

[75] R. Verdult and F. Kooman, “Practical Attacks on NFC Enabled Cell Phones,” in *Near Field Communication (NFC), 2011 3rd International Workshop on*, 2011, pp. 77 –82.

[76] NFC Forum, “Signature Record Type Definition, Rev 1.0,” Technical Specification, Nov. 2010.

[77] M. Roland and J. Langer, “Digital Signature Records for the NFC Data Exchange Format,” in *Near Field Communication (NFC), 2010 Second International Workshop on*, 2010, pp. 71 –76.

[78] M. Roland, J. Langer, and J. Scharinger, “Security Vulnerabilities of the NDEF Signature Record Type,” in *Near Field Communication (NFC), 2011 3rd International Workshop on*, 2011, pp. 65 –70.

[79] M. Q. Saeed and C. D. Walter, “A Record Composition/Decomposition attack on the NDEF Signature Record Type Definition,” in *Internet Technology and Secured Transactions (ICITST), 2011 International Conference for*, 2011, pp. 283 –287.

[80] C. Mulliner and C. Miller, “Injecting SMS Messages into Smart Phones for Security Analysis,” in *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT)*, Montreal, Canada, 2009.

[81] C. Mulliner, N. Golde, and J.-P. Seifert, “SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale,” in *Proceedings of the 20th USENIX Security Symposium*, San Francisco, CA, USA, 2011.

[82] C. Miller, “Defeating Android,” presented at the ShmooCon, 2009.

[83] S. P. T. Krishnan, L. W. Hao, S. A. Sathya, and L. Devi, “A Distributed Multi-Target Software Vulnerability Discovery and Analysis Infrastructure for Smart Phones,” in *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, 2010, pp. 1 –5.

[84] G. Madlmayr, J. Langer, C. Kantner, and J. Scharinger, “NFC Devices: Security and Privacy,” in *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, 2008, pp. 642 –647.

[85] E. Haselsteiner and K. Breitfuß, “Security in Near Field Communication (NFC),” in *Workshop on RFID Security*, 2006.

[86] *NFCIP-1 Security Services and Protocol - Cryptography Standard using ECDH and AES*. 2008.

[87] V. Coskun, K. Ok, and B. Ozdenizci, *Near Field Communication (NFC): From Theory to Practice*. Wiley-Blackwell, 2012.

[88] “RFID_Zapper.” [Online]. Available: http://events.ccc.de/congress/2005/static/r/f/i/RFID-Zapper(EN)_77f3.html. [Accessed: 24-Apr-2013].

[89] NFC Forum, “NFC Data Exchange Format (NDEF), Rev 1.0,” Technical Specification, Jul. 2006.

[90] NFC Forum, “URI Record Type Definition, Rev 1.0,” Technical Specification, Jul. 2006.

[91] NFC Forum, “Smart Poster Record Type Definition, Rev 1.1,” Technical Specification, Jul. 2006.

[92] NFC Forum, “NFC Record Type Definition (RTD), Rev 1.0,” Technical Specification, Jul. 2006.

[93] “Android API.” [Online]. Available: http://developer.android.com/reference/packages.html. [Accessed: 17-Jun-2013].

[94] ENISA and OWASP, “Smartphone Secure Development Guidelines,” Nov-2011. [Online]. Available: http://www.enisa.europa.eu/activities/Resilience-and-CIIP/critical-applications/smartphone-security-1/smartphone-secure-development-guidelines. [Accessed: 24-Apr-2013].

[95] OWASP, “Top Ten Mobile Controls.” [Online]. Available: https://www.owasp.org/index.php/OWASP_Mobile_Security_Project#Top_Ten_Mobile_Controls. [Accessed: 24-Apr-2013].

[96] Y. Zhou and X. Jiang, “Dissecting Android Malware: Characterization and Evolution,” in *Security and Privacy (SP), 2012 IEEE Symposium on*, 2012, pp. 95–109.

[97] C. Miller, “Exploring the NFC attack surface,” *Proceedings of Blackhat*, 2012.

[98] Visa, “Enable contactless payments in Mobile Applications.” [Online]. Available: https://developer.visa.com/paywavemobile. [Accessed: 12-May-2013].

[99] MasterCard, “Introducing Masterpass.” [Online]. Available: https://masterpass.com/. [Accessed: 12-May-2013].

[100] “Taxicab Metric.” [Online]. Available: http://mathworld.wolfram.com/TaxicabMetric.html. [Accessed: 12-May-2013].