Ivo Miguel da Silva Rocha

Bachelor in Computer Science and Informatics Engineering

A Mobile Secure Bluetooth-Enabled Cryptographic Provider

(Dissertation Elaboration Report)

Computer Science and Informatics Engineering

DI-FCT-UNL

Orientation: Henrique João Lopes Domingos,
Assistant Professor, DI/FCT/UNL,
Nova Lincs Research Center

Co-orientation: Nuno Felipe Sousa da Silva,
Departamento de Desenvolvimento,
Multicert S.A.

Examination Committee

Chairperson: João Carlos Antunes Leitão
Raporteur: Rui Miguel Soares Silva
Member: Henrique João Lopes Domingos

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2019

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To my grandmother Rosa.

Acknowledgements

I would like to start thanking to my supervisors, Professor Henrique Domingos e Nuno Silva, for the challenge and opportunity of working on the dissertation topic and the continuous support and motivation to achieve the objectives. Especially to Nuno, who is a great guy who tried to support me as much as he could. I also want to thank all Multicert collaborators for welcomed me in such a lovely way and for giving me the opportunity to learn and grow with them personally and professionally.

I am grateful to my whole family for the endless support and encouragement throughout all my student life. It has been a long journey and every little action and encouraging word meant the world to me. I must thank to: Roberto, Sílvia, Gil, Rosa, Júlio, Manuel, Leonilde, Paulo, Cláudia, Gilberto, Carlos, Isabel, Carlitos, Rui, Cheila, and Débora.

I must thank very much and appreciate the support, strength and love that my girlfriend gave me during this 5 years journey which were very difficulty for both of us. To you Inês, thank you for believing in me. Finally, I would like to thank to my second family that helped me to break many obstacles during the last couple of years and helped me to become the person that I am today. I must thank to my second brothers and sisters: Pedro, Daniel, Didier, Filipe, André, Dinis, Abílio, Diogo, Ana, Teresa, Maria, Helena, Marco, Wilson, Diogo. And also to all the other colleagues from the Faculty of Sciences and Technology of the NOVA University of Lisbon, in particular the ones that, for the last five years, shared their time, their knowledge, their experiences, but also their tears and sweat. In a final thanks, I would like to thank my cat for always being on the computer keyboard during the writing of this dissertation.

Abstract

The use of digital X509v3 public key certificates, together with different standards for secure digital signatures are commonly adopted to establish authentication proofs between principals, applications and services. One of the robustness characteristics commonly associated with such mechanisms is the need of hardware-sealed cryptographic devices, such as Hardware-Security Modules (or HSMs), smart cards or hardware-enabled tokens or dongles. These devices support internal functions for management and storage of cryptographic keys, allowing the isolated execution of cryptographic operations, with the keys or related sensitive parameters never exposed.

The portable devices most widely used are USB-tokens (or security dongles) and internal ships of smart cards (as it is also the case of citizen cards, banking cards or ticketing cards). More recently, a new generation of Bluetooth-enabled smart USB dongles appeared, also suitable to protect cryptographic operations and digital signatures for secure identity and payment applications. The common characteristic of such devices is to offer the required support to be used as secure cryptographic providers. Among the advantages of those portable cryptographic devices is also their portability and ubiquitous use, but, in consequence, they are also frequently forgotten or even lost. USB-enabled devices imply the need of readers, not always and not commonly available for generic smartphones or users working with computing devices. Also, wireless-devices can be specialized or require a development effort to be used as standard cryptographic providers.

An alternative to mitigate such problems is the possible adoption of conventional Bluetooth-enabled smartphones, as ubiquitous cryptographic providers to be used, remotely, by client-side applications running in users' devices, such as desktop or laptop computers. However, the use of smartphones for safe storage and management of private keys and sensitive parameters requires a careful analysis on the adversary model assumptions. The design options to implement a practical and secure smartphone-enabled cryptographic solution as a product, also requires the approach and the better use of the more interesting facilities provided by frameworks, programming environments and mobile operating systems services.

In this dissertation we addressed the design, development and experimental evaluation of a secure mobile cryptographic provider, designed as a mobile service provided in

a smartphone. The proposed solution is designed for Android-Based smartphones and supports on-demand Bluetooth-enabled cryptographic operations, including standard digital signatures. The addressed mobile cryptographic provider can be used by applications running on Windows-enabled computing devices, requesting digital signatures. The solution relies on the secure storage of private keys related to X509v3 public certificates and Android-based secure elements (SEs). With the materialized solution, an application running in a Windows computing device can request standard digital signatures of documents, transparently executed remotely by the smartphone regarded as a standard cryptographic provider.

Keywords: *Public-Key Cryptography*, *Digital Signatures*, *Certificates*, *PKI*, *Cryptographic Providers*, *Key Storage Providers*, *Bluetooth*, *NFC*, *Secure Element*, *HCE*

Resumo

Os certificados digitais de chave pública em formato X509v3, juntamente com os padrões existentes para construções criptográficas de assinaturas digitais seguras, são vulgarmente usados como mecanismos de estabelecimento de provas de autenticação entre principais. Uma das características associada ao uso reforçado de segurança desses mecanismos é a necessidade de se usarem dispositivos criptográficos físicos, como por exemplo HSMs (ou *Hardware-Security Modules*), cartões inteligentes (ou *smart cards*) ou outras soluções congéneres (como por exemplo, *secure hardware-enabled tokens*). Estes dispositivos permitem gerir e armazenar chaves criptográficas ou parâmetros sensíveis relacionados, e executar internamente as necessárias operações criptográficas, de forma fisicamente isolada. Deste modo, as chaves criptográficas e outros segredos sensíveis associados, nunca são expostos externamente aos dispositivos.

Entre os dispositivos portáteis mais vulgares encontram-se os USB-*tokens* (ou USB-*dongles*), os *smart cards* (que são usados como cartões de identidade, como é o caso do cartão nacional de cidadão, mas também como cartões bancários). É ainda vulgar o uso de outros dispositivos semelhantes, tais como cartões de títulos de transporte ou usados na área de aplicações de bilhética (ou *Mobile-Ticketing*).

Os anteriores dispositivos implementam de alguma forma a funcionalidade interna de um provedor criptográfico, tendo em vista a execução segura, contida e isolada de operações criptográficas seguras, associadas ao seu uso. Mais recentemente, embora menos vulgares, surgiram implementações especializadas de *tokens* ou *dongles* para interoperabilidade via Bluetooth, de modo a evitar o uso de conexões físicas USB, de modo a poderem ser usados de forma mais cómoda, como dispositivos móveis e ubíquos.

Entre as vantagens dos anteriores dispositivos está a sua portabilidade e ubiquidade. No entanto, como consequência, são também frequentemente perdidos ou esquecidos. O seu uso implica também a necessidade de leitores, nem sempre disponíveis aos utilizadores de dispositivos portáteis e uso quotidiano (sejam *laptops* ou *smartphones*). Por outro lado, muitas soluções possuem problemas de atualização de *software* e *firmware*, com dificuldade para acompanharem a evolução de normas e padrões criptográficos.

Uma alternativa prática para mitigar os anteriores problemas é a adoção de *smartphones* convencionais como dispositivos suportando provedores criptográficos, para utilização como serviço para aplicações-cliente remotas executando em computadores convencionais (sejam *laptops* ou *smartphones*). O uso de *smartphones* para armazenamento seguro e gestão de chaves criptográficas ou parâmetros sensíveis associados, requer uma análise cuidadosa das condições do modelo adversário, tendo ainda em vista o contexto de comunicação móvel, sem fios. Por outro lado, a implementação para diferentes dispositivos implica a adopção das facilidades de segurança mais apropriadas, como são hoje disponibilizadas como suporte de desenvolvimento de aplicações e serviços, ao nível das diferentes tecnologias de sistemas operativos móveis.

A presente dissertação aborda a concepção, desenvolvimento e avaliação experimental de um provedor criptográfico móvel, disponibilizado para *smartphones* Android, acessível através de comunicação Bluetooth. O provedor implementa, de forma transparente, as funções normalizadas de um *Key Storage Provider* (KSP), para utilização de aplicações executando em sistema operativo Microsoft Windows. Na solução proposta o armazenamento seguro de chaves criptográficas privadas utiliza o suporte de programação para *Secure Elements* (SEs), em *smartphones* Android.

Palavras-chave: *Criptografia de Chave-Pública*, *Assinaturas Digitais*, *Certificados*, *PKI*, *Provedores Criptográficos*, *Provedores de armazenamento de chaves*, *Bluetooth*, *NFC*, *SE*, *HCE*

Contents

List of Figures

List of Tables

Acronyms

APDU	Application Protocol Data Unit.
ATT	Attribute Protocol.
BLE	Bluetooth Low Energy.
BR	Basic Rate.
CA	Certification Authority.
CAP	Cryptographic Algorithm Provider.
CAPI	Cryptography API.
CNG	Cryptography API: Next Generation.
CSP	Cryptographic Service Provider.
CSRK	Connection Signature Resolving Key.
DH	Diffie-Hellman.
DLL	Dynamic Link Library.
DoS	Denial of Service.
DSN	Distributed Sensor Network.
DSS	Digital Signature Standard.
E2EE	End-to-End Encryption.
ECC	Elliptic Curve Cryptography.
ECDH	Elliptic Curve Diffie-Hellman.
EDR	Enhanced Data Rate.
FIPS	Federal Information Processing Standard.
GATT	Generic Attribute Profile.
HCE	Host Card Emulation.

HS	High Speed.
IoT	Internet of Things.
IRK	Identity Resolving Key.
KSP	Key Storage Provider.
L2CAP	Logical Link Control and Adaptation Protocol.
LE	Low Energy.
LMP	Link Manager Protocol.
LTK	Long Term Key.
MAC	Media Access Control.
MAC	Message Authentication Coding.
MITM	Man-In-The-Middle.
MKSP	Microsoft Key Storage Provider.
MTM	Mobile Trusted Module.
NDEF	NFC Data Exchange Format.
NFC	Near Field Communication.
NIST	National Institute of Standards and Technology.
OID	Object Identifier.
P2P	Peer-to-Peer.
P2PE	Point-to-Point Encryption.
PKCS	Public Key Cryptography Standards.
PKI	Public Key Infrastructure.
PSS	Probabilistic Signature Scheme.
RFCOMM	Radio Frequency Communication.
RFID	Radio Frequency Identification.
RTT	Round-Trip-Time.
SDP	Service Discovery Protocol.
SE	Secure Element.
SIG	Special Interest Group.

SSL	Secure Sockets Layer.
SSP	Secure Simple Pairing.
STK	Short Term Key.
TCB	Trusted Computing Base.
TCP	Transmission Control Protocol.
TEE	Trusted Execution Environment.
TLS	Transport Layer Security.
UUID	Universally Unique Identifier.
UWP	Universal Windows Platform.
WPF	Windows Presentation Foundation.
XAML	Application Markup Language.

Chapter 1

Introduction

1.1 Context and Motivation

The evolution of the digital era and the Internet leaded to a digital world where everyone can access and navigate through the world wide web, many times interacting under non-authenticated guarantees. In many cases, users interact as anonymous entities, and some times we saw identity thefts of digital identities of persons, which can cause severe damage in a person's life. A conventional way to prove the digital identity is through digital signatures and public-key certificates based on asymmetric cryptography [30].

Digital Signatures and Public-Key Certificates. A digital signature [72] is equivalent to a traditional handwritten signature, but is more difficult to forge and can provide non-repudiation, meaning that the person who signed cannot deny that he has signed. Also, cannot be copied, tampered or altered. They are created using the users' private key and the message to sign. A digital certificate [41] contains a set of standard attributes and values, bound to the exhibited digital identity. Those attributes are issued and typically authenticated by a Certification Authority (CA). Then, certificates and digital signatures can be used electronically to prove users' identity or to validate authentication of data and documents sent in electronic messages, assuming the validity of the CA involved. Among the different attributes, the certificate contains the owner's public key, which is used by the receiver to verify digital signatures as authentication proofs of data sent by the identity of the sender. The CA, acting as a trusted party, is responsible for issuing the certificates, with canonical representations (such as the X509v3 standard), confirming not only the identity of a subject but the bind of that identity to the respective public key. The functions and operations that take place in the lifecycle management of X509v3 certificates are currently addressed by well-defined operations in a Public Key Infrastructure

(PKI) and following the functions and entities in the PKIX framework model [85].

For example, when signing an email (or a document) we can prove our identity by signing the email content (or attached documents) using private keys and provide the respective public-key certificate. Then the receiver can verify the signature. Many software products and tools have integrated components for the management and validation of digital signatures and X509v3 certificates (e.g, Web Browsers, Gmail service and tools, Thunderbird Mail User Agent or Adobe Reader, among many other very common tools in everyday use). It is also current that different entities can exhibit multiple authenticated digital identities, each one mapped on the use of a certified public key. This happens possibly for multiple authenticated identities that are used for different purposes, for example, for signatures of documents with different professional roles or using different authentication and certification policies and methods.

Protection of Private Keys. The protection of private keys is of fundamental security importance, as this is the principal means by which the Principal is authenticated. Nowadays, there are two mainly ways to store a private key, as a secrecy concern [50]. In one hand, many applications tend to store digital certificates but also the related private keys on the hard drives of computers and in the supported file-systems, which can leave them vulnerable to attack by hackers performing intrusion attacks. On the other hand, private keys can be stored in persistent memory supported by specific tamper-proof physical devices, providing more protection. By using such devices, cryptographic operations are isolated and resistant to attack vectors against applications and operating system. This option provides additional protection against theft or impersonation, as the user is able to carry the private keys away from the computers and workstations. These security hardware devices, ranging from smart cards to different variants of cryptographic tokens, can ease the management of authentication proofs being used alongside or in the place of passwords, to prove the user identity by secure digital signatures. This way, we can avoid the need to manually insert credentials.

There are several types of security tokens with different objectives and approaches. However, the most used to attend the goal of securing private keys or digital signatures are USB tokens and smart cards [21], which are usually physically connected to computers with which the user is authenticating. For example, in Europe, each citizen have a citizen card, supported by smart-card technology that stores a certificate including cryptographic keys related to the remaining identification attributes of that citizen. Also, some countries enable the citizen to use citizen cards or other forms of smart cards, to sign contracts, to sign documents or to vote electronically. Other current devices, as USB cryptographic tokens, also can be used for the same purposes. These tokens are also currently used for authentication systems, such as Google Accounts [84] or for many two-factor authentication solutions in the market [28].

Drawbacks of hardware enabled cryptographic tokens. Hardware tokens are small enough to be carried in a pocket or bag, facilitating their transportation and ubiquitous use. However, these devices are also prone to losses or thefts, and also forcing users to have them permanently, which may counteract many cases assumptions in availability and convenience. The probability of losing tokens can be reduced in some available technology by using additional components and services, such as alarms or notifications, preventive locking mechanisms or body sensors detecting the absence of such devices. The possible malicious use of stolen or lost tokens is also mitigated providing access protection by passwords or PINs, and even biometric sensors. However, many times and in many situations users are reluctant to carry a "yet another" hardware device, beyond computers or smartphones. Therefore the use of software tokens for two-factor authentication procedures supported by SMS challenges or computed in applications running in laptops or smartphones are currently adopted. This way, we avoid those drawbacks [10] and improve commodity by reducing also the security guarantees. Another inconvenient founded by users, is the need to use additional devices (such as smart card or token readers) which involves biggest annoyance and more related costs.

In addition, past literature already revealed some vulnerabilities in the above devices. Bardou et al [13] successfully discovered a method to extract the private key from several Public Key Cryptography Standards (PKCS) #11 cryptographic devices. Also, firmware vulnerabilities can pose huge risk to USB device security as this could allow tokens to be reprogrammed to steal the contents of anything written to the drives and spread malicious code to any computer [5]. Although, security tokens are external devices connected to our computers, they are used to authenticate users may times trough untrusted access networks, being vulnerable to Man-In-The-Middle (MITM) attacks [48], requiring the complementary support of secure communication channels. Once the computer is compromised, it is possible to manipulate the interoperability with the token, as well as the client-side software, accessing to token functions. Even if the token device firmware is initially correct, it can be also potentially overwritten without the user's knowledge. In this situation, the user may unwittingly connect the device to additional systems, which can be subsequently compromised.

Flexible solutions for private-keys and cryptographic protection. A more flexible and versatile solution to avoid the discussed drawbacks, and to achieve security conditions for management of digital certificates and private keys, can be the use of smartphones. The idea is to provide such requirements with an object that we carry daily and we use regularly to do multiple tasks in our daily life [25]. As in token-devices, smartphones can also be lost and stolen, but the risk is lower: they are in general under a more permanent surveillance of users, used as ubiquitous supervised devices. Smartphones are also bigger than USB tokens for example, and most of the time they are close to users. Current technology also allows for the device access protection by passwords or PINs, locking-prevention after a reduced number of unsuccessful attempts. Current

smartphones also allow for the possible combination of access-control mechanisms using secret patterns as answers to cognitive challenges, biometric fingerprints, but also, native sensor-based data or voice-recognition patterns. Existing solutions for the certificate usage in today's smartphones already involve their correct installation under verification guarantees from the device OS (e.g, Android OS framework libraries), and the use of installed certificates for any supported application. For storing private keys and verifiable access to valid and correct certificates, mobile storage options include: the smartphone hard flash-disks; SD cards; Secure Element (SE) and related hardware chips, ARM TrustZone technology levered by the ARM processor and Host Card Emulation (HCE) software facilities and design patterns [24].

The flexibility of using smartphones for the management of certificates and private keys came from the potential use of such devices to implement cryptographic operations, particularly digital signatures. A possible idea is to use such devices as mobile cryptographic providers, accessible by wireless Bluetooth (or Bluetooth Low Energy), as well as, Near Field Communication (NFC) channels. This allows the use of their cryptographic functions, on demand, by different users' devices and applications, namely running on laptops or desktops. However, the approach to implement this idea must be aware of the possible vulnerabilities too, such as the ones pointed out by Entrust [92]. These include SMS-based attacks that can redirect SMS to exploit Android-based mobile devices for illegal financial gain or malicious users' attempts to clone properly secured applications to another device. Additionally, many other threats in the mobile execution environment can be found in literature, as well as, security concerns in protecting communication endpoints for Bluetooth and Bluetooth-enabled Security Operation [19, 26, 36, 51, 59, 67, 87, 88, 93] or NFC channels [3, 15, 27, 70, 73, 91], depending on the adopted communication technology to materialize the above idea.

1.2 Objectives, Contributions and Validation

The present dissertation addresses the design, implementation and experimental evaluation of a mobile cryptographic provider for Android smartphones, used to manage, store and execute digital signatures for external devices, namely Microsoft Windows computers. The approach of such a solution aims to study technical solutions that increase the security of Bluetooth connections to support the communication of cryptographic operations between Windows-based applications and the smartphone used as the mobile cryptographic provider. In our solution the mobile device is used to store and manage X509v3 certificates and cryptographic keys, providing its services to Windows-based applications, in a transparent way. For transparency criteria, the idea is that the mobile cryptographic provider must be used by Windows-based applications, in a similar way as using a local-installed crypto provider. Included in the dissertation objectives, security guarantees must be provided for integrity, authentication and authorization properties, in accessing the functions offered by the Android-enabled mobile cryptographic provider.

Some challenges arise with the dissertation goal. The major ones are the security of the communication channel and the secure management of certificates and private keys on the smartphone. The ability of the solution to be integrated in the Android framework by using the design options for Android Secure Elements (SEs)) in the Android Programming Framework, is another important concern.

In order to build the proposed solution, the following contributions were addressed:

- Analysis of the security properties of currently-enabled Bluetooth secure channels using the more recent standardized security modes and the required security enforcements. As such, we can protect the communication between the crypto provider hosted on the computer (client-side) and the crypto provider running in the smartphone environment (server-side).;
- Analysis of Android-based storage solutions to manage X509v3 certificates and secure solutions for the protection of private keys and security-associations with the stored certificates;
- Design of software attestation guarantees for the crypto provider solution running in the Android smartphone, with protection provided according to the Android SEs specification and development guidelines;
- Development of the crypto provider solution as an Android service running under the SE model;
- Design and development of a Windows crypto provider (more precisely, a Key Storage Provider), that virtually communicates by Bluetooth with the Android crypto provider service;
- Support for certificate registration and cryptographic operations, particularly, standardized digital signatures in the Windows OS environment (client-side), executed in the remote crypto provider (server-side).

From the system model and software architecture in the design solution, we implemented a prototype, addressed as a pre-product approach and funded initiative, in partnership between the FCT/UNL and NOVA-LINCS Research Center and Multicert S.A. With the developed prototype we conducted an experimental evaluation with different assessment criteria, to validate the proposed solution, including:

- Validity testing on the transparency of the provided solution when used by Windows-based applications, selecting Adobe Acrobat and Microsoft Word and Outlook as targeted applications;
- A test bench for performance evaluation of digital signatures provided by the designed and implemented Android mobile cryptographic provider. Measuring the

latency conditions observed by Windows-based applications and understanding the components contributing to latency and operation throughput metrics;

- Performance analysis in the operation of the Bluetooth-enabled communication channel in the solution, to observe the overheads imposed by the security requirements and their support;
- Additional experimental observations on the implemented mobile cryptographic provider solution, including: packaging metrics; runtime instrumentation and impact on resources allocation; as well as, power consumption observations.

1.3 Document Organization

This dissertation report is organized in six chapters including the current one. In Chapter 2, is presented the dissertation related work and state-of-the-art, which includes the approaches to secure the communication channel, the techniques to storage and manage private keys and the Windows Crypto Provider APIs. Chapter 3 discusses the system model and architecture, and its components for our proposed mobile crypto provider. Chapter 4 presents the prototype of the proposed solution. In Chapter 5, it is presented the experimental assessment to validate our proposal and implemented prototype. Finally, in Chapter 6, we summarize the main conclusions and possible future work directions from the implementation and observed results in this dissertation.

Chapter 2

Background and Related Work

In this chapter we address different state-of-the-art approaches to accomplish the objectives presented in the previous chapter. The two first Sections (2.1 and 2.2) present the possible solutions founded in literature to our main goals: communication channel and private key storage security. These include Near Field Communication (NFC) and Bluetooth to secure the wireless channel and Host Card Emulation (HCE) and Secure Element (SE) to store the private key securely. Then, in Section 2.3, we study the available APIs used to provide access to Windows cryptographic providers. Finally, in Section 2.4, we present a critical analysis about the related work techniques discussed and also how we handle their limitations and drawbacks in order to achieve our objectives.

2.1 NFC and Bluetooth Security

Multiple wireless network solutions are available to handle the communication between a computer and a smartphone such as Bluetooth, Near Field Communication (NFC), or Wi-Fi [57, 68]. In this dissertation, the goal is to establish a wireless communication channel without using Internet and that is effectively secure for transmitting sensitive data across it, i.e, is protected against malicious users. Thus, in this section we will discuss the Bluetooth and NFC approaches, indicating for each the existing threats and protection techniques.

2.1.1 Near Field Communication

NFC is a high frequency wireless communication technology which enables the exchange of data between devices under a 10 centimeter distance [27]. Is based on the "touching paradigm", which means approximating two NFC-enabled devices close to each other or simply touching to enable communications between them. This paradigm was firstly

employed by the Radio Frequency Identification (RFID) technology [45] that consists in a microchip (called RFID tag) attached to an antenna in a package that resembles a sticker used for identifying an object or a person. The data transmission between a RFID tag and a RFID reader is done via magnetic field induction. NFC-enabled devices also have integrated and are based on this technology.

With NFC users can perform safe and contactless transactions, access digital services or transfer a file from one device to another. Other great innovation is that credit cards or tickets can be integrated into the mobile phones, avoiding the need to physically carry them elsewhere we go [4]. At the same time, NFC can assure the secure storage of personal information, like private keys, and the storage and execution of NFC-based applications (e.g, card applications) through the hardware-based secure element (Section 2.2.1) of mobile phones. However, with HCE technology (Section 2.2.2), emerged the possibility of having virtual card applications running on the mobile phone's operating system.

According to ECMA-340 standard [64], NFC enables the establishment of bidirectional communication channels. In one side, we have a Initiator device that initiates and manages the interaction process. At the other side, we have a Target device that only responds to the Initiator's requests. NFC protocol specifies that the communication can happen in two modes, a mode where the Initiator and Target device generate their own radio frequency field to transmit the data and a mode where only the Initiator device creates the radio frequency field, respectively, Active Mode and Passive Mode. In addition, there are three operating modes for device communication [27], which are Reader/Writer (R/W), Peer-to-Peer (P2P) and Card Emulation. In R/W, data is transferred between a mobile and a tag; in P2P, data is transferred between two mobiles (or devices); and in Card-emulation, the data is transferred between a mobile and a reader. Besides the standardization of these modes, NFC data messages are also standardized by the NFC Forum [27] as NFC Data Exchange Format (NDEF) messages [37].

2.1.1.1 NFC Threats and Protection

Due to the proximity paradigm, attackers will encounter difficulties in attacking the data packets since the channel is very short. However, it does not ensure complete system security and user privacy [27, 70, 73]. The attacks can be directed to the NFC ecosystem or to a particular NFC component (i.e, a tag, a reader or a mobile phone).

NFC tags contains sensitive data that attackers can manipulate by replacing with malicious data, hiding the original tag with a fake tag or cloning the tag. Typical attacks include infecting the mobile phones with hidden NFC worms, phishing and spoofing attacks which provides the users with data that looks valid but in reality are fake, or NFC attacks using empty tags that cause a reaction on the device. A general solution to protect the tags is integrating digital signatures on NDEF messages to provide data integrity and authentication. [27, 75]. However, some vulnerabilities on the signature record types were discovered in another study [76], where some countermeasures to handle them were

discussed. Recent studies, propose to provide confidentiality to NDEF messages [40].

Despite the advantage of generating a short-range link, the NFC communications can be target of a variety of attacks due to devices interacting using radio frequency waves. The distance from which an attacker is able to make an attack can be extremely close depending on several factors [73]. Generally, the attack surface is composed by eavesdropping, Denial of Service (DoS), Man-In-The-Middle (MITM), relay and phishing attacks. Also, data modification and corruption is possible, for example, via RFID jammers [73]. In the case of eavesdropping, the attack success depends on the communication mode: if Active Mode is used, the attack can be done in less than 10 meters; otherwise, is very difficult to achieve eavesdropping. However, countermeasures for some of these threats already exist. For example, to protect NFC from eavesdropping and prevent data manipulation, the solution is to create a secure channel between devices using a key agreement protocol (e.g, Diffie-Hellman (DH) based on RSA or Elliptic Curve Cryptography (ECC) algorithms). In the case of MITM attacks, no countermeasure exists, because Passive and Active Modes implicitly protect the communications making these attacks almost impossible to happen, although it may happen in specific contexts.

NFC Readers are more sensitive to physical attacks, due to its placement in public areas, where they can being stolen, destructed or removed, but can also be target of software attacks given the critical information (e.g, cryptographic keys) they store.

Several NFC security standards have been developed in the past few years with the aim of providing a reliable basis for accessing NFC services and applications securely [43]. NFCIP-1 standard was the first to define a security protocol focused only on the data exchanged between two NFC mobiles (i.e, P2P mode). On top of that standardization, a new security standard (called NFC-SEC or ECMA-385) was built to improve the security capabilities [65]. This standard is defined as a common framework upon which future cryptography standards with more advanced security features can be implemented [43]. These standards can effectively deal with security threats, such as eavesdropping, data manipulation and MITM attacks, in order to guarantee the protection of NFC connections and the confidentiality, integrity and authenticity of data transmission [43].

2.1.2 Bluetooth

Bluetooth is an open standard for short-range radio frequency communication technology suitable to replace cables on devices like keyboards, mouses or printers [68]. This allows users to create an ad hoc network between devices to transfer data, called Piconet, which is a set of two or more Bluetooth-enabled devices close to each other operating in the same channel using the same frequency. Bluetooth technology is used in many business areas and consumer devices such as mobile phones, computers, headsets and watches. Benefits include disuse of extra gadgets to connect devices, automatic wireless synchronization between devices, non-proximity demand and ease of file sharing [67]. This technology has multiple versions and has two major implementations families standardized by the IEEE

802.15.1 [89]: Bluetooth Classic, which includes Bluetooth Basic Rate (BR), Enhanced Data Rate (EDR) and High Speed (HS), and Bluetooth Low Energy (BLE).

Bluetooth devices can operate with multiple data rates and change the Media Access Control (MAC) address and physical (PHY) layers. BR is the first Bluetooth version (v1.x series) and only supports transmission speeds of up to 1 megabit per second (Mbps). With version 2.0 + EDR, data rates in the order of 3 Mbps were reached, and Bluetooth EDR was born. In version 3.0 + HS, devices operate at higher data rates, up to 24 Mbps, by using alternative MAC/PHYs (AMP). This is known as Bluetooth HS. Nowadays, these several Bluetooth versions are used by commercial devices and all of them support backward compatibility. This means that older versions with low data rates are supported in later versions with higher data rates.

BLE [38], also called Bluetooth Smart, was introduced in the Bluetooth v4.0 specification with the intent of improving the energy consumption of mobile phones and, consequently, the throughput. Despite this, BLE also have other important goals, including reduced memory requirements, efficient discovery and connection procedures, short packet lengths and simple protocols and services. This standard is widely used for wireless BLE solutions, such as battery-powered sensors and devices.

Comparing with available versions, the adoption of Bluetooth v5.0 will be considered in the implementation baseline, although most devices use (and are optimized) for Bluetooth v4.x series. Bluetooth v5.0 primary goal is to provide enhancements related with range, data transfer speed, broadcast capacity and energy consumption, with BLE implementations emerging as the choice solution for Internet of Things (IoT) [22]. Although, even that BLE specification is the distinctive feature for such case, the Bluetooth Classic continues to be provided alongside, offering the devices concurrent support for both implementations.

2.1.2.1 Bluetooth Security

Bluetooth Security Architecture. This is illustrated in figure 2.1, which involves the participation of Bluetooth protocols to enforce the Bluetooth security policies such as Link Manager Protocol (LMP), Logical Link Control and Adaptation Protocol (L2CAP) and Radio Frequency Communication (RFCOMM) protocol. Link Manager participates in security procedures depending on the Bluetooth Security Mode: only the safest mode implies Link Manager to implement security [59]. L2CAP is a logical link and manages the creation and termination of virtual connections, called channels, with other devices. It also negotiates and determines security parameters for link establishment. RFCOMM enforces security policy for dial-up networking and other services relying on a serial port. The key component in the architecture is the Security Manager with functions as: storage of security-related information on devices and services; grant or refuse access requests by protocol implementations, such as L2CAP, RFCOMM, and applications; command the

Link Manager to enforce authentication and/or encryption before connecting to application; and initiates setting up "trusted" pairings and asks for PIN codes from users [88].

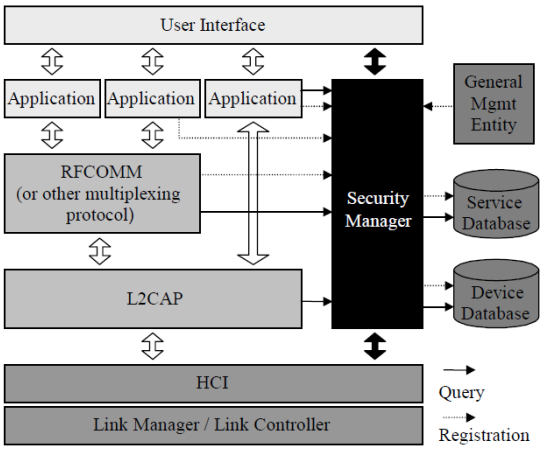


Figure 2.1: Bluetooth Security Architecture (taken from [88]).

Bluetooth Security Modes. Bluetooth standard [19] specifies four security modes in which a device can operate. These Security Mode indicate when a Bluetooth device initiates security procedures (e.g, authentication and encryption), not whether it supports security features [67]. In Security Mode 1, any security procedure are never initiated and devices do not prevent other devices from establishing connections. Security Mode 2 is a service level enforced security mode, where security procedures are initiated after physical and logical link setup. Security Mode 3 is a link level enforced security mode, in which is required that security services must be performed before any attempt to connect to other devices. That is, a device initiates security procedures, before the physical link is fully established. For Modes 1, 2 and 3, all v2.0 and earlier devices can support it, but v2.1 and later devices can only support it for backward compatibility. Security Mode 4 is service-based as Security Mode 2, in which encryption for all services except service discovery is required. This mode is mandatory for communications between v2.1 and later BR/EDR devices. However, for communications involving devices with no support for Security Mode 4, the Security Mode used can be reduced by backward-compatibility [67].

In addition, Bluetooth allows different levels of trust and service security [67]. A trusted device is a device authenticated to another device and has full access to all services. An untrusted device does not have a relationship with another device and has restricted access to services, although can be authenticated. In the other hand, three levels of service security are allowed (in Security Mode 2): services that require authentication and authorization; that require authentication only; and that are open to all devices. Security Mode 4 specifies multiple levels to classify the security requirements for services.

Each Bluetooth BR/EDR security mode also determines what stage of connection provides the related security properties and how [19]. In Bluetooth security exists six possible

security stages. Inquiry: a device in a new environment starts an inquiry to discover new devices. Paging: is a baseband procedure invoked by a device to synchronize with one of the responding devices. Link establishment: Link Manager executes its protocol (LMP) to establish a link with the other device. If Security Mode 3 is enabled, then Pairing begins in this stage. Service Discovery: LMP uses the Service Discovery Protocol (SDP) to find the available services. L2CAP channel creation: with information obtained from SDP, a L2CAP channel is created, which can be used by the application or the RFCOMM protocol. Pairing is the last stage if devices operate in Security Mode 2.

Bluetooth Security Features. Both Bluetooth specification families define a variety of security features, which are responsible for covering the encryption, trust, data integrity and privacy of the user's data, assuring the security and protection of the communication channel. This security mechanisms work differently in each implementation, but with the Bluetooth v4.1 and v4.2 releases, those differences have been minimized. The biggest feature is the Pairing mechanism (also called as link key generation), which is the process where the devices involved in the connection exchange their identity information to gain trust with each other and get the encryption keys ready for the future data exchange. However, this process differs between implementations [67]. Bluetooth BR/EDR establishes a link key by key agreement and BLE generates four different keys (Short Term Key (STK), Long Term Key (LTK), Identity Resolving Key (IRK) and Connection Signature Resolving Key (CSRK)) using a key transport protocol. All these keys will be essential to other security procedures, namely the link key (or the LTK), which dominates the security of Bluetooth, because these procedures will depend on it to accomplish their security goals. Figure 2.2 presents the BLE pairing process including the two possible pairing methods in phase 2 and the multiple keys distributed in phase 3.

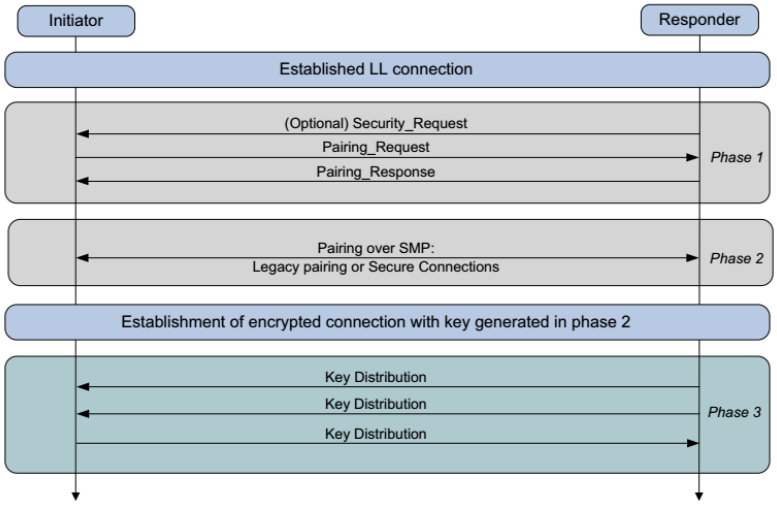


Figure 2.2: Bluetooth Low Energy Pairing Phases (taken from [19]).

Hence, the Bluetooth standard offers five basic security services: authentication, confidentiality, authorization, integrity and pairing. But, does not address other security properties such as non-repudiation or audit [67]. The authentication mechanism provided by Bluetooth is a process were devices are validated by a challenge-response protocol that verifies the knowledge of the link key. One-way and mutual authentication are both possible [67]. The security of this process is based on the secrecy of the link keys, which should never leave the device. Bluetooth provides a encryption feature to protect the data (or messages) exchanged through the communication channel against eavesdropping attacks. This confidentiality service is composed by three encryption Modes [67]: Encryption Mode 1 consists in not applying any encryption on the packets that contain the data; in Encryption Mode 2, data packets sent to individual devices are encrypted via encryption keys derived from link keys, but packets sent to multiple devices (i.e, broadcasting) are not encrypted; and Encryption Mode 3 is a generalization of the second mode. Yet, this procedure derives the encryption key from an artifact of the authentication procedure, so encryption cannot be done without authentication. In addition, authorization is the service that determines if a device had previously been authorized as a trusted device. Although, a device can pass authentication successfully, it could not be authorized to access restricted services.

Security Features in Bluetooth BR/EDR (Classic). The Bluetooth Core security architecture has evolved over time and has been specified in the different versions [19]. In early Bluetooth BR devices, pairing were performed via a method called PIN Pairing (or Legacy Pairing) for Security Modes 2 and 3. In Bluetooth v2.1 + EDR, the Security Mode 4 was introduced, defining a new pairing model called Secure Simple Pairing (SSP), which utilizes Elliptic Curve Diffie-Hellman (ECDH) key agreement for link key generation [87]. SSP also provides association models to simplify the pairing process, i.e, pairing methods that improve the flexibility in terms of device input/outputs - such as Numeric Comparison, Passkey Entry, Just Works and Out of Band [87]. In Bluetooth v3.0 + HS, another link key generation method was introduced. A new key, named AMP link key, is generated from a link key previously created (concatenated with itself) plus a ASCII key identifier. This process is done by the AMP Manager in the host layer of the Bluetooth protocol stack [67]. Bluetooth v4.0 do not present new significant improvements in the Bluetooth Classic security services already provided by older versions [19]. Bluetooth v4.1 improved the strengths of the BR/EDR technology cryptographic key, device authentication, and encryption by making use of Federal Information Processing Standard (FIPS)-approved algorithms [67]. More precisely, was introduced the feature Secure Connections, which consists in the use of P-256 Elliptic Curve (in SSP), HMAC-SHA-256 and AES-CCM algorithms for link key generation, authentication and encryption, respectively. Bluetooth v4.2 provided means to convert BR/EDR keys to Low Energy (LE) keys and vice versa.

Security Features in Bluetooth Low Energy. In the other hand, BLE offers security

services that are also expressed in terms of two security modes, called LE Security Mode 1 and LE Security Mode 2, which have multiple levels each one. Security Mode 1 has levels associated with encryption and authentication and Security Mode 2 has levels associated with data signing and authentication. BLE supports the ability to send authenticated data over an unencrypted link from one device to another based on a trusted relationship and still granting security [19]. This is achieved with Data Signing, which consists in signing data packets with the CSRK. Encryption uses AES-CCM cryptography to provide confidentiality, authentication and integrity. Then, there is no need to do challenge/response authentication, because LTK is used as input for the encryption key generation process, which implicitly grants authentication. However, Data Signing provides integrity and implicit authentication (i.e, devices have the correct CSRK), but does not provide confidentiality. In addition, BLE has integrated a mechanism called LE Privacy, which consists in mitigating the threat by which an adversary can track a BLE device by changing the device address frequently [94]. The mobile phone MAC address will be replaced by a random value (i.e, randomized MAC address) that changes periodically according to a timer implemented in the device firmware by the manufacturer. These randomly generated addresses are also called private addresses. Thus, any attacker would not be able to know that the several different addresses correspond to the same physical device and will not be possible to track your device. Only a trusted device that has been provided with the corresponding encryption key (IRK) can resolve the private addresses.

In Bluetooth LE devices before v4.2, the pairing process uses SSP (referred as LE Legacy Pairing after v4.2). However, no eavesdropping protection is provided unlike BR/EDR, because the LE association models operate in a different way and the pairing process itself is different, hence Just Works and Passkey Entry methods do not provide any protection [19]. However, with the release of the Bluetooth v4.2, security is highly enhanced and a new LE pairing model arises, named LE Secure Connections [19], which upgrades the LE pairing to utilize FIPS-approved algorithms (i.e, ECDH public key cryptography for key exchange) and adapts the Numeric Comparison association model to be used on BLE. With ECDH cryptography, eavesdropping attacks can be mitigated, given it's high degree of strength against those attacks [19]. Using Numeric Comparison, protection against MITM attacks is obtained as well as using the Out of Band or Passkey Entry methods [19]. Bluetooth v5.0 only introduces performance improvements related BLE while all security mechanisms implemented are inherited from v4.2 [19].

2.1.2.2 Threats and Protection

Bluetooth has multiple weaknesses that result in vulnerabilities that can be later exploited by attackers such as encryption key lengths restricted by the device that has shorter maximum length, PINs being easily brute-forced, Just Works model fragility, backward compatibility forcing the reduction of the Security Mode and no user-authentication support [26, 36]. Regarding BLE, Passkey Entry-based and LE Legacy pairing are considered

unsecure. Thus, Bluetooth is vulnerable to Bluejacking, Bluesnarfing, Bluebugging, MAC Spoofing or MITM attacks [67] and more verified attacks are enumerated in literature [36, 51, 60]. The main threats related with BLE concretely are passive eavesdropping, MITM attacks and identity tracking [19]. One important factor when discussing Bluetooth vulnerabilities is the version currently being used since many of them are specific to early versions [26, 67] and almost all of them mitigated in the last versions. Also, some vulnerabilities could still appear while using the built-in security features. Therefore, we need to put extra efforts to protect the Bluetooth channel and secure the user data against future attackers [68].

National Institute of Standards and Technology (NIST) [67] recommends several countermeasures based on the built-in security features in response to the threats presented. At a educational level, a users must have an adequate level of knowledge and understanding about Bluetooth devices and the organizations should establish security policies related with the use of Bluetooth devices and user's responsibilities. Also, organizations should reduce the risks against their Bluetooth implementations by applying methods to handle specific threats.

At a technical level, recommendations can include changing the default settings of the Bluetooth device to reflect the organization's security policy and setting the Bluetooth devices to the lowest power level so that transmissions remain within the secure perimeter of the organization. Using SSP with Just Works association model must be avoided, because it does not provide MITM protection (other model is preferred). Always use link encryption for all Bluetooth connections and with the key sizes configured to the maximum allowable (128-bit), otherwise, transmitted data are vulnerable to eavesdropping and brute force attacks. Employ mutual authentication for all connections to verify that all participates are legitimate. Ensure a secure environment to perform pairing where attackers cannot capture the data packets, since it is a vital security feature and requires that users are aware of possible eavesdroppers. Apply software and firmware patches and upgrades regularly, because new vulnerabilities can be discovered in some vendor products and they should be patched to prevent malicious exploits. Ensure that Bluetooth portable devices are configured with a password - such as a pattern, a fingerprint or a code - to prevent unauthorized access if the devices are lost or stolen. Also, antivirus should be installed to ensure that malware is not introduced in the Bluetooth connections.

In relation to Security Modes and levels, Bluetooth Classic devices should use Security Mode 4 Level 4 as it requires Secure Connections and provides the highest security available for v4.1 and later Bluetooth Classic devices. If not possible, Security Mode 3 should be the replacement. LE and v4.2 devices should use LE Security Mode 1 Level 4 as it implements LE Secure Connections and also provides the highest security available [67].

At application-level, NIST [67] recommends to re-encrypt the data before giving it to the Bluetooth layer and when is received by other device, it should be decrypted. One drawback of this technique is the time and computational power needed, since the

transmission will take longer than without encryption and a double encryption will be quite costly depending the algorithm used. We can also apply the same procedure for authentication, i.e, Bluetooth-independent re-authentication. Every time a user wants to access a secure service, re-authentication can be required, which is a repetitive process. However, fingerprint scanner can be used to ease the process. This type of mechanisms avoids attacks such as Bluebugging and provides user-authentication.

Many other protection methods can be found in literature [7, 26, 51, 60]. For example, Mutchukota et al. [60] proposes a improvement to the existing SSP method based on anti-jamming techniques despite the existing solutions for MITM protection.

2.2 Secure Elements and Host Card Emulation

Secure storage is fundamental when managing keys for cryptographic operations. Multiple techniques exist to achieve this goal which ensures integrity and confidentiality of data and enables a trusted environment to execute these operations. The most common are Mobile Trusted Module (MTM), Trusted Execution Environment (TEE) (e.g, ARM TrustZone technology leveraged by the ARM processor), Secure Element (SE) and Host Card Emulation (HCE) [24]. However, only the last two are in the scope of this dissertation.

2.2.1 Secure Element

SE is a tamper-resistant chip where critical code can execute and cryptographic keys can be stored [24, 77]. This component is based on the chip design that was built for use in contactless credit cards (e.g, smart cards) and provides the user with a level of security and identity management to assure the safe delivery of a specific service. The information stored in this special chip is impossible to read or copy by normal applications installed on the device, only special and trusted applications (e.g, virtual wallets [74]) are enable to do it. Also, SE communicates directly with end-applications without passing data to the smartphone operative system. Therefore, if a malware infects the device, the SE will be intact and no information can be intercepted by attackers. The main way to implement a SE is trough hardware [90]: embedded on the smartphone hardware at the time of manufacturing; or implemented in a removable card format (UUIC/SIM-card or SD-card), providing a secure environment for applications to execute.

Having a SE-based security architecture eliminates the vulnerabilities of single factor based authentication systems by adding another layer of security [81]. PKI is the best possible authentication method, but only if the certificates and keys are stored in the SE, because storing them out of the SE component makes them vulnerable to attacks.

Given the NFC lack of secure ways to store users' sensitive information (e.g, bank account details) in the mobile environment where other applications are running too, NFC begun to use SE and designed it to be the security base of NFC technology. In this

solution, SE is embedded together with NFC controller and can be accessed by internal (i.e, operative system applications) as well as external applications (e.g, NFC reader) [9].

One big drawback of SE is the tight control that mobile operators and manufacturers exercise over them [90]. This means that mobile developers that wish to access the SE to accomplish application requirements have no permission to do it. Another problem is that not all the devices have SEs and the use of UUIC-based SEs or their integration on mobile phones can bring extra costs to mobile builders and, consequently, to users.

2.2.1.1 Threats and Protection

Despite being safe and secure by themselves, using SEs alongside with NFC-enabled smartphones can cause the appearance of some threats as discussed by Roland et al. [77]. This paper presents two practical attack scenarios against a SE, given the risks imposed by the possibility of installing untrusted mobile applications and the mobile phone connectivity to a global network. These attacks are based on the assumption of unrestricted access to the SE. The first attack is a DoS on NFC-enabled mobile phones, in which successive authentication attempts are done for card management until it is no longer available. Then, a Relay attack is presented, in which an application is remotely installed in the victim's mobile phone without his knowledge and relays the communications between the victim's SE and mobile phone across a network to a card emulator (hosted in another device) that can then be used as if the victim's SE was implemented into it.

SE has strong countermeasures and is robust to a wide range of sophisticated attacks. As consequence, a few number of successfully attacks were registered in literature [82]. However, specific mechanisms must be employed to ensure this high level of security. The first line of defence to protect a SE is to restrict the access to it, employing access control policies. Almost all the devices that use a SE already implement this via SE APIs (as seen before). Cooijmans in [24] discussed that existing APIs to access SE in Android devices only grant access to them through special permissions only granted by a root user or the device manufacturer himself. Normal applications with basic permissions can't request access to the SE. The SE security can be also enhanced by combining it with a TEE [33], which can filter the access between the operating system and the SE to allow only trusted applications to access the SE, and uses well defined certification schemes to reduce the risk of fraud and malicious operations [82]. Despite the threats verified in the past, nowadays, using a embedded SE in combination with a NFC controller or a Bluetooth driver can be very useful for control and prevent remote relay attacks [9].

2.2.2 Host Card Emulation

HCE is a contactless technology that allows the launching of mobile NFC services and products without making use of SE by allowing the mobile device operating system (i.e, the "host") to communicate directly over the NFC interface in Card Emulation Mode [9].

HCE enables software emulation of card-based applications and to perform payment transactions when the users' sensitive informations is stored out of the SE. With HCE, no smart-card chip is required, because applications are held in the operative system and HCE relies on the mobile phone CPU for processing. Figure 2.3 illustrates the difference between using SE and HCE in a NFC-based application.

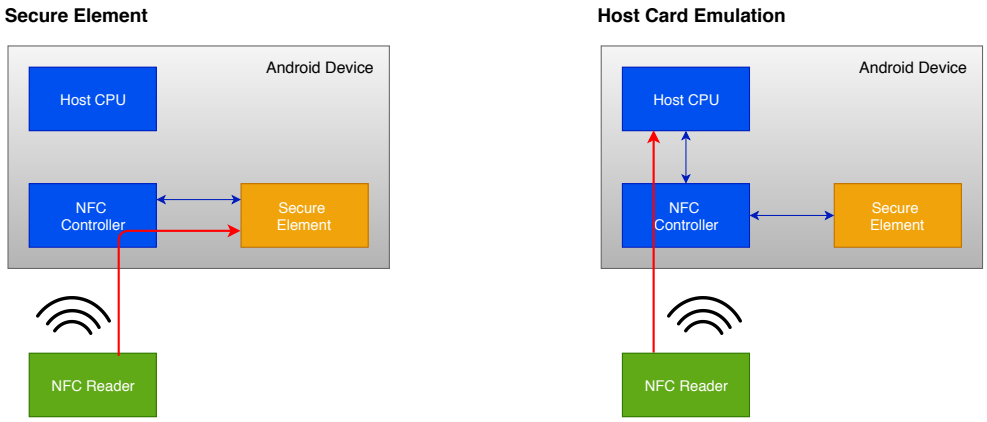


Figure 2.3: NFC Communications using SE and HCE (based from [9]).

In the SE-approach, the NFC controller routes all the messages from the NFC reader to the corresponding application residing in the SE. In the HCE-approach, messages can additionally be routed to an application running on the host CPU. The operating system is who decides where the messages go, which could be to the SE, but are more likely to be routed to the host. Besides this difference, HCE service can store the sensitive information used by in a location different from the application itself and from other secure locations such as TEE or SE. This new location can be a back-end server in the cloud from where is possible to retrieve data when devices need to communicate [9]. Unfortunately, network latency can cause a poor user experience, making this option not so good.

However, NFC Card Emulation services on the SE and in the host can coexist in the same mobile phone, but is necessary to integrate a mechanism to determine where to route the messages received from the NFC external reader [9, 29]. This mechanism allows the NFC controller to implement a routing table that lists the NFC applications stored in the SE and is used to choose where to forward the messages.

HCE is a benefit for the whole NFC ecosystem, since developers will be capable of developing innovative applications, creating new NFC use cases and enhancing the NFC service experience of the users. Another benefit is that companies that wish to deploy a NFC mobile product don't need to cooperate with mobile operators and manufacturers to implement a secure environment for storage and execution that implies extra costs [90].

2.2.2.1 Threats and Protection

The change on storage location of sensitive user data and on the routing of communications when using a HCE-based applications can introduce new vulnerabilities, because

an application running on the operating system is much more exposed to attacks than an application executing in the SE [9]. The communications between the host and the NFC controller can be attacked by malicious applications that live in the same ecosystem. The threats composing the adversary model can include malware able to exploit, root or jailbreak the device; spoofing attacks to induce users to do actions; attacks against the cloud storage if the credentials stored in applications are compromised; or DoS attacks.

The Smart Card Alliance white paper [9] enumerates various countermeasures to enhance the security of HCE. One technique to secure the applications can be the addition of software security in order to make it harder for an attacker to modify the software. Usually, this is accomplished with runtime integrity checking. If an attack is detected, the tamper-proofed system stops the application and communicates the occurrence.

An ascending technique in the recent years is the use of biometric factors - such as fingerprint, voice and face recognition - in addition to other authentication methods to enforce user authentication, providing a very friendly experience for the user.

HCE-based applications can use the TEE [33], which is a secure area in the main processor of a mobile phone that supports safe execution of trusted applications and enables the processing and storage of data. TEE offers protection against attacks coming from the main operating system, assists in the access control and hosts applications that need to be isolated from the main operating system. Other benefit of TEE is that it runs in its own operating system, hence it is not affected if the main operating system is attacked.

Methods for encrypting the data transmitted are also recommended. Data used by HCE-based applications can be encrypted and stored within the applications. End-to-End Encryption (E2EE) or Point-to-Point Encryption (P2PE) can be employed to ensure data encryption at the reader and protection while the data is being transmitted.

Another simple but effective method is tokenization, which consist in replacing credentials with high intrinsic value (e.g, a finance number) for a random value with no apparent value but equivalent. This method is typically used for masking a card identity.

2.3 Windows Cryptographic Providers

Cryptographic providers are the main way for implementing cryptographic functionalities in an operating system, which include cryptographic algorithms, key storage and generation and authentication of users. Microsoft Windows operating system provides two cryptographic APIs with which these providers are associated, Cryptography API (CAPI) and is long-term replacement Cryptography API: Next Generation (CNG). These SDKs are used to create Windows-based applications that need to use cryptographic functions to achieve their security requirements and data protection [54].

Providers associated with CAPI are named as Cryptographic Service Providers (CSPs) and implement cryptographic algorithms and key storage. In addition, CSP can be combined with smart cards to grant an higher level of security and can implement Secure

Sockets Layer (SSL) and Transport Layer Security (TLS) authentication protocols [54]. On the other hand, CNG separates the features of algorithm implementation and key storage in two different providers called Cryptographic Algorithm Providers (CAPs) and Key Storage Providers (KSPs). CAP implements hashing, key exchange, symmetric and asymmetric algorithms. KSP manages key operations - such as creating, deleting, exporting, importing and storing - and are used for asymmetric cryptography and signing.

Lina [49] discusses that CAPI has few algorithms and is becoming outdated with the current advances in technologies. CNG is more extensible and has support for more algorithms, including user-defined algorithms. As also, is the most recent SDK and is planned to be the substitute of CAPI. The next subsection will focus on the CNG architecture and KSPs design and implementation.

2.3.1 CNG and Key Storage Providers Overview

CNG is divided in six different classes of algorithms from the function perspective and each class exposes its own primitive API. For example, asymmetric encryption will use RSA and signature will use DSA or ECDSA [53]. Also, each algorithm can have multiple implementations offered by providers, but only one will be used at a given time.

KSPs are basically Dynamic Link Libraries (DLLs) [1] (i.e, libraries containing code that is only loaded at run time when a functionality is requested). We can create our custom KSPs with our own algorithm implementations, but is necessary to register them first with CNG, because only after registration it becomes available to system applications [54].

The main advantage of KSP is that it provides a model for key storage that allows applications to accomplish their asymmetric encryption and key storage requirements [49]. It could guarantee the security of private keys because after the key is stored, it can't be read. Applications access the KSPs on the system to accomplish its functional requirements through the key storage router which is the main routine in this model and provides details, such as key isolation, from both the application and the provider itself [49].

KSPs implemented by software define storage and execution mechanisms, but does not provide a high level of security. On the other hand, KSPs based on hardware implementation can grant a higher level of security, but are more expensive. To balance this trade-off, we can execute the security computing such as private keys and certificate storage in the hardware and implement the other functions in the operating system.

In PKI systems, encryption algorithms, certificates and signatures technology is extensively used to satisfy confidentiality, integrity and authentication requirements, and, as consequence, the approach used to store certificates and its private keys is very important [49]. Meanwhile, KSP provides a interface model to connect and communicate with one of the data storage solutions discussed, then the private key could be stored in a secure environment with specific function such as encrypt or signature.

[1] https://support.microsoft.com/en-us/help/815065/what-is-a-dll

2.4 Critical Analysis

2.4.1 Summary

In the context of the dissertation objectives, NFC and Bluetooth are possible solutions studied to establish a secure communication channel between the mobile phone (running the Mobile Cryptographic Provider supported by Android SEs on a conventional smartphone) and the computer (where applications running in a Windows-OS execution environment can request cryptographic operations through local Windows-Cryptographic Provider functions, transparently executed in the remote Mobile Cryptographic Provider).

NFC is a proximity-based technology to transfer data between devices, that can operate in a variety of modes and has few security and privacy issues, with the risk managed under physical awareness control by its users, communicating in very short physical distance. NFC and its enhancements on security standards are in an on-going in the current standardization effort, and research solutions to improve the security properties.

Bluetooth is also a wireless technology, with more flexible mechanisms to address different communication ranges, without the need of extreme proximity between devices. The standard feature Low Energy (particularly related to Bluetooth v4.0 and v5.0) emerged as a relevant reference designed for generic adoption, from smartphones or laptop computers to low-power IoT devices. Apparently, BLE v5.0 will have a strong impact in the unification of Bluetooth-Communication support, with the new smartphones enabling this version, with flexible support at the firmware level to operate the Bluetooth communication in different settings, for the optimization of trade-offs related to distance ranges, energy, and throughput rates. Despite that we must also consider the relevance of the installation base for Bluetooth v4.0 (4.1 and 4.2), Bluetooth is a more flexible and available solution for our goals, when compared with NFC.

NFC could have advantages over Bluetooth from the technological and economic viewpoints, for certain application scenarios. NFC is implicitly resistant to MITM attacks with the direct control of communicating devices operating in proximity assumptions. This makes NFC an ideal method for secure pairing for proximity payment systems, for example. On the other hand, Bluetooth has higher range and consequently a higher attack surface, depending on the covered ranges. However, it has an advantage over NFC since it comes with standard security protocols, parametrizable in different secure modes and enforcement levels. When using the most secure modes and maximized policy enforcements, it is not required to add extra security mechanisms to protect communications. However, system-level security can require extra-layers of security, to enforce security properties at system and application level, or to reinforce the security properties at the communication layer and wireless data-link level. Depending on the requirements, Bluetooth Classic and BLE implement security mechanisms to handle the threats against the channel, although may be not enough to mitigate other possible attacks.

To achieve a secure storage and mobile execution environment we studied two approaches for Android-based solutions: SE support and a virtual alternative, the HCE framework. SE provides a safe and secure environment to host applications, sensitive data and cryptographic credentials, and execute operations under isolated abstractions. Given its high security standards, the SE is mainly incorporated in NFC-enabled devices. Although, not all mobile phones have installed in hardware a SE chip (to minimize trustability assumptions to hardware-enabled support), or have UUIC-cards for the same purpose, the software design for SE can be used to address these possibilities in the future. HCE is a much more flexible solution that creates virtual SEs, bypassing the involvement of mobile operators and manufacturers in the provisioning of controlled SEs. However, is more vulnerable in terms of security, since it is executed as an operating system service.

To guarantee the communication between the computer and a mobile phone, a Key Storage Provider will be integrated, which will be responsible for forwarding requests for cryptographic operations to the remote mobile cryptographic provider running in a mobile phone, receiving the correspondent results, as happen in requesting digital signatures for documents managed in the client computer. This KSP will be registered through the Windows CNG to be available to other applications in a transparent way.

2.4.2 Discussion for Dissertation Approach

Our conducted study on the related work allowed for the comprehension of issues, to establish a rationale for the approach of our objectives.

On the choice between NFC versus Bluetooth. From the viewpoint of physical locations, the NFC proximity demand (which is essential for payment and ticketing applications) offers very limited device location flexibility in comparison with Bluetooth.

Rationale: NFC offers a good level of security, but for us is not a good option, because the proximity demand can be inconvenient to users and, like smart cards, would be necessary an additional reader to allow NFC on the computer. We desire to use our smartphone as a service host in an office environment, where we can be far from each other, without having to bring together the service and client devices to perform an action. Therefore, we intend to use Bluetooth technology as our secure communication channel. To mitigate attacks, a rigorous set of Bluetooth security mechanisms must be implemented.

On the different versions of Bluetooth and related security assumptions. Different versions of Bluetooth were promoted in the context of the activities of different Bluetooth Special Interest Group (SIG) working groups, with different implementations causing problems in interoperability of different devices from different vendors and different versions supported by different manufacturers. These problems were progressively corrected in Bluetooth firmware of different manufacturers, particularly, after the version 4.0. Additionally, Bluetooth security standardization define different security modes, complementary security levels for devices and services, and two trustability levels (trusted and

untrusted), which must be properly employed and depend on a concrete solution.

Rationale: For the objectives and regarding the differences in the above criteria and compatibility issues on different Bluetooth versions, we will address as our security baseline, Bluetooth Security Modes and levels, as defined for BR/EDR/HS devices, adopting the Security Mode 4 Level 4 as it requires Secure Connections. LE and v4.2 devices should use LE Security Mode 1 Level 4 as it implements LE Secure Connections. From this security baseline for Bluetooth authenticated pairing and key-establishment processes to setup the secure channel under mutual authentication assumptions, we can leverage an extra-layer of security, where data is encrypted before the transmission on the established channel. For this extra-layer at service-level we can base our solution on Numeric Comparison methods, where both Bluetooth devices can display a six-digit number and allowing a user to enter a "yes" or "no" response.

On mobile trust execution considerations. Mobile execution and storage solutions such as SE and HCE have similar goal but are implemented in different levels of abstraction, within the mobile environment. With SE, the data cannot leave the device itself and a strong level of security and an optimal level of interoperability is provided, with trusted execution environment guarantees possibly provided in isolated hardware-level, thus minimising the trust-computing model assumptions to hardware. The main problem is the need of extra hardware in the mobile phone, because not all mobile manufactures (or packaged devices promoted by manufacturer and operators) provide such solutions. HCE can be implemented by a wide range of mobile phones without software or hardware restrictions, but is less secure than a physical SE, since HCE-based applications use cloud storage services and run on the OS. The problem in this case is that the trust computing model assumptions are provided at the level of the Mobile OS services (e.g, Android OS services), which are vulnerable to attacks compromising the OS behaviour.

Rationale: We want to achieve a secure solution where the certificate private keys are never shared out of the control of the SE environment, and stored in a secure location where cryptographic operations that use them can execute. SEs can provide us with this isolation from the operating system, hence will be the secure storage approach to follow. Recent support of SEs for runtime services enabled by recent technology of mobile operating systems (e.g, recent versions of Android OS) enables a robust usage of SEs on almost every mobile phone, then hardware constraints will not be a problem as discussed above.

On Software Attesttation guarantees. As far as we studied, SEs don't have many security issues given their implicit high level of security, but one problem detected was the possibility of a client computer "to talk"with a fake SE-based crypto provider solution, after the compromise of its support, in a mobile OS.

Rationale: To verify or check if client devices (computers) are really integrating with the correct remote service, we must implement a Remote Attestation protocol. Thus, it will allow devices communicating with the SE to check the authenticity and the integrity of the software and/or hardware in the boot process and running in the SE environment of the mobile phone. The enforcement of the execution of such protocol will be considered as a key-point in the final design and implementation of our solution.

On managing data in the SE solution. To manage the data stored and the cryptographic operations executing inside SEs, we might need to implement a remote mobile crypto provider to run inside the SEs. A second crypto provider, particularly a Windows crypto provider, will be also necessary to send the cryptographic operations requests to the first crypto provider, allowing the Windows crypto provider to execute these operations remotely in the smartphone. In our implementation we will consider the client computer as trusted, focusing in the security design of the remote mobile crypto provider.

Rationale: The mobile crypto provider will be implemented as an application and SEs services running on the Android OS, and will be remotely used by the client computer whenever a cryptographic operation (namely, a standardised digital signature request) is demanded. The Windows crypto provider, more precisely, a key storage provider as studied previously, will implement proxy functions, transparently redirecting the local operations from applications (such as a signature of a PDF managed by Adobe Reader, for example) to the mobile remote cryptographic provider. We will give the necessary support to define policy enforcements at device-level (mapped on firmware configurations for Bluetooth security enforcement) and system-level (mapped on complementary designed security services, in an extra-layer of security).

Gomez et al. [38] provides evaluation criteria for BLE performance, regarding different metrics: (i) for power consumption, the average consumption decreases, since devices may be in sleep mode for a greater fraction of time, although, fast resynchronization methods can increase the values; (ii) for latency, with very low bit error rate, the average value of round-trip and one-way message exchanges are smaller than 2 ms and 1 ms, respectively, but for high bit error rate values, the average latency increases three times more. Without errors, the latency of a one-way exchange can reach 676.7 μs. The maximum throughput measured was 58.48 KBps which can be influenced by many factors.

Rationale: In our dissertation, a low energy consumption is not a primary goal because the number of cryptographic operations, including signing and certificate registration, is expected to be low, and does not force a significant energy consumption by the mobile phone. Thus, we intend to use the Bluetooth BR/EDR implementation as the primary approach to establish Bluetooth channels, because it offers a solution with better data rate and a bigger payload size, providing a faster transmission of data, analysing the impact of our designed solution in terms of throughput and latency conditions, for different security parametrizations.

A recent experience [90] with NFC-enabled devices, including two applications, one SE-based and the other HCE-based, measured the execution time that devices take to carry out non-trivial cryptographic operations such as encryption and signing via Application

Protocol Data Units (APDUs). For the SE-based application, the testing protocol took on average 1982 ms to fully execute with 85% of the time spent on digitally signing messages. For the HCE application, the testing protocol fully executes on average in 213 ms for a CPU running at 960 Mhz and 572 ms for a CPU running at 300 Mhz with similar signing and encryption execution times, but with a significant variation between the maximum and minimum values. HCE is faster than SE, but tends to be less secure, because no security mechanisms were employed and if they were the application would become much slower. We will observe the equivalent impact in evaluating the proposed solution.

Rationale: We will prefer to achieve a good trade-off between security and usability, in order to achieve the best security, instead of high execution speeds or long-range communications. We expect that cryptographic operations remotely executed in the mobile cryptographic provider will not be significantly slow, and maintain the same order of magnitude for throughput and latency, comparing with the studies in performance characterises of Bluetooth connections. Therefore, the SE approach is compatible with our objectives in the sense that it provides the desired level of security in the mobile environment to protect the sensitive information on Bluetooth security association parameters, as well as, on using and managing Private Keys.

Chapter 3

System Model and Architecture

In this chapter we present the designed solution to address the Bluetooth-based Mobile Key Storage Provider (KSP) (or BluetoothKSP), as a generic support for applications running in Windows machines. As stated before, the main use of the BluetoothKSP, is its ability to obtain standardized digital signatures of documents (file formats) that can be generated from Windows-based applications. For the purpose, the BluetoothKSP offers a transparent support for the local Windows applications, as a local Dynamic Link Library (DLL) acting as a proxy that implements the conventional cryptographic primitives as supported by local Microsoft Key Storage Providers (MKSPs), but supported by the interconnected mobile system (via Bluetooth) where the real BluetoothKSP runs. Then, from the Windows-based support, the BluetoothKSP does not implement its own functions, and the provided DLL acts as a transparent pass-through layer, facilitating the communication between the operating system and the remote BluetoothKSP implementation. In the remaining of the chapter we will present and discuss the designed solution, starting from the generic system model assumptions, including the studied threat model, and provided architecture (in Section 3.1) and going in the details of the supported facilities equivalent to conventional local Windows KSP functions, namely those implementing standardized digital signatures (in Section 3.3). Then, we discuss the particular architecture of the BluetoothKSP and its components executing on Windows OS (in Section 3.2) and the related configurations required for Windows-based applications use the mobile crypto provider (in Section 3.4). Finally, we discuss the possible security enforcements for the Bluetooth pairing functions (in Section 3.5) and present final considerations and remarks on the designed solution (in Section 3.6).

3.1 System Model and Architecture

In this section it is described an overview of the system model, including its adversary model, its core components and the system interactions between these entities.

3.1.1 System Model Overview

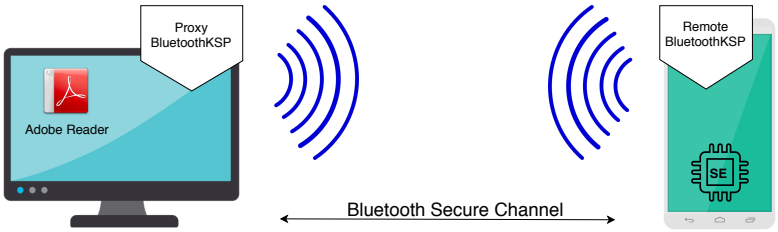


Figure 3.1: System Model Overview.

In figure 3.1 is illustrated a simplified version of the system model, composed by a computer and a smartphone that exchange information via a secure Bluetooth channel making use of both crypto providers in each node. The system built employs a client-server architecture between two nodes were the computer acts as client hosting the proxy BluetoothKSP and the smartphone as server hosting the remote BluetoothKSP (the real one). The Windows-side (proxy BluetoothKSP) is responsible for ensuring the communication between the smartphone and different Windows applications running in the computer. The remote BluetoothKSP is responsible for processing the cryptographic operations requested by the proxy BluetoothKSP and securely managing access to the certificate private keys inside the Secure Element (SE). This remote BluetoothKSP can work as a service running in the background (or not) in the smartphone waiting for remote signature requests from a client computer. The SE ensures that the signature requests issued by the proxy BluetoothKSP are securely executed in an isolated context from the Android OS and other mobile applications.

Besides the signature operations, the mobile application (i.e, the remote BluetoothKSP) accepts certificate requests from client-computers that desire to import new certificates into the Windows OS, configuring the crypto provider of the certificates with our BluetoothKSP, enabling Windows-based applications to make use of our remote BluetoothKSP. Plus, this mobile application, provides functionalities of certificate management such as certificate import into the Android OS, certificate entries removal from the Android Keystore and displaying of Android Keystore certificate entries. Hence, we can see the remote BluetoothKSP or mobile application as a certificate manager.

The Bluetooth communication channel has a security baseline composed by the Pairing mechanism and security levels, which is possible to be enforced with an extra layer of security, specially when working with less secure Bluetooth devices (i.e, mobile devices with old Bluetooth versions). The data exchanged between the nodes inside the secure channel is always public, such as a document hash or a certificate containing only the

public key, and no private data is transported through Bluetooth. All the private data to be used is stored inside the smartphone SE and never leaves it as mentioned before.

For initial validation, we considered the Adobe Acrobat Reader desktop application as the case-study of a Windows-based application, but our system has support for other applications such as Microsoft Outlook and Word. Except for applications that make usage of PKCS#11 uniquely (for example, Mozilla Firefox and other Linux-based applications). This is due to the fact that these applications do not support Windows APIs to interact with cryptographic token drivers, such as our BluetoothKSP. This way, our system guarantees only transparency for Windows-based applications.

Given this system model assumptions and all the basic mechanisms involved, we perform an analysis of the potential security vulnerabilities in the following subsection.

3.1.2 Threat Model

As described in section 2.1.2.1, Bluetooth has built-in security mechanisms that provide a certain level of security depending on the Bluetooth version of the computing device. However, we have shown that these basic security features do not provide a strong protection to data transported in the communication channels and some attacks are feasible.

We assume an adversary model where the Bluetooth communication channels are vulnerable to attacks that intend to tamper or modify the data being exchange such as Man-In-The-Middle (MITM) and Eavesdropping attacks. More concretely, an attacker with ability to eavesdrop or tamper the communications between the client and server endpoints could change the Bluetooth packets content to some data that they desired to be signed, leading to users signing forged documents without knowing it on the smartphone.

For the mobile execution environment, we assume that the user private keys can not be compromised by attackers, because only applications with permissions (in this case, our remote BluetoothKSP) can access the SE through the Android Keystore System to store keys or execute cryptographic operations. For example, an attacker that tries to compromise the sealed container used by our mobile application using another application that he/she has installed on the device will not have success. Thus, we consider the SE as our Trusted Computing Base (TCB).

For the scope of this dissertation, attacks coming from the Android OS or malicious code executing instead of the correct OS were not considered. This means that our solution nodes not rely on the integrity of the OS to secure their data. For example, attackers that can acquire the control of the Android OS through a root exploit, allowing them to run applications with root permissions and access its data and modify its content. Also, a malicious behaviour on the part of Windows OS, such as a corrupted KSP that changes the signatures requests or a malicious desktop application that makes use of the KSP to request forged signatures, was not considered and was leaved out of the scope. However, we plan to study these potential attack vectors with more detail in the future.

3.1.3 Architectural Components

Our solution is built as a service with the architecture depicted in figure 3.2. The system has several entities in each node (client and server), being the BluetoothKSP, the WindowsCertificateManager and the CertificateManager, the core components of our solution (dark blue boxes). BluetoothKSP is our proxy mobile crypto provider, WindowsCertificateManager is a graphical Windows desktop application that also has the ability to import and display certificates existing in the Windows OS and CertificateManager is the mobile application that has the remote BluetoothKSP and certificate management functionalities. The remaining (light blue boxes) are the built-in or black-box components, which are the components that support and are used by the core elements. All these entities will be described in this section for a better understanding of the proposed system model.

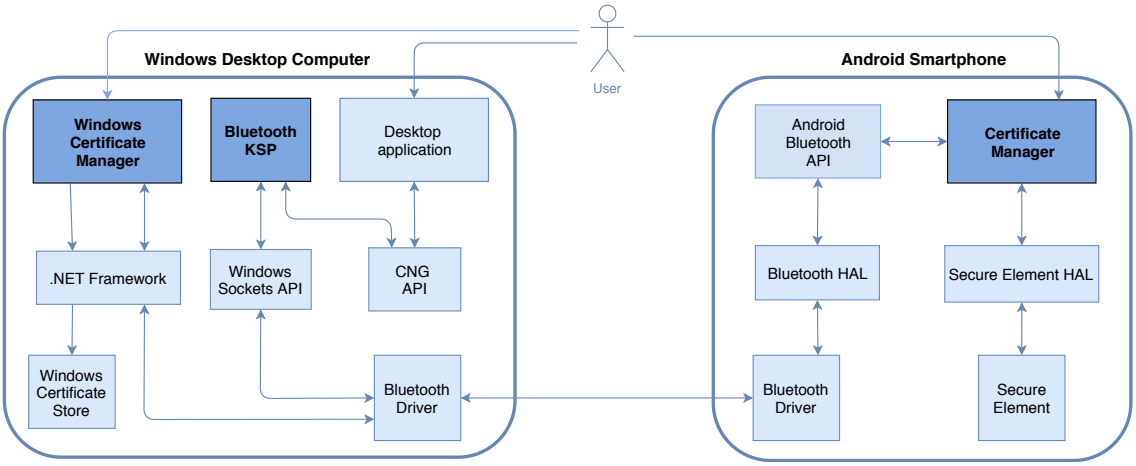


Figure 3.2: System Architecture and Interaction Flows.

WindowsCertificateManager. This client-side component allows the user to request certificates to import into the Windows OS from the remote BluetoothKSP running on the mobile phone, through the .NET framework which forwards the request packet and waits for the reply using the Windows Bluetooth Driver. Then, it imports the list of certificates received into the operating system (more precisely, into the Window Certificate Store) creating a link between each certificate and the *BluetoothKSP*. This certificates exchange is required to enable any desktop application to use our mobile crypto provider to request signatures later, and to use the corresponding user certificate that contains the public key to verify the signature executed by the remote BluetoothKSP.

BluetoothKSP. The proxy mobile crypto provider that forwards requests for cryptographic operations such as digital signatures from desktop applications (e.g, Adobe Acrobat Reader) to the remote BluetoothKSP and receives the correspondent signature result using the Windows sockets API to access the Windows Bluetooth driver. The signature content is then forwarded to the desktop applications. All these interactions between

the *BluetoothKSP* and user applications are done though the Windows Cryptography API: Next Generation (CNG). This means that, our *BluetoothKSP* communicates directly with the *CertificateManager* by sending unsigned and receiving signed documents hashes, respectively, forwarding them from and to the desktop application, acting as a proxy. When the *CertificateManager* receives the signature request through the Bluetooth Service linked to the Bluetooth HAL, it will execute the signature generation in the SE context, in which the private keys are stored. This generates a signed document hash that will be forward back to the *BluetoothKSP* also through the Bluetooth Service.

CertificateManager. The Android application that represents the remote mobile crypto provider and implements the features of a KSP, but also enables the importing of certificates (resident in PKCS#12 files) existing in the mobile device storage into the Android Keystore, consequently, inside the SE container, and the listing of certificates already imported. This *CertificateManager* will access the hardware-sealed SE through the Android Keystore System, which is responsible for managing all SEs integrated in the smartphone and enables the user to store private keys in these sealed containers, as well as, execute cryptographic operations. Additionally, these system offers the possibility of implementing control access policies to restrict how and when private keys can be used. Much of the mobile security relies on the SE which will implement all cryptographic operations and request our consent for any action evolving sensitive data stored in the SE. Figure 3.3 illustrates where this Android application is placed in the Android Framework, where the other mobile components are also.

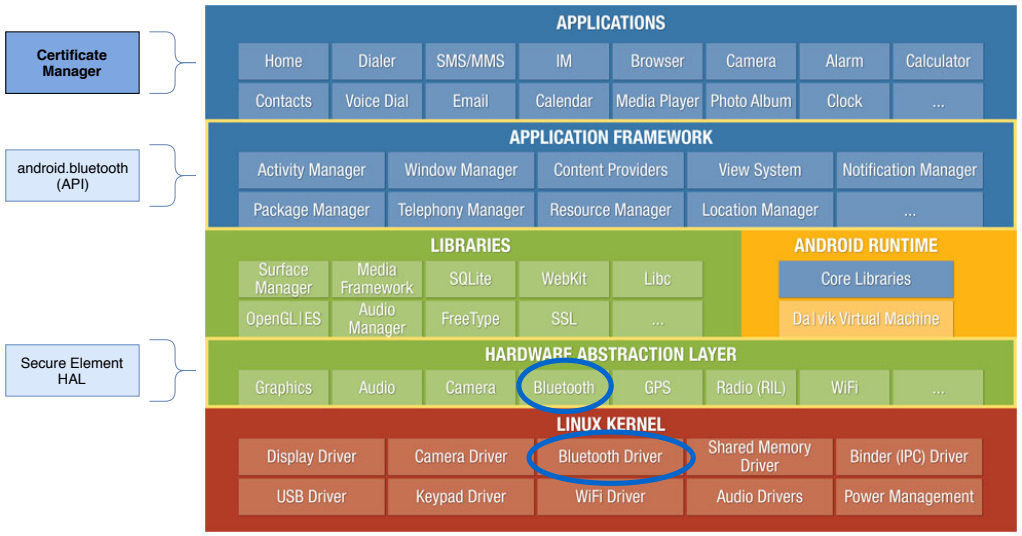


Figure 3.3: Mobile System Architecture and Android Framework.

Bluetooth HAL and Driver. These two are the low-level components abstracted by the Android Bluetooth API which is used by our *CertificateManager* to transmit and receive Bluetooth packets. While acting as a server, the smartphone runs the *CertificateManager* with a Bluetooth service on the background that waits for requests coming from a client computer and executes the operations demanded, responding with the operation result

and other useful data. Those requests packets are transported through the Bluetooth Driver and the Bluetooth HAL as well as the response packets.

Secure Element. The sealed-hardware part of the smartphone that provides a secure environment for storage, management and access control of cryptographic keys, and execution of cryptographic operations, reproducing the actual behaviour of a crypto provider, but using a sealed container for storage. This hardware piece is the main responsible for the execution of signatures in the context of our smartphone and solution since no other component implements signature algorithms neither executes digital signatures.

3.1.4 Components Interactions

Before executing the signature operations with the mobile crypto provider, the computer client must first obtain the respective certificates which will be used in the signature processes for requesting the signature and further verifying the signature. To obtain the certificates, the computer and the smartphone must first perform an initialization protocol by exchanging two Bluetooth packets (as shown in Table 3.1). This Initialization (or Configuration) Protocol is started by the *WindowsCertificateManager* when its demanded to import certificates into the Windows OS with our BluetoothKSP. A request packet (M_1) is sent to the *CertificateManager* containing an operation type (op) and the number of certificates requested (cert_num). Op indicates the Android application which operation the client is requesting: a certificates register or a signature operation. In this case, a operation code associated with the certificate register operation is placed in the op field of the packet. Cert_num provides the possibility to request a specific number of certificates from the *CertificateManager* or by default all certificates existing in the smartphone are retrieved. Then, the *CertificateManager* replies to the *WindowsCertificateManager* by sending a response packet (M_2) containing the list of certificates available in the mobile device. Finally, the user also has the possibility to select which certificates he/she wants to import and discard the rest.

Initialization Protocol
$(M_1) C \rightarrow S : (op, cert\_num)$
$(M_2) S \rightarrow C : (certs\_list)$

Table 3.1: Initialization Protocol.

After the configuration phase, a desktop application can request a signature to the mobile crypto provider and successfully verify the result. To accomplish this process, the computer and the smartphone must perform a signature protocol by exchanging other two Bluetooth packets (as shown in Table 3.2). This Signature Protocol begins when a desktop application starts a signature process with an imported certificate associated with the BluetoothKSP. A request packet (M_1) is sent to the *CertificateManager* containing an operation type (op), the document hash (hash), the certificate sequential number (sn),

the certificate issuerDN (idn) and the Object Identifiers (OIDs) of the hash and encryption algorithms. Similarly to the previous protocol, the op field is used to distinct the operation requested, in this case, a op code representing the signature operation is used. The hash attribute is the document hash computed by the desktop application that must be signed by the remote BluetoothKSP; the sn and idn (i.e, the combined unique identifier of the certificate being used in the signature) indicates the *CertificateManager* from which certificate it must obtain the private key to execute the signature; and the OIDs indicates which hash and public-key algorithms must be used to perform the signature, more precisely, which signature algorithm must be used.

Signature Protocol
$(M_1) C \rightarrow S : (op, hash, sn, idn, hash\_oid, enc\_oid)$
$(M_2) S \rightarrow C : (sign\_res)$

Table 3.2: Signature Protocol.

3.2 BluetoothKSP Architecture and Components

In this section, we cover the architectural components of the Cryptography API: Next Generation (CNG) KSPs [53] and the runtime support provided by CNG to them.

3.2.1 Key Storage Provider Architecture

CNG provides a plug-in model for private key storage that allows adapting to demands of creating applications that use cryptography features such as public or private key encryption, as well as storage of key material. That is, we can plug into CNG our own cryptographic key storage provider with our own implementations of the algorithms. All common Windows-based applications (e.g, Adobe Reader, Microsoft Word and Outlook) access the cryptographic functions exported by the custom KSP through the Windows CNG, more precisely, through the Key Storage Router. All access to private keys pass through the Key Storage Router, which provides a set of functions for storing and using private keys, and is audited by CNG. Additionally, the CNG API can communicate with the PKCS#11 API to invoke similar cryptographic functions demanded by common applications that use hardware cryptographic tokens as solutions for secure storage and execution. Figure 3.4 illustrates the CNG KSP architecture and all its components.

Our custom CNG BluetoothKSP is responsible for executing signatures remotely with private keys not stored in the computer, thus it does not manages or stores any private key, it only stores a reference or handler for the private key by creating a private key container without any key inside. This container is created when the corresponding certificate is imported into the Windows OS with BluetoothKSP as the cryptographic provider of

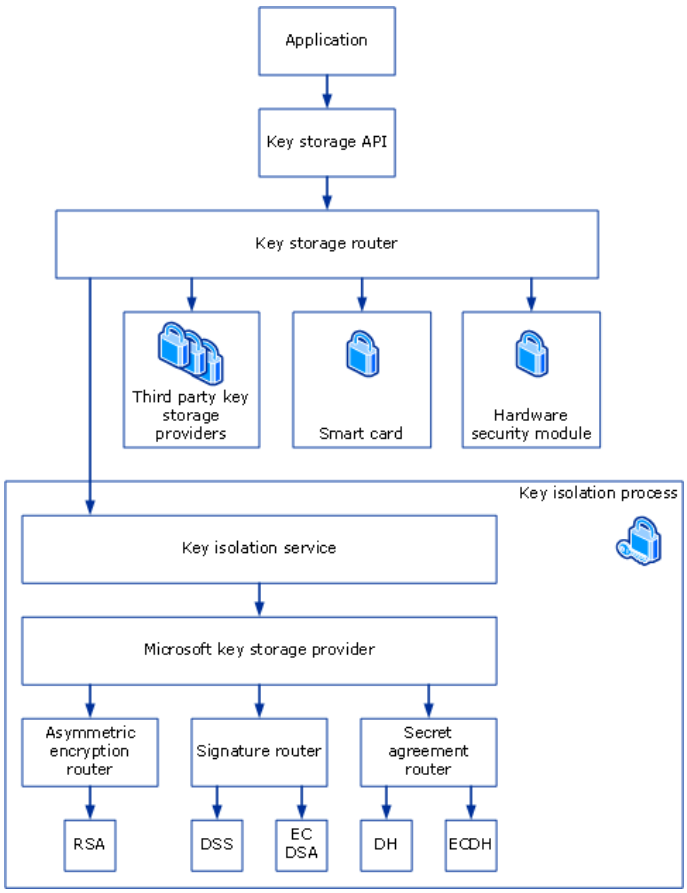


Figure 3.4: Architecture of the CNG Key Storage Providers (taken from [53]).

the certificate keys. As consequence, it does not perform signature generations, it only redirects the computed hash that needs to be signed to the *CertificateManager*.

CNG BluetoothKSP was implemented in a Dynamic Link Library (DLL) that exports an interface with a set of functions that the CNG Router calls to perform key storage operations. In general, each CNG KSP implements a list of callback functions provided by the CNG (available here [53]), so that the Router can call the callback functions implementations of each provider. The mapping between the generic callback functions and its implementation is defined using a specific table structure in the KSP DLL. Thus, every time a desktop application makes a call to CNG to perform a given cryptographic operation, the CNG Router will select the callback function implementation of the provider selected by the application correspondent to that operation.

In addition, this architecture provides isolation for the long term keys so that they are never shown to the application process. This key isolation feature is only available for the Microsoft KSP (as we can see in Figure 3.4), this mean that, only the Microsoft KSP is loaded with the key isolation service (or also known as LSA process). Therefore, the remaining third party KSPs, including our BluetoothKSP, are not loaded with this service.

As previously explained, to make custom third-party KSPs available for use with Windows-based applications they are required to be plugged into CNG through a provider

registration function (named *BCryptRegisterProvider*) that adds the provider to the CNG Router, making it available for future cryptographic operation calls.

3.2.2 Runtime Support

At runtime, the CNG KSPs can work in user and kernel mode. Such as CAPI, CNG can also be used in both modes to fully support the cryptography features, but not all the CNG functions can be called from kernel mode. Microsoft documentation explicitly states which functions can not be called from kernel mode [53]. Another constraint about this mode is that CNG does not support third-party providers and algorithms running in kernel mode. Only the algorithm implementations provided by Microsoft through the kernel mode CNG APIs are supported in kernel mode. For example, our BluetoothKSP cannot run in this privileged mode. The cryptographic services are provided to the kernel components through the Microsoft kernel security support provider interface (named *Ksecdd.sys*), which is a cryptographic module residing at the Windows kernel mode level and runs as a kernel mode export driver. The only exception at the level of cryptographic constructions supported by the interface *Ksecdd.sys* is the cryptographic algorithm DSA.

3.3 Cryptographic Functions

Our BluetoothKSP mainly provides digital signatures support, but also offers several basic cryptographic-related operations that different desktop applications may require to perform the signature operations, such as public key initialization (e.g, Acrobat Reader) or the public key exporting (e.g, Microsoft Word). Furthermore, it provides Key Storage Provider (KSP) management operations, such as KSP properties getter and setter functions, and KSP initialization operations. We can call these basic functions invoked by desktop applications to execute digital signature processes as KSP pre-operations. Each desktop application can invoked a different set of these operations.

3.3.1 Windows KSP Functions and Supported Operations

Regarding the Windows CNG, it provides a generic list of functions that each KSP must implement [53], so that, external desktop applications can invoke all the operations that satisfy their cryptographic demands through the CNG, independently of the current crypto provider being used. Our BluetoothKSP only implements or actually uses a small set of these functions, since we are only focused in cryptographic operations related with digital signatures and some operations are very application-specific, such that, we are only interested in functions that conventional Windows applications (such as Acrobat Reader, Microsoft Word and Outlook) use in order to provide transparency.

The functions provided by our BluetoothKSP API are summarized below:

- *OpenProvider*: Initializes the provider, retrieving a handle to the provider;

- *FreeProvider*: Releases or frees the provider handle created by the *OpenProvider* function;
- *GetProviderProperty*: Retrieves the value of a given property of the KSP;
- *SetProviderProperty*: Updates a property value of the KSP;
- *IsAlgSupported*: Determines whether the provider supports a specific cryptographic algorithm or not;
- *EnumAlgorithms*: Enumerates the names of the cryptographic algorithms supported by the KSP;
- *EnumKeys*: Enumerates the names of the keys stored by the KSP;
- *OpenKey*: Opens an existing and stored key, i.e, initializes a key object, retrieving a handle to that object;
- *FreeKey*: Releases or frees a object or key handle created by the *OpenKey* function;
- *ImportKey*: Imports a CNG key from a given memory BLOB;
- *ExportKey*: Exports a CNG key to a memory BLOB, more precisely, a public key BLOB. Notice that, this BLOB has a different format according with the key algorithm;
- *SignHash*: Forwards a Bluetooth packet containing a hash value (plus the remaining information described in Section 3.1) to the remote BluetoothKSP via a secure Bluetooth communication channel and waits until this remote provider signs the hash and sends back the signature value. In other words, it is performed the *Signature Protocol*;
- *VerifySignature*: Verifies a signature value over a given hash using a specific public key handle retrieved by the *OpenKey* or *ExportKey* function.

Therefore, when a desktop application is calling one of these functions, what it is really calling is the respective KSP callback function (*NCryptX*) provided by the CNG (where *X* is the name of one of the above functions). For example, when Acrobat Reader needs a signature, it calls the *NCryptSignHash* function from the CNG, indicating the KSP that provides the desired implementation of that function, which consequently calls the *SignHash* function of the chosen KSP.

Another supported feature available by our solution is the certificate import previously described in the *Initialization Protocol*. Basically, consists in importing a given certificate into the Windows operative system setting the cryptographic provider of the certificate private key as our BluetoothKSP, i.e, the private key will be stored and managed by BluetoothKSP. Although, the public key crypto provider is a Windows default

KSP. Usually, this operation implies that the KSP keeps the private key of the certificate in a secure storage area, but in our case only the certificate public part is persistently stored and a link between our BluetoothKSP and the certificate private key is created. This certificate registration process is similar to the one described in [49] with some differences. In general, the steps are: (1) read the certificate file which was already received through Bluetooth and create a certificate context object from the encoded portion; (2) set new certificate properties related with key storage (such as configuring the private key crypto provider as our BluetoothKSP and assign a name to the private key container which is used by KSPs to distinguish keys) and append them to the certificate context; (3) add the certificate to the Windows certificate store using the updated context object.

However, to achieve this certificate import from the Windows-side, the certificate and its respective private key must have been previously imported into the Android Keystore (more precisely, stored in the Secure Element (SE)). This means that, if no private keys exists inside the SE, the list of certificates received by the *WindowsCertificateManager* application during a certificate request will be empty and will be impossible to further perform remote digital signatures with the certificates using our BluetoothKSP. As such, before starting the Initialization Protocol, the user needs to import the certificates through the *CertificateManager* in order to be able to import certificates into the Windows OS, and as consequence, perform digital signatures using our BluetoothKSP. To accomplish this operation the following sequence of steps is required: (1) search the Android file system for the certificate file (i.e, a .p12 file); (2) retrieve the public and private key inside that file; (3) add a new entry in the Android Keystore for that cryptographic keys.

3.3.2 Provided Digital Signatures

By default, the CNG provides several built-in cryptographic algorithm implementations grouped in different classes according with their cryptographic operation. One of them is the algorithm class Signature which includes RSA, Elliptic Curve Cryptography (ECC) and DSA based digital signature schemes. However, our BluetoothKSP does not implement any of these algorithms, only redirects the signature operation requests to the real crypto provider running in the mobile environment, restricting in some way which type of signatures are allowed or supported by the remote BluetoothKSP.

In relation to the digital signatures schemes available by our BluetoothKSP, are included the following standard constructions: RSA-based signature schemes, such as RSA-PKCS#1 and RSASSA-PSS [58, 78, 79], DSA-based signature schemes [31, 61] and its ECC variant, ECDSA [1, 12, 69]. For this algorithms we selected a group of key lengths composed by RSA and DSA keys with 1024, 2048, 3072 and 4096 bits; and ECC keys with P-256, P-384 and P-521 elliptic curves standardized by the National Institute of Standards and Technology (NIST). These key lengths were chosen according with the algorithms and respective key sizes supported by the Android Keystore System since it is the component responsible for performing the digital signatures. The only algorithm

from the supported digital signatures that can't be executed with the Android Keystore provider is DSA, which is already not supported by the Android OS. Therefore, the use of DSA will be for testing purposes only. Additionally, the desktop applications that demand digital signatures through our BluetoothKSP must use for signature verification a crypto provider that offers the same digital signature schemes as BluetoothKSP in order to validate the signature. Any other desktop application that requires a different type of scheme, will have it's request rejected by the BluetoothKSP.

The signature scheme to use is dynamically selected in our solution when some desktop application invokes a signature request. That is, given the hash computed by this application and the certificate chosen to use in the signature, the BluetoothKSP selects the target signature scheme (i.e, the combination of the hash and encryption algorithms) to be used and forwards it to the smartphone. More concretely, the BluetoothKSP obtains the hash algorithm from the hash size and the encryption algorithm from the certificate public key. The BluetoothKSP, which needs the signature value, indicates the OIDs of the algorithms in a field of the packet sent to the remote BluetoothKSP, so that it knows which signature scheme the BluetoothKSP demands to be used in the signature generation. The next section presents a comparison between these supported digital signature schemes regarding the level of security offered by each one.

3.3.3 Comparative Analysis on Provided Digital Signatures

The experiments on digital signatures that will be later presented focus on the expected performance while using RSA, ECC or even DSA based digital signatures, in standardized constructions adopted for our prototype. Yet, we must noticed that standardized DSA-based constructions using SHA-1 [61] have been discontinued by Google from the Android OS reference, after version 6.0 (API level 23) [1], due to collision attacks against the SHA-1 hash function [23, 86]. Therefore, we restricted our following discussion to RSA and ECC public-key algorithms.

In this section, we intend to compare another interesting perspective on the comparison of RSA and ECC based signatures where we must additionally consider the security level comparison, beyond the performance analysis. As it is well known in general, the level of security in cryptosystems is becoming a primary concern as we would expect, with the key size criteria for different algorithms taking a particular relevance, complementarily to the robustness of each cryptographic algorithm itself and the usage efficiency. Security level consists in a measure of the strength that a crypto primitive - such as an algorithm or an hash function - achieves and is usually expressed in bits. Current cryptosystems provide a minimum security level in the form of 128 bits of security, which is required for systems that communicate and provide information with confidentiality

[1] https://developer.android.com/about/versions/marshmallow/android-6.0-changes

protection [14]. The security level of cryptographic algorithms comes from the combination of each specific algorithm mathematical construction and its key size. For example, it is known by cryptographic experts that a 128-bit security level can be achieved today with 128-bit Algorithm Encryption Standard (AES) keys (as reference for symmetric encryption), 256-bit ECC keys (for ECC-based encryption), and 3072-bit RSA keys (for RSA-based encryption) [14, 83]. If ignoring the implementation issues related to the mathematical instantiation of the cryptographic schemes, these algorithms alongside those specified key sizes will generally offer the same level of security.

Table 3.3 illustrates the comparison of security levels, when using today different algorithms and key lengths. As the table dictates, typical RSA implementations that currently employ 1024 and 2048 bit keys are less secure than the AES-128 reference, showing that asymmetric cryptography is not necessarily stronger than symmetric cryptography in each specific use. Also, ECC can provide higher levels of security with smaller keys comparing with RSA.

Table 3.3: Security Level Comparison for Conventional Cryptographic Algorithms (taken from [62]).

Security Bits	Symmetric Encryption Algorithm	Minimum Size (bits) of Public Keys	
		RSA	ECC
80	Skipjack	1024	160
112	3DES	2048	224
128	AES-128	3072	256
192	AES-192	7680	384
256	AES-256	15360	512

For our comparative analysis of the trade-off between efficiency and security of RSA and ECC based signatures, we must take into account the corresponding level of security of used keys. As it is well known, key lengths generally increase with time and the years to break a given key length increase with the key size, as the computational power available to possible attackers continues to increase, which is a manifestation of the so called cryptographic arms race. Nowadays, it is recommended for example that AES using 256-bit keys (AES-256) be employed for data encryption rather than the prior accepted AES with 128-bit keys (AES-128), since it only provides a near term protection (i.e, security for at least ten years) [83]. In the case of using ECC for key management (or key establishment) schemes of an AES-256 based session, it is recommended to use 512-bit ECC session keys to maintain the security level (as shown in Table 3.3), because the security strength that can be supported by sessions keys is determined by the weakest algorithm and key length. For example, if a 224-bit ECC key was used, the bits of security provided would decrease to 112. To achieve the same level of security strength of the AES-256 using RSA encryption, 15 360-bit keys must be required, although are computationally infeasible in current days, particularly when we use mobile systems with autonomous

battery or embedded systems and IoT devices. This limitation does not exist when using ECC, which has very small keys compared with RSA that enables faster processing times and lower demands on memory and bandwidth.

This notable contrast between the feasibility of ECC over RSA in relation with the security strength provided clearly indicates that ECC will be the algorithm more and more adopted in the future for asymmetric-cryptographic operations on systems where computational power or data storage capacities are limited, such as mobile and IoT devices. Except for these resource-limited systems, or where a strategic long-term migration is needed, there is no strong reason to employ ECC over RSA based signature schemes.

In the case of our evaluations for digital signatures, beyond the self robustness of secure hash functions used in standardized constructions (another point of security analysis), we must consider the comparative security level and efficiency for each achieved performance, regarding the use of 3072-bit keys for RSA-based signatures and 256-bit keys for ECC-based signatures (using curve P-256 from NIST), which is their equivalent.

It is important to notice that the security and key length of the cryptographic algorithms do not really matter if an attacker can obtain the secret and private keys through other methods. This way, we must emphasize the following: security starts and ends with how well the keys are managed and protected, and how good is the key storage. In addition, insecure and weak algorithms implementations, bad random number generators and various malicious attacks can also compromise security.

Finally, for the purpose of this dissertation it is interesting to discuss the use of cryptographic configurations in the combination of four relevant impact factors: the computation efficiency; the bandwidth (in the case, impacting the Bluetooth communication); the usability in the mobile environment and the relevance on government and industry standard recommendations.

As already mentioned, cryptographic algorithms execute two relevant computations in the process of digital signatures - signature (i.e, encryption using the private key) and verification (i.e, decryption using the public key) - which accordingly with the given algorithm have different performances. In general, RSA has an higher signature time than ECC since RSA keys are significantly larger than ECC keys; and has a lower verification time due to the fact that ECC executes more complex operations rather than RSA as stated in literature [42]. More precisely, they tell us that in relation to the performance or time costs at 128-bit security level, RSA is generally reported to be 10-times slower than ECC for private key operations such as signature generation, key establishment and key generation. The divergence in performance keeps increasing drastically as the key length increases, for example, at 256-bit security levels, RSA (using 15,360-bit keys) can be 50 to 100 times slower. On the contrary, for public key operations such as signature verification, RSA does not suffer big time variations as we increase the key size, which means that even doubling the key size the verification process is still fast compared with signature generation. The key generation of RSA is also significantly slow compared with

the key generation of ECC, with RSA being 100 to 1000 times slower and capable of draining completely the battery of a wireless device when using sufficient large key sizes. However, this may or may not be a relevant consideration to take into account in systems that rarely generate keys. In the case of this dissertation, no protocol or functionality requires the generation of keys frequently, therefore, it does not matter for our goals.

Regarding network bandwidth, the main concern relates to the symmetric algorithm used for message encryption and Message Authentication Coding (MAC) for integrity checking. For instance, small embedded system could start sessions often or the asymmetric-based authentication could represent a big part of the overall traffic and the size of keys and signatures would make a difference, since small key lengths do not increase the messages length and less bandwidth is consumed. The length of the public and private keys, and also signatures, is much shorter for ECC, for example, at 128-bit security levels, public keys and signatures are 6-times larger for RSA compared to ECC, and private keys are 12-times larger for RSA. As consequence, ECC has lower bandwidth demands. The key length generally has no impact on performance, but size matters when it comes to the cost of secure key storage. However, in our case, these differences are mitigated, because each cryptographic configuration used for digital signatures will only depend on the size of the secure hash of the document sent by the computer via Bluetooth to the mobile device, and the size of the returned signature obtained. In this case, ECC has also the advantage, because for the same level of security (128-bit) we will return a Bluetooth packet containing 70 to 72 bytes compared with a packet of 256 bytes in the case of RSA.

For the mobile environment, we consider now the suitability and efficiency of RSA and ECC based digital signatures with mobile phones. RSA algorithm is usually suspended from use with small wireless devices, because its usage will negatively affect their performance and consume a lot of resources such as memory and energy, delaying the verification process [6]. In the other hand, many studies and authors in literature, already have reported the advantages of employing ECDSA for resource-limited environments due to its performance and security [6, 17, 42]. Attending our dissertation case and goals, our performance evaluation helped us to gain more insight into the most suitable public-key cryptography algorithms for a mobile environment with resource limitations (such as memory, energy and CPU capability). In our case, the only cryptographic operation executed in the mobile environment is the signature generation with the private key, which was already demonstrated that its very slow for RSA and very fast for ECC. Only in a solution that performs signature verification in a mobile device, is ideally to employ RSA over ECC, because verification times are significantly different from each other and RSA scales well with the key length increasing. This way, we concluded that ECC-based digital signatures are more effective and feasible for our mobile environment in many aspects, since they provide a good security plus a good efficiency.

Considering for last the government and industry standard recommendations, there is an almost endless list of new standards that are recommending and requiring the use of public-key algorithms based on ECC rather than traditional key systems, such as RSA,

DSA and DH. Specially, for the case of constrained environments, where we need security without impacting severely the performance and resources. A small selection of these standards is presented below:

- ZigBee and the complementary IEEE 802.15.4 standard: these networking standards specify the ECDSA and ECDH asymmetric algorithms as the algorithms of choice for authentication and key management respectively in constrained environments [18]. Notice that, these standards focus on wireless networking protocols that are built to operate over ad-hoc networks, where other wireless technologies such as Bluetooth are less than ideal.
- Security Module PP Standards: the German Federal Office for Information Security (BSI in German language) agency has published a set of standards for energy metering gateway security which specifies different cryptographic functionalities based on ECC as the generation and verification of digital signatures and key agreement to be employed in Smart Metering Systems [34].
- Intelligent Transportation Systems (ITS) Standards: this set of standards documents the choice of the automotive standards industry for elliptic curves as the algorithm of choice for car-to-car and car-to-infrastructure communication security due to the low power consumption and space required, and because it offers the most security per bit of any public-key scheme [47].
- Suite B Cryptography Standards: this set of standard algorithms published by the US government (more precisely, NSA) is approved for use in non-defense applications [63]. We must notice that currently, this standard includes only ECC for digital signatures and key management, and RSA has been completely removed due to various reasons described in [62].

3.4 BluetoothKSP Initialization and Setup

In this section, we group all the configuration or installation processes required to fully use the BluetoothKSP to perform cryptographic operations executed in a smartphone environment demanded by Windows-based applications.

Firstly, we should define or configure which Bluetooth security mechanisms (mainly, the Pairing method), and also the usage of security enforcements, the BluetoothKSP will provide to the Windows-based applications that use it. Secondly, we must do the BluetoothKSP registration in the Windows OS, so the OS recognizes it as a valid Windows cryptographic provider as the other installed cryptographic providers. Besides this registration operation, the representative DLL of the BluetoothKSP must be copied to the *System32* folder of the Windows OS, where common Windows DLLs are also maintained [53]. Consequently, Windows starts to recognize our DLL as another Windows

DLL. Also, it is important to note that Windows only trust DLLs that were previously code signed. To ease this whole BluetoothKSP configuration process we could produce a registration executable program that registers the BluetoothKSP as a KSP and copies the respective DLL file into the *System32* folder. Finally, the Initialization Protocol its performed through the graphical *WindowsCertificateManager* application to accomplish the certificates registration into the operating system with our BluetoothKSP. After these few configurations, we are able to fully use the BluetoothKSP to sign documents remotely using a smartphone.

3.5 Bluetooth Security Considerations and Enforcement

Bluetooth Security is a very complex domain and is very dependent on the Bluetooth version of mobile devices, given that each version implements different security mechanisms and the backward-compatibility feature always tries to provide compatibility for low Bluetooth versions. Although, it is possible to increase the baseline security provided to communication channels between devices with low Bluetooth versions via an Extra-layer of Security at an higher level, such as a secure Transport Layer Security (TLS) channel. Furthermore, it is possible to discard a security enforcement as the TLS and only use the most recent and complete version of Bluetooth (i.e, v4.2 or v5.0) that can provide a low-energy and secure protocol for Bluetooth communications. Unfortunately, we need mobile devices with these versions or a strong security baseline can't be provided. These approaches to Bluetooth Security and secure communication channels are described throughout this section.

Bluetooth Classic. In a first approach, we implemented the Bluetooth Classic protocol between both devices - one acting as server and the other as client - with the security mechanisms that compose the Bluetooth baseline security. Figure 3.5 represents the establishment of the Bluetooth communication channel. Before the formation of the link, the client device scans the local area for Bluetooth-enabled devices that are currently accepting connection requests. This procedure is called Device or Service Discovery. These devices running Bluetooth services (through the *bluetooth_accept* blocking method) are also called discoverable devices, such as our server device. Using the information exchanged in this process, the client device can initiate a connection with the discovered device (invoking the *bluetooth_connect* method) to establish Radio Frequency Communication (RFCOMM) channels to exchange data with a selected device. Then, the Pairing process initiates if this is the first time a connection is made and a pairing request is automatically presented to the user according with the Pairing method used (e.g, PIN or SSP). In this process, both devices exchange security informations (such as the security features that each one supports), both devices establish a shared link-key to be used for authentication and to create an encrypted connection with each other. When the devices are paired, the informations (mainly, the security keys) are saved and can be re-used

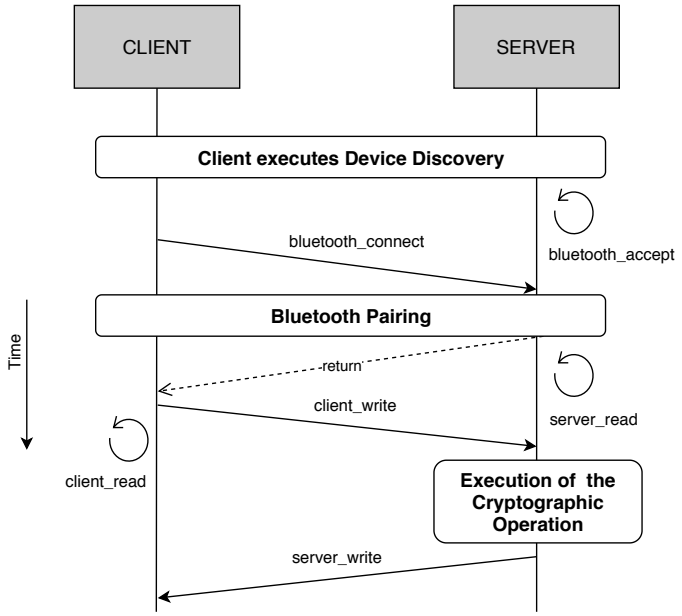


Figure 3.5: Bluetooth Stack Operations Flowchart.

through the Android Bluetoothy APIs, and is no longer needed to execute another Pairing unless the devices became unpaired. This process is automatically performed when a connection is initiated or received with the Android Bluetooth APIs. After the pairing and connection processes are completed, the two devices start exchanging data trough the Bluetooth socket streams: the server device invokes a blocking method (*server_read*) that reads data from the input stream; the client device invokes *client_write* to write to the output stream the data to send to server-side and waits for the response invoking *client_read*; when the server device reads the data from the stream, executes the cryptographic operation demanded and writes the response to the output stream, so the client can receive the result from the operation. Once the session is completed, the device that initiated the pairing request releases the channel created with the other device, but they remain bounded to each other so the can reconnect automatically during future sessions.

In our system, there are two Bluetooth communications channels that are established: one during the certificates configuration and other during the remote signature. In situations where both devices have high Bluetooth versions, these channels are created using the most recent security mechanisms defined in versions 4.1, 4.2 or 5.0, such as the Secure Connections (with Numeric Comparison) pairing mechanism required by the Security Mode 4, which is employed by default by the Bluetooth APIs or libraries used. This way, we can mitigate the MITM and eavesdropping attacks. Therefore, confidentiality, authentication and integrity are provided for mobile devices that make use of Security Mode 4 (i.e, mobile devices with Bluetooth v4.1 beyond). Although, only public data is transmitted via Bluetooth while critical data (such as private keys) never leave the device (i.e, the SE), it is important to use mechanisms that grant high security to achieve a more reliable and secure solution, mainly, in the case of devices with low Bluetooth versions.

The Pairing mechanism can be parametrizable (according with the Security Mode employed by the device) in order to grant flexibility to the solution in terms of which security features can be used. In the Classic Bluetooth case, the supported Pairing mechanisms are PIN Pairing (i.e, Legacy Pairing), SSP and Secure Connections (using a given association model). Although, these mechanisms can be parametrizable for recent mobile devices, for older devices is impossible since those devices don't support Secure Connections or even SSP. Also, some of these mechanisms are more secure than others as we presented in the previous chapter. PIN Pairing is used by v2.0 or older Bluetooth devices and is classified as insecure, SSP is used by v3.0 to v4.0 devices and despite its enforced security algorithms has some well-known faults, and Secure Connections is used by v4.1 or recent devices and its classified as the most secure Pairing mechanism. This problem can be aggravated given the high use scale of old Bluetooth versions (below v4.0), which is very high for the Android devices [20]. Yet, it is estimated that until 2023, 90% of all Bluetooth devices will include BLE and also the presence of Bluetooth BR/EDR, enhancing the rapidly replacement of single-mode BR/EDR for Dual-mode radios.

Bluetooth Security Enforcement. In order to grant support for various Bluetooth versions ensuring a minimal security base for all Bluetooth mobile devices was implemented an Extra-layer of security in our solution. This layer has the goals of avoiding to work only with the Bluetooth security baseline and increasing the protection granted to the connections between mobile devices with lower Bluetooth versions. Also, it is possible to enable or activate this Extra-layer of security given the Bluetooth version of the mobile device. This means that, for old devices we activate the Extra-layer, but for recent mobile devices with versions equal or higher to v4.0, it has no need given that it would create an useless overhead over the communication channel.

The security protocol or mechanism that composes the security enforcement is the TLS protocol, which enables us to create a neutral layer between the application layer and the network layer (in this case, the Bluetooth channel) as represented in Figure 3.6. TLS or SSL consists in the establishment of a secure point-to-point channel where data exchanged between two endpoints is always encrypted. Generally, this protocol provides three security properties: authenticity, confidentiality and integrity.

The combination of TLS protocol and Bluetooth is analogous to the use of HTTPS, but

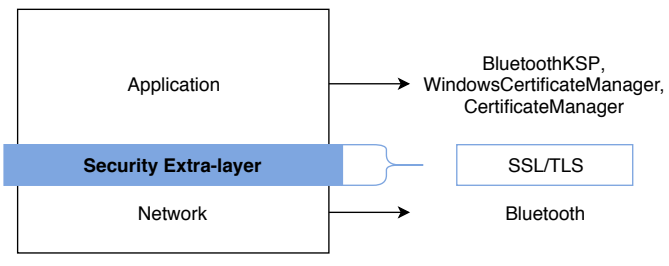


Figure 3.6: TLS/Bluetooth Stack.

with the difference that in this case we create a secure communication channel double-encrypted that enforces the Bluetooth Security Baseline.

Figure 3.7 represents the establishment of the TLS channel over Bluetooth. The creation of this secure link is similar to the establishment of the Bluetooth connections, because a connection request is also executed from the client-side, and the server-side also is blocking and waiting for these TLS connection requests. Although, in this case, the difference is that instead of performing the Pairing process, the TLS Handshake protocol is executed between both devices. After the creation of this secure and encrypted channel over the already encrypted Bluetooth communication channel, we can perform the execution of a particular cryptographic operation (such as digital signature or certificate registration). The data exchange procedure is also equal to the normal Bluetooth protocol, but using the *client_write*, *client_read*, *server_write* and *server_read* from the TLS Protocol.

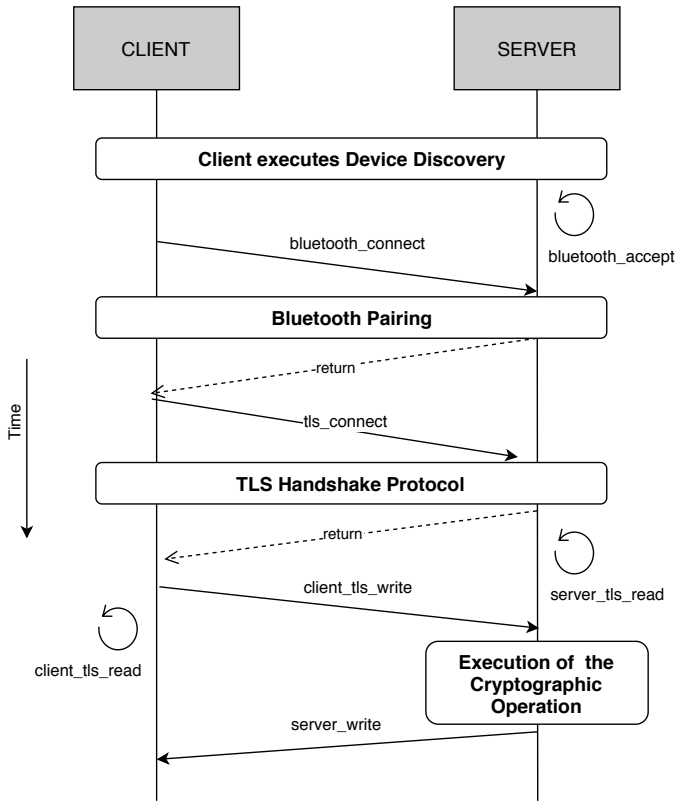


Figure 3.7: TLS/Bluetooth Stack Operations Flowchart.

Bluetooth Low Energy. In a final approach we handled the second specification of Bluetooth technology: the Bluetooth Low Energy (BLE), which is designed to provide low power consumption for wireless devices that have stricter power resources such as heart rate monitors, proximity sensors and fitness devices. Regarding the specification protocol, BLE technology uses different protocols compared with Bluetooth Classic that uses RFCOMM protocol, which are focused in the functionality and features of the IoT devices. The main protocol consists in sending and retrieving small portions of data (also

known as attributes) using the minimal bytes possible, being called as Attribute Protocol (ATT) [19, 38]. On top of this protocol, is built a generic specification for exchanging these attributes over a BLE secure link named Generic Attribute Profile (GATT). This notion of profile consists in the specification of how a device works with a specific application. Bluetooth Special Interest Group (SIG) provides various conventional profiles for Low Energy (LE) devices, such as heart monitors and proximity sensors, and these can implement more than one profile. This combined protocol and specification is commonly referred as GATT/ATT.

The attribute values transmitted through the ATT are formatted as characteristics and services that are available on a LE device working as server (or GATT server), and are uniquely identified by a Universally Unique Identifier (UUID). A characteristic is similar to a type or class and contains a unique value and a descriptor that specifies the value description in a human-readable way. A service is basically a collection of characteristics. LE devices acting as client (or GATT client) can search for services hosted in other LE devices and read or write the characteristics values or descriptors of that service. We can think of this protocol as a key-value mapping that is readable and/or writeable by a LE client devices. For example, this protocol is useful for situations where IoT devices - such as smart bands, heartbeat sensors or beacons - are constantly measuring metrics such as heart rate, temperature or proximity, or are simply hosting important information that a further LE client device can query to know the values of those metrics.

Before establishing a connection, GATT client or server devices adopt one of two roles: Peripheral or Central. A device in the first role is responsible for advertising its service to other LE devices and a device in the second role scans, searching for LE advertisements. For example, if a given device wants to communicate data to other LE devices, it may start broadcasting advertisements and a second device that wants to received them, starts looking for that service advertisements in order to further write or read the service information (i.e, the characteristics values). Notice that, is required to have both roles for establishing a BLE connection, because two devices in Peripheral role or two devices that could only support Central role can't talk to each other.

In relation to the two Bluetooth communication links existing in our model, only one was created using the BLE technology, which is the channel established during the certificates configuration. Because this link is based on Bluetooth versions that include LE (i.e, versions 4.1, 4.2 and 5.0), it can also be created using the most secure mechanisms available, such as the Secure Connections (with Numeric Comparison) Pairing method, providing the confidentiality, integrity and authentication of Security Mode 4.

3.6 Summary Remarks

In the beginning of this chapter we presented an overview of the system model and a vulnerability analysis considering some of the existing threats in both computer and

mobile side. We defined a thread model where the packets transmitted can be tampered, the Secure Element (SE) can be replaced by a malicious one, root exploits can clean all the SE cryptographic content and a corrupted SE can produce incorrect data to be signed by the user. Given the adversary model, we described in more detail our architectural components and how they relate with each other by exchanging information, for example, executing protocols such as the certificates configuration (Initialization Protocol) and the remote signatures execution (Signature Protocol).

Next, we defined the principal component of the KSPs architecture (i.e, the CNG Router), which is the responsible for the management of different KSPs registered in the Windows OS and the access to all cryptographic content of the respective KSP demanded by desktop applications. Our BluetoothKSP is a third-party KSP, represented as a Dynamic Link Library (DLL), that implements a common interface from which the CNG Router can invoke function implementations.

The primary cryptographic operation provided by our mobile crypto provider is signature generation using different parametrizable digital signature schemes with different performance gains, although other operations, as the KSP pre-operations, may be necessary to complete the signature operation when using different desktop applications. We also presented, the certificate import operation in the Windows (only the public key) and Android OS (both public and private keys).

Initialization and setup of the BluetoothKSP includes the registration of the provider in the Windows OS to be available to the CNG Router and the placement of the KSP DLL in the Windows file system.

Finally, we presented the Bluetooth security mechanisms employed and parametrizable by our solution, such as the Pairing method, and the enforcement techniques, such as TLS, used to increase the communication channel security and protection. As an alternative to the Bluetooth Classic specification, we employed the secure and energy-efficient Bluetooth Low Energy (BLE) for devices with more recent Bluetooth versions, avoiding the need of adopting costly security enforcements.

Given the risks and threats that exist in our mobile-computer environment and how our solution can be used to mitigate them, in the next chapter, we discuss the implementation of the system model and architecture, addressing a prototype to be used for experimental evaluation.

Chapter 4

Implementation

In this chapter, we present the implementation details and decisions related to the prototyping effort of the system model proposed in Chapter 3. This way, we start by presenting an overview of the developed prototype (in Section 4.1), followed by a description of the software building blocks and technologies used by the prototype (in Section 4.2). Then, it is shown the development technologies and environment used to build this prototype (in Section 4.3), and the prototype implementation effort (in Section 4.4). The chapter concludes with a discussion of the issues that arose during prototype implementation and related observations (in Section 4.5).

4.1 Prototype Overview

Following the design of the BluetoothKSP system model proposed, we implemented a prototype in which a mobile phone acts as a crypto provider, enabled with two Bluetooth communication modes and with parametrizations for digital signature schemes, Bluetooth Pairing methods and Security Enforcement, and being possible to be used transparently with real Windows desktop applications.

As said before, we intended to develop a prototype with strict existing software building blocks. Regarding the computer operating system, we developed the prototype to run on a Windows machine, since it is required the usage of the Cryptography API: Next Generation (CNG) to implement the BluetoothKSP. In relation to the smartphone operating system, we adopted the Android OS due to its high usage rate in today's mobile market and also because Android has a flexible way to communicate with the smartphone's Secure Element (SE) through the Android Keystore System. For the Bluetooth technology, the prototype is based on the Bluetooth versions 4.1 and 5.0, because are, respectively, the versions of the computer and smartphone adopted for development and testing.

In practice, this prototype was developed essentially as three software programs, representing the main building blocks or components of the system proposed:

- *CertificateManager*: a mobile application that acts as a certificate manager and represents the remote BluetoothKSP, which waits for certificate registration and signatures requests incoming from Windows desktop applications;
- *WindowsCertificateManager*: a UI-based desktop application in the Windows-side to import certificates requested to the mobile application;
- *BluetoothKSP*: the proxy Dynamic Link Library (DLL) responsible for redirecting the signature requests from Windows desktop applications to the mobile application.

In addition to these components, our prototype also includes the implementation of a benchmark client that executes certificate registration and/or digital signatures operations. And can be parametrizable in terms of public-key certificates, signature schemes, Bluetooth-related security mechanisms, tls-based security enforcement, Bluetooth Low Energy (BLE) and test documents with different sizes for system validation and experimental observations. These benchmark clients will be later detailed.

4.2 Building Blocks and Technology

As a starting point for the development of the prototype, was required to choose which technologies were to be used to develop the main building blocks. This included choosing the technology on which we would implementing the *WindowsCertificateManager* and the *CertificateManager* application, as the *BluetoothKSP* library must be strictly a Windows DLL. The following Table 4.1 describes the technologies used for each on of these building blocks. In the next subsections, we describe how the building blocks in each computing side were implemented using these technologies.

Table 4.1: Technologies of the Building Blocks.

	Building Blocks	**Technologies**
Server-side	CertificateManager	Android app (in Java)
Client-side	WindowsCertificateManager	.NET app (in C#)
	BluetoothKSP	DLL library (in C++)

4.2.1 Android-Side or Server-Side

On the Server-Side, the *CertificateManager* application, or remote BluetoothKSP, was implemented using the Java programming language and the Android package manager

Gradle, targeting smartphones with Android 7.0 (Nougat) or later operating systems. The adopted Java version was 8 and the Gradle version was 4.5.1.

All support libraries used in this application to develop the functionalities belong to the Android Framework, including the Bluetooth APIs for creating and managing the Bluetooth Service, and the Keystore System API for storing, accessing and manage cryptographic data and execute cryptographic functions. The adopted crypto providers to satisfy these cryptographic demands were the AndroidKeystore and the Spongy/Bouncy Castle provider, being the second added to the Android project as a Gradle dependency. As an exception-case, this second provider was used to enable the possibility of using digital signatures schemes that are not supported or were impossible to execute in the desired way by the AndroidKeystore provider, such as DSA and RSA-PSS. Also, it was used to implement the Security Enforcement, i.e, the server-side of the TLS/Bluetooth secure channel.

As a sub-component of this application, there is Bluetooth Service supporting Classic (using RFCOMM protocol) and LE specifications (using GATT/ATT protocol), which is simply a thread running in the background of the application waiting or listening for Bluetooth connection requests from the *WindowsCertificateManager* application or the *BluetoothKSP* library. This service is uniquely identified with a UUID and only applications with its knowledge will send connection requests successfully. In the prototype, this service is always initiated when the application starts, but the main goal would be to put this Bluetooth service running in the background of the Android OS along with other system services. As a result, end users wouldn't always need to have the application running to complete signature processes and through notifications they would know when a signature operation is requested. In the case of BLE, there is a slightly different, which is the Bluetooth Service advertises the service that it is providing and other client devices must scan for these advertisements to request some operation to the service.

The server-side implementation of the TLS/Bluetooth secure channel was done through the Bluetooth sockets already being used in the Bluetooth Service, or more precisely, through the input and output streams from these sockets. After the server device accept a connection request, the socket streams are initialized and passed to our TLS service, where is executed an accept call on the input stream to wait for TLS handshake requests from clients that invoked a TLS connection request on the output stream from the client-side. After the handshake is completed, the server device start listening for data in the encrypted input stream to read.

This mobile application is able to respond to certificate and digital signature requests via Bluetooth as it acts as a cryptographic provider, import certificates and respective privates keys in local .p12 files into the Android OS and display of the current certificates installed. Beside these main functionalities, it is possible to configure the application to work with BLE-based communication channels or TLS/Bluetooth secure links. For implementation and testing purposes, this configuration is done through a config.properties file, but in a real deployed application, it would be configured in the settings menu of the

Android application.

4.2.2 Windows-Side or Client-Side

On the Client-Side, the *WindowsCertificateManager* application was implemented using the .NET Framework and the C# programming language with the goal of providing a graphical and intuitive application to enable users to manage their certificates (i.e, visualizing them or registering new ones). The versions adopted were, respectively, 4.7.2 and 7.0 for .NET Framework and C#.

In relation to the UI and aesthetics part of the implementation, we adopted the Windows Presentation Foundation (WPF)[1] Framework, which is part of .NET . This Windows UI framework enables the creation of desktop client applications with a broad set of development features such as controls, graphics, layout, data binding and security. Other remark is that WPF uses the Application Markup Language (XAML) to create the layouts for the application.

This WPF desktop client application includes as support libraries the *32Feet* for the Bluetooth Classic client-side implementation and the *Bouncy Castle* for the client-side implementation of the TLS/Bluetooth secure channel. As we used .NET Framework, these dependencies were added to the WPF application project as NuGet packages. Notice that, we adopted a third-party Bluetooth library to enable the use of Windows 7 or 8 as the operating system of the client computer due to the fact that the most recent Bluetooth APIs in .NET and Windows are only supported in Windows 10 OS, because Microsoft is fully moving to Universal Windows Platform (UWP) applications and the widespread adoption of Windows 10, which can be impractical for many developers in many ways. However, the *32Feet* library does not support yet the BLE technology or, more precisely, the GATT/ATT protocol specification. Therefore, we had to adopt one of these new Bluetooth APIs in order to implement the BLE support in our prototype. As consequence, the LE feature will only work on our solution if a Windows 10 computer is used, besides the implicit restriction that the devices involved in the Bluetooth communication must support LE technology.

The client-side implementation of the TLS/Bluetooth secure channel was also done through the Bluetooth sockets streams, where the client device first sends a connection request to a server device with the target service, obtaining a socket stream if the connection is successfully established, and then sends a TLS connection request via the TLS Service, initiating the TLS handshake with the server device. After the TLS handshake is completed, the client can write the data to the new encrypted stream.

The Bluetooth Classic client-side implementation (using RFCOMM protocol) consists, firstly, on discovering Bluetooth devices and, secondly, connecting to the devices that provide the target service represented by a given UUID, for then exchange information. For example, in this case the WPF application sends a connection request for the certificate

[1] https://docs.microsoft.com/en-us/dotnet/framework/wpf/

import operation and the server retrieves the set of certificates to be selected and imported into the Windows OS. Regarding the implementation of BLE (using GATTT/ATT protocol), the *WindowsCertificateManager* application scans for LE advertisements of a specific service, and when it finds the device advertising that service, it sends a connection request and then a request for read or write a given characteristic/attribute.

Due to some implementations issues regarding the certificate import operation in C# code, we had to implement it using C++. This is possible because .NET Framework cryptographic functions usually redirect the function call to CNG functions that are implemented in C++. This means that, in most of the cases, a given sequence of operations in C# code can be reproduced in C++ code. Our certificate import operation implemented in C++ was encapsulated in an intermediary DLL to be accessed by C# code. The sequence of instructions performed by this operation was already explained before, but in general consists of configuring the cryptographic provider of the certificate private key as our BluetoothKSP, creating only a link without storing any key material, and registering the certificate in the Windows Certificate Store.

In the other hand, the *BluetoothKSP* library was implemented using the C++ programming language as any other DLL library in the Windows OS, which also needs to be registered as a valid Windows cryptographic provider. The version employed was version 17 and the compiler used was Clang. This library follows and implements the CNG callback functions for KSPs as defined previously, but only using some of them to complete digital signature operations via Bluetooth and using a SE as the real crypto provider.

Besides CNG and CAPI library support at C++ level, other important C++ libraries were used to accomplish the required functionalities, such as the Windows Sockets interface (or WinsockAPI v2.2) library to conduct the implementation of the Bluetooth Classic client-side (using RFCOMM protocol[2]) and the *Botan* library[3] which offered a way to enable the implementation of a TLS/Bluetooth secure channel. Related with the BLE support at this lower-level technology, was adopted the same library as in the C# code (i.e, WinRT APIs, but implemented in C++) to establish a BLE link between the computer and smartphone. In this case, the use of this library restricts the use of BLE technology to perform digital signatures only with Windows 10 computers.

Additionally to the *BluetoothKSP* library implementation, we developed a small *BluetoothKSP* configuration client (using C++) to perform the registration of the BluetoothKSP in the Windows OS and to validate the implementation of the *BluetoothKSP* functions. This client program was also based from a configuration client provided by the Windows Cryptographic Provider Development Kit.

[2]https://docs.microsoft.com/en-us/windows/desktop/bluetooth/windows-sockets-support-for-bluetooth

[3]https://botan.randombit.net/

4.3 Other Development Tools

Our development environment included computing devices such as Windows 10 desktop computers with support for Bluetooth connections (though a Bluetooth interface) and Android smartphones with embedded-SEs. We consider that the intended solution must be designed and developed for the support of other Bluetooth versions (classic or BLE-based), to cover the majority of current devices in the market. In [20] it was announced that 100% of all smartphones, tablets and laptops shipped in 2018 (estimated in 2.1 billion devices) include Bluetooth natively, predominantly with Bluetooth v4.0 and v5.0. We evaluated in our development how the current Android platforms include support for the Bluetooth network stacks, and how we can transparently support different Bluetooth versions. Therefore, in our approach we guided our developments possibly looking for the more representative platforms in the Portuguese Android market today: Android OS 6 (Marshmallow), 7 (Nougat) and 8 (Oreo) versions, trying to obtain data from the Portuguese operators or from other possible sources.

4.4 Implementation Effort

In terms of implementation effort, the whole source code of the three different software projects developed has a total of 26.444 lines of code (loc) as shown in Table 4.2. The Android and .NET applications, which were developed from zero, resulted in around 12,630 and 3,422 lines of code, respectively. The BluetoothKSP was implemented based on a Sample KSP available through the Cryptographic Provider Development Kit from Microsoft. The addition of new features and changes to the original functions represented a total of 1,136 lines of code, since the original codebase from the Sample KSP was totalled with 9,256 lines of code.

Table 4.2: Prototype Implementation Effort in terms of Lines of Code (LoC).

Building Block	**LoC**
CertificateManager	12,630
WindowsCertificateManager	3,422
BluetoothKSP	10,392
Total	26.444

In developing this solution, the difficulty felt was somewhat high given the numerous technologies, programming languages and languages paradigms used and the purpose of bringing these different software and hardware components together into a flexible computer-mobile cryptographic solution. Furthermore, some of these technologies have not allowed us to fully meet some of the initial objectives due to the lack of technology

support or the inability of these technologies to provide the means to implement the required features. In the next section, we describe some of these situations in more detail.

4.5 Implementation Issues and Final Remarks

There are some final issues related with the building blocks and their components that must be emphasized as relevant implementation concerns, when addressing the developed prototype.

The first issue was the registration of a new certificate on the Windows OS with our BluetoothKSP as the cryptographic provider of the certificate private key, because there are few examples of how to do this with the Cryptography API: Next Generation (CNG) using C# on the Web. Hence, to solve this problem, we had to implement this operation in C++, where more samples and better documentation are available to assist the development of the operation, and in C# code call it through an intermediate library that has the actual implementation of the certificate registration function.

Other problem was knowing which cryptographic algorithms the desktop application with signature demands were requiring when calling the BluetoothKSP signature function. Specially, which hash and public-key algorithm was being used. The point of this is that the remote BluetoothKSP needs to know which digital signature scheme must be used in the signature generation process on the Secure Element (SE) context. This issue was easily solved by appending the respective algorithm OIDs in the Bluetooth packet sent during the signature process as seen before.

An recurrent obstacle in the prototype development was without doubts the interoperability between the computer-side (C#, C++ and .NET) and the mobile-side (Android and Java). For example, the DSA and ECDSA based signature schemes have different signature format on each platform. All Microsoft technology employs the IEEE P1363 signature standard format [46] while Java employs the DER format such as OpenSSL. This constraint has been overcome by converting from DER to P1363 format after the signature generation in the Android side. Related with this issue was also the format used in the Bluetooth-based communications, i.e, which format is necessary to encapsulate multiple informations that need to be sent to the other endpoint. This format needed to be generic to the three building blocks for ease of reading and writing data that its differently formatted in each platform, so, JSON-formatted messages or packets were employed to prevent interoperability issues. For example, in Java, the bytes are signed while in C# and C++ the bytes are unsigned.

Regarding the Bluetooth Low Energy (BLE) implementation, we faced some issues when sending write requests for a given characteristic to a target service. In the first place, the write request function that enables a responsive execution, i.e, a waiting for a response from the server-side, was not working correctly given successive errors. In the second place, the BLE packets exchanged are so small that it is almost impossible

to execute digital signatures, unless the signature result size does not surpasses the BLE maximum packet length.

A relevant future issue from the Windows support is the common generalization that Microsoft is doing of using UWP applications and Windows 10 APIs for the future applications in order to remove completely the use of older Windows operating systems.

Overall, this solution implementation is complex, mainly in terms of interoperability, and is very dependent from the cryptographic support from the employed platforms, in the sense that is not easy and direct to employ a new cryptographic algorithm in the system since we have to implement the support for it in the different components and some of them may not support it at all through their respective libraries or APIs. Also, we have to be careful with the data exchange between these different platforms and how it is processed.

Chapter 5

Experimental Evaluation

In this chapter, it is presented the experimental evaluation of the prototype implementation described in Chapter 4. Several experiments were made and conducted using a well-defined testing environment and a set of parametrizations which included different hash functions, public-key algorithms, key sizes and security enforcements. First, we describe the components of the test bench environment (in Section 5.1) and present a test bench to measure the cost of the configuration phase of our solution (in Section 5.2). Then, we present a use-case of our solution with conventional Windows applications (in Section 5.3) and proceed to present the results of the experimental benchmarks regarding digital signatures (in Section 5.4). Next, we present the performance evaluation of Bluetooth, more precisely, the Low Energy (LE) impact on the execution of digital signatures and also the benchmark observations on the overhead imposed by the Transport Layer Security (TLS) enforcement (in Section 5.5). Additionally, we present other validation metrics, such as packaging, memory usage and energy consumption (in Sections 5.6). For last, a brief summary on the results obtained is presented (in Section 5.7).

5.1 Test bench Environment

In this section we present the generic test bench environment, including the system topology, used resources, test assumptions and test conditions, and also the technical specifications of the computational systems used.

5.1.1 Generic Test bench

The testing environment for evaluating our prototype, was mounted using the same computing devices presented in the previous Chapters, more precisely, a Windows computer and an Android smartphone. We set up the following topology between the two: the

computer was hosting and executing the Windows *CertificateManager* application and the Benchmark clients, besides the installed and pre-configured BluetoothKSP; the mobile phone was hosting and running the *CertificateManager* application, working as a tiny server. These two devices were positioned less than ten centimetres apart.

In terms of resources for testing, we used a set of sample documents (i.e, .pdf and .docx files) with different sizes (more precisely, ranging from 1 MB to 100 MB) as signature test objects and a set of certificates with keys of one of the three different public-key algorithms employed(i.e, RSA, Elliptic Curve Cryptography (ECC) and DSA) to be used in the signing operation. The range of documents size was chosen based on the samples obtained from an analysis of documents used by 43 users in a typical office environment - such as in Multicert[1]. Regarding the certificates, we used the KeyStore Explorer [2] tool to generate RSA, ECC and DSA self-signed certificates, distributing them in the form of Public Key Cryptography Standards (PKCS)#12 files to user's smartphones, so they can import the certificates into the Android OS. It is important to notice that, these certificates are self-signed and, consequently, are not accepted as trusted by common applications and services, they are just for development and testing purposes. In normal cases, the certificates are distributed to users via Certification Authorities (CAs), which issue trusted certificates signed by the CA itself and other root CAs that authenticate the credibility of the issuing CA. In our testing environment, we assume that certificates have already been obtained by conventional ways.

To obtain better observations and results during the experiments, we executed all benchmarks in a clean environment where only our applications were running and we disabled some features which imply user interaction in order to remove the entropy when measuring the metrics. For example, the Bluetooth Pairing mechanism was already executed and the devices were paired when the benchmarks were running, the Bluetooth device discovery was configured to search for a specific device and the notifications from *CertificateManager* to user to accept certificate and signature requests were turned-off. In the case of signature requests, this would imply inserting the password of the private key stored inside the Secure Element (SE) through a PIN or Fingerprint based authentication.

Regarding the Bluetooth security features such as the pairing method, algorithms for key generation and key establishment and others, they were also negotiated before any pairing or data communication occurs, but after the Bluetooth devices were selected. In this case, having one device with Bluetooth v4.0 and other with v5.0 implies the usage of security features from v4.0 due to the backward-compatibility feature of Bluetooth devices, as explained before. In this case, the finite security baseline agreed between the two devices included the usage of Security Mode 4, which uses SSP as pairing method with P-192 ECC algorithm and HMAC-SHA-256 hash function, and employs the association

[1] https://www.multicert.com/pt/
[2] https://keystore-explorer.org/

model named Numeric Comparison. As said before, with this Bluetooth security baseline we achieve an authenticated pairing and key establishment processes to setup a secure channel under mutual authentication.

5.1.2 Software Environments and Devices

The following Table 5.1 summarizes the technical specifications of each computational system.

Table 5.1: Technical Specifications of the Test bench Environment.

	Windows Computer	**Android Smartphone**
OS	Windows 10 Professional	Android 7.0 Nougat
CPU	Intel Core i5-3230M 2.60GHz	Qualcomm MSM8998 Snapdragon 835
RAM	8GB	4GB
Model	Toshiba Pórtege R930-17U	Galaxy S8
Bluetooth	4.0 (with LE)	5.0 (with LE)

5.2 Setup and Benchmarking

In this section, it is presented the configuration phase (or setup) benchmark and the measured performance results - in terms of latency - of our solution using a variable number of certificates and Bluetooth Classic without any security enforcements.

This benchmark consists in the evaluation of the impact of configuring the certificates for registration (or import) in the Windows OS before starting the execution of digital signatures. The main goal of this benchmark was to obtain the latency of the Bluetooth channel and the time taken by the Windows OS to execute the cryptographic operations associated with the certificate registration in this phase. Plus, this test allowed us to measure how much time is taken to validate certification chains, considering also the possibility of existing intermediary certificates already revoked.

To perform this benchmark, we implemented a small Setup Benchmark client which executes a sequence of certificate registration requests to the *CertificateManager* (i.e, the Initialization Protocol) over the same number of certificates to register existing in the smartphone environment. This sequence of requests or benchmark runs was done with the intent of obtaining the average and standard deviation of latency. It is important that the chosen number of benchmark runs is high so that statistically unique points or effects can be discarded and the uncertainty principle can be reduced, for example, 20 runs as in our case. Regarding the number of certificates, was used a set of 3, 5 and 10 certificates.

The results from this experiment are shown in Figure 5.1. This graph (or Boxplot) illustrates the distribution of latency values obtained from benchmark runs based on a five number summary (minimum, first quartile(Q1), median, third quartile(Q3) and maximum). Additionally, we can estimate the standard deviation by looking at the skewness of the central rectangle. As we can observe, we achieved an overall low latency, more precisely, less than half a second on average to complete a certificate registration with a variable number of certificates and only obtaining few outlier samples (i.e, individual high latency peaks). Only with 10 certificates, the latency is higher than 0.5 seconds. The existing outliers are due to the fact that the first connection that is established between the devices for each different number of certificates takes longer than the subsequent ones. This is because, an high number of BluetoothKSP functions are invoked as pre-operations before the certificate registration. As Figure 5.1 suggests, by varying the number of certificates, the latency in general tends to increase as the size of exchanged Bluetooth packets tends to be higher, as we expected. However, the variation isn't noticeable.

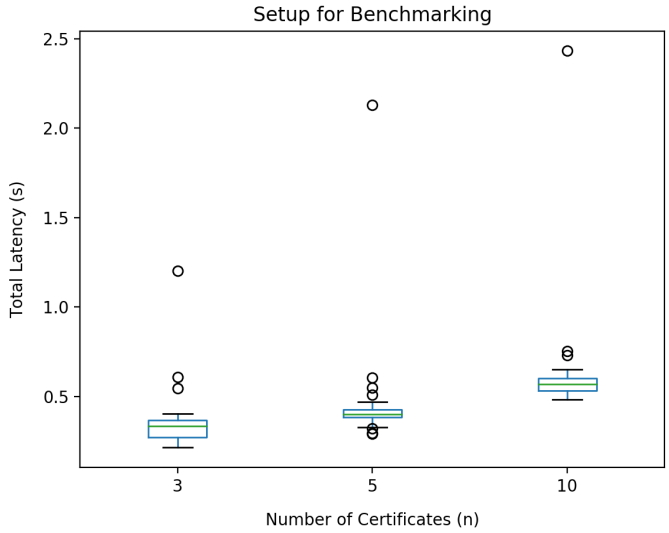


Figure 5.1: Setup for Benchmarking.

For better understanding of what composes the total latency, we divided it into two relevant temporal intervals (i.e, time chunks) - the Bluetooth Round-Trip-Time (RTT) and the certificate import duration - and measured how high was the percentage of each time chunk in the total latency. Note that, other meaningless times while executed auxiliary functions and computations are discard and not take into account for the total latency, only the measuring of these two relevant periods of times was done. Table 5.2 presents the average values of the components in the observed latency. Bluetooth RTT keeps rounding the same values, not varying to much even when increasing the number of certificates to exchange. Certificate import time also increases, but more significantly as it can double the time consumed. As such, we can conclude (on average) that approximately 70% of the setup time corresponds to time consumed while exchanging small data packets via

Bluetooth, producing the so called Bluetooth overhead. However, when increasing the number of certificates, this overhead can be reduced to proximately 50%.

Table 5.2: Latency of Certificate Imports

	Total latency (s)	
# Certs	**Bluetooth RTT**	**Cert Import**
3	0.269	0.065
5	0.325	0.118
10	0.358	0.253

The following Figure 5.2 presents the latency results from the Setup Benchmark using certificates containing certificate chains. We can observe that the latency increases because the import function must import the certificate chain of each certificate. In this case we used certificates with a chain containing a total of 3 certificates (including the root certificate itself), as consequence, a certificate import process, in reality, is a 3-certificate import process. For example, when importing 5 certificates retrieved from the smartphone, in fact, we are performing an import operation of 15 certificates.

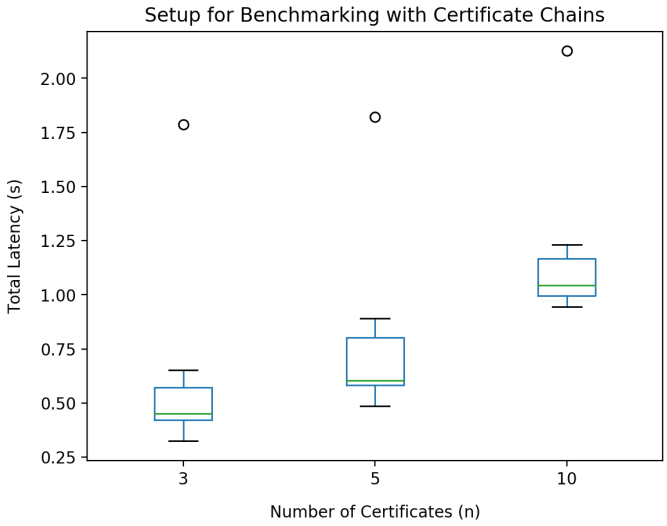


Figure 5.2: Setup for Benchmarking of Certificate Chains.

In a perspective of human time perception, the measured Bluetooth overhead is meaningless to an user, although in general it can perceive time differences until 20-30 milliseconds depending on some factors [32]. As such, the average 0.6 seconds obtained as the setup latency measured is a small time cost that users are willing to pay (or wait), which can be almost considered as instantaneous and provides a feeling of pleasure to the users. However, in real use cases where the users only need to install certificates one time

per different computers to further execute signature operations, the real observed latency will be the outliers' values (i.e, approximately 1 to 2.5 seconds) due to the pre-operations' slowness. Although, these values continue to be irrelevant for common users. Thus, we can conclude that even using Bluetooth as a wireless communication channel to exchange information during the setup phase, this configuration is not an heavy process in terms of time cost and that the Bluetooth overhead would not be a big bother to people. Yet, it is not a flexible process to further accomplish digital signatures, because it implies that this manual configuration must happen, for each new computer, before the computation of digital signatures through Bluetooth, between that computer and the smartphone containing the certificates and respective private keys.

5.3 Use of BluetoothKSP with Conventional Applications

In this section, we describe some use cases of our BluetoothRemoteKSP with conventional Windows desktop applications. It is described the observations that were made and the results obtained regarding the transparency validation of our solution. With theses qualitative experiments, we intended to observe how much our solution is transparent to the conventional applications that make use of it, i.e, how many different applications can use our BluetoothRemoteKSP to achieve their digital signature demands. Alongside the toy-application already adopted (Acrobat Reader), we selected two more applications to conduct this transparency experiment - such as the Microsoft Word and Microsoft Outlook applications.

5.3.1 Adobe Acrobat

In the case of Acrobat Reader, it offers users a way to protect pdf files through signatures using certificates from the Windows Certificate Store [2]. This feature is provided as a tool called "Certificates" among others. The Acrobat Reader digital signatures are performed by selecting the "Certificates" feature from the available tools of the application, where is prompted a list of installed and validated Windows certificates to select which one the user pretends to use to perform the signature. When selecting a certificate previously installed through our Certificate Manager application, Acrobat Reader will invoke our BluetoothKSP to complete the cryptographic operation. For this test, we performed the execution of RSA, ECDSA and DSA based signatures successfully with the key lengths presented before. RSA-PSS based signatures are not possible to perform in this context or application, because Adobe Reader does not provides any way or feature to enable the user to choose the padding to use in the case of RSA.

5.3.2 Microsoft Word and Outlook

Microsoft Word also enables users to add digital signatures to docx files [55]. This feature is provided in the options separator with the name "Protect this document". When

selected, the list of certificates installed in the operating system is shown, so the user can select the certificate to use in the signature process of the document. However, Microsoft Word digital signatures are performed differently from Acrobat Reader, i.e, the sequence of pre-operations invoked from our BluetoothKSP is different. For example, Microsoft Word calls the method *KSPEnumAlgorithms* to retrieve and show to the user the list of certificates existing in the Windows certificate store and *KSPExportKey* in order to obtain the public key to execute the verification operation. For this Word test, we only executed successfully RSA signatures since ECDSA and DSA signatures are strongly dependent from the implementation of the *KSPExportKey* function of the BluetoothKSP, which was left to implement in future work.

Similarly to Microsoft Word digital signatures feature, Microsoft Outlook enables users to add digital signatures to outgoing messages or emails [56]. Plus, provides the possibility to encrypt emails' contents and attachments. Although, in the case of Outlook, the signatures feature is provided in a different way, i.e, before creating a digital signature, the user must first configure the signing certificate and the algorithms (hash and encryption) that desires to be used. Then, when the user creates a new email and hits the send button, the signature is generated automatically with the signing certificate previously configured. In the moment of the email creation, the user only chooses the appearance of the signature, since the signing certificate must be configured always first. The certificates feature is available in the "Trust Center" separator which exists in the options menu. Under "Trust Center" menu, exists another separator named "Email Security", where we can import/export and manage our certificates (or Digital IDs) and all security settings related. In terms of functions invoked from our BluetoothKSP, Outlook behaves identical to Word calling also the functions *KSPEnumAlgorithms* and *KSPExportKey*, which is comprehensive because both applications are owned by the same company (i.e, Microsoft). For this Outlook test, we successfully signed emails only using RSA-based signatures with the same reason described in Word experiment.

5.4 Evaluation of BluetoothKSP Digital Signatures

In this section, we present the experimental benchmarks on digital signatures and the performance results (also in terms of latency) of our solution, using a set of parametrizations that include secure hash functions, key lengths of public-key algorithms, and digital signature schemes and security enforcements.

These benchmarks relate to the tests made to measure the latency of the BluetoothKSP supported digital signatures using different combinations of parametrizations. Our goal with these tests, beside measuring latency, is to compare key sizes, hash functions, types of signatures and even the time chunks that compose the total latency of executing a digital signature, as it was done in Section 5.2.

To conducted these benchmarks, we implemented a Signature Benchmark client that

executes a sequence of digital signatures using a given certificate over the set of sample variable documents with a given signature algorithm defined through the certificate public-key algorithm and a hash function. With this client we pretend to simulate a Windows desktop application that communicates with our BluetoothRemoteKSP to complete digital signature processes. After the execution of a predefined number of signatures (i.e, benchmark runs) for one document, we calculate the average latency and the standard deviation. This step is repeated for each test document, which in our case are a group of 5 with sizes of 1MB, 5MB, 20MB, 50MB and 100MB, respectively. As in the setup benchmarking, we also executed 20 runs of signing requests for each test to discard floating points and obtain a consistent average latency.

It is also worth to note that each first benchmark run always takes longer than the subsequent ones due to the higher number of BluetoothKSP pre-operations that are executed before the signing operation. When the subsequent runs are executing, some of the pre-operations results are already cached by the provider in memory, so there is no need to execute them again and the signature process will have a lower latency. Yet, these first runs are not visible in the result graphs since we are interested in the average results of the execution of multiple signature runs. However, some of these outliers can be seen through the Boxplots that will be presented.

5.4.1 Client-side Latency Observations

For a first benchmark, our objective was to compare the secure hash functions available in the SHA-2 hash algorithm family with each other to measure the impact of each hash function in the signature process. This comparison is extensively discussed in past and current literature and many experiments were already done [35, 39, 52], although, we are interested to know in which ways the hash functions could influence our solution. The results obtained from this experiment are illustrated in Figure 5.3, which shows the total average latency of digital signatures, using a specific hash function over a specific document size. Regarding the test conditions, we used a well-known public-key algorithm, i.e, the RSA algorithm with 2048-bit keys, and the set of documents described previously.

At first sight, the results clearly confirms the intuition that when the document size increases, the average latency also increases for each hash function, but slightly in the case of SHA-256. We can see that (1) for smaller documents below and up to 5MB, the latency is stable for all hash functions; (2) the SHA-384 and SHA-512 perform worse when the document size increases and always side by side; and (3) the SHA-256 shows a good performance for all documents size.

Besides this average of total latency for each hash function and document size, it is also important to have a good indication of how the latency values obtained from the benchmark runs are distributed, as done in the previous section. This distribution is illustrated in Figure 5.4. Notice that, in this Boxplot the latency values include the runs

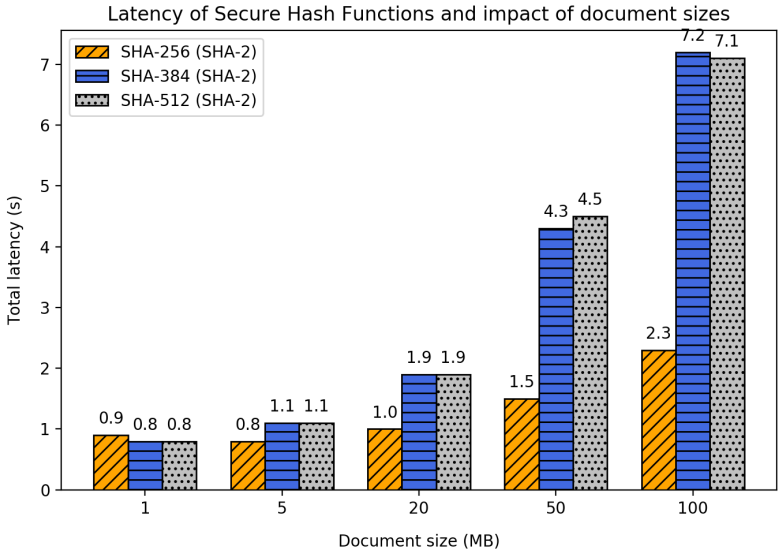


Figure 5.3: Latency of Secure Hash Functions and Variable Document Sizes.

for every document size used. The results show that SHA-384 and SHA-512 have the highest range of values and the highest latency values, reaching to maximums of almost 9 seconds to execute a digital signature. But remember, all the singular or outlier points correspond to the first execution and in this case, we only focus on the median values (i.e, the green lines inside the boxes in Figure. 5.4). In general, we can achieve an average latency independently of the document size of 1 and 2 seconds for SHA-256 and SHA-384 or SHA-256, respectively.

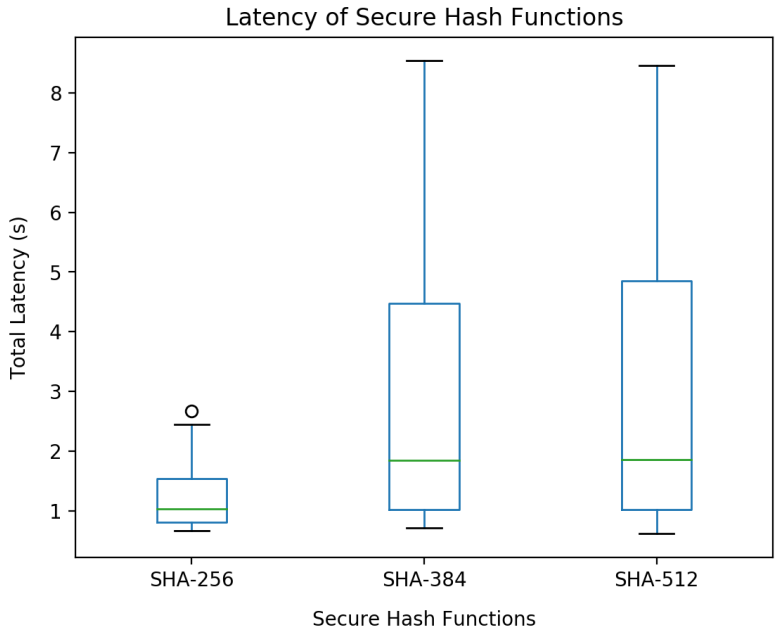


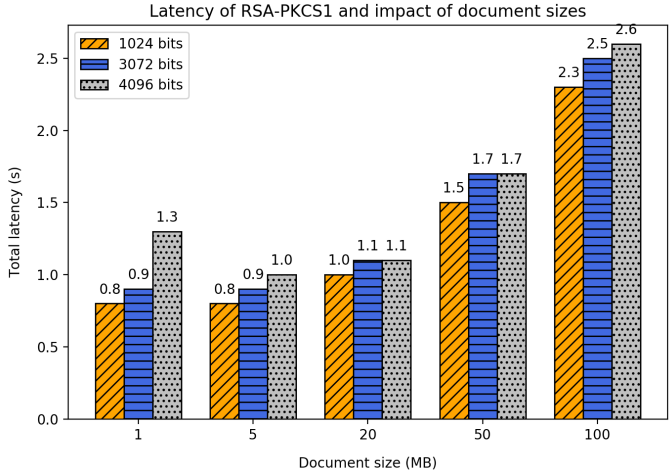
Figure 5.4: Latency Specifically Induced by Secure Hash Functions.

In a second benchmark, we intended to understand how much the key size of the cryptographic algorithms would influence the signature process time in relation with the document size. This way, we started by comparing the key sizes of the RSA and the ECC based algorithms in order to know what key length may imply in latency with our solution. Many authors in literature [8, 42] already have documented that larger key lengths increase execution times. As consequence, it is expected that they will increase the overall latency in our solution environment. In this experiment, we decided to put aside the DSA because it shares some similarities with RSA in terms of key length and performance in general [8, 71]. Figure 5.5 shows the experiment results of RSA with three different key sizes (1024, 3072 and 4096 bits) and ECDSA with three different NIST curves (P-256, P-384 and P-521), over a specific document size. For this test case, we choose SHA-256 to be used on the digital signature, which is the hash function measured with best performance in the previous benchmark.

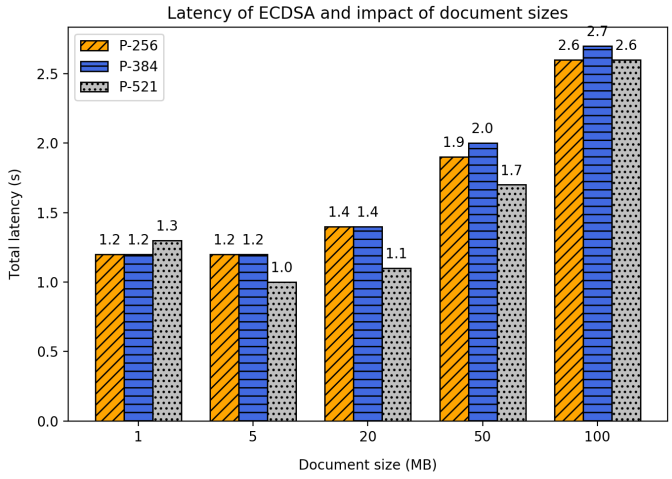
In general, the results show that the total latency tends to increase with the key size and the document size, as expected. This increase in latency is mostly consequence of the key size, since with larger keys the signature execution time quickly rises, delaying the transmission of Bluetooth packets. The time required for this operation increases for both algorithms, more significantly for RSA due to the larger keys comparing with ECDSA, although the results for both algorithms are very similar. This is due to the fact that the verification execution for ECDSA takes longer than RSA and also increases with the key size since ECDSA executes more complex operations, inversely, to the signature execution. Thus, the two algorithms have both heavy cryptographic operations that increase the latency in similar way. These observations can be seen in the next sub-section.

Similarly to the previous benchmarks, we illustrate the distribution of latency values through a Boxplot for each of the cryptographic algorithms used, in Figure 5.6. The results suggests that, the median value oscillates between 1 and 1.5 seconds (for RSA and ECDSA, respectively) and the values dispersion is similar in each algorithm between key sizes, except in the case o RSA-4096 and ECDSA-521. For these last algorithms, the maximum value easily surpasses the 3 seconds and the median value is slightly above the median of the other algorithms. Our results lead us to conclude that RSA-3072 and ECDSA-256 have the best performances providing the same level of security.

For the third benchmark, we present the performance comparison of the adopted digital signature schemes with the corresponding key length in terms of security, i.e, the RSA key size equivalent to the ECC key size in terms of the security level that they offer. As previously discussed, RSA or even DSA based encryption needs larger key size while ECC-based encryption requires significantly smaller key size for the same level of security. Opposite to the previous benchmark test, we decided to run an experiment which includes the DSA algorithm using a key length equivalent in terms of security level to the RSA and ECDSA with the intent of testing this algorithm with our solution. However,



(a) Latency of RSA and Variable Document Sizes.

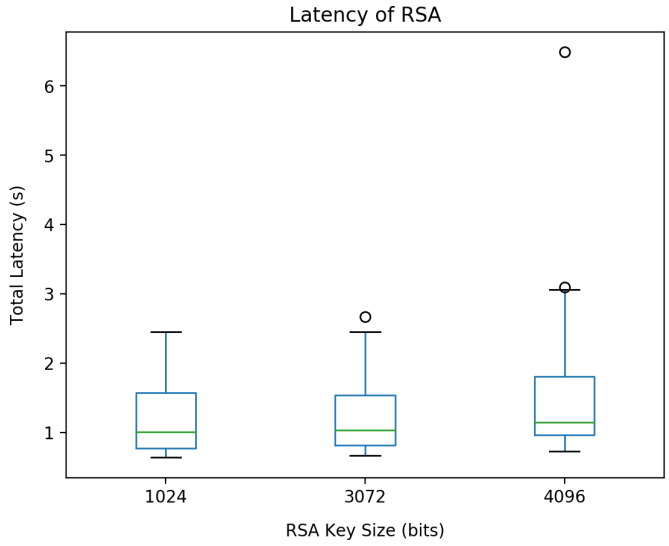


(b) Latency of ECC and Variable Document Sizes.

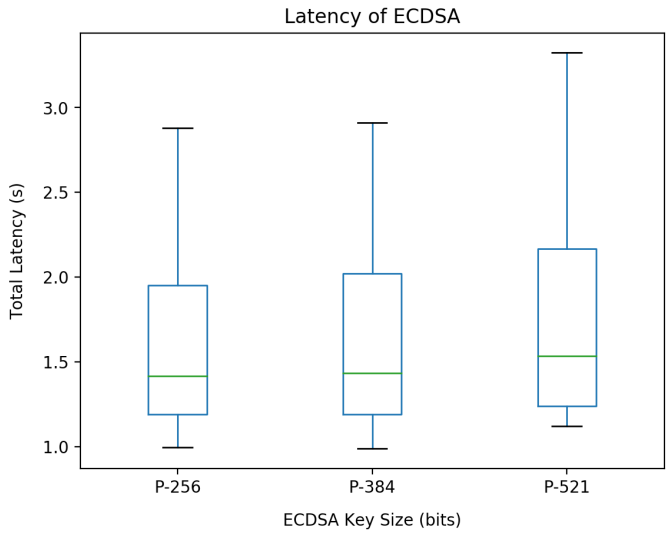
Figure 5.5: Comparison with Different Public-key Cryptography Algorithms and Variable Key Sizes.

this algorithm is no longer supported by Android mobile phones due to attacks against cryptography constructions, for example, SHA-1. As such, we stored a DSA certificate inside a local PKCS#12 file (in the application resources) to simulate a provisional key store. Consequently, the private keys are not stored in a secure location as in the SE and can be compromised, therefore, it is not recommended to employ this workaround in real-world applications. The experiment results for the supported digital signature algorithms are illustrated in Figure 5.7. These benchmarks were also conducted using SHA-256 as the hash function of the signature algorithms.

Surprisingly, the results present RSA as the algorithm with best performance in general for all documents size, even performing better than ECC. This seems somewhat contradictory to what we discussed earlier, that the signature generation time of ECC is



(a) Latency of RSA.



(b) Latency of ECC.

Figure 5.6: Performance of RSA and ECDSA Signatures.

generally faster than the signature generation time of RSA. In the next sub-section, we will present new pieces of information that will help us to understand and justify this result. In the other hand, DSA is the algorithm with worst performance, although DSA is usually presented as fast signature and verification times as seen in documented in literature [11, 71]. RSA using Probabilistic Signature Scheme (PSS) padding (i.e, RSA-PSS) [58] also presents a bad performance, but less than DSA. The reason behind the high latency values for both DSA and RSA-PSS is that our solution needs to access the resource file which represents a key store to execute digital signatures with these algorithms. To accurately understand how much time the read file operation takes, we measured this

operation time individually and we found that the application can take on average of 1.9 seconds to access the resource files containing the certificates. If we decrement this resource overhead in the average latency of DSA and RSA-PSS, we can preview that these algorithms will have a performance only a bit worst than RSA and ECDSA.

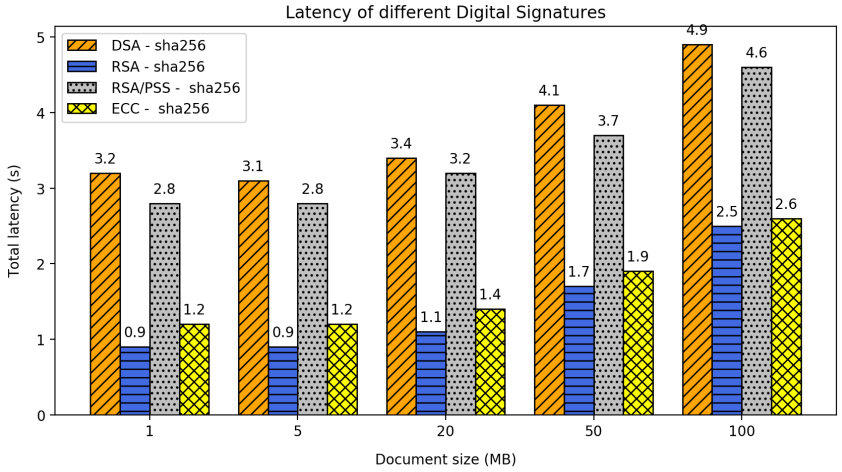


Figure 5.7: Comparison of Different Constructions for Digital Signatures.

Figure 5.8 illustrates the distribution of latency values through a Boxplot for these digital signature algorithms. The distribution enables us to see more clearly that DSA and RSA-PSS have the worst performance since the maximum, minimum and median values are always higher than for the other algorithms, more precisely, achieving signatures with 5 seconds of latency. These algorithms can execute signatures with 1 to 2 seconds of latency and in extreme cases less than 3 seconds. RSA and ECC based signature have the lowest value dispersion and the lowest median values, where RSA has the lowest median value but the highest dispersion.

Despite these results for signature schemes, it is important to notice that the evaluation of the number of operations and data transmissions involved in a security protocol is required to conclude which is the best algorithm to use on each particular case. Our mobile-computer environment has an use case, where the user receives signature requests non-sequentially but with a variable time-interval (i.e, opposite to the Signature Benchmark), executing one single signature from time to time. The algorithm that produces this signature must be secure and fast in signature generation and verification. Thus, we can concluded that ECDSA is the signature algorithm that best fits our requirements and objectives.

For a final test bench for latency, we compared the performance between our solution (i.e, BluetoothRemoteKSP) against the Windows-default provider (i.e, Microsoft Software Key Storage Provider) and a *token* (i.e, SafeNet Smart Card KSP) or *smart card* (i.e,

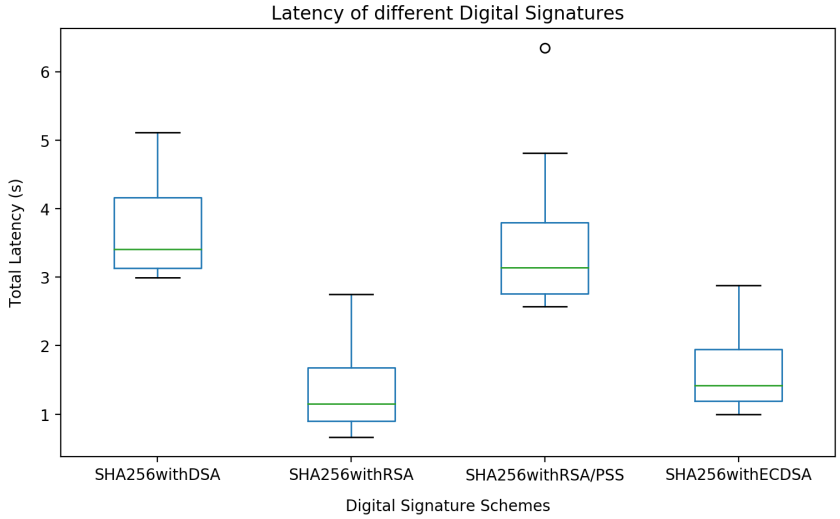


Figure 5.8: Performance Comparison of Constructions of Digital Signatures and Algorithms, using SHA-256.

Gemalto Classic Card CSP) based provider. Our solution is analogous to token and smart card based provider solutions in the sense that we use the mobile device as the crypto provider while in the token solution the provider is installed in a USB dongle, and in the smart card solution the provider is installed in a card chip, but they all are based in sealed-hardware devices. The main objective of this ultimate test is to show how our solution can be better or not in terms of average latency than these existing solutions. The test conditions for this final benchmark include the cryptographic constructions that demonstrated the best results in the previous benchmark tests, i.e, SHA-256 with ECC using a P-256 elliptic curve, for our solution. In the case of the remaining solutions, we used SHA-256 with RSA-2048 for token and smart card based solutions, and we used SHA-256 with ECC using P-256 elliptic curve for Windows solution. The results from this experiment are presented in Figure 5.9, where is shown the average latency of each provider solution per document size using the most secure and performance-efficient algorithms in each solution.

Our evaluation results follow the expectable trend that wireless communications impose an implicit overhead in latency measurements and is worst than others. In this case, is the Bluetooth communication channel that produces this overhead (i.e, the Bluetooth overhead) and delays the whole signature process of our solution, increasing the latency. As seen from the previous benchmarks, the Blueetooth RTT has always values between 0.2 and 0.3 seconds, therefore, this is the implicit overhead that Bluetooth imposes in our solution. However, it performs better than the smart card solution due to the fact that smart cards are slow chips and are the only solution that needs a card reader, which imposes a significant overhead. Analysing the results, we can conclude that the Bluetooth overhead makes our solution, in worst case, 1 second slower than token and Windows

solutions for any document size, which is not a real problem since this delay is meaningless to the final user. Furthermore, the ace of our solution is to provide flexibility, user-experience and on top of this, security as a priority.

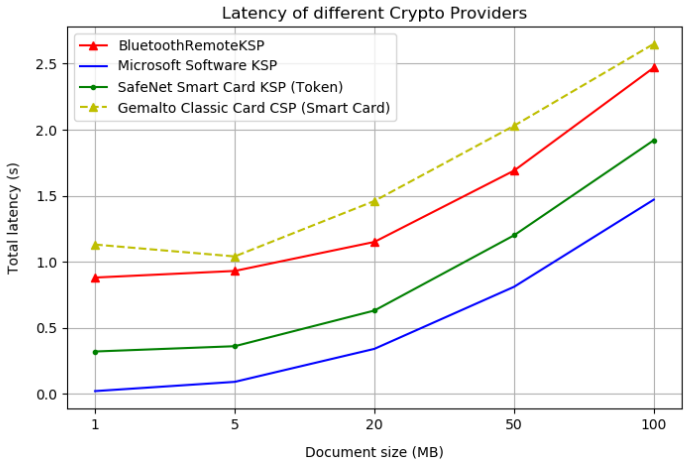


Figure 5.9: Evaluations with Different Cryptographic Providers.

5.4.2 Components in the Observed Latency

Now, we present the components in the observed latency for each one of the signature benchmarks as a complementary information to help understand the previous results and observations. This way, we divided the total latency in time intervals (or time chunks), so that we know which step takes longer time to execute and delays the signature process, increasing the total latency.

Following the same ordering of the previous sub-section, we start with the secure hash functions. In this case, we partitioned the latency of the secure hash functions to understand how much time their computation takes and which percentage of the total latency it corresponds. Table 5.3 presents the average time duration of each component in the total latency, where we can see this increase proportional to the hash function. For Secure Hash Functions, the components are the hash time, the Key Storage Provider (KSP) pre-operations time and the Bluetooth RTT.

It is normal to think that the time will not increase to much since the only thing that is transported through the Bluetooth channel is the hash of the document and not the document itself. And, independently of the document size, the hash will always have the same size if using the same hash function and as consequence the total latency should be stable and not increasing with the document size. However, this depends on the computation of the hash, that takes longer for bigger documents and, consequently, increases the total latency of the signature. Our results suggests that SHA-384 and SHA-512 are 4-times slower than SHA-256. Bluetooth RTT continues to be similar, only increasing

with SHA-384 and SHA-512 due to the larger document hashes. KSP pre-operations time is low and constant for all the secure hash functions.

Table 5.3: Average Latency for RSA Digital Signatures using Different Secure Hash Functions

	Total latency (s)		
Hash function	**Hash**	**KSP pre-Ops**	**Bluetooth RTT**
SHA256withRSA	0.549	0.229	0.312
SHA384withRSA	2.042	0.227	0.610
SHA521withRSA	2.025	0.232	0.634

Secondly, we present the components in the latency of RSA and ECDSA using different key sizes. In this case, the total latency was partitioned in order to obtain the signature and verification times for each algorithm. These components can be observed in the following Table 5.4. We also present the KSP pre-operations and Bluetooth RTT components to observe how these components change by varying the public-key algorithm and key sizes used.

Observing the results we can see clearly that ECDSA has always better signature time than RSA for all key sizes and the verification time tends to be similar to RSA with lower key sizes, but slightly higher with larger key sizes. However, ECDSA has very large KSP pre-operations times rather than RSA, where it can be 3-times slower than RSA. These KSP operations before the signature process delay the ECDSA signature generation in such way that RSA can perform better than ECDSA in terms of overall latency. Although, the results presented enhance the efficiency improvement of ECC over RSA for the same level of security regarding signature generation and verification times. The Bluetooth RTT continues to be constant once again.

Thirdly, we present the components in the latency of different signature schemes. Now, we partitioned the total latency in the same way as the previous experiment, but with the intent of observing the KSP pre-operations, signature and verification times for different signature schemes. These components in the latency observed are presented in the following Table 5.5.

As we can see, and as expected, the signature time of RSA is much higher than signature time of ECC. However, the KSP pre-operations invoked for ECDSA take much more time to execute than for RSA with a much higher difference than in signature time. Therefore, RSA has better performance than ECDSA for each document size. In the other hand, DSA has fast signature and verification times, although has worst performance due to the access to the resource file and has an high KSP pre-operations time such as ECDSA. RSA-PSS has the same problem of accessing a resource file, but has low KSP

Table 5.4: Average Latency for RSA and ECC Digital Signatures using Different Key Sizes

	Total latency (s)			
Signature Algorithm	**KSP pre-Ops**	**Bluetooth RTT**	**Sign**	**Verify**
SHA256withRSA (1024 keys)	0.224	0.304	0.044	0.001
SHA256withRSA (3072 keys)	0.231	0.321	0.109	0.002
SHA256withRSA (4096 keys)	0.230	0.332	0.166	0.002
SHA256withECDSA (256 keys)	0.595	0.324	0.036	0.002
SHA256withECDSA (384 keys)	0.604	0.333	0.055	0.003
SHA256withECDSA (521 keys)	0.656	0.309	0.067	0.006

pre-operations time such as RSA-PKCS#1.

Table 5.5: Average Latency with Different Digital Signatures Schemes.

	Total latency (s)			
Signature Algorithm	**KSP pre-Ops**	**Bluetooth**	**Sign**	**Verify**
SHA256witDSA (1024 keys)	0.753	0.332	0.001	0.002
SHA256withRSA (3072 keys)	0.231	0.321	0.109	0.002
SHA256withRSA/PSS (2048 keys)	0.293	0.359	0.0047	0.0015
SHA256withECDSA (256 keys)	0.595	0.324	0.036	0.002

5.5 Bluetooth Performance Evaluation

In this section, it is compared the impact of using Bluetooth Low Energy (BLE) technology and security enforcements, such as Transport Layer Security (TLS), for executing certificate and digital signatures requests.

5.5.1 Bluetooth Low Energy

Regarding BLE, the goal of this experiment is to understand how this technology could improve or impact our solution. It was performed a test benchmark with different digital signature algorithms (equally to third benchmark in Section 5.4) to compare the overall latency of the execution of digital signatures with BLE. This benchmark was conducted through the Signature Benchmark client with the same configuration described before,

varying only the signing certificate with the security equivalent key sizes.

BLE is the second implementation or protocol of Bluetooth technology mainly designed and modelled to accommodate Internet of Things (IoT) applications and low-battery devices. With this goal in mind, the BLE packet length (more precisely, the maximum length of the data to be transmitted) is smaller than the packet length used in Bluetooth Classic. The default and maximum length for the first one is 512 bytes and for the second is 1021 bytes [19]. Notice that, in the case of BLE, this maximum length are negotiated between the two Bluetooth devices moments before a connection and data exchange take place, so the data maximum length is dependent from what both devices support individually. Bluetooth Classic handles packet length in a different way, because the communication is based in streams and we can specify the maximum length through the API. With our test environment, the mobile phone and computer negotiate a maximum length of 516 bytes for BLE and we specified a maximum length of 1024 bytes for Bluetooth Classic in the prototype implementation. It is also important to notice that, if packets have a length higher than the maximum length, they are partitioned in chunks with the maximum length allowed and are transmitted sequentially.

Regarding the specification protocol, BLE technology uses different protocols compared with Bluetooth Classic, which are focused in the functionality and features of the IoT devices. The most important one is called GATT/ATT, which is, in fact a combination of two protocols that consist of providing attributes through a service hosted in a LE device, from which client LE devices can read current values or write new ones. Therefore, only attribute values are transmitted through Bluetooth, confirming the reduction of the packet size and a less power consumption for devices.

In fact, this BLE specification protocol does not correspond to our requirements and goals, because our solution implies transmitting data over Bluetooth with lengths higher than the maximum packet length of BLE. For example, in the case of certificate requests, the data exchanged between our prototype applications are JSON strings that can achieve lengths higher than 2000 bytes, depending on the number of certificates retrieved. Another reason is that using JSON constructions to transmit multiple informations (as in our solution for certificate and signature requests) in one time is against the goal of the ATT protocol which is to send one attribute value each time when a LE client device interacts with a LE server device.

Yet, we successfully employed the BLE technology in our solution and completed digital signature processes with success. An experiment with certificates requests was not done, because of the huge exponential packet length when retrieving certificates existing in the mobile phone, which rounds 4154 bytes for only 5 certificates (even represented as base 64 strings). This packet length exceeds to much the maximum packet length in a way that dividing it by chunks would imply only a significant delay in the latency. Is also important to notice that was only possible to perform digital signatures which length

combined, with other pieces of information (such as the timestamps from Android side) required to transmit in the JSON packet, were less than the maximum packet length (516 bits). For example, RSA-2048 produces 256-bit length signatures, ECDSA-256 produces 70 to 72 bit length signatures and DSA-1024 produces 40-bit length signatures. This signature length combined with the remaining status and timestamps information length cannot surpass the maximum packet length. Support for key lengths which produce signature lengths that cause the JSON packet length to be higher than the maximum packet length was not completed or finished due to some implementation issues on our prototype in the moment when ATT protocol generates the chunks of the data to be transmitted, although the protocol handles the packets fragmentation as stated in documentation.

Plus, the Signature Benchmark performed to measured the performance was done only with 10 benchmark runs for each document size, because the time to perform a single signature using BLE is to high. The Bluetooth sockets get stucked in disconnected mode for a long time delaying the whole signature process. Figure 5.10 shows the results obtained from this BLE experiment.

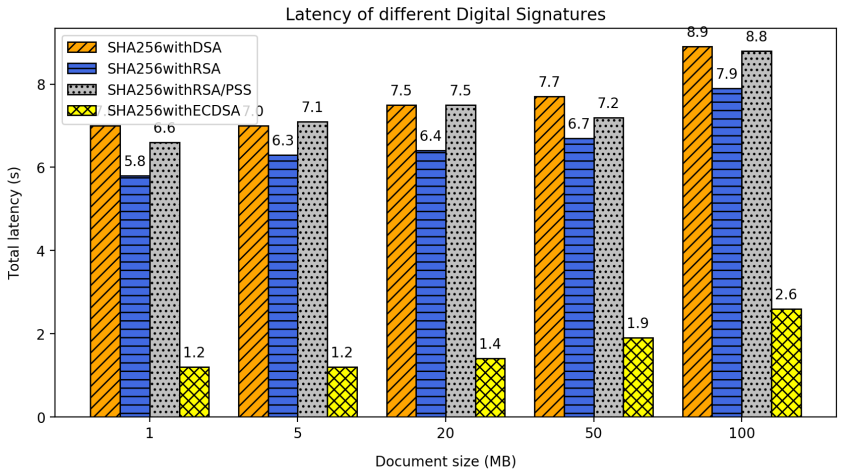


Figure 5.10: Average latency of Remote Digital Signatures using BLE.

The results suggest that the usage of BLE with our solution may lead to bad performance independently of the signature algorithm used, except for ECDSA, which maintains its performance throughout the size of the documents due to the low signature lengths produced. Although JSON constructions tend to be small and less than the maximum packet length, the LE specification protocol based in advertisements and in the ATT protocol imposes a significantly delay in the latency when executing digital signatures. One of the main causes of these high latency values is because LE protocol has a default time of 30 seconds to search for advertisements, but it is also possible to re-define a specific time. Since the default time is to elevated, we defined a specific discoverability time of 2 seconds in order to have time to discover our target device and Bluetooth service without increasing the latency to much in the process. This means that besides the Bluetooth

overhead we will also have another implicit overhead due to the Bluetooth discoverability time. Another reason for these results its because of the inefficient implementation of BLE support on our prototype, which has a workaround to successfully perform attribute write requests with responses. This is by default provided by the Windows 10 APIs through a specific function named *WriteWithResponse*, but without knowing why the function its not working and due to time restrictions we didn't investigate completely the origin of this issue. Therefore, we implemented a workaround that basically consisted in performing a Write request to demand the execution of a digital signature, and sequentially, a Read request to obtain the digital signature result. However, this involves the establishment of a connection for each one of the requests, turning the process very heavy and delaying the whole signature process.

5.5.2 TLS/Bluetooth Enforcement

In relation to the security enforcement, it is compared the performance results of the security enforcement implementation, more precisely, the TLS channel over the Bluetooth communication channel. The goal of this evaluation is to know the overhead caused by the security extra-layer on the process of digital signatures. Due to implementation and time limitations, it was only performed a test benchmark for the certificate requests and using only a default cipher suite for the TLS channel (i.e TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384). This test was conducted using the Setup Benchmark client with the same configuration explained in Section 5.2. The results of this experiment are represented in Figure 5.11.

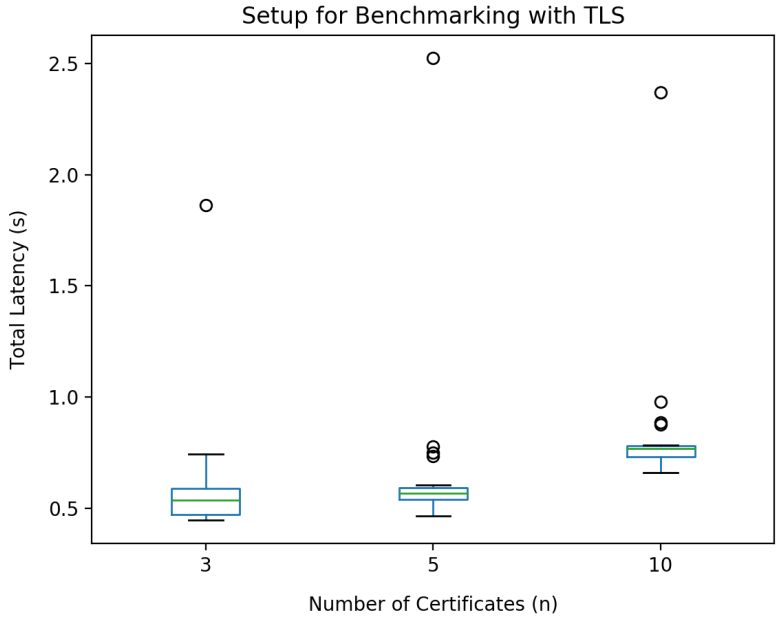


Figure 5.11: Latency measures using TLS Layering Enforcement.

Comparing with the results presented in Section 5.2, using an extra-layer of security as the TLS channel slightly increases the overall latency by 1.5 to 2 seconds in average for each different number of certificates. The results also show that latency continues to be proportional with the number of certificates, for example, responses containing 10 certificates have an higher dispersion of values and median than responses with less certificates and it will increase further if the number of certificates doubles or triples.

Notice that, the main objective of this security enforcement is to conceal a secure and protected communication channel for Bluetooth-enabled devices with lower and insecure versions of Bluetooth that have privacy and protection demands regarding the data exchanged. Our results show that a security enforcement as the TLS implies an acceptable overhead in the Bluetooth communications and enhances the security provided by the solution. Although, in current Bluetooth versions (v4.0 to v5.0), the security and protection levels provided by Bluetooth security mechanisms are already considered very high, leading to security enforcement layers being considered as unnecessary overhead.

5.6 Other BluetoothKSP Evaluation Metrics

In this section, it is presented additional evaluation metrics such as prototype packaging, memory usage, and energy consumption, which help to enhance the validation of our mobile cryptographic provider.

5.6.1 Packaging Metrics

In relation to the packaging of our mobile crypto provider to be further distributed to computer users, we need to package each prototype component and its dependencies in its own package. For example, the *WindowsCertificateManager* was packaged as a executable file which can be executed in the user Windows computer to install or run the .NET application; the *BluetoothKSP* was also packaged as an executable file that executes an installer to register our Key Storage Provider (KSP) in a particular Windows computer; and *CertificateManager* was converted to an APK file (i.e, the file extension that represents an Android application) to be further installed in a particular smartphone.

These packagings are generally easy to be generate, except in the case of *BluetoothKSP*, where a complete installer, that executes all the KSP register operations, must be created. In the case of the Android application, the APK file is automatically generated by the package manager Gradle when building the application. The APK file of the *CertificateManager* application has a total size of 2,5 MB. The Windows .NET applications also generates the executable file automatically when building the application. The executable or packaging file of *WindowsCertificateManager* has a total size of 81 KB. For *BluetoothKSP*, the process to generate an executable setup file that will install the C++ library on other computers is a bit more difficulty since we need to create a new setup project (besides our *BluetoothKSP* project) to accomplish the BluetoothKSP deployment.

However, From the user perspective, any of these packagings distribution can be seen as a lightweight process since all the components are not very big, except the DLL library.

5.6.2 Memory Resources and Utilization

In this section, we show other experimental observation on the mobile environment, in this case, the volatile memory (or RAM) used by the CertiticateManager when computing signatures or running the Bluetooth service. The objective of this experiment is to evaluate the memory usage of different digital signature schemes with our solution and which is the impact compared with the normal execution of the application, i.e, running the Bluetooth service. Similarly to the previous benchmark, we are also concerned with the memory used by our solution to complete signature processes in mobile devices due to their small volatile storage space. As such, we conducted a memory benchmark evaluation to measure the real memory cost of executing signature generation in mobile devices with our CertiticateManager.

This memory benchmark consisted in executing the Signature Benchmark client with the same test conditions described in the previous section. For measuring the CertiticateManager memory usage we used the available "Running Services" tool under the "Developer Options" in the smartphone settings. Notice that, this experiment discards the data stored on disk or by the Android Keystore in the SE - such as the private key.

Before measuring the memory usage, we measured first the memory cost when starting the *CertificateManager* application and without receiving any signature requests (just listening the Bluetooth Socket). The average used memory obtained is 41 MB, which corresponds to 0.010% of the total volatile memory (4GB).

Table 5.6: Average Memory Usage of the Supported Digital Signatures with Different Security Levels.

	Memory Usage (MB)		
Signature Algorithm	**128 bits**	**192 bits**	**256 bits**
SHA256withRSA	45	-	-
SHA256withDSA	49	-	-
SHA256withECDSA	44	45	45

Table 5.6 presents results obtained of the memory usage of the supported digital signature schemes. Note that, the values not appearing in the table are due to the fact that the respective algorithm key size is not supported by the Android OS. As the results suggest, when the signature operations start to be executed in sequence, the average memory used slightly increases independently of the signature algorithm. Our results

are very coherent with results from other studies [6, 71] in the sense that ECDSA requires less storage space than RSA and DSA in both ways: key length and signature length. As seen before, this is because, ECDSA has small keys and computes the signature faster, which leads to less memory usage. The only difference is that ECDSA with larger keys has equal memory usage as RSA.

With this results, we can conclude that our mobile crypto provider solution uses on average 0,011% of the total memory available to generate a signature, which does not seem to be a big deal comparing with other Android applications even if it is running as a service in the background listening the Bluetooth Sockets.

5.6.3 Power Consumption Observation

In this section, we present the experimental observations done on the mobile environment in relation to the energy consumed by the *CertificateManager* when executing signature operations and listening for connection requests through the Bluetooth service. The goal of this experiment is to evaluate the power consumption of different digital signature schemes with our solution and which is the impact compared to the smartphone common usage - such as interacting with and navigating through Android applications. This concern is extremely important due to the limited energy resources that wireless devices have (for example, the mobile phone battery) which restricts their power use. Although, in current days, wireless communication technology provides protocols and resources that lead to low-power and inexpensive Distributed Sensor Networks (DSNs) [71, 80]. Also, there are standard digital signatures considered as efficient and assumed as less power consuming for these environments such as ECDSA [6, 71]. Thus, the energy impact of the signature operation in the mobile phones is expected to be low.

5.6.3.1 Power Consumption of BluetoothKSP Digital Signatures

Then, we conducted an energy benchmark evaluation to know the real cost of executing digital signatures in mobile devices by our *CertificateManager*. This energy benchmark consisted in executing the Signature Benchmark client with 20 runs of signature requests per document size to generate the average consumption for each digital signature algorithm using SHA-256 and different key sizes corresponding to the security levels: 128, 192, and 256 bits. For measuring the mobile phone energy consumption we used a third-party mobile application called AccuBattery [3], which calculates the energy consumption of the mobile device and the individual consumption of each Android application in milliampere (mA) units. The mobile phone used for the benchmark is equipped with a Li-ion battery [66] that has a maximum capacity of 3000 mA per hour.

Before we analyze the impact of the supported digital signature schemes, let us first highlight the measured power consumption when not having any user application running in first or second plan (i.e, in idle mode); and for a common pattern of use. The

[3] https://www.accubatteryapp.com/

average measurements obtained are 139,7 mA and 362,2 mA, respectively. Additionally, we measured only the power consumption of the *CertificateManager* application listening for Bluetooth connection requests, which achieves a consumption of 150,9 mA. Note that, all these metrics were measured in time intervals of 5 minutes.

Table 5.7: Average Power Consumption of the Supported Digital Signatures with Different Security Levels.

	Power Consumption (mA)		
Signature Algorithm	**128 bits**	**192 bits**	**256 bits**
SHA256withRSA	209,5	-	-
SHA256withDSA	310,5	-	-
SHA256withECDSA	182,2	182,7	184

Table 5.7 shows the power consumption of the supported digital signature schemes. Considering the initial energy costs measured, we can conclude in general that the impact of our solution is small and that it can perform better than other applications. This is due to the fact that our CertiticateManager is an Android application that only listens the smartphone Bluetooth socket for packets directed to the Bluetooth service that it runs; executes a cryptographic operation or not when a request is received; and sends back a response packet through the socket.

Considering that the signature benchmark was executed 20 times for each one of the five sample documents, we can say that the average value presented in Table 5.7 is the average of 100 digital signatures over documents with variable length. In the case of RSA, we have an average of 209,5 mA per signature. Also, to drain completely the smartphone battery it will be necessary the execution of more than 10.0000 RSA-based digital signatures.

5.6.3.2 Comparative Analysis

Energy costs of signature generation depends mostly on the security level (i.e, key size) and the operation execution time, therefore, the energy consumption is proportional to these two variables. Our results show that RSA and DSA signatures are energetically more expensive due to the high computation cost of the private key operation, being DSA the less energy efficient; and ECDSA is the most energy efficient signature algorithm for all security levels. These results are consistent with previous literature experiments such as presented in [71].

Thus, we can conclude that our mobile crypto provider solution can impact the energy consumption on average by 30.42% regarding the idle state energy consumption, but

consumes much less than other applications running as they can impact the consumption by 159.27%.

5.7 Summary

In this chapter we presented and discussed our experimental results over the implemented prototype, in six different types of evaluation benchmarks. These experiments were conducted on both the *BluetoothKSP* (computer) and *CertificateManager* (mobile phone) components of the prototype and two testing benchmarks were implemented to execute them. A detailed analysis was performed for each benchmark section, which is briefly summarized next:

- The configuration or setup phase, i.e, the certificates request issued to the BluetoothRemoteKSP tends to produce a reasonable average latency for a variable number of certificates taking into account the human time perception, although is necessary to execute this configuration for each new computer where is further desired to perform remote digital signatures.

- The hash function SHA-256 has an efficient performance for all document sizes while SHA-384 and SHA-512 perform worse and worse as the document grows larger. RSA has a bad performance proportional to the key length and documents size, i.e, it performs worse as the key length increases and the document grows larger. The most secure (i.e, 128-bit security level) and efficient key length is 3072. In the other hand, ECDSA has similar performance with any key length for a specific document size, being ECDSA-521 a few times the fastest one. For the same security level as RSA, the most efficient ECDSA key length is 256.

- The signature average latency of our solution tends to be better when using RSA-3072 algorithm with the hash function SHA-256 for all documents size. ECDSA is the second with the best performance and with a low margin regarding RSA. RSA-PSS and DSA are the signature algorithms with worst performances due to the need of accessing the resource key stores, which increases latency to much. However, ECDSA is the algorithm with the best performance at 128-bit security level and that best fits our requirements and goals simultaneously.

- The ECC-based signatures normally perform better than RSA and also provides an equal security level with smaller keys, offering faster computations, less storage space and less demands on bandwidth, and is ideal for constrained environments such as our mobile phone. RSA has excellent signature verification times and ECDSA has great signature generation times. One of the few weaknesses of ECDSA is only the verification time.

- The performance of our solution is clearly worst than a Windows default provider or a a Token-based provider, but only with a difference of 1 second for each different document size. On the other hand, our solution offers flexibility and transparency to the final user and protection and security for their data, which helps disguise this performance flaw.

- The security enforcement, more precisely, the TLS can bring an maximum latency overhead of 2 seconds independently of the data size (for example, the number of certificates retrieved), which do not seems to be a great disadvantage regarding the security and protection that is obtained as trade-off.

- Given the Bluetooth Low Energy (BLE) specification protocol and our objectives - such as retrieving a variable list of certificates from the mobile phone and requesting signature operations - this technology is not worth for our use case and environment, since to take full advantage of BLE we needed to compromise the security of the solution by reducing the key sizes in order to achieve the required packet length and change our protocols in order to adapt them to the ATT protocol. Also, this data exchange can take longer than if it was done with Bluetooth Classic if the data size to be retrieved is very high. Thus, the use of BLE with our solution worsens the performance and delays the certificates retrieving and signature generation.

- The prototype software components (and their packaging) are not very heavy, although the final user will need to install two applications in different devices and one cryptographic provider.

- The *CertificateManager* application in the mobile environment has less power consumption than other Android applications and only has a small increase during signature processes compared with the consumption in idle mode. ECDSA and DSA are the digital signature algorithms with less and most energy consumption, respectively. The ECDSA consumption stays constant when varying the security bits.

- Regarding memory usage, our mobile application uses less memory comparing with other applications when is listening for certificate or signature requests via Bluetooth or even when its executing signatures generation. The difference between listening for requests and computing signatures corresponds to only 8MB in the worst case. ECDSA uses less memory such as RSA while DSA is the signature algorithm that more memory uses.

Chapter 6

Conclusions

In this chapter we present the main findings in the conclusion of the dissertation, regarding the investigation conducted, our system model assumptions and the instantiation in its prototype implementation. From these conclusions we discuss open issues and limitations, as well as, opportunities for future work directions in enhancements, optimizations and refinements.

6.1 Main Conclusions and Remarks

In this dissertation we addressed the design, prototyping and experimental evaluation of a mobile cryptographic provider for Android smartphones, used to manage, store and execute digital signatures for external devices, namely Microsoft Windows desktops. The proposed solution, although oriented towards support and testing in particular technological environments, followed the approach of a generic model that can later be generalized to other computing environments and other supporting technologies. Included in the dissertation objectives, we addressed the security guarantees provided for integrity, authentication and authorization properties, in accessing the functions offered by the Android-enabled mobile cryptographic provider.

In the Related work we survey the characteristics, properties, threats and protection related with the state-of-the-art solutions in approaching wireless communication channel and private keys management and storage security. The study oriented the choice of Bluetooth and smartphone's Secure Element (SE) to leverage the objectives of the dissertation. We studied the APIs and architecture of cryptographic providers of the chosen platform (i.e, Windows) to be considered as the base for our proposal, considering the most recent and secure Cryptography API: Next Generation (CNG), although not yet fully

implemented by some technologies. During the dissertation work, we also studied technical solutions, such as Bluetooth Low Energy (BLE) and Transport Layer Security (TLS), that increase the security of Bluetooth connections to support secure communication channels and involved cryptographic operations, between Windows-based applications and the smartphone mobile cryptographic provider, with insecure Bluetooth versions.

From the studied Related Work, we proposed a System Model and Architecture for the remote mobile cryptographic provider. As a corollary, we designed the cryptographic provider services that enable storage and management functions for X509v3 certificates and cryptographic keys, providing these services to Windows-based applications, in a transparent way. For Windows-based applications, the solution is regarded as a Windows-based local-installed crypto provider.

We provided an implementation of the proposed solution. The implementation was addressed as a ready prototype that is usable as a pre-product in a funded initiative and partnership between the FCT/UNL and NOVA-LINCS Research Center and Multicert S.A. Using the provided prototype we conducted an experimental evaluation, following different assessment criteria, to validate the designed solution, targeting the following observations: (1) validity testing on the transparency of the proposed solutions regarded by Windows-based applications; (2) performance evaluation of digital signatures provided by the designed Android mobile cryptographic provider, observing latency and operations' throughputs; (3) Performance analysis of the provided Bluetooth-enabled communication channel and impact of their security mechanisms, and (4) additional experimental analysis on the implemented mobile cryptographic provider solution to observe packaging metrics, runtime instrumentation and use of resources; as well as, power consumption evaluation.

Overall, the dissertation achieved the defined objectives, as well as the specific contributions beyond the planned goals. We addressed the problems and limitations of using a smartphone as a cryptographic provider and Bluetooth as the wireless environment to transmit sensitive data and created a model and proof-of-work of that same model. The implemented prototype is capable of providing itself as a cost-effective and flexible alternative to traditional USB tokens or dongles and smart cards, handling independent end-user scrutiny and better control of their data, with the additional security enforcements together with Bluetooth security mechanisms, thereby preventing tampering and corruption by a malicious entity.

6.2 Future Work

We were able to implement a functional prototype of our system model, although there are some features that were not totally implemented according to relevant theoretical design assumptions, and some related issues were left opened, which could be improved upon in the future. Besides the effort in a more extensive evaluation and benchmarking

using different and heterogeneous devices from different origins or manufacturers, we emphasize the following future work initiatives:

- **Overlaying enforcement of TLS/Bluetooth communication.** Related to our BluetoothKSP component, it was left to implement the Transport Layer Security (TLS) support that would allow in enhancing the security of the communication channel between the Certificate Manager and the BluetoothKSP components, while executing digital signatures. This can be integrated in the short term, as relevant extended overlaying security enforcement, particularly in the circumvention of possible fragilities of certain Bluetooth implementations or solutions based in old-versions. The implementation task for this feature started a bit late than initially expected and we found some technical issues in the chosen library (Botan[1]). For example, the integration of this library in the mobile crypto provider solution was causing runtime conflicts within software components causing crashes when a signature requests is performed. A future solution for the implementation problems would increase the reliability and consistency of the results obtained for the impact of TLS on our solution (more precisely, on the signature process). Furthermore, we can address in the future new Signature Benchmark evaluations with TLS, enriching the completeness and validation of the current prototype. As a direction the detected problems we must investigate the origin of the conflict or to take a new decision for a new open source C++ library for TLS to fulfill our requirements and allowing to us to manipulate TLS/Bluetooth data streams;

- **Refinement of standardization formats for Digital Signatures.** Regarding the implementation of the Signature Benchmark client, our solution invokes on-demand requested digital signatures creating on-the-fly hashes for the sample documents. Our formats are not using the formal standard digital signature formats and procedures, such as those formats formalized for the Digital Signature Standard (DSS) construction [61]. Due to the lack of support from .NET documentation and specific runtime support to provide a consistent API for DSS, as the formal standard addresses, we were unable to implement the related formalization that could help us better understanding the latency values obtained from the signature benchmarks and the investigation of impact factors. Also, this feature would help to elevate the Signature Benchmark resemblance to any real-world application, which expects to use the DSS standard. Anyway, for the initial purposes the refinement was not considered as priority, because what really mattered in the latency and related throughput observations was the signature processing time (with the dominant exponentiation time of the DSS signature) and the Bluetooth Round-Trip-Time (RTT), and not the format processing of final signature objects. A suggestion to address this as future work could be the reimplementation of the Signature Benchmark Client

[1] https://botan.randombit.net/

using the Java programming language, where exists reasonable documentation and various open-source code samples for creating digital signatures using the DSS construction. Unfortunately, Java (more precisely, the SunMSCAPI provider) does not yet have full support for Cryptography API: Next Generation (CNG) and a future Signature Benchmark in Java would not be able to execute signature requests [2].

- **Remote attestation facilities.** One security feature that was not addressed timely in the dissertation was the Remote Attestation protocol, which would consist in validating the Bluetooth crypto provider source code installed and loaded in the Windows OS library and attesting the remote mobile crypto-provider. The main objective of this protocol is to mitigate attacks against the execution after the possible injection of tampered data to the user mobile phone in order to obtain a valid signature, recognizing that the attestation proofs are not correct, stopping malicious KSPs from requesting forged signatures by the tampered remote crypto-provider components. This protocol is relevant as a security enforcement of our present solution and could enhance the protection of the implemented mobile crypto provider. In this future direction a possible initial approach could consist in modeling the Remote Attestation protocol, validating its specification and integration in the current solution, and finally conclude the related implementation.

- **Bluetooth-enabled pairing enforcements.** Our implementation components depend on the secure establishment of a Bluetooth connection. As stated before, except the Pairing method, all other Bluetooth security mechanisms are usually defined automatically, during the initial negotiation of devices configurations and cannot be changed programmatically. These security mechanisms - such as the pairing algorithm, the authentication algorithm and the encryption algorithm - are all negotiated depending on the Bluetooth versions of the devices. During the Pairing first-phase, both devices exchange pairing information (for example, capacities and requirements, as specified in the Bluetooth device configuration). This done via pairing request and response packets. In each device configuration, the Bluetooth protocol computes the final security parametrizations for further key generation, key establishment and connection processes. In the Pairing-first-phase, it is possible to intercept pairing packets through the Bluetooth service and force the pairing configuration desired for further connections. The initial idea to deal with the possible programmatic enforcement of parametrizations was considered, bit were not finished due to time restrictions. A task in the future work direction would be the investigation and validation of Pairing methods that can be actually proposed and the supported ones in different versions of Bluetooth implementations, to create a smart and secure dynamic negotiation handshake, addressing security default

[2] https://bugs.openjdk.java.net/browse/JDK-8026953?subTaskView=unresolved

settings and establishing the acceptable or the enforced levels of security, whenever possible and supported.

- **Refinements in transparent use of DSA signatures for Microsoft Word and Outlook applications.** Due to time restrictions, we were not able to implement completely the *KSPEnumAlgorithms* and *KSPExportKey* functions which are required for the digital signature processes of Microsoft Word and Outlook applications (as well as other Microsoft Office Applications). Unlike Adobe Reader. This leaves our mobile crypto provider a little bit limited in the transparency support, an issue that will be more relevant in offering the solution as a future product in the Multicert S.A portfolio. Even that this is more a concern for a final product perspective, it makes sense to address a future work task in finalizing the implementation of those two Key Storage Provider (KSP) functions, and finally validate the full-fledged transparency support by performing digital signatures with ECDSA and DSA algorithms requested by Microsoft Office applications.

- **New digital signature schemes.** There is also space for future research on the improvement of the signature schemes supported by our solution. For example, it is interesting to further investigate and employ the support for signature schemes based on the elliptic curve named Curve25519 [16], which is an elliptic curve designed for use with Elliptic Curve Diffie-Hellman (ECDH) key agreement scheme, but is also used in a Edwards-coordinate signature system named Ed25519 [44]. This is an elliptic curve that has gaining considerable interest in recent years. This support was added to OpenSSL v1.1 2 and also announced as part of the National Institute of Standards and Technology (NIST) Special Publication 800-186 3 and considered as the use of secure and efficient elliptic curves in the state-of-the-art, being an alternative to the used efficient P-256 elliptic curve. Actually, we began implementation and validation work to integrate this elliptic curve in our solution, but this is a on-going task that require a future work stream.

- **Other refinements and optimizations.** Also related with Bluetooth, but in this case with BLE technology, we successfully implemented the BLE support on our prototype and performed digital signatures with the supported algorithms via a BLE-based secure channel. Although, the current implementation has some flaws and issues that were left to investigate in more detail. More precisely we addressed initially a workaround to deal with Read/Write requests, to deal with the generation and establishment of two connections and the problem of JSON packet lengths that exceeds the BLE maximum packet length specification. A direction for future work would be to continue the investigation of the root cause of these issues and to correct and refine them in a new prototype, with expected performance gains.

Bibliography

[1] *256-bit Elliptic Curve Cryptography (ECC) - secp256r1 (alias NIST P-256 Algorithm))*, *OID 1.2.840.10045.3.1.7*. Global OID Reference Database, International Organization for Standardization (ISO). URL: http://oid-info.com/get/1.2.840.10045.3.1.7 (visited on 08/17/2019).

[2] Adobe. *Securing PDFs with certificates*. 2019. URL: https://helpx.adobe.com/uk/acrobat/using/securing-pdfs-certificates.html (visited on 08/17/2019).

[3] N. Akinyokun and V. Teague. "Security and Privacy Implications of NFC-enabled Contactless Payment Systems." In: *Proceedings of the 12th International Conference on Availability, Reliability and Security*. ARES '17. Reggio Calabria, Italy: ACM, 2017, 47:1–47:10. ISBN: 978-1-4503-5257-4. DOI: 10.1145/3098954.3103161. URL: http://doi.acm.org/10.1145/3098954.3103161 (visited on 09/20/2019).

[4] H. A. Al-Ofeishat and M. A. Al Rababah. "Near field communication (NFC)." In: *International Journal of Computer Science and Network Security (IJCSNS)* 12.2 (2012), p. 93.

[5] M. Al-Zarouni. "The reality of risks from consented use of USB devices." In: (2006). URL: https://ro.ecu.edu.au/cgi/viewcontent.cgi?referer=https://scholar.google.pt/&httpsredir=1&article=1061&context=ism (visited on 01/16/2019).

[6] M. Al-Zubaidie, Z. Zhang, and J. Zhang. "Efficient and Secure ECDSA Algorithm and its Applications: A Survey." In: *arXiv preprint arXiv:1902.10313* (2019).

[7] M. A. Albahar, O. Olawumi, K. Haataja, and P. Toivanen. "A Novel Method for Bluetooth Pairing using Steganography." In: *International Journal on Information Technologies & Security* 9.1 (2017), pp. 53–66.

[8] M Alimohammadi and A. Pouyan. "Performance analysis of cryptography methods for secure message exchanging in VANET." In: *International Journal of Scientific & Engineering Research* 5.2 (2014), p. 912.

[9] S. C. Alliance. "Host card emulation (hce) 101." In: *A Smart Card Alliance Mobile and NFC Council White Paper* (2014).

[10] F. Aloul, S. Zahidi, and W. El-Hajj. "Two factor authentication using mobile phones." In: *Computer Systems and Applications, 2009. AICCSA 2009. IEEE/ACS International Conference on*. IEEE. 2009, pp. 641–644.

[11] P. Anantharaman, K. Palani, D. Nicol, and S. W. Smith. "I Am Joe's Fridge: Scalable Identity in the Internet of Things." In: *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE. 2016, pp. 129–135.

[12] *ANSI X9.62 Elliptic Curve Digital Signature Algorithm (ECDSA) signatures and modules, OID 1.2.840.10045.4*. Global OID Reference Database, International Organization for Standardization (ISO). URL: http://oid-info.com/get/1.2.840.10045.4 (visited on 08/17/2019).

[13] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J.-K. Tsay. "Efficient padding oracle attacks on cryptographic hardware." In: *Advances in Cryptology–CRYPTO 2012*. Springer, 2012, pp. 608–625.

[14] E. Barker. "SP 800-57 Part 1 Rev. 4 Recommendation for Key Management, Part 1: General." In: *NIST special publication* 800 (2016), p. 57. DOI: 10.6028/NIST.SP.800-57pt1r4.

[15] C. Bermejo and P. Hui. "Steal Your Life Using 5 Cents: Hacking Android Smartphones with NFC Tags." In: *arXiv preprint arXiv:1705.02081* (2017).

[16] D. J. Bernstein. "Curve25519: new Diffie-Hellman speed records." In: *International Workshop on Public Key Cryptography*. Springer. 2006, pp. 207–228.

[17] A. S. Bhala, V. P. Kshirsagar, M. B. Nagori, and M. K. Deshmukh. "Performance comparison of elliptical curve and rsa digital signature on arm7." In: *Proceedings of International Conference on Information and Network Technology (IPCSIT), Singapore.(4),(2011)*. 2011, pp. 58–62.

[18] M. Blaser. *Securing ad hoc embedded wireless networks with public-key cryptography*. 2006. URL: https://www.edn.com/design/other/4025638/Securing-ad-hoc-embedded-wireless-networks-with-public-key-cryptography (visited on 08/17/2019).

[19] *Bluetooth Core Specification - v5.0*. Tech. rep. Bluetooth Special Interest Group, 2016. URL: https://www.mouser.it/pdfdocs/bluetooth-Core-v50.pdf (visited on 02/02/2019).

[20] Bluetooth SIG. *Bluetooth Market Update Report*. 2019.

[21] F. F. Brasser, S. Bugiel, A. Filyanov, A.-R. Sadeghi, and S. Schulz. "Softer Smartcards." In: *International Conference on Financial Cryptography and Data Security*. Springer. 2012, pp. 329–343.

[22] M. Collotta, G. Pau, T. Talty, and O. K. Tonguz. "Bluetooth 5: A concrete step forward toward the IoT." In: *IEEE Communications Magazine* 56.7 (2018), pp. 125–131.

[23] L. Constantin. *Stop using SHA1 encryption: It's now completely unsafe, Google proves*. 2017. URL: https://www.pcworld.com/article/3173791/stop-using-sha1-it-s-now-completely-unsafe.html (visited on 08/17/2019).

[24] T. Cooijmans, E. E. Poll, E. E. Verheul, and T. T. P. ter Gunne. "Secure key storage and secure computation in Android." In: *Master's thesis, Radboud University Nijmegen* (2014).

[25] T. Cooijmans, J. de Ruiter, and E. Poll. "Analysis of secure key storage solutions on android." In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. ACM. 2014, pp. 11–20.

[26] P. Cope, J. Campbell, and T. Hayajneh. "An investigation of Bluetooth security vulnerabilities." In: *Computing and Communication Workshop and Conference (CCWC), 2017 IEEE 7th Annual*. IEEE. 2017, pp. 1–7.

[27] V. Coskun, B. Ozdenizci, and K. Ok. "A survey on near field communication (NFC) technology." In: *Wireless personal communications* 71.3 (2013), pp. 2259–2294.

[28] M. L. Das. "Two-factor user authentication in wireless sensor networks." In: *IEEE transactions on wireless communications* 8.3 (2009), pp. 1086–1090.

[29] A. Developers. *Android 4.4 APIs*. Tech. rep. 2019. URL: https://developer.android.com/about/versions/android-4.4 (visited on 01/15/2019).

[30] W. Diffie and M. Hellman. "New directions in cryptography." In: *IEEE transactions on Information Theory* 22.6 (1976), pp. 644–654.

[31] *Digital Signature Algorithm (DSA) subject public key*. Global OID Reference Database, International Organization for Standardization (ISO). URL: http://oid-info.com/get/1.2.840.10040.4.1 (visited on 08/17/2019).

[32] D. M. Eagleman and A. O. Holcombe. "Causality and the perception of time." In: *Trends in cognitive sciences* 6.8 (2002), pp. 323–325.

[33] J.-E. Ekberg, K. Kostiainen, and N Asokan. "Trusted execution environments on mobile devices." In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 1497–1498.

[34] Federal Office for Information Security. *Protection Profile for the Security Module of a Smart Meter Gateway (Security Module PP)*. Version 1.03. 2014. URL: commoncriteriaportal.org/files/ppfiles/pp0077b_pdf.pdf (visited on 08/17/2019).

[35] M. Feldhofer and C. Rechberger. "A case against currently used hash functions in RFID protocols." In: *OTM Confederated International Conferences"On the Move to Meaningful Internet Systems"*. Springer. 2006, pp. 372–381.

[36] D. Filizzola, S. Fraser, and N. Samsonau. *Security Analysis of Bluetooth Technology*. 2018.

[37] N. FORUM. *NFC Data Exchange Format (NDEF), Technical Specification, Version 1.0*. Tech. rep. NFC Forum, 2006.

[38] C. Gomez, J. Oller, and J. Paradells. "Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology." In: *Sensors* 12.9 (2012), pp. 11734–11753.

[39] S. Gurjar, I. Baggili, F. Breitinger, and A. Fischer. "An Empirical Comparison of Widely Adopted Hash Functions in Digital Forensics: Does the Programming Language and Operating System Make a Difference?" In: *Proceedings of the Conference on Digital Forensics, Security and Law*. 2015, pp. 57–68. URL: https://commons.erau.edu/adfsl/2015/tuesday/6/.

[40] S. Hameed, U. M. Jamali, and A. Samad. "Protecting NFC data exchange against eavesdropping with encryption record type definition." In: *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*. IEEE. 2016, pp. 577–583.

[41] R. Hunt. "PKI and digital certification infrastructure." In: *Networks, 2001. Proceedings. Ninth IEEE International Conference on*. IEEE. 2001, pp. 234–239.

[46] Ieee. *IEEE Standard Specifications for Public-Key Cryptography*. 2000. DOI: 10.1109/IEEESTD.2000.92292. URL: https://standards.ieee.org/standard/1363-2000.html.

[42] A. A. Imem. "Comparison and evaluation of digital signature schemes employed in NDN network." In: *arXiv preprint arXiv:1508.00184* (2015).

[43] E. International. *On 17 June 2015, the 109th Ecma General Assembly approved new editions of NFC Standards*. 2015.

[44] S. Josefsson and I. Liusvaara. "Edwards-curve digital signature algorithm (EdDSA)." In: *Internet Research Task Force, Crypto Forum Research Group, RFC*. Vol. 8032. 2017.

[45] A. Juels. "RFID security and privacy: A research survey." In: *IEEE journal on selected areas in communications* 24.2 (2006), pp. 381–394.

[47] M. Knežević, V. Nikov, and P. Rombouts. "Low-latency ECDSA signature verification—a road toward safer traffic." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.11 (2016), pp. 3257–3267.

[48] B. Krebs. "Citibank Phish Spoofs 2-Factor Authentication." In: *The Washington Post* (July 10, 2006). URL: http://voices.washingtonpost.com/securityfix/2006/07/citibank_phish_spoofs_2factor_1.html (visited on 12/22/2018).

[49] Z. Lina. "Design and Implementation of KSP on the Next Generation Cryptography API." In: *Physics Procedia* 33 (2012), pp. 1640–1646.

[50] J.-C. Liou and S. Bhashyam. "On improving feasibility and security measures of online authentication." In: *Int. J. Adv. Comp. Techn.* 2.4 (2010), pp. 6–16.

[51] A. Lonzetta, P. Cope, J. Campbell, B. Mohd, and T. Hayajneh. "Security Vulnerabilities in Bluetooth Technology as Used in IoT." In: *Journal of Sensor and Actuator Networks* 7.3 (2018), p. 28.

[52] A. Maetouq, S. M. Daud, N. A. Ahmad, N. Maarop, N. N. A. Sjarif, and H. Abas. "Comparison of Hash Function Algorithms Against Attacks: A Review." In: *INTERNATIONAL JOURNAL OF ADVANCED COMPUTER SCIENCE AND APPLICATIONS* 9.8 (2018), pp. 98–103.

[53] Microsoft. *Cryptography API: Next Generation*. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/seccng/cng-portal (visited on 02/15/2019).

[54] Microsoft. *Understanding Cryptographic Providers*. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/seccertenroll/understanding-cryptographic-providers (visited on 02/15/2019).

[55] Microsoft. *Add or remove a digital signature in Office files*. 2019. URL: https://support.office.com/en-us/article/add-or-remove-a-digital-signature-in-office-files-70d26dc9-be10-46f1-8efa-719c8b3f1a2d#__toc311526848 (visited on 08/17/2019).

[56] Microsoft. *Secure messages by using a digital signature*. 2019. URL: https://support.office.com/en-us/article/secure-messages-by-using-a-digital-signature-549ca2f1-a68f-4366-85fa-b3f4b5856fc6 (visited on 08/17/2019).

[57] Z Mngomezulu, S Rimer, K Ouahada, and A. Ndjiongue. "A review of Bluetooth and NFC for financial applications." In: *Sixth International Conference on Advances in Computing, Control and Networking - ACCN 2017* (2017), pp. 48–51. DOI: 10.15224/978-1-63248-117-7-11. URL: https://www.seekdl.org/conferences/paper/details/8674.

[58] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. Nov. 2016. DOI: 10.17487/RFC8017. URL: https://rfc-editor.org/rfc/rfc8017.txt (visited on 08/17/2019).

[59] T. Muller. "Bluetooth security architecture." In: *White Paper Version* 1 (1999).

[60] T. R. Mutchukota, S. K. Panigrahy, and S. K. Jena. "Man-in-the-middle attack and its countermeasure in bluetooth secure simple pairing." In: *Computer Networks and Intelligent Computing*. Springer, 2011, pp. 367–376.

[61] "National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 186: Digital Signature Standard (DSS)." In: *Information Technology Laboratory, NIST, Gaithersburg, MD* (1994).

[62] National Security Agency (NSA). *The Case for Elliptic Curve Cryptography*. 2009. URL: http://www.nsa.gov/business/programs/elliptic_curve.shtml (visited on 08/17/2019).

[63] National Security Agency (NSA). *Suite B Cryptography*. 2014. URL: https://www.nsa.gov/ia/programs/suiteb_cryptography/ (visited on 08/17/2019).

[64] *Near Field Communication - White Paper*. Ecma/TC32-TG19/2005/012. Ecma International. 2005.

[65] *NFC-SEC, White Paper*. Ecma/TC47/2008/089. Ecma International. 2008.

[66] N. Nitta, F. Wu, J. T. Lee, and G. Yushin. "Li-ion battery materials: present and future." In: *Materials today* 18.5 (2015), pp. 252–264.

[67] J. Padgette, K. Scarfone, and L. Chen. *NIST Special Publication 800-121 Revision 2, Guide to Bluetooth Security*. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-121r2.pdf.

[68] S. K. Panigrahy, S. K. Jena, and A. K. Turuk. "Security in Bluetooth, RFID and wireless sensor networks." In: *Proceedings of the 2011 International Conference on Communication, Computing & Security*. ACM. 2011, pp. 628–633.

[69] *Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA)*. ANSI X9.62. Nov. 2005.

[70] A. Rahul, G. Krishnan G, U. Krishnan H, and S. Rao. "Near Field Communication (NFC) Technology: A Survey." In: *International Journal on Cybernetics & Informatics* 4 (Apr. 2015), pp. 133–144. DOI: 10.5121/ijci.2015.4213.

[71] H. Rifa-Pous and J. Herrera-Joancomartí. "Computational and energy costs of cryptographic algorithms on handheld devices." In: *Future internet* 3.1 (2011), pp. 31–48.

[72] R. L. Rivest, A. Shamir, and L. Adleman. "A method for obtaining digital natures and public-key cryptosystems." In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[73] M. Riyazuddin. "NFC: A review of the technology, applications and security." In: *ABI research* (2011).

[74] M. Roland. "Applying recent secure element relay attack scenarios to the real world: Google Wallet Relay Attack." In: *arXiv preprint arXiv:1209.0875* (2012).

[75] M. Roland and J. Langer. "Digital signature records for the NFC data exchange format." In: *Near Field Communication (NFC), 2010 Second International Workshop on*. IEEE. 2010, pp. 71–76.

[76] M. Roland, J. Langer, and J. Scharinger. "Security vulnerabilities of the NDEF signature record type." In: *Near field communication (NFC), 2011 3rd International Workshop on*. IEEE. 2011, pp. 65–70.

[77] M. Roland, J. Langer, and J. Scharinger. "Practical attack scenarios on secure element-enabled mobile devices." In: *2012 4th International Workshop on Near Field Communication*. IEEE. 2012, pp. 19–24.

[78] *RSA Digital Signature Standard with PKCS #1 Scheme and SHA-based Hash Family, OID 1.2.840.11354.1.1.11 to 1.2.840.11354.1.1.13*. Global OID Reference Database, International Organization for Standardization (ISO). URL: http://oid-info.com/get/1.2.840.113549.1.1.11 (visited on 08/17/2019).

[79] *RSA Digital Signature Standard with Probabilistic Signature Scheme (RSASSA-PSS), OID 1.2.840.113549.1.1.10*. Global OID Reference Database, International Organization for Standardization (ISO). URL: http://oid-info.com/get/1.2.840.113549.1.1.10 (visited on 08/17/2019).

[80] S. Seys and B. Preneel. "Power consumption evaluation of efficient digital signature schemes for low power devices." In: *WiMob'2005), IEEE International Conference on Wireless And Mobile Computing, Networking And Communications, 2005.* Vol. 1. IEEE. 2005, pp. 79–86.

[81] SIMalliance. *Secure Authentication for Mobile Internet Services - Critical Considerations v1.1*. 2011.

[82] SIMalliance. *Secure Element Deployment & Host Card Emulation v1.0*. 2014. URL: https://simalliance.org/wp-content/uploads/2015/03/Secure-Element-Deployment-Host-Card-Emulation-v1.0.pdf (visited on 01/15/2019).

[83] N. Smart et al. "Algorithms, Key Size and Protocols Report (2018)." In: *ECRYPT—CSA, H2020-ICT-2014—Project* 645421 (2018).

[84] S. Srinivas, D. Balfanz, E. Tiffany, F. Alliance, and A. Czeskis. "Universal 2nd factor (U2F) overview." In: *FIDO Alliance Proposed Standard* (2015), pp. 1–5.

[85] W. Stallings. *Network Security Essentials: Applications and Standards*. 4th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2010. ISBN: 0136108059, 9780136108054.

[86] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. "The first collision for full SHA-1." In: *Annual International Cryptology Conference*. Springer. 2017, pp. 570–596.

[87] D.-Z. Sun, Y. Mu, and W. Susilo. "Man-in-the-middle Attacks on Secure Simple Pairing in Bluetooth Standard V5.0 and Its Countermeasure." In: *Personal Ubiquitous Comput.* 22.1 (Feb. 2018), pp. 55–67. ISSN: 1617-4909. DOI: 10.1007/s00779-017-1081-6. URL: https://doi.org/10.1007/s00779-017-1081-6 (visited on 09/20/2019).

[88] J.-Z. Sun, D. Howie, A. Koivisto, and J. Sauvola. "Design, implementation, and evaluation of Bluetooth security." In: *Wireless Lans And Home Networks: Connecting Offices and Homes*. World Scientific, 2001, pp. 121–130.

[89] J. M. Tjensvold. "Comparison of the IEEE 802.11, 802.15. 1, 802.15. 4 and 802.15. 6 wireless standards." In: *IEEE: September*. Vol. 18. 2007.

[90] A. Umar, K. Mayes, and K. Markantonakis. "Performance variation in host-based card emulation compared to a hardware security element." In: *Mobile and Secure Services (MOBISECSERV), 2015 First Conference on*. IEEE. 2015, pp. 1–6.

[91] Z. Wang. "Information Security Vulnerabilities of NFC Technology and Improvement Programs." In: *Proceedings of the 2018 International Conference on Information Science and System*. ICISS '18. Jeju, Republic of Korea: ACM, 2018, pp. 196–199. ISBN: 978-1-4503-6421-8. DOI: 10.1145/3209914.3226165. URL: http://doi.acm.org/10.1145/3209914.3226165 (visited on 09/20/2019).

[92] *Why Mobile is the Next Digital Identity*. 30129-2-0216. Entrust Datacard. Jan. 2016. URL: https://www.entrust.com/wp-content/uploads/2013/08/Mobile-Perception-vs-Reality_JAN16_WEB.pdf (visited on 12/22/2018).

[93] T. Willingham, C. Henderson, B. Kiel, M. S. Haque, and T. Atkison. "Testing Vulnerabilities in Bluetooth Low Energy." In: *Proceedings of the ACMSE 2018 Conference*. ACMSE '18. Richmond, Kentucky: ACM, 2018, 6:1–6:7. ISBN: 978-1-4503-5696-1. DOI: 10.1145/3190645.3190693. URL: http://doi.acm.org/10.1145/3190645.3190693 (visited on 09/20/2019).

[94] M. Woolley. *Bluetooth Technology Protecting Your Privacy*. Apr. 2015. URL: https://blog.bluetooth.com/bluetooth-technology-protecting-your-privacy (visited on 01/23/2019).