Ref. Ares(2017)3453059 - 08/07/2017

TagItSmart!
Smart Tags driven service platform for enabling ecosystems of connected objects

Grant agreement 688061

Secure and reliable FC-scanner usage

Deliverable ID:	**D 3.2**
Deliverable Title:	**Secure and reliable FC-scanners usage**
Revision #:	**1.0**
Dissemination Level:	**Public**
Responsible beneficiary:	UNIPD
Contributing beneficiaries:	UNIS, EVT, UPC, SIE
Contractual date of delivery:	30.06.2017
Actual submission date:	07.07.2017

Start Date of the Project: 1 January 2016 Duration: 36 Months

European Commission

Contents

Section 1 - Introduction 3

1.1 Objectives 3

1.2 Organisation 4

Section 2 - State-of-the-art on security of mobile applications 5

2.1 Mobile Malware Detection Techniques 5

2.2 Mobile Data Protection and Privacy Preservation Techniques 6

2.3 Security in NFC/RFID based communication channels 8

2.4 Discussion on Related Work 9

Section 3 - Secure and reliable scanning 10

3.1 System level security 10

3.1.1 Sandboxing Techniques 10

3.1.2 Remote attestation 20

3.2 Communication reliability and security 21

3.2.1 Vulnerabilities in the FC-Scanner to backend communication 21

3.2.2 Authentication and API Keys 21

3.2.3 Secure communication 22

3.2.4 Role-based authorisation 23

Section 4 - Enhancing security and user privacy based on context 24

4.1 Privacy Protection in Mobile Applications 24

4.1.1 User-in-the-Loop Verification System 25

4.2 Security and privacy based on user context 30

4.2.1 Current State of the Art in Context Recognition 30

4.2.2 Context Recognition in TagItSmart! 31

Section 5 - Conclusions 34

Section 6 - References 35

Section 1 - Introduction

The overall objective of the TagItSmart (TIS) project is to create an open, interoperable cloud-based platform that could be effectively used by various consumer-oriented service providers for managing the lifecycle of everyday mass-market objects, and for composing and offering value added services to customers. These mass-market objects when combined with our SmartTags will create a connected Internet of Things (IoT) ecosystem. The SmartTags combine the capabilities of functional inks with the pervasiveness of digital QR-codes and electronic printed circuits (e.g. NFC tags). The resultant smart objects then act as intelligent markers for a variety of environmental conditions by changing their status in response to these and can also be seamlessly tracked throughout their lifecycle, through embedded tag readers (i.e., FC-scanners) in the mobile devices.

The data extracted from the SmartTags observations, in conjunction with contextual information, enables a new ecosystem of SmartTag-enabled services and intelligent applications for new markets. Hence, the security and communication reliability in the front-end system of TagItSmart that includes the billions of FC-scanners and SmartTags becomes an essential task. In particular, the following functionalities of the front-end system should be insured: (i) the communication between the FC-scanner and SmartTag must be secure and reliable in order to extract correct information from these tags; (ii) the FC-scanner installed in a mobile device must execute in a secure environment i.e., the underlying OS and other applications running in parallel in the same device are not able to affect its normal functionalities; and (iii) an adversary should not be able to compromise the FC-scanner in order to leak any private information (such as GPS or camera data) of the user that is also stored on the same device. Additionally, solutions that ensure security of communication between FC-scanner and TIS back-end, and the use of contextual information are also envisioned to enhance the security in the whole system. For instance, dynamically block unintended (e.g., device in pocket) or even intended scan attempts (e.g., scan while in front of an ATM or scan/transmit while in a questionable environment such as when using unsecured/public WiFi) based on the context.

1.1 Objectives

The main aim of this deliverable is to research and develop various techniques that will allow the users of FC-scanner to securely and reliably extract the information from SmartTags, use it, and securely share it with the TIS service platform. The goal is to ensure the secure and reliable use of the FC-scanner application. Particular attention will be devoted to the identification of possible threats the system could be exposed to during the SmartTags scanning process. For the identified threats, we will investigate and propose possible security solutions. To this end, we will define solutions:

- to allow FC-scanners to verify the authenticity or maliciousness of components involved in scanning applications such as device sensors (audio, camera, GPS, to name a few) and a building block to guarantee essential security features which includes data flow integrity (i.e., a “correct” behaviour of the FC-scanner software) and remote attestation (i.e., verify that the platform is running an authorized FC-scanner software). The communication reliability and security characteristics of the wireless channel (i.e., NFC or RFID) that will be used for data communication between the FC-scanner and SmartTags will be investigated to report the impact on the system for the cases, where the communication channel is compromised or misused by an adversary i.e., relay attack, eavesdropping and jamming. In particular, a secure communication between the SmartTags and the FC-scanner as well as

between the FC-scanner and the existing infrastructure should be addressed in order to prevent participating users from being the target of maliciously deployed SmartTags.

- at system level by using sandboxing techniques for executing FC-scanner applications in order to support permission-based and context-based access control on sensitive data that is stored on the same device, and to address the security concerns about data sharing, leakage and loss.

1.2 Organisation

The rest of this document is organized as follows. In section 2, we discuss the state-of-the-art related to the research topics that we are addressing in this deliverable (i.e., a secure and reliable usage of FC-Scanner application). This includes discussion on the mobile malware detection techniques, the mobile data protection and privacy preservation techniques, and the security in NFC/RFID communications. In section 3, first, we present a new sandboxing technique that ensures the correct behaviour and functionality of the FC-Scanner application. Our approach ensures that the functionality of FC-Scanner application remains unaffected in the presence of a strong adversary. Then, we present a novel technique that allows the developers to implement a stealthier and modular malware analysis architecture that could be used to further secure the FC-Scanner application from possible advanced malware attacks. We also discuss the possible use of remote attestation process for authenticating the FC-Scanner application. Furthermore, we propose solutions to ensure secure communication between the FC-Scanner and the TIS platform. In section 4, we propose various innovative techniques that could be envisioned to enhance the security of the FC-Scanner and the privacy of user mobile data using the contextual information extracted from mobile and other sensors. Finally, in section 5, we present the conclusion and directions for future work.

Section 2 - State-of-the-art on security of mobile applications

With the continuous digital advancements, smartphones have become very essential components for many routine tasks such as paying bills, transferring money, instant messaging, emails, extracting product information while shopping, to name a few. This extensive use of mobile devices make them a very attractive attack surface for cyber thieves as these devices hold personal details (i.e., accounts, locations, contacts, and photos) and have potential capabilities for eavesdropping (i.e., unauthorized real-time interception of a private communication) by using various means such as cameras/microphone, malicious applications, wireless connections, to name a few. Android, being the most popular mobile platform, is the target of malicious hackers who are trying to use Android app as a tool to break into and control the device.

In this section, we present a brief overview of the various security and privacy aspects that are closely related to the mobile applications (such as FC-Scanner) and the mobile devices usage. We discuss the existing vulnerabilities in mobile devices that lead to various threats to the security and privacy of their users. We will also discuss the corresponding state-of-the-art countermeasures that are used to detect or avoid these security threats.

2.1 Mobile Malware Detection Techniques

The proliferation of mobile devices has opened the mobile era and rapidly increased the popularity of mobile apps. Smartphones have become pervasive due to the availability of office applications, Internet, games, vehicle guidance using location-based services apart from conventional services such as voice calls, SMS messages, and multimedia services. The explosive growth of mobile communication brings substantial burden to the mobile security management. A recent report shows that the number of apps in the Google Play Store has risen from 16 thousand in December 2009 to more than 2 million in February 2016. Notably, the mobile data traffic amount has reached 3.7 exabytes per month in 2015. However, the increase in the number of applications is highly impaired by the prevalent malware. Among different operating systems, Android becomes the most popular platform due to the open architecture of its system. Unfortunately, smartphones running on Android system have gradually become the main target of attackers [1] and are infected by malicious apps. This circumstance reveals the urgency of enforcing mobile app security.

At the same time, academic researchers and commercial antimalware companies have realized that the conventional signature-based and static analysis methods to detect mobile malwares are vulnerable. In particular, the prevalent stealth techniques, such as encryption, code transformation, and environment-aware approaches, are capable of generating variants of known malware [2] [3]. This has led to the use of behaviour-analysis, anomaly-analysis, and dynamic-analysis based methods. Since a single approach may be ineffective against the advanced techniques, multiple complementary approaches can be used in tandem for effective malware detection. Existing survey papers [4] [5] [6] extensively cover the smartphone OS security, malware growth, study of anti-analysis techniques, and existing detection methodologies.

Mobile platforms (i.e. Android) security solutions such as vulnerability assessment, malware analysis and detection techniques are broadly classified into: 1) static; 2) dynamic and 3) hybrid. Static analysis methods analyze code without actually running it, hence they are quick, but they have to deal with false-positives (i.e. when benign is classified as malware). Dynamic analysis techniques monitor the executed code and inspect its interaction with the

system. Though time-consuming they are effective against malware obfuscation [7]. Hybrid approaches leverage the good of both the static and dynamic analysis methods.

Other security solutions can be categorized as rule-based [8] or feature extraction based machine-learning models [9]. Inappropriate feature selection can degrade the performance of the model, to generate false-positives. Moreover, the number of features under the problem must be small sized and effective as an on device anti-malware solution. Feature reduction methods combined with statistical measures such as mean, standard deviation, chi square, and *haar* transforms can be used to identify the prominent attributes responsible for malicious actions. Learning models can be created by analyzing the features such as processor, memory usage, battery consumption, system call invocation, network activity etc. that can be used with the clustering or classification algorithms to predict anomalous behaviour.

Finally, following is the list of malicious activities that have been reported or can be employed across mobile devices that are infected with some sort of malicious activities:

- Privilege escalation attacks were leveraged by exploiting publicly available Android kernel vulnerabilities to gain root access of the device [10]. Android exported components can be exploited to gain access to the dangerous permissions.
- Privacy leakage or personal-information theft occurs when users grant dangerous permissions to malicious apps and unknowingly allows access to sensitive data and ex- filtrate them without user knowledge and/or consent [13] [14].
- Malicious apps can also spy on the users by monitoring the voice calls, SMS/MMS, bank mTANs, recording audio/video without user knowledge or consent [15].
- Malicious apps can earn money by making calls or subscribing to premium rate number SMS messages without the user knowledge or consent.
- Compromise the device to act as a Bot and remotely control it through a server by sending various commands to perform malicious activities.
- Aggressive ad campaigns may entice users to download potentially unwanted apps (PUA's), or malware apps [11].
- Colluding attack happens when a set of apps, signed with same certificate, gets installed on a device. These apps would share UID with each other; also any dangerous permission(s) requested by one app will be shared by the colluding malware. Collectively, these apps perform malicious activities, whereas, their individual functionality is benign. For example, an app with READ_SMS permission can read SMS messages and ask the colluding partner with INTERNET permission to ex-filtrate the sensitive information to a remote server.
- Denial of Service (DoS) attack can happen when app(s) overuses already limited CPU, memory, battery and bandwidth resources and restrains the users executing normal functions.

2.2 Mobile Data Protection and Privacy Preservation Techniques

Mobile devices equipped with powerful sensing, computing and networking capabilities have proliferated lately, ranging from popular smartphones and tablets to Internet appliances, smart TVs, and others that will soon appear (e.g., watches, glasses, and clothes). One key feature of such devices is their ability to incorporate third-party apps from a variety of markets. This poses strong security and privacy issues to users and infrastructure operators, particularly through software of malicious (or dubious) nature that can easily get access to the services provided by the device and collect sensory data and personal information.

We now briefly describe the four major techniques or/and frameworks that have been widely used in the literature to provide protection from data loss and leakage, and privacy for sensitive user data usage specifically in Android platforms:

- *Android source code modification:* Android's source code is openly available and this makes it easy to implement security extensions by modifying the Operating System directly. Doing so allows researchers to alter significantly the existing system and is the most flexible option among the existing ones.

  The Android ecosystem is getting mature; however, its permission model does not show signs of moving towards more fine-grained permissions. In fact, around 44.8% of the Apps have been found to request permissions that they do not need, thus violating the least-privilege principle [16]. This, combined with lack of flexibility in the security model configuration, is what led researchers to propose several Android security extensions. In [12] the authors describe CRePE, a context-aware system capable of enforcing fine-grained policies in Android phones. The context is defined as a set of variables such as position, time, light, etc. This approach is interesting but makes a limited use of sensors, requires the user (or a third-party) to manually define a set of rules and doesn't take into account interaction of the user with the device.

  The automatic use of context tracking to dynamically switch security profiles has been investigated in MOSES [17]. This work introduces the concept of "mode of use" and security profiles to enforce different policies for accessing applications and data. Switching profiles doesn't require user intervention but the system still requires the user, or a third-party, to provide the profile definitions.

  In order to reduce the need for pre-defined policies, [18] introduces a context-aware and adaptive security framework for Android which makes use of sensors data to inform a user about whether a requested action poses a low or a high security risk. They model the risk assessment as a Multi-criteria Decision Making problem and use Analytic Hierarchy Process to deal with it. This system could be used to bypass the need for user-defined policies, but the authors don't consider user awareness in their model.

- *Firmware Modification:* A less invasive technique when it comes to implementing security extensions to the Android OS is firmware modification. This technique involves modifying some files on the system to extend the capability of the OS without flashing a complete ROM. The two main drawbacks are that root privilege is required and that heavily-customized ROMs may network well together with the extension. Still, this approach usually works on a wide range of devices without requiring changes to the source code and is therefore an interesting method.

  DeepDroid [19], a dynamic enterprise security policy enforcement scheme, allows for the definition of fine-grained context-based policies. Their approach is based on the dynamic instrumentation of both Dalvik and Native code. In addition to the ordinary information about the permissions requested by an App, it can extract extra information (e.g., an URL associated with an http request). Being targeted at enterprise, DeepDroid requires a pre-defined set of fine-grained policies to be effective. Moreover, it fails to consider the user interaction and to exploit user feedback.

- *Application Repackaging:* Application repackaging is a technique which bypasses completely the need to modify the Android source code or the firmware installed on a device. Instead of extending the security model of the operating system, repackaging works by modifying the Apps before installing them. Hooks and policy enforcement logic are injected into the App binaries and therefore systems using this technique do not need to adapt the Android OS to their needs. The main drawback of repackaging systems is that the security guarantees are only as good as the tool that analyses and repackages the Apps. If a malicious App is obfuscated well enough, it could prevent the hooks from being inserted into its code, thus making the system useless. Aurasium [20] is an example of this technique.

- *Sandbox-based approaches:* Boxify [21] leverages Android's isolated processes to run applications in a sandbox. A key advantage of this work is that it doesn't require any system nor application modification as it is implemented as a normal Android application. As most systems it suffers from the lack of kernel support for policy enforcement, as that would require custom kernel extensions. Moreover, the authors themselves acknowledge that Boxify requires a wide set of permissions to be able to provide them to the sandboxed Apps. This could be exploited by malicious Apps due to the fact that Android doesn't provide any mean to drop permissions at runtime. While being an interesting general purpose Access Control mechanism, the authors don't provide information about applications of their system. Another novel approach is introduced by [22] with the FlowFence system. FlowFence performs the equivalent of taint analysis by making data and control flows of the Apps explicit. This system is targeted at IoT security, but its core concepts could also be applied to the Android OS.

2.3 Security in NFC/RFID based communication channels

Today, we use contactless smart keys to unlock our cars, even start the engine without inserting the keys. This is one of the many industrial applications which grant services or privileges based on the physical proximity between the communicating devices. These industrial applications uses the most popular short range communication technologies called Radio Frequency Identification (RFID) and Near Field Communication (NFC) for establishing contact between the communicating pairs. Other widely used applications that use short-range contactless smart cards based on RFID or NFC includes supply chain management, e-passport, access cards (such as building, parking, highway toll fee collection and public transports), medical implants, and Point-of-Sale (PoS) systems [23].

Unfortunately, due to the inherent weaknesses in underlying wireless communication, the RFID/NFC systems are exposed to a wide variety of security and privacy attacks. Thus, it subverts the security and usability advantages provided by these authentication systems. In particular, so-called relay attacks [24] [25] are one of the many distance hijacking attacks that exploits the radio communication technology of RFID/NFC systems. In relay attacks, to impersonate a victim's card within the proximity to the reader, a proxy device (or often referred to as ghost) which emulates a contactless smartcard is placed near the reader. On the other end of the communication point, a mole (or often referred to as leech) which acts as reader is placed nearby the victim card. The proxy forwards all the messages to the mole which acts as a fake of an authentic reader to the victim card. With respect to the FC-Scanner application, the relay attacks could make a FC-Scanner to scan a SmartTag even in the cases, where, the owner of the smartphone is away from the product that carries the SmartTag. This could be done without the user knowing about the attack. Using relay attacks, an adversary could perform too rapid scanning, and it can also disturb the correct

functionality of the context-based service composition applications running on the TIS platform by feeding it the incorrect context. The distance between such a proxy and mole can be increased as far as the communication delay is kept sufficiently short. Example instances that show wide existence of relay attacks are, in [26], where authors showed a successful relay attack over more than 300 miles, and in [24], where authors demonstrated relay attacks on passive keyless entry and start for over 50 meters. Further, in [27], successful relay attacks over more than 110 meters are demonstrated using three NFC smartphones.

2.4 Discussion on Related Work

The focus of TagItSmart is on the objects tagged with the SmartTags and the resulting information about the objects and the surrounding environment that can be retrieved through FC-Scanners. The collected information is then used to provide the specific services and at the same time to create other innovative services for the customers. Hence, it requires ensuring the security and reliability in the usage of the FC-Scanner enablers. In the related work section, we have discussed the most popular security threats and their corresponding state-of-the-art solutions with respect to the security of mobile applications and privacy of user data, both residing on the mobile devices. The techniques discussed include, mobile malware detection, ensuring security and privacy of sensitive user data, and the reliability and security issues with the communication channels. These security and privacy measures are required because the FC-Scanner application will be running on the user mobile devices and it will be operating on the sensitive user data (i.e., camera data, GPS etc.) along with the access to Internet connectivity. Thus, an adversary could exploit the vulnerabilities in the: (i) underlying OS, (ii) applications running parallel to FC-Scanner, (iii) FC-Scanner software, and (iv) communication channels used for the communication with the SmartTags and TIS platform.

TagItSmart takes these existing security techniques and concepts as its starting point, with extensions defined to meet the security and privacy requirements coming from its overall implementation system as well as from its specific use-cases. To this end, we research and develop various security solutions to ensure the secure usage of the FC-Scanner enablers. Our solutions cover an array of security threats that could potentially harm the working mechanisms of an FC-Scanner. We start with the novel technique that protects the sensitive data from being misused by a malicious FC-Scanner application via the use of sandboxes. As there are various malware detection frameworks available in the literature, we take a step further, and we propose stealthier and modular malware analysis architecture that is robust against advanced malware attacks. We further discuss the methods that can be used for ensuring the protection from adversaries targeting the communication channels that are used in the TagItSmart system for accessing the data from SmartTags and FC-Scanners. Furthermore, to ensure the authenticity of the FC-Scanner application, we propose the possible use of large scale software attestation technique. Finally, we propose various techniques to ensure the security and privacy of user data (in terms of data loss and leakage) by using context-aware access control techniques.

Section 3 - Secure and reliable scanning

In this section, we present a new sandboxing technique that ensures the correct behaviour and functionality of the FC-Scanner application. Our approach ensures that the functionality of FC-Scanner app, while it accesses the sensitive data, remains unaffected in the presence of a strong adversary. We also present a malware analysis framework that helps the mobile app developers to implement stealthier and modular malware analysis architecture. We show that our proposed framework could be used to further secure the FC-Scanner application from possible advanced malwares (i.e., the malwares that could bypass the existing malware detection frameworks). Furthermore, we discuss the possible use of remote attestation process for authenticating the FC-Scanner application. Finally, we propose solutions to ensure secure communication between the FC-Scanner and the TIS platform.

3.1 System level security

In this section, we investigate and design various techniques that will help us to ensure: (i) the correct behaviour of an FC-scanner application while operating on sensitive data which is either extracted from SmartTags or from mobile sensors, and (ii) the integrity of the FC-Scanner application against malware and code-injection attacks. A mobile device running a malicious or compromised FC-scanner software may risk data modification attacks (i.e., the FC-scanner's extracted data from SmartTag will not match with its send data to TIS platform) and data leakage attacks (i.e., send users' sensitive data to third party), thus threaten user privacy and TIS functionalities.

3.1.1 Sandboxing Techniques

Current smartphone frameworks use permission-based access control for data sources and sinks, but they do not control flows between the authorized sources and sinks. In particular, these frameworks only support permission-based access control on sensitive data, which is ineffective at controlling how apps use data once they gain access. This method has already proved to be inadequate, as is evident from the growing reports of data-stealing malware in the smartphone [28] and browser extension spaces [29], [30]. The fundamental problem is that users have no choice but to take it on faith that an app will not abuse its permissions. Instead, we need a solution that forces apps to make their data use patterns explicit, and then enforce the declared information flows, while preventing all other flows.

Let us consider, for instance, a barcode scanner app such as the FC-scanner. This app obviously needs the *CAMERA* permission to scan barcodes and QR codes along with the access to the GPS data. It also needs the *INTERNET* permission to search information on a code. These are all legitimate uses, and thus permissions are required, but a hypothetical malicious barcode app could also misuse the granted permissions to send all the user data to a remote collecting server without user consent. To address this misbehaviour of apps, a system is needed that requires consumers of sensitive data (i.e., in this case, a mobile application) to declare their intended data flow patterns, which the system can enforce with low overhead, while blocking all other undeclared flows.

3.1.1.1 FlowFence: Operating on Sensitive Data via Sandboxes

We propose a practical data protection system called *FlowFence*, which provides a framework for an alternative permissions system for mobile OS, in the form of a trusted service that helps apps operate on sensitive data safely, in a manner specified by the

developer, while allowing users to monitor or prevent the flow of sensitive information out of the device. FlowFence achieves this by explicitly embedding data flows and the related control flows within app structure, thus ensuring that an app is completely isolated from the sensitive data it operates upon, while still allowing the app to perform operational access on the data. FlowFence allows the app developer to define his own operations on private data, and provides a safe and isolated environment (referred as a sandbox) to execute those operations, while never disclosing actual sensitive data to the application.

The results of the operations on sensitive data are provided to the app in the form of a token, without disclosing the sensitive data itself. Tokens can be used for subsequent operations within FlowFence's safe execution environment. FlowFence tracks the permissions used in generating each token at runtime. Users can define a policy that restricts dangerous combinations of permissions, without hindering safer combinations. For example, GPS data or user contact data could be restricted from being sent over the network by the app, while the app could still send the scanned code to the TIS platform, or retrieve other information from the network. In short, FlowFence allows:

- an app developer to execute code on sensitive data safely, with more assurance that the data remains protected, by preventing or detecting flows of sensitive data with harmful combinations of permissions.
- the user to monitor the flow of sensitive data to off-device destinations.
- the user or an app developer to define a policy that specifies which combinations of permissions can and can't be used.

3.1.1.1.1 FlowFence Overview

Our proposal, FlowFence, offers a trusted component that manages sensitive data, preventing apps from accessing the data directly, while allowing them to use the data in their code. One way to do this would be to use a simple delegation pattern, where the trusted service directly exposes the functionalities needed. However, this solution would imply having a public API on the trusted component able to satisfy all possible usages of private data in an Android application, which seems a requirement very difficult to satisfy.

Instead of designing a "good enough" API, FlowFence allows app developers to define their own operations on private data, in the form of a SODA (Sensitive Operation Defined by the App) - a self-contained, stateless function that produces an output from specified inputs. Our system would then execute this SODA within a trusted and isolated environment, while returning the result of the computation as an opaque handle to the app. The app cannot access the results directly, since it only gets back a handle, but it can pass the handle to subsequent SODAs for further operations on sensitive data. Inputs to a SODA can be ordinary data or handles, and the result returned is arbitrary, but is always returned as a handle. The FlowFence service keeps tracks of the real values associated with the handles, as well as the permissions used in building up the handle. Only the disclosure of the real data that is embodied in the handle is subject to policy, not the operations on the data itself.

This achieves four key goals:

- access to private data is kept separate from the application itself;
- the transformations over private data are defined by developer;
- since access to private data is done in a safe and controlled environment (via a trusted service), we can precisely track the combination of permissions used by an app when private data is externalized.

- permissions from the user are not needed for sensitive data accessed via FlowFence, except when sensitive data is desired to be sent out of the trusted service, e.g., to the network, in unencrypted form.

The FlowFence service offers functionality to declassify handles and output raw values to destinations such as the device's display or the network (sink); since it is the trusted component itself that does this declassification, and since a handle carries all the permissions used in its computation, FlowFence can control and monitor these flows via policy. Such control and monitoring is simply not possible with the standard Android permissions system that only has knowledge about the data (and sinks) that will be available to the app.

3.1.1.1.2 FlowFence Architecture

FlowFence uses a client-server model, where the client is an Android app (i.e., FC-Scanner) and the server is a service that exposes functionality to execute SODAs on sensitive data. An app requests the execution of a SODA via a blocking inter-process communication (IPC) call to the service. The app sends the SODA and any required additional parameters as arguments of the call to the service, and then waits for the result from the service. The *FlowFence* service tracks usage of sensitive data during the SODA's execution, checking against a policy whether those uses are allowed or not.

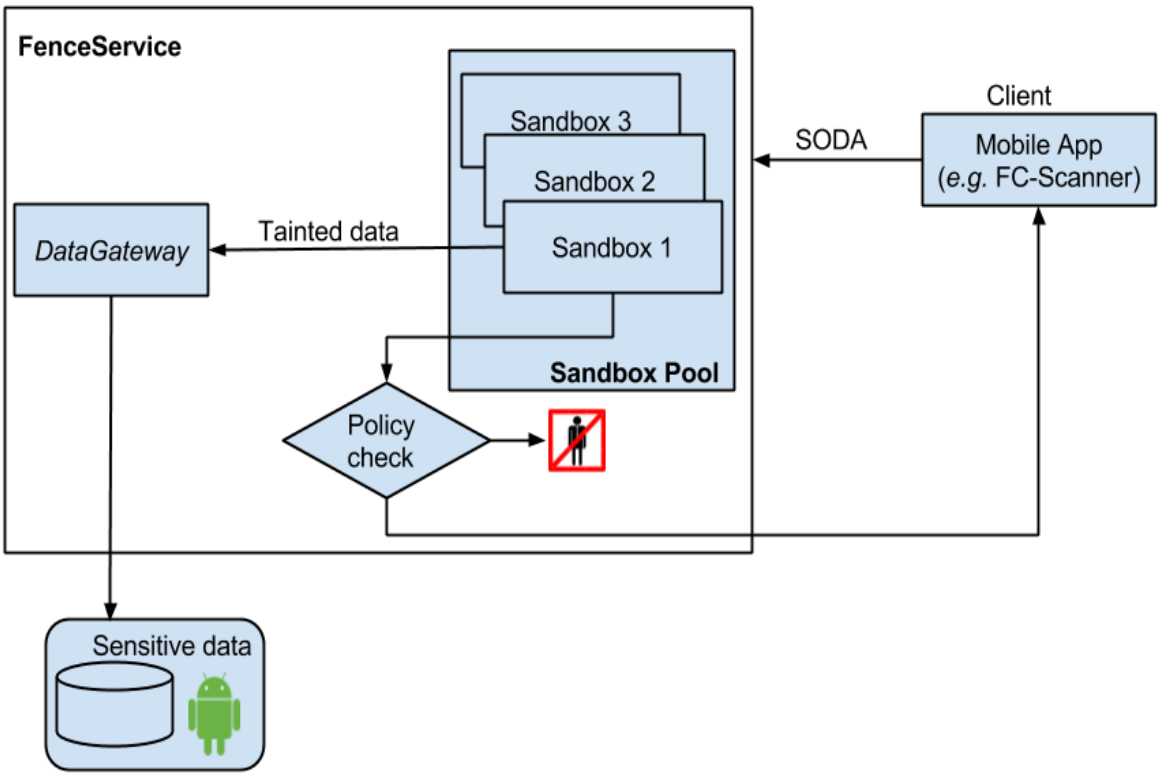

Figure 1 *FlowFence: Architecture Overview*

The architecture of *FlowFence* is illustrated in Figure 1. In particular, the service is composed of three main Android services:

- Sandbox pools, which realizes the isolated environment for the SODA execution;

- *DataGateway* service, used to access private data from the Sandbox;
- *FenceService*, the service which encapsulates the other two and is called by the application.

We use the *DataGateway* service as a way to keep the Sandbox isolated and prevent an app from accessing sensitive data directly. Apps need to use the *DataGateway* service to retrieve or output the sensitive information. Thus, at the *DataGateway*, we can monitor and track the sensitive data ("sources") accessed by a SODA as well as where this data flows to ("sinks").

FenceService receives the calls from an app to execute a SODA. It then selects a Sandbox from a pool of identical sanitized sandboxes. The architecture uses a pool of sandboxes to scale better in case of multiple concurrent requests and for resiliency. The parameters given to *FenceService* for the SODA call are passed by value into the Sandbox - this prevents SODAs from modifying parameters to leak data from the sandbox. When a SODA has finished executing, its result is then returned to *FenceService*, which checks (with a call to the *DataGateway* service) the sources and the sinks accessed by the SODA. *FenceService* stores the result and returns an opaque handle to the result back to the app. *FenceService* also calculates the taint of the SODA, starting it at the union of the taints on the inputs, and adding any inputs used directly by the SODA as it executes. The final taint of the SODA is also associated with the returned handle.

If the SODA accesses a sink (e.g., the network), policy is checked to see if the access should be logged or prevented. For example, if the taint set when the network is accessed includes the camera, which implies that there is a potential flow from an image acquired by the camera to the network.

The results from the SODAs are returned in the form of an opaque handle, called *FenceToken*. A *FenceToken* is composed of two fields: the encrypted result of a SODA execution, and a taint field that tracks the cumulative set of permissions used (i.e., sensitive sources used) to obtain the token. *FenceService* also adds a digest (generated with a cryptographic hash function) to the token, in order to verify the integrity of the token if used as input in subsequent computations. By encrypting information and returning it to the caller application, we can offload the data management to the application itself, which can use this *FenceToken* as input for other SODAs or simply store it. In the future, we will avoid the cost of encrypting computation results by keeping them in the *FenceService* (up to a maximum amount of time) and only returning a symbolic token to the application. Encrypted values will be available to applications only with an explicit request.

To prevent data leakage between SODAs with different taints, SODAs cannot store data between executions, except by returning that data at the end of the execution. *FenceService* also offers functions to display information on the screen (e.g., via dialogs or toast notifications). Depending on whether one trusts Android to display sensitive information securely, these may or may not be subject to policy. *FenceService* also includes operations to declassify information and send it through a sink (e.g., sending GPS coordinates to the network), subject to policy. These are needed to keep the app functional while being able to track or prevent risky flows of sensitive data.

Apps can get operational access to sensitive data without requesting access to the raw sensitive data. This potentially reduces the number of permissions that apps need to get approved from users.

To use *FlowFence*, the developers of FC-Scanner app needs to write a SODA script that will contain all its operations that this app wants to do on sensitive user data. Inside the SODA, the developer can retrieve sensitive data with calls to the *DataGateway* service. The *FenceToken* resulting from a SODA execution can be displayed by calling *FenceService*. The end user would simply need to install the *FlowFence* application (and optionally define a policy) to use the system.

3.1.1.1.3 FlowFence Working Methodology

When the service executes the *runSODA* method, it retrieves the calling application package name. The service then generates an *IBinder* object to be used to identify this *SODA*. Subsequently, it chooses a clean Sandbox from the pool and calls the Sandbox method *execSODA(String sourceAPK, String sClass, Map inputTokens, Bundle args, IBinderauthToken, IBinderDataGW)* (number 2 in Figure 2). Inside this method, the Sandbox binds to the *DataGateway* service, after retrieving a proper binder from the *IBinder* it receives as argument. The Sandbox is implemented as an Android service with the property *isolatedProcess* set to true. Such a service runs in a separate process with no permissions of its own, and the only communication possible is through the service API. Since the only binder object that is passed to the Sandbox is the *DataGateway* binder, the only way to retrieve or send out sensitive information is by calling the *DataGateway* service. The isolation of the process prevents any unapproved I/O communication, even if the *SODA* is written poorly.

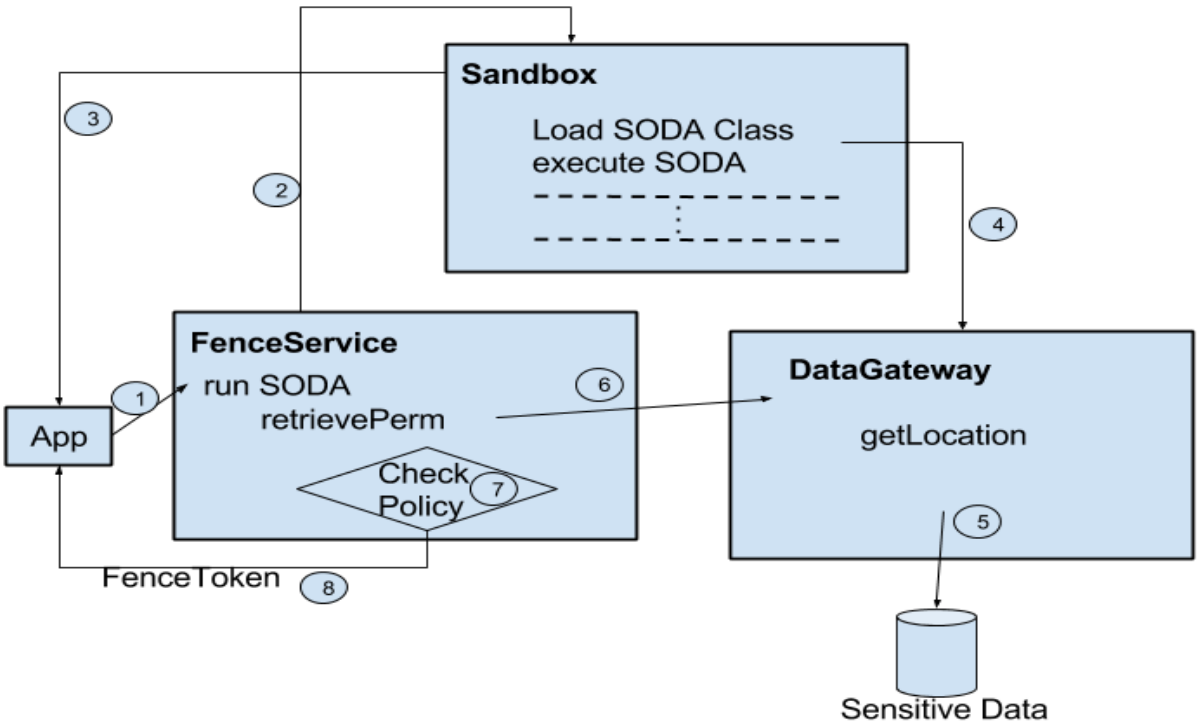

Figure 2*Sample execution of a FlowFence*

The Sandbox loads the class implementing a *SODA* from the caller's *APK* (number 3 in Figure 2). We then use Java reflection to instantiate a new object of the *SODA* class, and

execute the method that runs the *SODA*. The result of this execution will be returned to *FenceService* as a *String*.

The DataGateway service is a local service that can only be called inside FenceService application. This service acts as interface for the *SODA* to gather sensitive data, e.g., a developer would call *datagw.getLastLocation(IBinderauthToken)* to get the current location in a SODA (number 4 in Figure 2). The DataGateway will call the Android LocationManager (number 5in Figure 2) and retrieve the desired data, which will be returned to the SODA. The authToken is necessary to identify the SODA. Every call made by a SODA sets one or more permission flags in the DataGateway. This allows tracking every data access during the *SODA* execution and applying security policy. The DataGateway service also offers a registration method to FenceService. During the registration, the DataGateway acquires a token that is used to authenticate to FenceService for subsequent calls.

At the end of the execution of a SODA, FenceService will make a call to the DataGateway method *intretrievePermissionsUsed(IBinderauthToken)* to retrieve the permissions used by that SODA, using the previously generated IBinder as an identifier (number 6 in Figure 2). FenceService encrypts the result of the SODA execution and creates a new FenceToken. The retrieved permissions are added to the permissions required by all the FenceTokens used as input to the SODA, and result in the final set of permissions associated with the SODA and the returned FenceToken. This FenceToken object contains the encrypted result, the permissions used by the SODAs and a digest of the contents added by FenceService to verify the token integrity. The token is returned (number 8) to the calling app, where it can be stored, used as input of another SODA or be displayed by calling a FenceService output method.

FlowFence uses notifications and dialogs as a way to display information on top of another application without disclosing sensitive data. The ability to display information on top of the calling apps without disclosing sensitive data will be added in the future, either by using Access Control Gadgets (like the ones described in [31]) or by drawing on top of the app window using available Android APIs.

3.1.1.2 Stealthier and Modular Malware Analysis Sandbox

Mobile devices can access and store personal information (e.g., location, photos, and messages), and thus, are appealing to malware authors. One of the most promising approaches to analyse malware is by monitoring its execution in a sandbox (i.e., via dynamic analysis). The majority of malware sandboxing solutions for Android rely on an emulator, rather than a real device. This motivates malware authors to include runtime checks to detect whether the malware is running in a virtualized environment. In that case, the malicious app does not trigger the malicious payload. The presence of differences between real devices and Android emulators started an arms race between security researchers and malware authors, where the former want to hide these differences and the latter try to seek them out. In this section, we present *Mirage*, a malware sandbox architecture for Android focused on dynamic analysis evasion attacks. We designed the components of *Mirage* to be extensible via software modules, to build specific countermeasures against such attacks. To the best of our knowledge, *Mirage* is the first modular sandbox architecture that is robust against sandbox detection techniques. As a representative case study, we present a proof of concept implementation of *Mirage* with a module that tackles evasion attacks based on sensors API return values.

3.1.1.2.1 Motivation

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

Similar to other mobile apps, the FC-Scanner could also be used to infect the user mobile device. For instance, an adversary could insert some malicious code using the repackaging technique and publish the FC-Scanner at Google Play or other similar marketplaces. However, the malware analysts will be examining all the uploaded apps for possible malwares by using static analysis and dynamic analysis techniques. During such analysis, the sample (i.e., an app submitted for the analysis) runs in a sandbox. A sandbox is an isolated environment where malware analysts can execute and examine untrusted apps, without risking harm to the host system. Unfortunately, sandboxes run on top of the emulators (such as QEMU and Genymotion) which present some hardware and software differences (i.e., artefacts) with respect to real devices, which can also be recognized at runtime by apps. Artefacts are imperfections that make a sandbox distinguishable from a real device. By detecting these artefacts, the malicious FC-Scanner can easily recognize whether it is running or not on a real device. Thus, a malicious app can exploit emulator detection to evade dynamic analysis and show a benign behaviour, instead of the malicious payload. Relying on such mechanism, malware authors might spread a new generation of malicious apps, which they would be hardly detectable with current dynamic analysis systems.

3.1.1.2.2 Proposed Architecture

To address the aforementioned security threat, we propose *Mirage*, a malware analysis sandbox which is robust against evasion attacks. One of the key features of *Mirage* is that it is composed of processes that execute inside the operating system, and software that runs outside the emulator. This feature allows Mirage to be not tied to a specific analysis system.

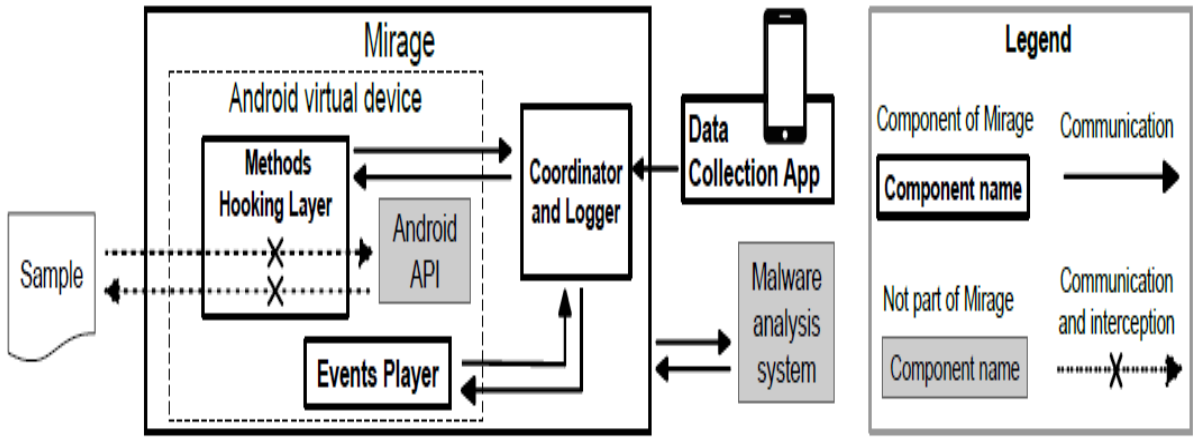

Figure 3*Mirage architecture: Components and their interactions*

In Figure 3, we illustrate the four main components of Mirage which are the Methods Hooking Layer, the Events Player, the Coordinator and Logger, and the Data Collection App. The details of these four components along with their functionality and interaction with each other and with the system are as follows:

- **Methods Hooking Layer:** The first component of *Mirage* architecture is the Methods Hooking Layer. This component executes as a process in the Android operating system. The main function of Methods Hooking Layer is to intercept calls to methods of Android API and manipulate their return value. Such manipulation occurs just whenever the original returned value may reveal the presence of the underlying emulator. Relying on this component, we can address the majority of behavioural differences. As an example, we can return a well-formed telephone number when a sample asks for TelephonyManager.getLine1Number(), instead of the default one (which in an emulator always begin with 155552155, followed by two random digits). Since it is possible to predict which artefacts the Methods Hooking Layer introduces, we can use such component to hide them as well. Moreover, hooked methods should perform minimal computation to reduce the risk of detection via computational timing attacks.
  The code of the Methods Hooking Layer executes directly on a compiled operating system image. Hence, such code is “debuggable” without modifying and compiling every time the Android source code. In compliance with the modularity of sandbox components requirement (such as “stealthiness” and modularity of the components, consistency of bogus data, monitor known evasion attempts, and no modifications to the Android source code and emulator), the modular sub-architecture of the Methods Hooking Layer makes it flexible with respect to changes. The Methods Hooking Layer divide hooks by target artefacts, thus they are editable without touching the other hooks. Moreover, such sub-architecture allows researchers to share their proof of concepts or mature modules in a common framework. However, system constants expose some artefacts as well (e.g., the ones contained in A ndroid OS Build).
- **EventsPlayer:** Real mobile devices generate many events in response to external stimuli, hence hooking methods calls and manipulation of their return value is not enough to simulate such asynchronous behaviour. In order to make our runtime environment as realistic as possible, we need the Events Player replay recorded or generated streams of events in the emulator. Besides the touch screen, the main sources of events are sensors (e.g., accelerometer, thermometer) and multimedia interfaces (e.g., camera, microphone).

The Events Player replays tidily the streams of events, respecting their order. The accuracy of values domain is crucial to build a stealthy sandbox. Indeed, the sandbox would be vulnerable to detection and fingerprinting, if the injected events do not resemble the ones that come from a real sensor (e.g., they are out of range). Similarly to the Hooking Layer methods, the Events Player uses only tools from Android, Android SDK and emulators, without requiring any modification.

- **Coordinator and Logger:** The Coordinator and Logger, as it is clear from its name, has two roles: to coordinate and to log. Its first role as coordinator consists in ensuring consistency of bogus data, which the other components inject into the emulator. Whenever the Methods Hooking Layer loads a new module, or when the Events Player opens an events stream, we have to instruct the coordinator on how to manage such hooks or events stream in accordance with the other modules. A deep study of the interaction between Android features lead to a set of rules, which the coordinator feature is able to interpret. For example, data that sensors acquire are interdependent (e.g., accelerometer and GPS). Moreover, actuators on the device (i.e., the screen, the notification LED, the ash, speakers and the vibrator) can also influence data that sensors record (e.g., speakers may influence the microphone). The second role of this component consists in logging what happens inside the sandbox. This logging feature of the Coordinator and Logger is useful to have an insight on which detection techniques the samples are probably exploiting. In addition to that, the logging feature is even more useful to signal whenever a sample attempts to use a known technique which the sandbox is not able to cope with yet. In this way, Mirage is able to monitor all possible evasion attempts. The Methods Hooking Layer reports to the Coordinator and Logger every suspect or evidence about the analysed sample. The Coordinator and Logger could manage the analysis process entirely. As an example, this component could handle tasks such as sample submission or the presentation of results.
- **Data Collection App:** The task of the Data Collection App is to collect information from real mobile devices. Then, the Coordinator and Logger will inject such information into the Methods Hooking Layer and into the Events Player. The goal of this process is to hide artefacts in the emulator. Indeed, acquiring data from different smartphones and tablets models allows creating emulator instances with different characteristics. At the same time, this approach also reduces the risk that malware authors detect a particular image. The app is also responsible of capturing events streams on the real device, and stores them in a compact and easy to replay representation. The Data Collection App can retrieve information from real mobile devices available in a laboratory, but a real advantage would be to collect data with crowd-sourcing. On one hand, in a laboratory scenario researchers could ask their colleagues or students to kindly give their help by installing the app and uploading data. On the other hand, in a crowd-sourcing scenario companies could include the Data Collection App in their mobile app. Adopting a freemium pricing strategy, companies can freely distribute their software for free in exchange for data collected from the device. With an app with a wide user base, it is also possible to acquire \disposable" data on demand. As an example, an antivirus app may offer to the user an extension of the license or a month of premium features, if she agrees to share with the company her sensors events for the next ten minutes. In both scenarios, we highlight that data collection must be respectful of the privacy of the participants, e.g., applying perturbation on collected data [10]. Such perturbation is meant to alter information in such a way that avoids exposing the contributing user's identity (e.g., biometrics, habits) and, at the same time, preserves the characteristics of the device.

3.1.1.2.3 Tackling Evasion Attacks with *Mirage*

In this section, we start with artifacts discovery and analysis, and end with the implementation of the *Mirage*. Artefacts are imperfections that make a sandbox distinguishable from a real device. To put ourselves in attacker's shoes, we studied the Android sensors API to find out which sensors artifacts malware could leverage to evade dynamic analysis.

Device	getName	getVendor	getFifoMaxEventCount
Real	BMI160 accelerometer	Bosch	5736
QEMU	Goldfish 3-axis Accelerometer	The Android Open Source Project	0
Genymotion	Genymotion Accelerometer	Genymobile	0

Table 1 *Example of return values for Nexus 5X accelerometer in real devices, and vanilla emulators (i.e., QEMU and Genymotion)*

In our analysis, we considered the sensors embedded in real devices and the ones simulated by virtual devices. For each sensor, we called all methods available in the Sensor class. As an example, in Table 1 we show the discrepancies in terms of return values for accelerometer methods on real and emulated Nexus 5X. Malware authors can rely on those discrepancies to develop simple detection techniques (a single conditional statement is enough). We refer to these techniques as *static heuristics*, since they exploit an artifact due to the Android API, which is not related to events streams. The accelerometer, thanks to its wide availability, is particularly well suited for broad-spectrum heuristics.

Android sensors	Portrait			Landscape		
getStringType	*values[0]*	*values[1]*	*values[2]*	*values[0]*	*values[1]*	*values[2]*
accelerometer	0	9.776	0.813	9.776	0	0.813
magnetic field	0	0	0	0	0	0
light	0	n/a	n/a	0	n/a	n/a
pressure	0	n/a	n/a	0	n/a	n/a
proximity	1	n/a	n/a	1	n/a	n/a
relative humidity	1	n/a	n/a	0	n/a	n/a

Table 2 *Constant values produced by sensors in QEMU, grouped by screen orientation*

In the literature, researchers already pointed out the feasibility of dynamic heuristics, in which they exploited sensors events that emulators generate [49]. We investigated further: for each real mobile device at our disposal, we registered callback methods to receive changes in sensors state. By applying the option SENSOR DELAY FASTEST, we got those states as fast as possible. In our experiments, we observed that collecting an incoming stream of events for ten seconds is enough for our purpose. We collected sensors data from real mobile devices in three different scenarios: lying on a table, while typing and leaving them in a pocket while walking. Then, we repeated the data collection task on QEMU and

Genymotion emulators. Such emulators allow only two modes of screen rotation: portrait and landscape.

During our initial experiments, we were able to observe some differences between real and emulated motion sensors. In real mobile devices, we noticed that motion sensors (e.g., the accelerometer) quickly oscillate among a small range of values, even when the device is lying on a flat surface. In emulators, we noticed that it is possible to stimulate the accelerometer by changing from landscape to portrait mode. In contrast, without rotating the screen, each motion sensor in emulators produces the same value. Table 2 records the constant values that each sensor in QEMU produces. It is worthy of note that some sensors in QEMU produce values only along one axis, so in Table 2 we mark the cells related to the other two axes as n/a.

In order to tackle the evasion attacks based on mobile sensors with *Mirage*: (i) we included in our *Data Collection App* the code we used for artifacts analysis, and (ii) we patched the discrepancies in return values using information we obtained from the *Data Collection App*.

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

For the evaluation of our proposal, we will be developing a *SandboxStorm* app. Such app includes the aforementioned static and dynamic heuristics. The *SandboxStorm* app could be considered as a genuine mobile app such as FC-Scanner, which contains malware code along with the capabilities that helps it to detect a difference between an emulator and a real device. Hence, it can fool the existing emulators during dynamic malware detection process.

3.1.2 Remote attestation

Large numbers of smart connected devices, also named as the Internet of Things (IoT), are permeating our environments (homes, factories, cars, and also our body—with wearable devices) to collect data and act on the insight derived. Ensuring software integrity (including OS, apps, and configurations) on such smart devices is then essential to guarantee both privacy and safety. A key mechanism to protect the software integrity of these devices is remote attestation: A process that allows a remote verifier to validate the integrity of the software of a device. This process usually makes use of a signed hash value of the actual device's software, generated by dedicated hardware. While individual device attestation is a well-established technique, to date integrity verification of a very large number of devices remains an open problem, due to scalability issues.

In this section, we will investigate possible directions for the use of Trusted Platform Model (TPM) 2.0 to develop a secure application for smartphones. Increasing adoption of smartphones in recent times has begun to attract more and more malware writers towards these devices. Among the most prominent and widely adopted open source software stacks for smartphones is Android that comes with a strong security infrastructure for mobile devices. However, as with any remote platform, a service provider or device owner needs

assurance that the device is in a trustworthy state before releasing sensitive information to it. Trusted Computing (TC) provides a mechanism of establishing such an assurance. Through remote attestation, TC allows a service provider or a device owner to determine whether the device is in a trusted state before releasing protected data to or storing private information on the phone. However, existing remote attestation techniques cannot be deployed on Android due to the unique, vm-based architecture of the software stack. Therefore, we will first investigate novel solutions to perform software attestation on remote IoT devices as we believe that similar solutions could later be used for authenticating the mobile applications that runs on smartphones with embedded TPM chips.

3.2 Communication reliability and security

In this section, we discuss the security strategies that we will apply on how the FC-scanner, namely a mobile app running in a specific device, securely interacts with services and repositories in the backend.

3.2.1 Vulnerabilities in the FC-Scanner to backend communication

Because the Internet is an open network and security was not included by design in the original IP layer, additional protocols must be used at the transport layer as well as at the application layer. This, to guarantee that two communicating devices keep communicating with the same devices (i.e., prevent hijacking). Furthermore, still at the transport layer, it must also be guaranteed that the data that two devices exchange cannot be intercepted, read or modified during the exchange. This service is provided by Transport Layer Security protocols, or TLS, and its predecessor, Secure Socket Layer protocols. However, while security at the transport layer secures the point-to-point communication between two devices in an IP-based network, of which the Internet is the most prominent, TLS does not say anything about the legitimacy of the communication at the application level. In the context of the FC-Scanner, TLS would make sure that the mobile device and the backend are securely connected, and even if the communication is intercepted by a third party, all that the third party would see is seemingly random *0s* and *1s*. However, how does the backend know that the FC-Scanner is used by a legitimate owner? Or that the FC-Scanner hasn't been compromised and, while the owner is legitimate, the FC-Scanner app tries to open a second backdoor, to circumvent the security mechanisms that are outlined in this deliverable? This is why authentication at the application layer is required, which is implemented with API keys. At the application level, FC-Scanner and backend communicate via HTTP. HTTP is the de-facto standard on the Web to transfer files in a request-response manner. Each time the FC-Scanner wants to communicate with the backend, it sends its API key as part of the header payload.

3.2.2 Authentication and API Keys

Within a mobile device's space, there are always possibilities of various security threats. Hence it is important that, should an app be compromised, swift and decisive actions can be taken to prevent the app from being used. After all, a mobile device could be stolen or infected by a malware. So, how can it be ensured that an app is legitimately permitted to communicate with the backend. While there are numerous ways to do this, API keys are an ideal choice for RESTful services.

An API key is a unique and crypto-secure token given to an application after the authentication process to which a role is assigned. An API key is the only value that the mobile application needs to authenticate and authorise a request. The advantage of using API keys is that personal access credentials (i.e. username and password) are not shared,

as would be the case with basic access authentication for example. If an API key is compromised, a new API key can be generated and the compromised API key be invalidated, without affecting a user account. Furthermore, using API keys prevents cross-site request forgery attacks, because each individual request is authenticated. The backend does not keep a state between requests.

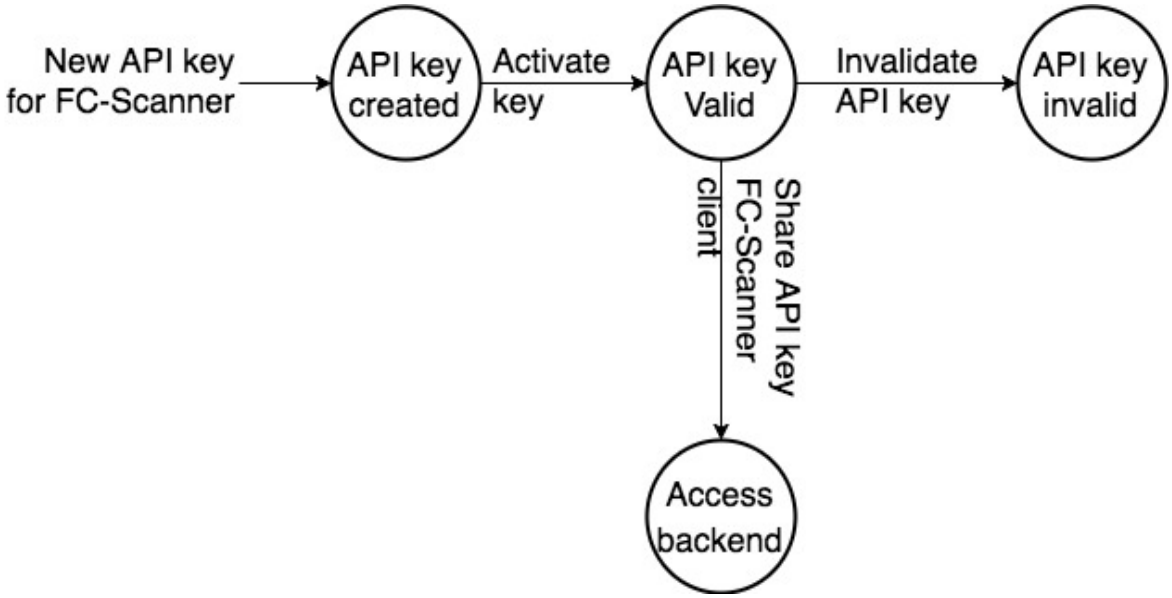

Figure 4 *Lifecycle of an API key*

The backend exposes endpoints to virtual entities as RESTful APIs. Each time anFC-Scanner makes a request to the backed, the request must be authenticated and authorised by a valid API key. Based on the API key, the backend determines which resources the application is permitted to access and the operations applications can perform on those resources. This is shown in Figure 4. A new API key is created for the FC-Scanner. This API key is uploaded to an FC-Scanner app. Each time the FC-Scanner accesses the backend, it uses the API key. The backend will allow the FC-Scanner mobile app to access predefined resources under well-defined permissions. The API key can be invalidated, for example if the mobile phone is stolen or the API key is published. An invalid API key cannot be used. However, all other API keys remain valid.

3.2.3 Secure communication

The open architecture and philosophy of the Internet, which was key for its global success, means that if two parties want to communicate securely, they must find a way to exchange keys to establish a secure channel in plain sight. This problem has been solved with a public key infrastructure, which means that two actors trust a third party, the certificate authority, to vouch for their identity. Tools are provided for most platforms that implement transport layer security (TLS) protocols. Users will mostly not be aware that two devices establish a secure tunnel to enforce communication security enforced using TLS. But TLS establishes a private and secure tunnel between two applications in an open network, even if the entire communication, from negotiation, to transmission, to ending transmission is intercepted by a third party. TLS guarantees the identity of the mobile application and the backend (using

public key cryptography) and guarantees that the communication cannot be intercepted and then manipulated.

```
GET /products HTTP/1.1..Authorization:
<FC_SCANNER_API_KEY>..Content-Type: application/json..Accept:
*/*..Host: api.fc-scanner-backend.com..accept-encoding: gzip,
deflate..Connection:keep-alive.
```

Figure 5 *HTTP header with FC-Scanner API key*

At the application layer, FC-Scanner and backend communicate via HTTP, respectively HTTS, “S” standing for the secure transportation layer. The FC-Scanner authenticates itself by sending its FC-Scanner specific API key in the header. An example for such a header is shown in Figure 5. The backend will verify if the API key provided in the header is valid and if the request if valid, i.e. is the resource the API key requested within its scope and does the API key have the necessary permissions.

3.2.4 Role-based authorisation

The backend uses a role-based access control model. The key concepts are scopes and permissions. Access is managed by creating roles, which assign permissions to a scope:

- Role scope: A scope is a group of zero or more resources (e.g. virtual entities)
- Role permission: Permission specifies the operation that can be performed on resources.

The role-based access paradigm does assign permissions to users that users assume through API keys.

Section 4 - Enhancing security and user privacy based on context

The role of the contextual information is to offer information regarding the situation in which the user of a service is currently in. This includes physical characteristics of the context (e.g. location, address, time and hour), as well as technical ones (device model, carrier information, frequency, hardware information and protocol versions). Miele et al. have proposed in [46] a contextual model specific to mobile devices, meant to extend data services by minimizing the amount of information required to be stored on phones.

In PaaSword [45], a complex and updatable semantic model [47] is available to a central application owner, allowing him to define complex rules for allowing the execution of certain actions or services. Thus, authentication and authorization mechanisms have been proposed and developed. The models, subsequently being transformed and enforced dynamically into a XACML format, allow the straightforward definition of context-based policies; contextual information is typically location, network, device, usage and pattern related. E.g., requests between certain time frames and from known geographical locations can be considered valid or typical for a specific user, while any deviation or reporting frequency, not.

In this section, solutions will be investigated and presented that use the contextual information collected from mobile sensors (i.e., sensor modalities) or user-defined policies, to enhance the security of the overall system (i.e., front-end which includes FC-scanner and SmartTags), and to ensure the proper functionality of all its components. In particular, we include security and privacy techniques that protect users' privacy not only against malicious applications, but also against benign ones (such as FC-scanner) by keeping the user-in-loop while operating on sensitive data, and context-based security profiles that are associated with a set of policies to control the access to applications and data.

4.1 Privacy Protection in Mobile Applications

Recent history has shown that the users' privacy must not only be protected against malware, but also against benign Applications. In fact, cases have been reported of widely used Apps requiring too many permissions or leaking user information to their servers intentionally [32]. To make things even worse, several models of cheap Android handsets have been found to be transmitting personally identifiable information, including call logs and the content of text messages, to remote servers [33].

Despite the introduction of several extensions to the Android security model, little attention has been paid to possible mechanisms able to check whether a sensitive action is intended by the user or not, based on contextual information. This shortcoming severely limits the security of a system and its ability to protect users. For example, let us consider the case of a geo-tagging photo App. Such an App would typically require camera and location information for legitimate purposes. Moreover, Internet access might be needed to share the pictures among its users. In this scenario, nothing prevents the App from leaking the user location continuously while running in background or while the screen is off.

The permission granting process could be improved by using sensors' data and other contextual information, thus making it a well-informed task. Examples of such information could know whether the screen is on/off, the time since last user input, and the presence of the user in front of the screen or the rotation/position of the phone itself. Permission enforcement systems should exploit this data to try to block privileged but unsolicited actions requested by an App, therefore limiting potential privacy breaches.

4.1.1 User-in-the-Loop Verification System

In order to ensure that the FC-scanner application does not misuse the user data that are also residing on the same device, we propose the use of *"UserLoop"*, which is a User-in-the-Loop verification system for Privacy Protection in Mobile Applications. *UserLoop* makes use of sensor data and human feedback to enforce user awareness of sensitive actions performed by mobile applications. We present the architecture of the proposed system, describe our prototype implementation and report on the thorough evaluation we run to assess its performance both with regards to its effectiveness and its runtime footprint.

The concept of context-based access control has already been investigated on mobile devices by the research community, but no system so far has exploited contextual information to put the user and its interaction with the device in the centre of the security model. Moreover, existing systems have a limited definition of context which is mostly static and is used only to change the set of policies depending on environmental attributes such as the time of the day or the location of the device. *UserLoop*, on the other hand, can apply different policies based on the instantaneous value of any of its contextual variables, thus adapting automatically to the user's needs. Furthermore, the *UserLoop* client App provides an easy and effective way for the user to detect suspicious Apps which perform sensitive requests behind his back.

With *UserLoop*, we achieve a fine-grained control policy over Android Apps by making the default Android permission model context-dependent. Our system doesn't require installing custom ROMs or radical changes to the OS, thus giving it the ability to be deployed on different devices and different Android versions with minimal to no changes.

4.1.1.1 Architecture

The architecture of *UserLoop* is shown in Figure 6. *UserLoop* is composed of three main modules: an *Xposed* module named *XUserLoop*, a *service* and a *client App*. XUserLoop is the Policy Enforcement Point (PEP) and is used to relay Android's UID-based permission check to the *UserLoop* service. The service is itself composed of different modules which encapsulate its main functionalities. The Logger module is in charge of storing permission requests together with contextual information in order to provide useful statistics to the front-end App. The Context Tracker is the module which gathers data from the sensors and keeps track of the current context. The Policy Manager acquires contextual information from the Context Tracker and provides *XUserLoop* with policies to enforce. It also communicates with the Policy Provider, which is where the policies are stored. Finally, the client App, the user-facing part of the system, is used to manipulate policies and show the user useful information on the Apps and their permission requests. The client can also update the *UserLoop* configuration whenever the user makes a change to a policy.

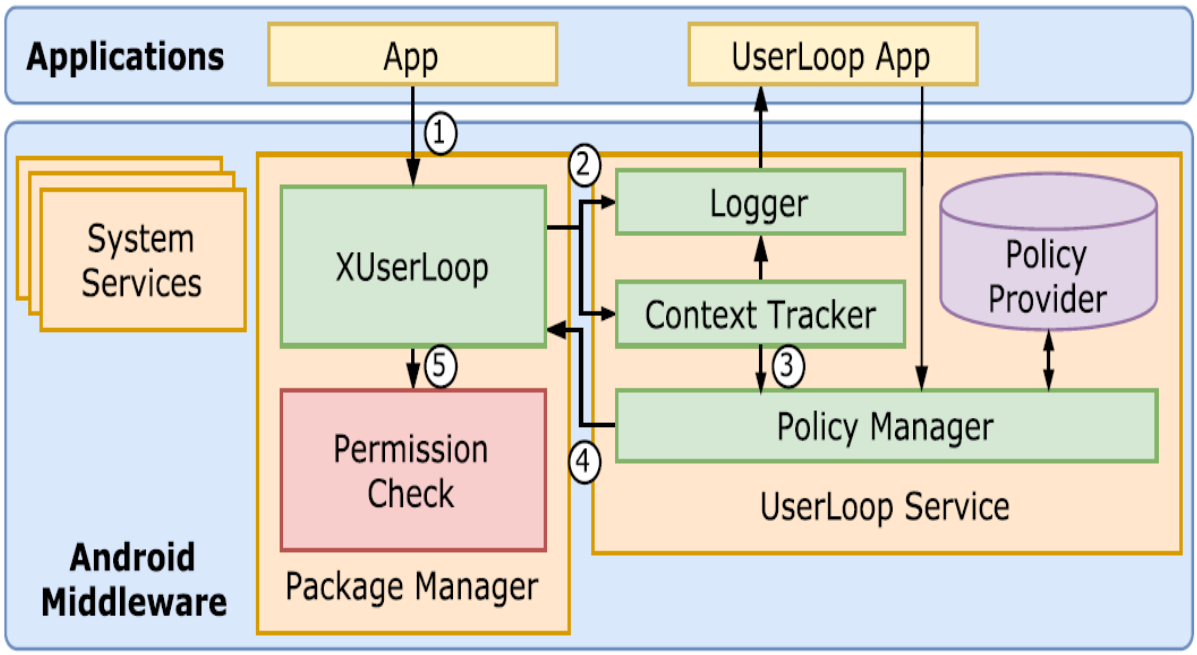

Figure 6*UserLoop Architecture*

The numbered arrows in the diagram follow the flow of a permission request inside the system. Whenever an App requests a privileged action through the Android Framework, a permission check is performed inside the *PackageManager.checkUidPermission(String permName, intuid)* method (1). The string *permName* uniquely identifies the permission being requested, while *uid* is the Linux UID of the requesting App. If the check performed by Android has a positive outcome, *XUserLoop* intercepts the return value and performs additional checks by forwarding the request to the *UserLoop* service (2). Here the request is logged in a database and handed to the Context Tracker, which appends contextual information and hands it to the Policy Manager (3). At this point, the system goes through the policies to determine if one of them applies and the result is sent back to the *Xposed* module (4). The final step is the actual enforcement of the policy (5) which can result in oneof the following scenarios: (i) No policy applies, and (ii) At least one policy applies.

In the first case, the system applies the default policy. Such policy is user-defined and could grant or deny the permission based on the configuration set by the user inside the *UserLoop* App. In the latter case, the first matching policy is applied. *UserLoop* also provides a *whitelist* and a blacklist that can be used to bypass the rule system in special cases. The entries to those lists are tuples composed of a package name and a set of permissions. The use of a *whitelist* is especially useful in that it can be used to grant vital permissions to important services like messaging Apps. At the same time, the *blacklist* can be used to manually deny specific permission to Apps that the user does not trust.

4.1.1.2 Working methodology

Here, we explain the working methodology of the major components of *UserLoop* Architecture in detail:

- **XUserLoop–**

XUserLoop is the module in charge of placing the hooks into the Android OS. As the name suggests, it relies heavily on the *Xposed* framework to achieve this goal.

XUserLoop intercepts the Android booting process and, at the right moment, registers the *UserLoop* service, thus making it available to the client App. It also gets a reference to the system server Context instance, as it needed for most operations involving the Android framework.

By hooking the *checkUidPermission(String permName, intuid)* method, *XUserLoop* becomes the PEP of our security model. Whenever the Android PackageManager grants permission to an App, this module intercepts the return value of the check in order to let the *UserLoop* service perform its additional checks. If Android's permission checking routine denies a request, *XUserLoop* simply returns the original value as we are only interested in making the Android security model stricter. Thus, *UserLoop* never grants a request that was denied by Android.

- **Context Tracker –**

The Context Tracker is one of the main modules of our system and provides contextual information to the rest of *UserLoop*. In *UserLoop*, the context consists of a mix of data gathered from the sensors and of information about software-based states. This module is one of the key elements of our system. All the policy enforcements performed by *UserLoop* uses the information about the context to grant or deny a permission request. Thus, modelling the user context in an effective way is crucial in making the user more secure.

Being interested in understanding whether the user is using the phone or not, the most valuable sensors are the rotation vector sensor, the accelerometer, the light sensor and the proximity sensor. The rotation vector and the accelerometer are used to estimate the position of the device in the 3D space and also to estimate the level of physical engagement of the user. For example, if the accelerometer reports that the phone is in a still position it is very likely that the user is not actually using it, even if the screen is on. The proximity sensor is instrumental in checking if the phone is upside-down on a table, in a pocket or if the user is facing the screen. Finally, the light sensor can augment the proximity data and can also be used to check whether the user is sleeping or not.

In addition to these common Android sensors, *UserLoop* makes use of other information from the OS. We identify five values which can significantly improve the value of *UserLoop's* security model: the screen state, the lock state, the visibility state of the App, the process state and the time since the last user input. The first two are simple on/off boolean values and are used to represent, respectively, if the screen is on or if it is locked. The visibility state of the App is also a boolean value which encodes whether the App is in foreground or in the background when it makes a permission request. The process state takes a value of either system or user and enables us to differentiate between system Apps and user-installed Apps. Finally, the time since the last user action is recorded. This is useful to recognize scenarios in which the screen is on but the user is not paying attention and is not actually interacting with the device. For this variable, we keep track of both the global time since the last input and the time since the last input inside the App making the permission request.

- **Policy Manager –**

Whenever an App requires permission, the request reaches the Policy Manager module with contextual information attached to it. Policies are defined as an ordered list of tuples

(permissionType, permission, dangerous, conditions, action),

Where

- *permissionTypeϵ{SINGLE,GROUP,NULL}* defines if permission refers to a permission or to a permission group.
- *permissionϵPermissions*UPermissionGroupsU*{NULL}* is the name of a permission or a permission group.
- *dangerousϵ{TRUE, FALSE,NULL}* defines if the rule applies to dangerous permissions, normal permissions or both.
- *conditions* is a list of conditions composed of a *UserLoop* contextual variable, a comparison operator and a value. Such conditions are concatenated through the use of multiple *AND* boolean operators.
- *actionϵ{GRANT, DENY}* defines the action that must be taken if all the conditions are satisfied.

In the policy definition, NULL attributes are considered to be matching regardless of their actual value. We note that the user is spared from dealing directly with the policy implementation language thanks to the client App.

UserLoop eliminates the risk of overlapping or conflicting policies by checking them in a cascading manner. Because of this, the closer the policies to the start of the policy list the higher their priority. Policy Manager also holds a global policy which has an empty condition and therefore defines the default outcome when no other policy applies. The final user can decide if this global policy should grant or deny requests not covered by any rule. The Policy Manager interacts with the Policy Provider to retrieve and store policies. Thus, the latter is basically storage for policies.

In order to provide a more effective control on the permissions, they are grouped according to the Android permission grouping. Such grouping differs from an Android version to another and so groups are created at runtime by inspecting the Manifest class. Using groups allows, for example, using of a single *UserLoop* policy for both coarse and fine location requests.

For the evaluation, we chose 6 free Apps: *Facebook, Skype, ES File Explorer, 360 Security - Antivirus, Text Free by Pinger, Inc* and *Glide*. The first four are widely known and have more than 1M downloads while the others are less known but still have over 100,000 downloads. *Facebook* and *Skype* were chosen because, even if they come from well-respected companies, their history raises privacy concerns. ES File Explorer is a famous file manager which has recently started showing intrusive ads and including adware, according to many user comments on the Google Play Store1. 360 Security - Antivirus is a free antivirus which claims to also optimize background processes, memory space and battery power. We chose to include it in our tests as an example of a “boosting” app, because such Apps often come with intrusive ads and nagging notifications2. The last two Apps, Text Free and Glide, were reported to send potentially sensitive data to both its servers and to third-parties [34].It is

worth noting that as a next step, project developed FC scanner applications will be used for evaluation.

- **Client App -**

We provide a front-end app to *UserLoop* in order to let the user review and tune the policies shipped by default. Moreover, since we gather statistics about the permission requests, we are able to show some useful information to let him quickly grasp if an App is suspicious. For this reason, the *UserLoop* client App contributes to our security model by giving feedback to the system.

Table 3*Policy set*

ID	Condition	Action
R0	Location when App in foreground, screen on, phone unlocked	GRANT
R1	Dangerous permission, App in background	DENY
R2	Dangerous permission, no touch in the last 20 seconds	DENY
R3	App in foreground, screen on, phone unlocked	GRANT
R4	Screen on and unlocked, recent input inside the App	GRANT
R5	Screen off, low light	DENY
R6	Screen off, low proximity	DENY
R7	SEND SMS permission, screen off	DENY
R8	CALL PHONE permission, screen off	DENY
R9	Non-dangerous permission, screen on, phone unlocked	GRANT

For the assessment of our system, we deployed the set of policies defined in Table 3. During our preliminary testing we found that accelerometer and gyroscope data wasn't very useful with regards to our aim of making sure the user is in the loop. This is because the current iteration of our system uses the instant value from the sensors, neglecting the time dimension. Further work is required to see if an activity recognition-based approach would yield better results.

In the initial phase of the security evaluation, we downloaded and installed the Apps on our test device. We then proceeded by manually opening them and creating accounts where required. After that, we performed manual testing by opening each App and interacting with it. In this phase, we modified some contextual values on purpose in order to activate the policies we deployed. Finally, we rebooted the phone and recorded the activity of the Apps for two days.

4.2 Security and privacy based on user context

The previous section detailed an approach for security based mainly on the context of the *phone*. In this section, we will provide an overview of security based on the context of the *user* currently in possession of the phone. This involves using the smartphone to determine the context that the user is currently in.

In the early 2000s, much research in the area of activity and context recognition focussed on using wearable sensors mounted on the body to obtain data. These sensors were often specifically for one task, such as gait recognition, and were mounted on the body in the specific place where the most appropriate data for the problem at hand could be obtained. The advent of the smartphone with it ubiquitous use and multiple sensors has meant that researchers have focussed on using this device for the opportunistic gathering of sensor data. A smartphone contains many sensors, such as a microphone, accelerometer and gyroscope and is usually carried at all times by the user making it an ideal device to gather information. The smartphone has enabled the vision of ambient intelligence to become a reality where the goal is to perform context recognition in the background using the available sensors opportunistically on a smartphone.

4.2.1 Current State of the Art in Context Recognition

Context recognition aims to determine the current context of the user in order to make it available to applications. The primary types of context awareness are; location, identity, time and activity. Context-aware applications are able to alter actions of the smartphone or present options to the user based on the current context. For the context module in the TagItSmart!, the current environment that the user is in will be the focus of our attention. This might also include an indication of the current activity of the user.

Research has shown that one of the key modalities for determining the context a user is in is sound[35]. Sound has been used to determine the activities and context that a user is in over the course of a day. In locations such as bathroom [36], office and workshop [37], kitchen [38], and public spaces [39], sound has been shown to be effective at identifying the current environment. Mesaros et al. [40], performed an evaluation of an ambient sound classification system for a set of over 60 classes, illustrating the kind of performance that can be achieved.

The determination of the context of a user from their smartphone requires a framework with several components. For each component, there are a myriad of options which a researcher can use in order to provide the required performance or determine the required context. The first component of the framework is the pre-processing of the sound data obtained from the microphone. Pre-processing involves taking the raw data and transforming them into a format that can be used by machine learning algorithms. A common method is to separate the data stream into sections of contiguous data instances, termed *windows*. These are then used as the base data to extract relevant features of the sound data. To extract features from the data there are a myriad of methods that are used such as Mel-frequency cepstral coefficients (MFCCs).After the data set has been formed from the sound data stream, a classifier is required to perform classification. Many methods have been used to perform classification that range from early machine learning methods such as k-nearest neighbour to state of the art techniques such as deep learning. Dubey et al. [41] aim to identify the mood of the wearer by unobtrusively recording the ambient sound and then processing the noise signal and using a C4.5 decision-tree classifier to determine the mood of the user. Chu et al.

[42] use the Mel-frequency cepstral coefficients (MFCCs) which describe the audio spectral shape of a signal. The aim is to identify environmental sounds such as restaurants, casinos and inside a moving vehicle. Overall accuracy is in the 70% range. Rossi et al. [35] present AmbientSense, which performs real-time ambient sound recognition on a smartphone. The application analyses the ambient sound sampled from the smartphone's microphone. The system can run on the smartphone or in combination with a server. There were 23 ambient classes that were classified with an overall accuracy of 58%. A system which aims to use a deep learning framework is DeepEar, Lane et al. [43]. Four coupled deep neural networks of stacked Restricted Boltzmann Machines perform the classification of sound data. It performs four classification tasks; acoustic scene classification, stress detection, emotion recognition and speaker identification. The feasibility of using smartphones is examined, with the algorithm being deployed on a low power digital signal processor in a smartphone to reduce power consumption. Results show that if the application runs continuously it only uses 6% of the battery a day.

An alternative to using sound to determine context is used by Liang et al. [44]. Scene classification is performed using the data from smartphones sensors. The data from the microphone, light, WiFi and Bluetooth sensors are used to build an ensemble recognition model which is based on dynamically-weighted majority learning. Results show that the most accurate sub-model is the audio classifier, which is significantly more accurate than the light, WiFi and Bluetooth classifier. This justifies the focus on audio data to perform context recognition.

4.2.2 Context Recognition in TagItSmart!

The aim of the context module in TagItSmart! is to identify the current context of the user. This will enable taking context of the user into account during SmartTag scanning process and other relevant smartphone operations. The aim is to provide security through context recognition.

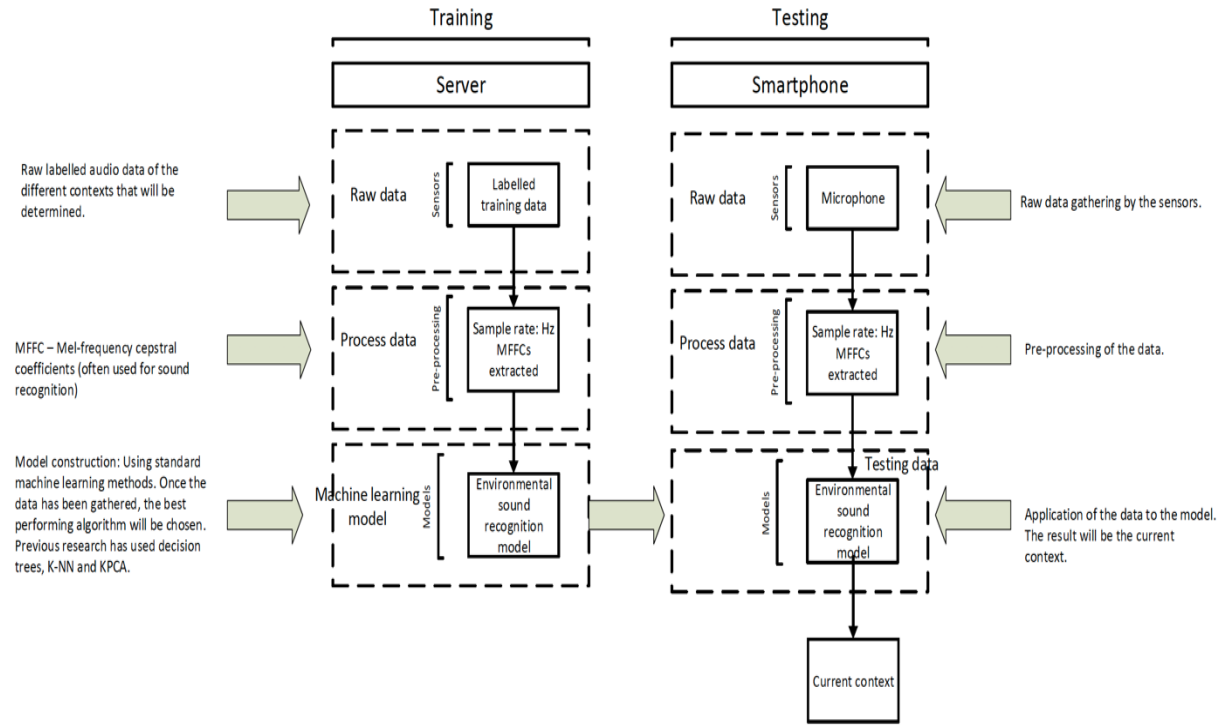

Figure 7*The framework of the context recognition module*

4.2.2.1 Context recognition module

The aim of the context recognition module is to determine the current context of the user by using the audio stream from the microphone to identify the environment that the user is currently in.

The framework for the context recognition module is detailed in Figure 7. The first phase is to construct the models of the sound data. Audio recordings from the environments are labelled. Pre-processing is the next step. The aim of pre-processing is to derive features that will enable the machine learning algorithm to distinguish between different contexts. The Mel-frequency cepstral coefficients (MFCCs) have been shown to be effective in distinguishing between contexts, and therefore these will be extracted from the raw data. After pre-processing, the next step is to construct the model of the data using a machine learning algorithm. Current research has used a variety of algorithms, including support vector machines and decision trees. A number of algorithms will be evaluated, with the one that performs best on the problem of context identification being selected. Once the models have been constructed, the training phase is complete. The models will be made available within the context application for the phone to use to classify audio data.

The app on the phone will provide the algorithm for context recognition to occur using the testing phase. The smart phone will use the microphone to collect samples of sound data representing the ambient environment that the user is currently in. This data is pre-processed and then the data is applied to the model and the resulting label will give the current context.

4.2.2.2 Security through context recognition

The context recognition module will determine if the user is in one of the contexts that are recognized. The aim of the context module is to use context to provide an additional measure of security through the disallowing of the scanning of SmartTags in certain environments. The environments can be set by the user from a closed set of contexts that the module is able to detect. The aim is for scanning to be prevented in situations where the security of private and confidential transactions is at risk. For example, banking transactions should only be conducted at home or at work, not on the street on in other locations. Limiting risky transactions in places where there are a large number of people, for example supermarkets. Unsafe environments for sensitive operations are prohibited.

For example, if the user is in a busy crowded place, it might be determined that banking services on the phone are disabled. This is so that sensitive and secure transactions are not performed where other people may be able to discover information, such as a bank account number and screen swipe pattern, about a user.

In the context of TagItSmart this will involve blocking the scanning of certain codes if security and privacy rules are not met.

- Crowded places where shoulder surfing can occur.
    - At an ATM
    - On a train
    - In a crowded public space

At an ATM, actually the problem can be not only that someone may be shoulder surfing (they can do this regardless if someone is scanning or not) but even if the person who is scanning is alone, if the FC-scanner application is compromised, it can see the PIN being typed or what the account balance on the screen is. Another similar example could be a public toilet; even without shoulder surfing a compromised FC-scanner application can take pictures in such environments.

Additional situations of interest are the use of additional context to further improve the performance of identifying if a human is involved in the scanning progress as well as protecting from too rapid scanning attempts, which in some situations can be attributed to malicious attempts or be detrimental to the backend.

In summary, a smartphone that is context-aware will add a further layer of intelligence to the user environment. The recognition of the current situation that the user is in will allow the smartphone to tailor its actions, and in some situations, stop the user or phone performing an action that is deemed to be insecure.

Section 5 - Conclusions

We have defined the core functions of the FC-Scanner and implemented enablers for them as part of our previous WP3 deliverable (i.e., D3.1).In this deliverable, we investigate and report various security and privacy threats related to the usage of the FC-Scanner application. Based on the identified threats, we propose the use of the various novel security and privacy preserving techniques to strengthen the security of our FC-Scanner application. Our proposed solutions to enhance the security of FC-Scanner application include an adequate mixture of research oriented as well as practical techniques.

The FC-Scanner application running on a user mobile device interacts with the TIS platform using Internet for following reasons:(i) to send the scanned data to the platform, and (ii) to fetch the required information from the platform to provide a service to the end users. Furthermore, the FC-Scanner application also interacts with the SmartTags for scanning purposes through user mobile camera, and it has access to the sensitive user data (such as camera data, GPS, other data required for user-oriented service composition, etc.) residing on the same mobile device. Therefore, ensuring the security and reliability of the FC-Scanner is an important aspect of the TagItSmart system where an adversary can perform various attacks by compromising the FC-Scanner functionality or behaviour, such as leaking sensitive user data to a remote server through the Internet.

Due to the aforementioned reasons, a careful security analysis of the usage of various scanning and authentication enablers of FC-Scanner app (please refer to deliverable D3.1) is essential. Hence, in this deliverable, we investigate and analyse various security aspects of the FC-scanner usage, and we design the adequate set of security solutions to countermeasure the identified security problems. In the remaining months of this task, we will also perform the implementation of our proposed FC-Scanner security solutions in our use-cases to analyse their effectiveness and correctness in real-world scenarios.

The FC-scanner uses a large set of contextual information (extracted by using various sensors in user smartphones) to provide enriched or differentiated services to the users. These increases the FC-Scanner’s access to the sensitive user data, thus additional security measures to enforce proper access control/rights on the user data are required. In particular, the use of contextual information exposes the users to various privacy risks, thus calls for inclusion of privacy preserving mechanisms. In order to address these challenges in an exhaustive manner, we will analyse and envision various solutions (in deliverable D3.3 in M24) to ensure, the user privacy-preservation, and the correct working of the context-aware management system both for security and service customization purposes. D3.3 will also present the final enablers for FC-scanners while the integrated FC-scanner environment will be reported in D3.4 at M30.

Section 6 - References

[1] G. Inc., Android Smartphone Sales Report, 2013, (Online; Last Accessed March 17, 2014). Available: *http://www.gartner.com/newsroom/id/2665715*.

[2] Android. Bgserv, (Online; Last Accessed February 12, 2011). Available: *http://www.symantec.com/security_response/writeup.jsp?docid=2011-031005-2918-99.*

[3] Backdoor.AndroidOS.Obad.a, (Online; Last Accesed Dec. 25, 2013). Available: *http://contagiominidump.blogspot.in/2013/06/ backdoorandroidosobada.html.*

[4] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "*RiskRanker: Scalable and accurate zero-day Android malware detection*," in Proc. 10th Int. Conf. MobiSys, New York, NY, USA, pp. 281–294, 2012.

[5] G. Suarez-Tangil, J. Tapiador, P. Peris-Lopez, and A. Ribagorda, "*Evolution, detection and analysis of malware for smart devices*," IEEE Commun. Surveys Tuts., vol. 16, no. 2, pp. 961–987, 2014.

[6] Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M. S., Conti, M. &Rajarajan, M, "*Android Security: A Survey of Issues, Malware Penetration, and Defenses*," IEEE Communications Surveys and Tutorials, 17(2), pp. 998-1022, 2015.

[7] ParvezFaruki, Hossein Fereidooni, Vijay Laxmi, Mauro Conti, Manoj Singh Gaur, "*Android Code Protection via Obfuscation Techniques: Past, Present and Future Directions*," CoRR abs/1611.10231, 2016.

[8] W. Zhou, Y. Zhou, and X. Jiang, "*Hey, you get off my market: Detecting malicious apps in official and third party android markets*," in Proc. Annu. NDSS, New York, NY, USA, pp. 1–13, 2012.

[9] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "*Andromaly: A behavioural malware detection framework for android devices*," J. Intell. Inf. Syst., vol. 38, no. 1, pp. 161–190, 2012.

[10] CVE, (Online; Last Accessed Feb. 11). Available: http://cve.mitre.org/

[11] G. Andre and P. Ramos, "*Boxer SMS Trojan*," ESET Latin American Lab, Bratislava, Slovakia, Tech. Rep., 2013.

[12] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo, "*CRePE: Context-Related Policy Enforcement for Android,*" pp. 331–345, 2011.

[13] Mohammad Nauman, Sohail Khan, and Xinwen Zhang, "*Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints Mohammad*," In Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security - ASIACCS '10, p. 328, 2010.

[14] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi, "*Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies,*" In Proceedings of the 22nd USENIX Security Symposium, pp. 131–146, 2013.

[15] Sven Bugiel et al., "*XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks*". In TechnischeUniversit¨at Darmstadt, Technical Report, pp. 1 –18, 2011.

[16] Xuetao Wei et al. "*Permission Evolution in the Android Ecosystem,*" In ACSAC '12 Proceedings of the 28th Annual Computer Security Applications Conference, pp. 31–40, 2012.

[17] YuryZhauniarovich, Giovanni Russello, Mauro Conti, Bruno Crispo, Earlence Fernandes," *MOSES: Supporting and Enforcing Security Profiles on Smartphones*, In (IEEE) Transactions on Dependable and Secure Computing, 11(3): 211-22, 2014.

[18] YaserMowafi et al, "*A Context-aware Adaptive Security Framework for Mobile Applications,*" In Proceedings of the 3rd International Conference on Context-Aware Systems and Applications - ICCASA '14. ICST, pp. 147–153, 2014.

[19] Xueqiang Wang et al, "*DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices,*" In *Symposium on Network and Distributed System Security (NDSS)*, pp. 8–11, 2015.

[20] Rubin Xu et al, "*Aurasium: Practical Policy Enforcement for Android Applications*," In Proceedings of the 21st USENIX conference, pp. 1 – 27, 2012.

[21] Michael Backes et al, "*Boxify: Full-fledged App Sandboxing for Stock Android*," In 24th USENIX Security Symposium (USENIX Security 15), pp. 691–706, 2015.

[22] Earlence Fernandes et al, "*FlowFence: Practical Data Protection for Emerging IoT Application Frameworks*", In Usenix Security, pp. 531–548, 2016.

[23] "*Mastercardpaypass,*" Available: *http://www.paypass.com*.

[24] A. Francillon, B. Danev, and S. Capkun, "*Relay attacks on passive keyless entry and start systems in modern cars*," 2010, Available: *http://eprint.iacr.org/2010/332*.

[25] L. Francis, G. Hancke, K. Mayes, and K. Markantonakis, "*Practical NFC peer-to-peer relay attack using mobile phones*," pp. 35–49, 2010.

[26] L. Sportiello and A. Ciardulli, "*Long distance relay attack*," pp. 69–85, 2013.

[27] T. Korak and M. Hutter, "*On the power of active relay attacks using custom-made proxies*," pp. 126–133, April 2014

[28] ZHOU, Y., AND JIANG, X, "*Dissecting android malware: Characterization and evolution*," In IEEE S&P, 2012.

[29] HEULE, S., RIFKIN, D., RUSSO, A., AND STEFAN, D, "*The most dangerous code in the browser*," In 15th Workshop on Hot Topics in Operating Systems (HotOS XV) (KartauseIttingen, Switzerland,), USENIX Association, May 2015.

[30] CARLINI, N., FELT, A. P., AND WAGNER, D, "*An evaluation of the google chrome extension security architecture*," In 21st USENIX Security Symposium (USENIX Security), 2012.

[31] F. Roesner and T. Kohno, “Securing embedded user interfaces: Android and beyond,” In USENIX Security, 2013.

[32] Arstechnica.com, “*Your iPhone calendar isn’t private*,” 2012. Available: *http://arstechnica.com/apple/2012/06/your-iphone-c*.

[33] New York Times, “*Secret Back Door in Some U.S. Phones Sent Data to China, Analysts Say*,” 2016. Available: *http://www.nytimes.com/2016/11/16/us/politics/chinaphones-software-security.html*.

[34] Jinyan Zang et al, “*Who knows what about me? A survey of behind the scenes personal data sharing to third parties by mobile apps,*” In Technology Science 30, 2015.

[35] Rossi, Mirco, et al, “*AmbientSense: A real-time ambient sound recognition system for smartphones*,” Pervasive Computing and Communications Workshops (PERCOM Workshops), *IEEE International Conference on*. IEEE, 2013.

[36] J. Chen, A. H. Kam, J. Zhang, N. Liu, and L. Shue, “*Bathroom activity monitoring based on sound*,” in In Third International Conference on Pervasive Computing, pp. 47–61, 2005.

[37], M. Stager, P. Lukowicz, and G. Troster, “*Implementation and evaluation of a low-power sound-based user activity recognition system*,” in Eighth International Symposium on Wearable Computers (ISWC ’04), pp. 138–141, 2004.

[38] E. Miluzzo, N. D. Lane, K. Fodor, R. Peterson, H. Lu, M. Musolesi, S. B. Eisenman, X. Zheng, and A. Campbell, “*Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application*,” in Proceedings of the 6th ACM conference on Embedded network sensor systems , pp. 337–350, 2008.

[39] A. J. Eronen, V. T. Peltonen, J. T. Tuomi, A. P. Klapuri, S. Fagerlund, T. Sorsa, G. Lorho, and J. Huopaniemi, “*Audio-based context recognition*,” IEEE Transactions on Audio, Speech, and Language Processing , vol. 14(1), pp. 321–329, 2006.

[40] A. Mesaros, T. Heittola, A. Eronen, and T. Virtanen, “*Acoustic event detection in real-life recordings*,” in 18th European Signal Processing Conference , pp. 1267–1271, 2010.

[41] Dubey, Harishchandra, Matthias R. Mehl, and KunalMankodiya, “*Bigear: Inferring the ambient and emotional correlates from smartphone-based acoustic big data*,” *Connected Health: Applications, Systems and Engineering Technologies (CHASE), IEEE First International Conference on*. IEEE, 2016.

[42] Chu, Selina, Shrikanth Narayanan, and C-C. Jay Kuo, “*Environmental sound recognition with time–frequency audio features*,” *IEEE Transactions on Audio, Speech, and Language Processing*, pp. 1142-1158, 2009.

[43] Lane, Nicholas D., PetkoGeorgiev, and Lorena Qendro. “*Deepear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning*,” *Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 2015.

[44] Liang, Shuang, Xiaojiang Du, and Ping Dong, "*Public scene recognition using mobile phone sensors,*" Computing, Networking and Communications (ICNC), International Conference on. IEEE, 2016.

[45] YiannisVerginadis, AntonisMichalas, Panagiotis Gouvas, Gunther Schiefer, Gerald Hübsch, IraklisParaskakis,"*PaaSword: A Holistic Data Privacy and Security by Design Framework for Cloud Services*," In Journal of Grid Computing, Springer, March, 2017.

[46] Antonio Miele, Elisa Quintarelli, Letizia,"*Tanca: A methodology for preference-based personalization of contextual data,*" In Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, ACM, 2009.

[47] Simeon Veloudis, YiannisVerginadis, IraklisPraskakis,"*GregorisMantzas: Context-aware Security Models for PaaS-enabled Access Control*," In Proceedings of the 6th International Conference on Cloud Computing and Services Science - CLOSER, SCITEPRESS, 2016.

[48] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis,"*Rage against the virtual machine: hindering dynamic analysis of Android malware*," In ACM EUROSEC, 2014.