UiO : Department of Informatics
University of Oslo

An eMRTD inspection system on Android

Design, implementation and evaluation

Halvdan Hoem Grelland

Master's Thesis Spring 2016

An eMRTD inspection system on Android

Halvdan Hoem Grelland

2nd May 2016

Abstract

All modern passports, ID cards from all around the world and a wide range of other identification and travel documents contain a contactless integrated circuit (IC).

The IC contains the data printed on the document as well as biometrics of the holder (face image, fingerprints) and is protected by multiple cryptographic mechanisms. The collective term used for such documents is *electronic Machine Readable Travel Document* (eMRTD).

This Master's project is given by the Norwegian National Directorate of Police (POD). They wish to demonstrate and evaluate the use of an off-the-shelf Android smartphone as a mobile Inspection System (IS) for eMRTDs. The envisioned system is an app which uses the phone's NFC interface to read and verify the eMRTD IC contents.

In answer to this a prototype Android app has been researched, designed, implemented and evaluated.

The app, "MRTD Inspector", uses the phone's camera and optical character recognition (OCR) to read access keys from data printed on the eMRTD, and performs the subsequent contactless inspection over NFC, as specified in the relevant standards.

Observations and experiments have shown that the solution performs well under certain conditions, but leaves room for improvement in others. Data transfer rate is determined to be a major bottleneck, and sensitivity to movement and positioning of the device is also identified as being a challenge.

It is therefore concluded that current handsets are not quite suitable for use as an eMRTD inspection system. However, current developments in NFC technologies holds great promise for future applications.

Acknowledgements

I would like to thank my supervisor Professor Audun Jøsang for his valuable guidance and contagious enthusiasm. Our discussions, both on and off topic, have been of great help and are much appreciated. He also deserves credit for his work in orchestrating supervision meetings with everyone involved.

A big thank you goes out to everyone involved at POD, Dr. Tage Stabell-Kulø, Jon Ølnes, and Øivind Næss, for sharing their deep domain knowledge and expertise. Our meetings and discussions have provided valuable insight into an otherwise quite inaccessible field. Also, thanks for giving me such an interesting and challenging project assignment.

Next, I'd like to thank my friends for their companionship through the hard times, and to my colleagues for constantly reminding me to just finish the damn thing. A special thanks goes to Mark Polak for providing much needed input when I was at my most bewildered.

Furthermore, I would like to thank my family for their unconditional support and encouragement throughout these years.

Last, but certainly not least, a special thanks is given to Nadia for all her love, understanding, encouragement and boundless patience.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

Over the past few decades, globalization has given rise to a need for strong and efficient, internationally interoperable identity verification and management.

The first steps towards this was taken in the early 1980s with the introduction of the *Machine Readable Passport*, the first release of *Document 9303* by the *International Civil Aviation Organization* (ICAO). Doc 9303 has since evolved to become the de facto standard for travel documents, collectively referred to as *Machine Readable Travel Documents* (MRTDs).

The mid-2000's advent of the *electronic* MRTD (eMRTD, ePassport) marks the next generational leap for the MRTD standard with the introduction of a contactless integrated circuit (IC) embedded into the data page of the MRTD.

The IC contains a digital copy of the information printed on the document as well as one or more biometrics of the holder (portrait image, fingerprints), all protected with a suite of specialised security protocols.

As a result eMRTDs are cryptographically verifiable and, coupled with biometric authentication of the holder, offer a much higher and more efficient assurance of authenticity and originality than the non-electronic MRTDs, which rely on physical security features alone.

Virtually all passports issued are now ePassports. In fact, they are issued by over 100 States[1] and economies [1] around the world. Also, in many States the same or adjacent technologies and standards are being integrated into a wide range of identity and travel documents such as ID cards and residence permits.

1.1 Motivation

With the sophistication of eMRTDs also comes the complexity of the technology. In order to support the issuance and verification of eMRTDs States have to implement and manage specialized PKIs, and inspection

[1] The capitalized S in *State* signifies an independent State as opposed to part of a country (i.e. "the state of Illinois"). This convention is used throughout this Master's thesis.

of documents is done with purpose-built systems employed at border crossings and ID checkpoints.

These *Inspection Systems* (IS) must efficiently, reliably and securely read and authenticate eMRTDs issued by a large number of States, implementing the standards to varying degrees of compliance and with differing features. Additionally, leveraging the biometric data dictates the need for the IS to perform biometric authentication of the holder, including capturing biometric samples and performing comparisons.

Due to these requirements traditional ISs, deployed for border control, are often bulky, stationary and costly devices (example given in Figure 1.1a).

(a) ABC at Oslo Airport, from [2] with permission.

(b) Mobile ePassport verification terminal, from [3].

Figure 1.1: Examples of ePassport inspection systems: 1.1a shows the Automated Border Control system at Oslo Airport Gardermoen. 1.1b shows the *Coesys Mobile eVerification Terminal* by Gemalto.

For border control in an international airport, where travellers are funnelled through a permanent checkpoint, these stationary machines are up to the task[2].

There are, however, a range of scenarios where one might want to inspect eMRTDs in remote locations, away from a power outlet and without the ability to carry heavy equipment. This could be a make-shift immigration checkpoint on a ferry dock, a police officer performing on-the-spot verification of an identity or even a (civilian or governmental) Internet service utilizing ePassports for user authentication. Clearly, there are many potential applications of a mobile solution for ePassport inspection.

For this reason various portable IS solutions are currently available. However, they are commonly expensive, proprietary and do not necessarily meet the expected degree of mobility. That is, though mobile they still require the operator to bring along a dedicated and fairly large piece of equipment (example given in 1.1b). This makes many of the existing mobile systems impractical for certain scenarios, such as the aforementioned on-the-spot verification of identity.

[2]In fact, bulkiness is an effective anti-theft measure.

Today's conventional mobile phones (smartphones) have become what is essentially affordable, ultra-portable and powerful computers. In addition they commonly contain a range of sensors and different means of connectivity, making them usable for a multitude of purposes beyond communication.

Most modern smartphones contain an interface for *Near Field Communication* (NFC). The NFC capability was initially intended for simple use cases but the potential for more advanced applications quickly became apparent as the capabilities of smartphones developed.

The banking and payment industries were among the first to take advantage of this with payment services based on NFC-capable phones such as *Apple Pay* and *Google Wallet*[3]. In fact, mobile payment is by many considered to be the *killer app* of NFC phones, and is one of the main catalysts for further advancement of NFC capabilities in consumer grade devices.

As NFC is fully compatible with the contactless interface in eMRTDs, an attractive prospective solution to the lack of truly mobile inspection systems presents itself: implement an IS as an NFC smartphone app.

The possible advantages are significant: truly mobile, connected, considerably less expensive and potentially easier to manage. Additionally, since most citizens own an NFC capable device the landscape for services based on electronic identity documents opens up.

The Norwegian National Directorate of Police (POD) recognizes the potential in a smartphone based Inspection System for eMRTDs.

As part of the *G3kko* project, which is a subordinate of the *IDeALT*[4] programme at POD they wish to gauge the practical feasibility of a smartphone based solution for eMRTD inspection.

1.2 Goal

As stated in the previous section, POD wishes to investigate an envisioned smartphone based eMRTD inspection system. In the initial project description given by POD (Appendix A) the concrete assignment is stated as follows:

> *"The Police wishes to test the use of a standard mobile phone as an inspection system for MRTDs. The assignment is, in short, to create an app for Android which shows that this is a functionally satisfactory solution."*
>
> *POD Master's project description*
> *(Appendix A, translated from Norwegian)*

We understand from this that the core of the assignment is to investigate the feasibility of such a system. That is: to build a prototype, identify the

[3] Now *Android Pay*.

[4] IDeALT is a POD programme established to aid the fight against identity fraud.

practical challenges and limitations and propose solutions.

The questions we wish to answer by doing so are:

- How can an eMRTD inspection system on Android be realized?
- What are the limitations of our solution? What are the strengths?
- Are there generic limitations for such a system?

The goal of this Master's thesis is to fulfil the assignment given by POD through answering these questions.

1.3 Approach

Our primary approach is to design and implement a prototype Android app for the inspection of ePassports. The design and implementation process comprises the following:

- Gain understanding of the field: researching existing solutions to this or adjacent problems, reviewing the relevant literature and the technologies we are working with.
- Limit the scope: identifying and selecting the key functionalities needed for our implementation and formulate our system requirements from these.
- Review and select the core technologies, hardware and software, which we wish to use in our implementation.
- Implement a minimum prototype and iteratively improve upon it with the goal of fulfilling our requirements.
- Document and describe findings, problems, solutions and design choices.

Throughout the process we aim to identify both the problems and benefits of our implementation with respect to our system requirements and, ultimately, our research questions.

Additionally we design and execute a small series of experiments in order to quantitatively gauge the performance of key parts of our implementation.

The final evaluation of our implementation is based on the sum of these parts. That is our observations and experiences from the development cycle as well as the indications given by our experiments.

1.4 Work done

Our work has consisted of researching, designing and implementing a prototype Android app which implements eMRTD inspection as defined in ICAO Doc 9303 [4] and BSI TR-03110 [5] for NFC handsets.

We have also designed and conducted a small series of experiments which give indication of performance and stability for key parts of the inspection procedure in our implementation.

Part II

Background

Chapter 2

MRTDs and ICAO Doc 9303

In this chapter we give an introduction to *Machine Readable Travel Documents* (MRTDs). We start off by introducing the standard for MRTDs, namely *Doc 9303* which is published by the *International Civil Aviation Organization* (ICAO). Continuing from this a brief survey of the MRTD standard is given, focusing on the parts which are most significant to this project.

Seeing that *Electronic MRTDs* (eMRTDs) is the main field of study for this Master's project, Chapter 3 has been dedicated to the subject.

2.1 ICAO and Doc 9303

The *International Civil Aviation Organization* (ICAO) is a United Nations specialized agency, established in 1944 to carry out and manage the *"Convention on International Civil Aviation"*, also known as the *"Chicago Convention"*.

One of the functions of ICAO is as a standards organization for many aspects of civil aviation, one of which is the standardization of MRTDs.

The ICAO standard for MRTDs is a twelve-part series of documents known as *Doc 9303* [4]. It specifies everything from physical construction to the specifics of technical implementation, manufacture and issuance of MRTDs.

The standard was first released in 1980 and became the initial basis for issuance of machine readable passports by Australia, Canada and the United States [6]. Today it is published in its seventh edition and is the internationally recognized standard for travel documents issued by States throughout the world.

Doc 9303 compliant electronic passports (*ePassports*), which is the de facto standard for passports, are issued by over 101 States and economies [1] and growing.

2.2 MRTDs

Machine Readable Travel Document (MRTD) is a collective term which refers to the range of travel documents covered by Doc 9303. This includes machine-readable passports (MRPs), visas (MRVs) and ID cards.

As stated in Doc 9303, an MRTD is principally characterized by the presence of two mandatory zones: the *Visual Inspection Zone* (VIZ), which is human-readable, and the *Machine Readable Zone* (MRZ), which is in machine-readable format and contains a summary of the data contained in the VIZ.

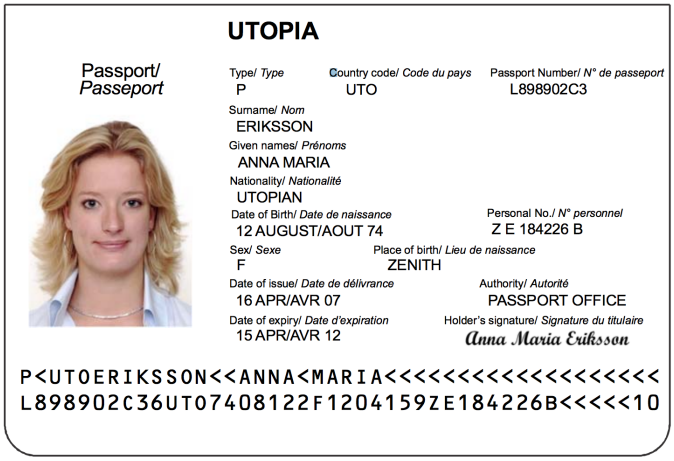


Figure 2.1: Example of MRTD, from [7]

The common specification for all MRTDs is given in part 3 of Document 9303 [7]. Though data contents, area of use and features vary between the different subsets of MRTDs, they all conform to and extend from this standard.

There are three physical sizes for MRTDs defined in Doc 9303: *TD1*, *TD2* and *TD3*. The *TD1* size correspond to a standard ID card ("smart card format"), whilst *TD3* dimensions correspond to what most will recognize as a standard passport.

Figure 2.1 shows an example of an MRTD data page containing a VIZ (top part, printed information) and MRZ (bottom). The example is a TD3 MRTD.

2.2.1 MRZ

The introduction of the MRZ was the primary enabler of machine-readability for the initial generation of MRTDs.

At the time, the MRZ was the sole mechanism for machine-readability, and it is still an intrinsic part of the standard.

Essentially, the MRZ is a printed, machine-readable representation (or summary) of the data printed on the data page (VIZ). Reading is done optically: capturing an image of the MRZ and using optical character recognition (OCR) to extract the data.

The data is then parsed by reading the data elements from fixed segments and applying a set of encoding rules, specified by Doc 9303. An example of the MRZ layout can be seen in Figure 2.2, which shows a break-down of the data elements encoded in a TD3 MRZ.

In order to aid machine-readability the MRZ itself is printed in the *OCR-B* type face, which is optimized for OCR [1]. Only the uppercase `A-Z` and `0-9` subset of characters are used, as well as the "<" filler character. This makes a total of 37 characters that the OCR system must recognize and differentiate.

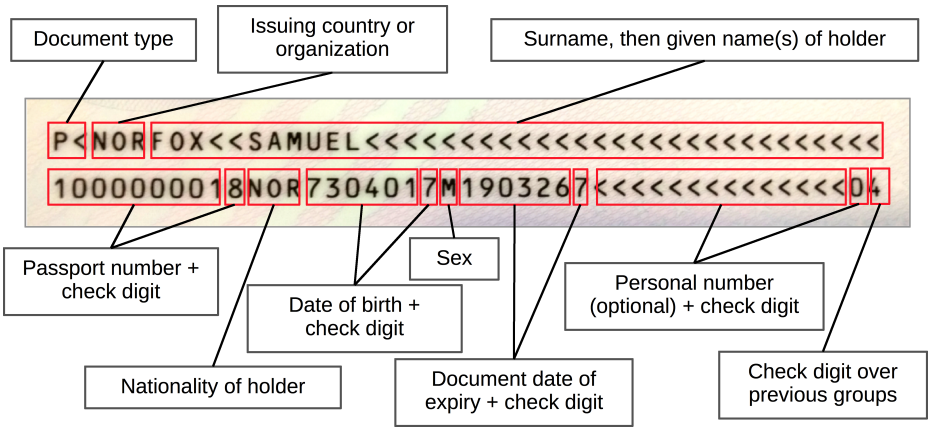


Figure 2.2: Break-down of a TD3 MRZ.

As can be seen from Figure 2.2, additional check digits are provided for the data elements on the lower line of the MRZ. These serve to increase the reliability of reading those data elements, making detection of misreads possible.

2.3 Machine Readable Passports

Machine Readable Passports (MRPs) are MRTDs in the TD3 booklet format (125 x 88 mm). In addition to fulfilling the MRTD specification, the specifics of MRPs are defined in part 4 [8] of Document 9303, including the layout and contents of data elements, physical dimensions and so forth. In practice almost all passports issued today conform to the MRP specification.

[1] The OCR-B font is commonly used for a range of applications where both human- and machine-readability is desired, such as airline tickets, bills and cheques.

Chapter 3

Electronic MRTDs

Electronic MRTD, *eMRTD*, *ePassport* and *biometric passport*: These names all refer to the same thing and are often used interchangeably, even in the official ICAO standards. In essence, an ePassport is an MRP which has a contactless integrated circuit (IC) embedded in the data page.

The IC contains the printed data of the ePassport data page, one or more biometric measures (face image, fingerprint, iris scan) of the passport holder and a security object which employs public key cryptography to protect the contents against tampering, forgery, copying and unauthorized access. The IC gives assurance of the authenticity of the passport data page as well as enabling biometric authentication of the passport holder.

Electronic MRTDs are standardized in Doc 9303 parts 9 (deployment of biometrics) [9], 10 (IC storage) [10], 11 (security mechanisms) [11] and 12 (PKI) [12]. Additionally, EU passports implement an extension to these standards known as *Extended Access Control* (EAC). It is standardized by the German *Federal Office for Information Security* (BSI) in the technical report TR-03110 [5].

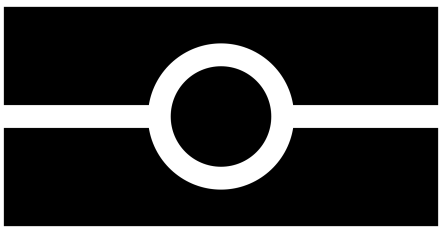


Figure 3.1: The ePassport logo (also known as the "Chip inside" symbol) is printed on all MRTDs which contain an ICAO 9303 compliant contactless IC.

In this chapter we cover key aspects of the aforementioned eMRTD standards, with special attention given to the data storage, the security protocols employed by the IC for data protection and access control and the PKI which those protocol rely upon.

The following sections assume that the reader has basic knowledge of contactless smart cards technologies. A review of these can be found in

Chapter 4.

3.1 Logical Data Structure

For the purpose of universal interoperability, all data is stored on the IC in a standardized data structure known as the *Logical Data Structure* (LDS), specified by part 10 of Doc 9303 [10].

The LDS is organized as a collection of data groups which in turn consist of standardized data elements. The standard in [10] specifies 16 data groups, named *DG1-16*, of which the two first are mandatory whilst the rest are optional. In addition there are two unnumbered mandatory elementary files: *EF.COM* and *EF.SO$_d$*.

Data stored in the LDS is encoded using BER-encoding (which is a format for ASN.1). Table 3.1 lists all data groups of the LDS.

Data group	Description	Required	Encoding
DG1	MRZ contents	✓	BER
DG2	Encoded face	✓	CBEFF
DG3	Encoded fingers	✗	CBEFF
DG4	Encoded irises	✗	CBEFF
DG5	Displayed portrait	✗	BER
DG6	Reserved for future use	✗	BER
DG7	Displayed signature or usual mark	✗	BER
DG8	Data features	✗	BER
DG9	Structure features	✗	BER
DG10	Substance features	✗	BER
DG11	Additional personal details	✗	BER
DG12	Additional document features	✗	BER
DG13	Optional details	✗	BER
DG14	Security options for secondary biometrics	✗	BER
DG15	Active Authentication public key info	✗	BER
DG16	Persons to notify	✗	BER
EF.COM	Common data for the LDS	✓	BER
EF.SO$_d$	Document security object	✓	BER

Table 3.1: The data groups of the LDS.

Following is a rundown of the most significant data groups and elementary files [1]. Those which have been left out are classified as *additional data* and

[1] Data groups are sometimes referred to as elementary files, which is imprecise.

most are not commonly used. The exception is data groups 14 and 15 which are required for certain security protocols (discussed in 3.3).

3.1.1 DG1 - MRZ contents

Data group 1 contains the MRZ in its entirety, encoded as a single data element.

3.1.2 EF.COM - Common information

EF.COM contains version information for the LDS and a list of all present data groups. Thus, it acts as an index for the LDS contents and is typically the first to be read in an inspection scenario.

3.1.3 EF.SO_d - Document Security Object

The Document Security Object contains hashes of all present data groups in the LDS. It is stored as a *SignedData* object as defined by [10], which contains a digital signature over the contained list of hashes alongside the Document Signer Certificate.

The purpose of EF.SO_d is thus to make the content of the LDS cryptographically verifiable by the reader, and of the security mechanisms for eMRTDs have a baseline reliance on it.

3.2 Biometric data groups

Data groups 2, 3 and 4 of the LDS contain the biometric features of the passport holder. DG2 is mandatory for all ePassports whilst DGs 3 and 4 are defined as "additional biometrics" and are optional.

3.2.1 Encoding

All of the biometric data groups are structured and encoded in accordance with the standard given in Doc 9303 part 10 [10].

A *Biometric Information Template* (BIT) group template is specified, which allows nesting BITs and storing multiple biometrics [2]. This format harmonizes with the *Common Biometric Exchange Formats Framework* (CBEFF), and the actual biometric payload is encoded in compliance with this.

3.2.2 DG2 - Face

Data group 2 contains the encoded face image of the passport holder. The image is photographed and cropped to meet the specifications of Doc 9303

[2] In fact nesting is always used, even for single biometrics.

part 9[3]. The format of the image is specified to ensure compatibility with facial recognition systems [9].

The "encoded face" is simply an image of the passport holder's face. Storing a facial recognition template instead is disallowed by the standard since these are vendor specific, making a facial recognition-optimized source image the only viable option to ensure interoperability.

3.2.3 DG3 - Fingerprints

Data group 3 contains the encoded fingerprint(s) of the passport holder. As the standard specifies optional multiple biometrics, more than one fingerprint can be encoded.

3.2.4 DG4 - Iris

Data group 4 contains the encoded iris image of the passport holder. The specification allows for one or two encoded irises.

3.3 Security protocols

As previously mentioned, eMRTD ICs employ a suite of dedicated security protocols. These protocols, seen in Table 3.2, protect various aspects of the IC in order to prevent unauthorized access and proving authenticity of the document.

According to Doc 9303 only *Passive Authentication* is the only mandatory protocol for all eMRTDs. EU passports, however, are mandated to support BAC/SAC and EAC (when using secondary biometrics). Note that the implementation of EAC is not specified in Doc 9303, but is outlined as a concrete security mechanism. Hence, the EAC protocols given in Table 3.2 are external to the Doc 9303 eMRTD standard but adhere to BSI TR-03110 [5].

Protocol	Protects
Basic Access Control	Confidentiality
Supplemental Access Control	Confidentiality
Passive Authentication	Authenticity
Active Authentication	Originality
Extended Access Control	
Chip Authentication	Originality + confidentiality
Terminal Authentication	Authenticity

Table 3.2: The eMRTD Security protocols. Note that Chip Authentication and Terminal Authentication make up the two components of the Extended Access Control protocol.

[3]The encoded face image is for all intents and purposes the same image which is displayed in the passport data page VIZ (specified in [7]).

For the following sections we establish some nomenclature:

1. **IS:** Inspection System - terminal
2. **IC:** Integrated Circuit - eMRTD

3.3.1 Passive Authentication

The goal of Passive Authentication is to prove the authenticity of the data contained in the LDS. Recall that the LDS contains a Document Security Object $\mathrm{SO_d}$ which holds the hash of all contained DGs as well as digital signature computed over this list of hashes.

In essence PA consists of the inspection system verifying the authenticity of these hashes, and thus the data groups. The IC itself does not actively participate; all processing in performed on the IS. Hence the name *Passive* Authentication.

For PA an inspection system (IS) carries out the following steps:

1. Read $\texttt{EF.SO}_\texttt{d}$ from the eMRTD.
2. Look up and retrieve the corresponding Document Signer Certificate C_{DS} from either $\texttt{EF.SO}_\texttt{d}$ itself or some other source.
3. Look up and retrieve the *Country Signing CA Certificate* C_{CSCA} and the Certificate Revocation List CRL.
4. Verify C_{DS} using C_{CSCA} and asserting it is not in the CRL.
5. Verify the signature of SO_d using C_{DS}.
6. Compute the hash values of each DG of the LDS and compare them to their counterpart in $\texttt{EF.SO}_\texttt{d}$.

The trust anchor of the verification process is the CSCA Certificate C_{CSCA}, which is assumed by PA to be genuine. Consequently, an inspection system implementing PA must make sure C_{CSCA} and CRL are retrieved from a trusted source (the underlying PKI is described in 3.4.1).

Passive Authentication enables the inspection system to certify the integrity of the data groups, but does not prevent cloning or physical substitution of the chip.

Since PA is required and verifies the authenticity the eMRTD against the PKI, other security protocols (described in the following sections) rely on PA as a source of trust.

3.3.2 Active Authentication

Active Authentication is a countermeasure for chip substitution[4] and copying attacks. It lets the IS verify the authenticity of the IC by means of a digital signature-based challenge-response protocol.

[4]Physically changing the chip of the eMRTD.

Support for AA is indicated by the presence of DG15, which contains the public key PK_{AA} needed by the IS to verify the response. The private key $Kpriv_{AA}$ is, of course, only accessible to the IC itself.
Assuming the IS has read the public key PK_{AA}, the steps of AA are as follows:

1. The IS generates a random nonce RND_{IS} and sends it to the IC.
2. The IC creates the message M[5], signs it using $Kpriv_{AA}$ and sends the signature σ to the IS.
3. The IS verifies σ using PK_{AA}.

By successfully passing AA the IC has proved that it knows $Kpriv_{AA}$. Since the integrity of DG15, which contains PK_{AA}, has been verified by PA it is therefore implicitly proven that the IC is genuine.

3.3.3 Basic Access Control

The purpose of Basic Access Control is to ensure that the IS has physical access to the eMRTD's data page, that is, to prevent reading the IC without the holder's knowledge. It does so by requiring the IS to authenticate using the *Document Basic Access Keys* which are derived from information readable in the MRZ. By demonstrating knowledge of the keys the IS proves to the IC that it has physically read the data page of the eMRTD. Once authenticated, session keys are generated and Secure Messaging is started between the IC and IS.

The Document Basic Access Keys KB_{Enc} and KB_{MAC} are stored in private memory of the IC, and have to be derived by the IS prior to initiating the BAC protocol. The key derivation process is as follows:

1. The IS obtains three fields of the MRZ: the Document Number, the Date of Birth and the Date of Expiry. Typically the IS will do so by optically reading the MRZ, but the data is readable from the VIZ as well, making manual reading and entry by an operator possible.
2. The key seed K is generated by concatenating the fields and applying the SHA-1 hash function.
3. The ICAO KDF as defined in [11] is used by the IS to derive the two 3DES keys KB_{Enc} and KB_{MAC} from K.

These are the steps of Basic Access Control:

1. The IC chooses the nonce RND_{IC} and key seed K_{IC} randomly. RND_{IC} is then sent to the IS.

[5]The composition of M depends on the signature algorithm used, but always contains RND_{IS}.

2. The IS chooses the nonce RND_{IS} and key seed K_{IS} randomly. It then sends the encrypted challenge e_{IS} to the IC, which is derived using an encrypt-then-MAC scheme over the plaintext $RND_{IS}||RND_{IC}||K_{IS}$ with the Document Basic Access Keys K_{Enc} and K_{Mac}.

3. The IC decrypts e_{IS} using K_{Enc} and K_{Mac} and verifies that RND'_{IC} extracted from the plaintext matches RND_{IC}.

4. Next, the IC responds with the encrypted challenge e_{IC} which is derived with the same scheme and keys as e_{IS}, but over the plaintext $RND_{IC}||RND_{IS}||K_{IC}$.

5. The IS decrypts e_{IC} using K_{Enc} and K_{Mac} and verifies that RND'_{IS} extracted from the plaintext matches RND_{IS}.

After successfully completing BAC all subsequent messages are protected by Secure Messaging. The SM session keys KS_{Enc} and KS_{MAC} are derived from the common master secret $K_{Master} := K_{IC} \oplus K_{IS}$. The Send Sequence Counter for SM is initialized from RND_{IC} and RND_{IS}.

3.3.3.1 Weaknesses

BAC has received much criticism for its low levels of protection.

As stated, its purpose is to protect against skimming attacks from parties without physical access to the document. However, the protection relies solely on the Document Basic Access Keys, which are derived from the MRZ and are ultimately used as keying material for the session keys.

The first two MRZ fields, the date of birth and the date of expiry, are not randomly distributed but in fact quite limited in terms of possible values, and the document number often follows known conventions in structure, limiting the key space further.

The result of these factors is that the entropy of the derived keys are very low and opens the possibility for brute-force skimming attacks.

As a matter of fact, practical attacks in this vein have been successfully mounted against BAC [13], as well as a replay attack which enables tracing individual BAC eMRTDs [14].

3.3.4 Supplemental Access Control

"Supplemental Access Control"[15] (SAC) is the name of an ICAO Technical Report which defines access control mechanisms *supplementary* to BAC. It is based on the *Password Authenticated Connection Establishment* (PACE[6]) protocol, and is essentially a framework which fixes a set of parameters and options for PACE in eMRTDs.

In SAC, PACE has an analogous role to BAC: preventing reading of the IC without physical access and setting up an encrypted session. Thus, SAC is used in lieu of BAC, and is, in fact, devised partly to address the aforementioned BAC weaknesses.

[6]For the sake of simplicity we're referring to PACEv2 as PACE.

3.3.4.1 Password Authenticated Connection Establishment

PACE is password authentication protocol based on Diffie-Hellmann key agreement (DH). As stated in [15] it is a generic protocol which is not necessarily exclusive to eMRTDs. However, as our focus in indeed eMRTDs and SAC, we are describing the protocol within that context.

Similarly to BAC, the IS and IC share a static password π which is visually or optically read from the eMRTD data page. Knowledge of π therefore serves as proof of physical access. The shared password π is in turn used to derive the shared key set K_{pi}.

In [15] the following options for π are defined:

- **MRZ:** π is derived from the MRZ in a similar fashion to the Document Basic Access Keys in BAC, meaning it is based on the Document Number, Date of Birth and Date of Expiry fields.
- **CAN:** π is a *Card Access Number* (CAN). The CAN is printed in human-readable format on the data page of the eMRTD. It must be chosen randomly[7].

According to [15] only the MRZ option is required, whilst supporting a CAN is optional. The advantage of supporting a CAN is, of course, that it is much more convenient to deal with in the case of manual entry into the IS.

[15] defines three *mappings* for PACE:

- **Generic Mapping (GM):** Diffie-Hellman key exchange.
- **Integrated Mapping (IM):** direct mapping of a field element.
- **Chip Authentication Mapping(CAM):** an extension of GM which integrates PACE with Chip Authentication.

According to the SAC specification: If PACE-CAM is supported by the IC, at least one of PACE-IM and PACE-GM must also be supported. An IS is required to support PACE-IM and PACE-GM whilst support for PACE-CAM is optional.

In order to initiate PACE the IC must first choose parameters from those available for the eMRTD. The PACE parameters are made available in the elementary file `EF.CardAccess`. This EF is located directly under the Master File (outside the LDS) and is publicly readable.

SAC specifies a set of parameters which must be supported by inspection systems:

- Key agreement: Diffie-Hellman (DH) or Elliptic Curve Diffie-Hellman (ECDH).

[7] Using a cryptographically strong method of random number generation.

- Mapping: GM, IM or CAM.
- Symmetric cipher: 3DES or AES.
- Key length: 112 for 3DES. 128, 192, 256 for AES.
- SM scheme: CBC/CBC for 3DES, CBC/CMAC for AES.
- Auth. token: CBC for 3DES, CMAC for AES.

A suite of PACE parameter combinations are specified for SAC in [15, p. 215, Table 2]. An IS must support all of these[8], whilst an eMRTD must support at least one. The parameter suites are made available as a list of Object Identifiers (OIDs) in `EF.CardAccess`, and the IS is free to choose from these.

Once the IS has acquired the password π and has chosen a set of PACE parameters, the protocol can be initiated. Following are the (simplified) steps of PACE:

1. The IC chooses the random nonce RND_{IC} and derives the shared key K_π from π. RND_{IC} is encrypted using the chosen symmetric cipher with K_π and the ciphertext c_{RND} is sent to the IS.
2. The IS derives K_π and decrypts c_{RND}, recovering RND_{IC}.
3. Both the IC and IS do the following:
   a. Choose a random ephemeral key pair, $[SK_{DH,IC}, PK_{DH,IS}]$ and $[SK_{DH,IS}, PK_{DH,IS}]$ respectively.
   b. Exchange the data required for mapping the nonce. The data depends on the mapping:
      - **GM and CAM:** the IC and IS exchange the ephemeral public keys $PK_{DH,IC}$ and $PK_{DH,IS}$.
      - **IM:** The IS chooses an additional random nonce RND_{IS} and sends it to the IC.
   c. Use the mapping function as defined in [15, p.16, Section 3.4.2] to compute the ephemeral domain parameters D_{Map} from the exchanged parameters.
   d. Generate the shared secret K by performing the anonymous DH key agreement with D_{Map}.
   e. Derive session keys KS_{MAC} and KS_{Enc} using the specified key derivation functions.
   f. Compute the authentication tokens $T_{IS} := MAC(KS_{Mac}, PK_{DH,IC})$ and $T_{IC} := MAC(KS_{Mac}, PK_{DH,IS})$, respectively, and exchange them. The tokens are verified on each side.
4. In the case of PACE-CAM the IC computes the CA data CA_{IC}, encrypts it using the shared session key K_{Enc} and sends it to the IS. The IS recovers CA_{IC} from the ciphertext and authenticates the chip.

[8]The exception is PACE-CAM, for which support is optional.

After successfully performing PACE, SM is started using the session keys K_{Enc} and K_{Mac}. The SSC is initialized to zero.

3.3.4.2 A better BAC?

As mentioned, SAC (and PACE) has been developed in answer to the known weaknesses of BAC. Whilst the entropy of BAC session keys are directly dependant on the entropy of the MRZ-derived static access keys, PACE uses the MRZ-data for initial access and employs asymmetric cryptography to establish strong session keys instead.

Clearly, the implementation of PACE in eMRTDs raises the level of security. There is an interesting paradox with this, however: the ICAO requirement for backwards compatibility with non-SAC eMRTDs dictates that both protocols must be available and provide the same level of access. Many would argue that this defeats the purpose, but it is seemingly a necessary step on the long road towards BAC deprecation.

3.3.5 Extended Access Control

With Doc 9303 definition of the additional biometrics [9] as *more sensitive* comes a recommendation for stricter protection. *Extended Access Control* is suggested as one way of achieving a higher level of protection[10]. However, Doc 9303 does not require nor specify EAC and leaves the implementation up to the issuing States.

The EU decided to include the fingerprint additional biometric in all new passports starting from mid-2009[11], and to protect it with EAC as specified by the German BSI *"TR-03110"* [5].

[5] specifies two protocols for EAC: *Terminal Authentication* (TA) and *Chip Authentication* (CA). TA and CA fulfil different requirements of EAC and only the successful execution of both is deemed sufficient to elevate access of the inspection system (IS).

Like the ICAO protocols specified in Doc 9303, both of the EAC protocols rely on a PKI. The CA protocol, like AA, relies on Passive Authentication. Terminal Authentication, however, is dependent on an entirely separate PKI, often dubbed the EAC PKI. The EAC PKI is outlined in 3.4.2.

3.3.5.1 Chip Authentication

Chip Authentication has two purposes:

- Authenticate the IC to ensure it is genuine.
- Establish a strongly encrypted SM session using a chip-specific key pair.

[9] DG3 and DG4.

[10] Data encryption being the other.

[11] This new EAC passport is commonly referred to as the 2nd generation ePassport.

Recall that the purpose of Active Authentication (see 3.3.2 is also to authenticate the IC. Thus, CA is an alternative to AA, but has the added benefit of providing strong session keys[5].

The CA protocol itself is an ephemeral-static Diffie-Hellman key agreement protocol. It establishes SM using static keys stored in the IC, whilst ephemeral keys are used on the IS side. The steps of the protocol are as follows:

1. The IC sends its static DH public key PK_{IC} along with the domain parameters D_{IC} to the IS.
2. The IS creates its ephemeral DH key pair $[SK_{IS}, PKIS, D_{IS}]$ and sends the public key PK_{IS} to the IC.
3. Both parties proceed to do the following:
    a. Perform the DH key agreement algorithm to compute the shared secret K.
    b. Derive the shared session keys K_{MAC} and K_{Enc} by applying the key derivation function to K.
    c. Compute the IS' compressed public key $Comp(PK_{IS})$ (used for TA).

As stated in [5], the IS must ultimately verify the authenticity of PK_{IC} by performing Passive Authentication. If PA fails the IC may not be considered genuine, regardless of whether CA was successful or not.

Once CA has successfully been completed, Secure Messaging is re-initialized using the newly derived session keys.

3.3.5.2 Terminal Authentication

Terminal Authentication authenticates the IS and grants explicit access to sensitive data groups. It is a challenge-response protocol based on *Card Verifiable Certificates* (CVCs, see 3.4.2.4).

An IS holds a CVC and corresponding private key which have been issued and signed by a *Document Verifier* (DV). The DV certificate is in turn granted by a *Country Verification Certificate Authority* (CVCA) (the trust anchor). The CVC contains the Access Rights given to the IS, which is ultimately used by the IC to determine access. This system is referred to as the *EAC PKI*, and is explained in further detail in 3.4.2.

In order to successfully authenticate using TA an IS must provide a valid[12] chain of CVCs which matches the CVCA of the IC and prove knowledge of the IS private key.

Additionally, [5] states that since the IS may access sensitive data after TA has been completed, the ephemeral public key PK_{IS} (which was used to set up SM by CA) must be authenticated as well. By doing so the access rights granted as a result of TA are bound to the SM session.

[12] Within the validity period and not revoked.

The identifier ID_{IC} used in TA is dependent on the whether BAC or PACE was used to set up the SM session. If BAC was used ID_{IC} is the Document Number. If PACE was used ID_{IC} is computed from the ephemeral PACE public key $PK_{PACE,IC}$.[13]

In preparation for TA the IS must resolve the needed chain of certificates C_{IS}. This entails:

- Reading the root CVCA public key PK_{CVCA} from DG14 of the IC[14].
- Building a chain of certificates which starts with a DV verifiable by PK_{CVCA} and ends with the IS certificate.

The steps of the TA protocol are as follows:

1. The IS sends the certificate chain C_{IS} to the IC.
2. The IC verifies C_{IS} and extracts PK_{IS} from the IS certificate.
3. A random nonce RND_{IC} is chosen by the IC and sent to the IS.
4. The IS creates the signature S_{IS} by signing $ID_{IC}||RND_{IC}||PK_{IS}\prime$[15] using the IS private key SK_{IS}. S_{IS} is then sent to the IC.
5. The IC verifies S_{IC} using PK_{IS}.

After successfully performing TA the IC grants access in accordance with the Access Rights of the IS certificate. It must also check that the compressed IS public key $PK_{IS}\prime$ matches the one computed upon completing CA, ensuring privileges are bound to the same session.

3.4 Public Key Infrastructure

The verification of authenticity and originality for eMRTDs relies on Passive Authentication (3.3.1), which in turn relies on the signed Document Security Object SO_d. Of course, creating and verifying the signature has to be supported by a PKI.

Similarly, EAC (3.3.5) is also dependent on a (separate) PKI, which is referred to as the *EAC PKI*. An overview of the ICAO eMRTD PKI is given in 3.4.1, whilst the EAC PKI is reviewed in 3.4.2.

[13] This is referred to as *dynamic binding*. There is also *static binding* which directly uses the Document Number (as for BAC) or the CAN, but it is deprecated.

[14] Typically done for CA.

[15] $PK_{is}\prime$ is the compressed IS public key.

3.4.1 ICAO eMRTD PKI

ICAO Doc 9303 Part 12 [12] defines a PKI standard for eMRTDs. As stated in [12], it is based on generic PKI standards[16]. However, since the eMRTD application is comparatively simple many elements from multi-application PKIs (such as the Internet PKI) are irrelevant and therefore not supported. As such the eMRTD PKI is a rather uncomplicated affair. Notably:

- Only a single root CA (CSCA) is supported per issuing State.
- An eMRTD is certified using a single certificate (the Document Signer). Complex certification paths are not needed nor supported.

[12] defines the following entities for the eMRTD PKI:

- Country Signing Certificate Authority - CSCA
- Document Signer - DS
- Inspection System - IS
- Master List Signer
- Deviation List Signer

3.4.1.1 Country Signing Certificate Authority

In the eMRTD PKI each issuing State has a single CSCA. The CSCA is the root CA and is therefore the trust anchor for all eMRTDs issued by that State. CSCAs are used to issue certificates for Document Signers, Master List Signers and Deviation List Signers as well as issuing Certificate Revocation Lists (CRLs).

3.4.1.2 Document Signer

Document Signers are, as is deductible from the name, responsible for signing eMRTDs. More precisely, the DS signs the contents of SO_d. Consequently, verifying SO_d (essentially PA, see 3.3.1) is done by means of the DS public certificate.

3.4.1.3 Inspection System

In the eMRTD PKI an Inspection System (IS) defined as the system which performs Passive Authentication and thereby performs validation of the DS certificate (based on trust in a known CSCA certificate).

3.4.1.4 Master List Signer

The Master List Signer (MLS) signs a list of both domestic and foreign CSCA certificates, known as a CSCA Master List. [12] defines the MLS as optional.

[16] X.509, RFC 5280

3.4.1.5 Deviation List Signer

The Deviation List Signer (DLS) signs a Deviation List. As the name implies a Deviation List is a signed list issued in order to inform of anomalies with the keys/certificates, LDS, MRZ or IC of an eMRTD. The use of a Deviation List is optional for the eMRTD PKI.

3.4.1.6 Certificates and keys

The public keys for the CSCA, DS, MLS and DLS are issued as X.509 certificates, as per the specification in [12].

[12] also defines validity periods for the private and public keys. As the private key is used for issuing (signing) and the public keys are used for validation, it is clear that there is an asymmetric relationship in the validity of the two. For example, a typical MRTD has a validity of 10 years, but the issuing Document Signer's private key is only valid for 3 months. Of course, the DS public key certificate must be valid for the lifetime of the signed MRTD, but cannot exceed it significantly as that would imply that the MRTD IC itself can be validated past its expiry. Likewise, the CSCA public key must be valid for the lifetime of all documents signed by its subsidiary Document Signers. Table 3.3 outlines the relationships between private key usage and certificate validity.

Entity	Use of Private Key	Public Key Validity
CSCA	3-5 years	13-15 years
DS	Up to 3 months	approx. 10 years
MLS	Discretion of issuer	Discretion of issuer
DLS	Discretion of issuer	Discretion of issuer

Table 3.3: The key usage and validities for the eMRTD PKI, assuming a 10 year passport validity. The table is reproduced (and slightly altered) from [12, p. 5, Table 1].

This asymmetric relationships between private key usage and public key validity means that an issuing State will at any given time have multiple valid CSCA certificates and DS certificates in their eMRTD PKI.

3.4.1.7 PKI Distribution

It is worth noting that each eMRTD issuing State runs its own, entirely independent PKI. As stated previously this PKI manages multiple public certificates of varying and overlapping validity.

The verification of an eMRTD relies on trust in and knowledge of the issuing PKI. Thus, verifying documents issued by foreign States requires the verifying party to have (trusted) access to the relevant PKI. The requirement of global interoperability coupled with the co-existence of all of these PKIs suggests the need for standardized mechanisms for exchange and distribution of certificates between issuing States.

The eMRTD PKI standard in [12, p. 9] gives the following short-list of certificate distribution mechanisms:

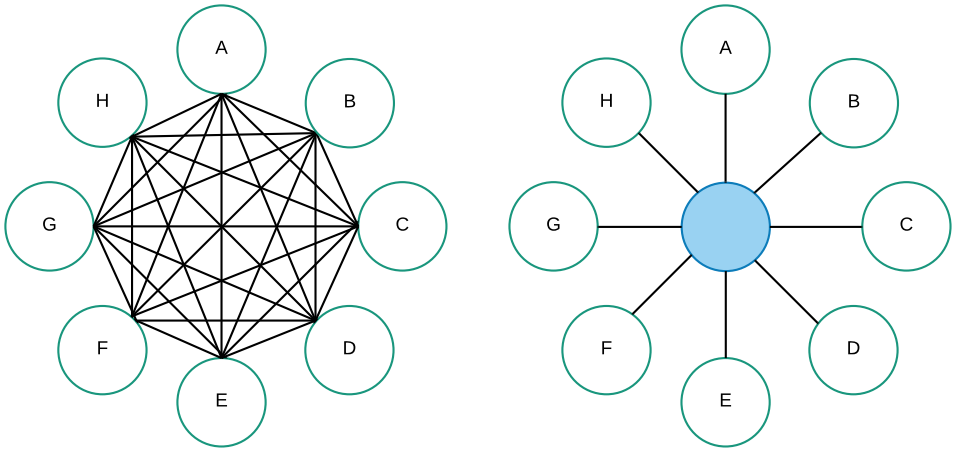
- Public Key Directory (PKD)
- Bilateral exchange
- Master Lists
- Deviation List
- The eMRTD IC (contains the signing DS certificate)

Bilateral exchange refers the direct exchange or handover of certificates, which might entail a range of methods (physical delivery, public download, LDAP server etc.). Common for all of these is that no third party is directly involved in distribution.

Master Lists, or CSCA Master Lists, is a subset of the bilateral distribution scheme[12, p. 11]. They are signed lists of CSCA certificates which are trusted by the signing party. As such, the Master List is a supportive mechanism which simplifies bilateral CSCA distribution: a Master List issued by a trusted party can provide implicit trust in third-party CSCA certificates. As an example, the German CSCA[17] makes the current German CSCA Master List available as a public download.

Of course, the distribution methods which do not employ a central broker are inherently harder to manage. For these, distribution of certificates is essentially a many-to-many problem which quickly grows with exponential rate. A central broker such as a PKD helps alleviate this. The problem is illustrated in Figure 3.2.

[17]The German eMRTD issuer, operated by the Federal Office for Information Security (BSI).



$$N = \frac{n\,(n-1)}{2} \qquad N = n$$

N : *Number of root certificate distributions*
n : *Number of States*

Figure 3.2: The certificate distribution problem. The left side illustrates the situation for bilateral exchange of certificates (which grows quadratically with n) and the right side illustrates the central broker model (which grows linearly), such as a PKD.

3.4.1.7.1 ICAO Public Key Directory

In order to enable global interoperability through PKI distribution, ICAO operates and provides a service known as the ICAO Public Key Directory [12]. In short, the ICAO PKD accepts certificates, Master Lists, CRLs and so forth from participants. These objects are then made available to the participating States through the PKD service. In practical terms the ICAO PKD offers updated collections of DS certificates, CRLs and CSCA Master Lists for download to participants.

The goal of the service is to act as a central broker and thus minimize the need for bilateral exchange. Currently there are 50 participating States[16].

The ICAO PKD data is publicly available for download[18] in the LDIF[19] file format from [17].

3.4.2 EAC PKI

Extended Access Control (see 3.3.5) is a mechanism for role and privilege based access control of sensitive data groups on the eMRTD. As the existing eMRTD PKI has no concept of privileges or roles, EAC relies on an entirely separate PKI. As is the case for the eMRTD PKI, each State issuing EAC eMRTDs must also run an EAC PKI.

[18] Under a non-commercial, personal license.
[19] LDAP Data Interchange Files

BSI TR-03110 part 3 [5] specifies the PKI for EAC. It is, as shown in Figure 3.3, a three-tiered hierarchy consisting of these entities:

- Country Verifying Certificate Authority (CVCA)
- Document Verifiers (DVs)
- Inspection Systems[20] (ISs)

These entities are closely related to the Roles defined for EAC in [12]: CSCA, $DV_{Domestic}$, $DV_{Foreign}$ and IS.

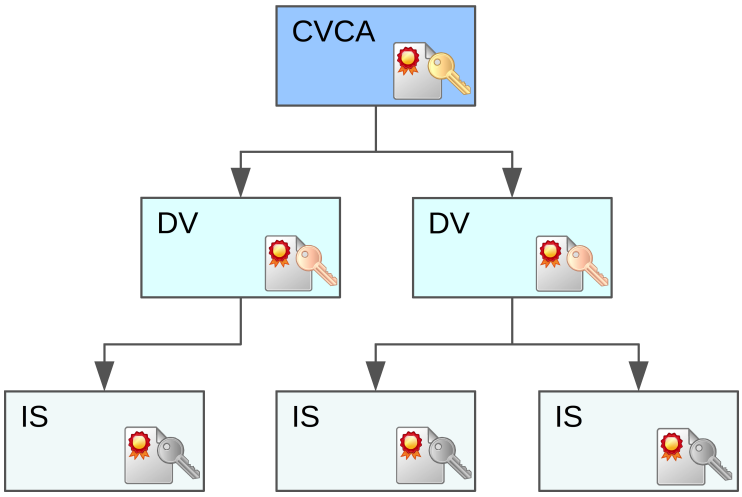


Figure 3.3: The EAC PKI hierarchy.

3.4.2.1 Country Verifying CA

The CVCA is the root trust-point in the EAC PKI. Consequently, each State implementing EAC must establish a CVCA.

The CVCA issues Document Verifier certificates, which contain information on exact EAC access privileges and the validity period for DVs. The specification in [5] does not explicitly state specific validity periods for DV certificates, but does say the validity must be short to diminish the consequences of potentially stolen or lost terminals.

It is worth noting that the EAC CVCA and the eMRTD PKI CSCA are two logically separate entities, operating in separate environments. However, [5] states that they *"MAY be integrated into a single entity"*, but must still use separate key pairs.

3.4.2.2 Document Verifiers

The purpose of the Document Verifier is to manage a set of terminals, i.e. a border control inspection systems[21]. It does so by issuing IS certificates,

[20] Also referred to as *terminals*.

[21] For consistency we refer to terminals as inspection systems or IS.

which means the Document Verifier is also a CA.

As previously stated the DV certificates granted by the CVCA contain access privileges and a short validity period. The DV issues IS certificates which contain a subset of these. In practical terms: IS certificates inherit the access and validity of the DV certificate. The DV might also further restrict access for an IS if so desired.

3.4.2.3 Inspection Systems

The Inspection System (IS) is the entity which performs EAC against the EAC eMRTD. As stated in 3.4.2.2 the IS holds an IS certificate. This certificate is verifiable via the chain of CVCA and DV certificates from which it was issued. The process of presenting this chain of Card Verifiable Certificates to the eMRTD IC is a core mechanism in Terminal Authentication (described in 3.3.5.2).

At inspection time (i.e. when performing TA) the IS must have access to both the chain of certificates and the private key of the IS certificate.

3.4.2.4 Card Verifiable Certificates

The eMRTD IC must verify a chain of CVCA, DV and IS certificates during Terminal Authentication. Due to the very limited computational power of the ICs, however, the EAC specification in [5] elects to not use traditional certificate formats such as X.509, but instead relies on Card Verifiable Certificates (CVCs).

CVCs are designed explicitly to be easily processed on resource-constrained devices such as smart cards. The ease of processing is achieved through using the TLV[22] encoding with fixed fields. This makes parsing significantly easier than for ASN.1 encoding (used for X.509), which requires the device to keep more in-memory state while parsing.

EAC CVCs are defined in terms of the Role and Access Rights (privileges) granted to the holder. These are encoded in the CVC itself, as well as the validity period.

[22] Tag-Length-Value.

3.4.2.5 Cross State certification

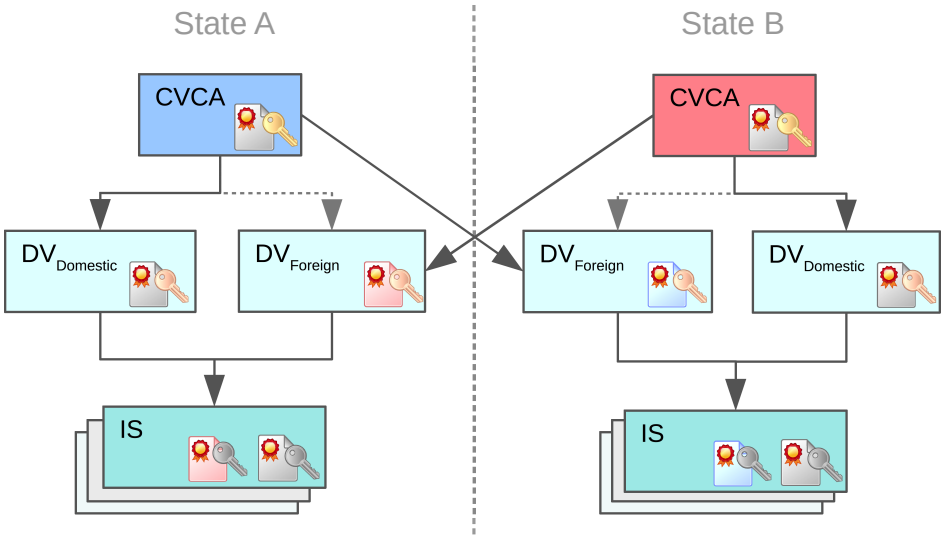


Figure 3.4: EAC PKI cross certification. State A provides State B with a signed foreign DV certificate and vice versa. Each State uses both foreign and domestic DVs to issue IS certificate/key pairs.

A core business case for eMRTDs is the verification of foreign documents (e.g. border control of foreign nationals). For the base eMRTD PKI described in 3.4.1 this is a matter of managing and distributing public certificates. For EAC, however, the picture is a little different as privileges are explicitly granted.

A CVCA grants access to a foreign State by issuing a *foreign* DV certificate. The foreign State can, in turn, use the DV certificate to issue IS certificates which grant access to sensitive data in the eMRTDs of the other party. The implication for inspection systems is that it must hold a DV certificate, IS certificate and IS private key for each foreign State for which it has been granted access. From a PKI management point of view the challenges are similar to those of the ICAO eMRTD PKI: effective and coordinated management of certificates is needed.

Figure 3.4 illustrates cross State certification.

3.5 Inspection procedures

In order to conduct inspection of an eMRTD, two[23] major steps are performed:

1. Acquire the static access keys (MRZ or CAN)[24].

[23]Due to the scope of this thesis we are disregarding physical checks, identity database lookups and biometric authentication of the holder.

[24]Non access-protected eMRTDs not considered.

2. Carry out procedure between Inspection System and eMRTD IC.

The first is the preliminary acquisition of the static document access keys. This is done either by optically reading the MRZ or manually reading the MRZ or CAN and inputting it into the IS.

The second step encompasses the entirety of the procedure between the Inspection System (IS) and the eMRTD IC: access control, reading data and performing verification of the document.

The details of the IS-IC procedure depends on the supported features of both. Due to the many varying eMRTD implementations with varying security protocol support and LDS contents, there is no one inspection procedure which covers all use cases. Therefore, the IS must actively check for features during inspection time to decide what access and authentication protocols to execute and what data to read out of the IC.

BSI TR-03110 [5], which is the adopted standard for EU EAC passports, refers to this step as the *Inspection Procedure*, of which two major variations are defined:

- Standard Inspection Procedure: for access to less-sensitive data groups.
- Advanced Inspection Procedure: for access to less-sensitive and sensitive data groups.

As defined by [5] which procedure to perform depends on the compliancy of both the IS and the eMRTD to the standard. *Compliancy* in this context refers to fulfilment of the BSI specification (EAC support) as opposed to a system which only complies with the Doc 9303 standard (non EAC).

Figure 3.5 illustrates the flow of eMRTD inspection, including the branching paths for the Standard and Advanced procedures.

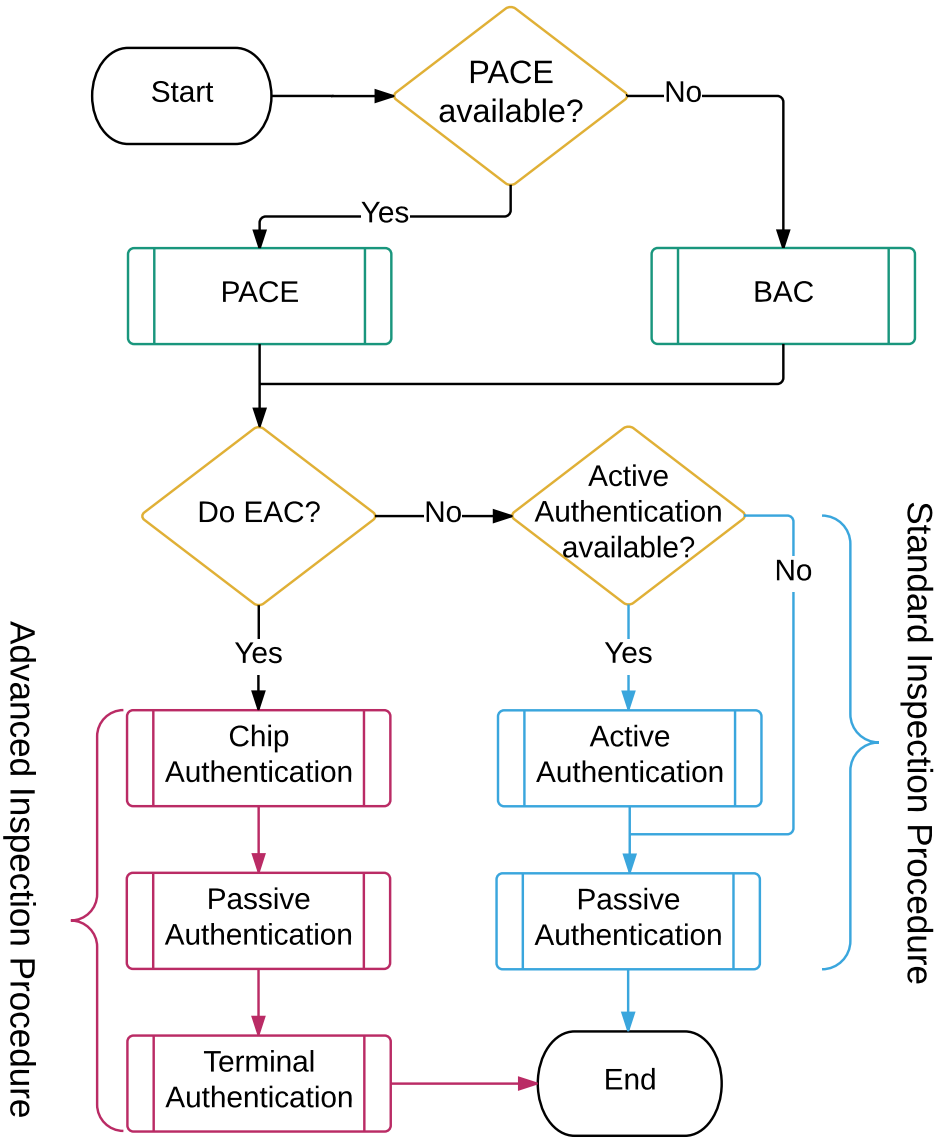


Figure 3.5: The standard and advanced inspection procedures.

For simplicity the flowchart shows the access control and authentication steps, but omits the actual reading of data groups. Also note that Passive Authentication (PA) is a continuous process for most applications, where data groups are verified as they are read.

Both procedures are preceded by successfully passing access control through either BAC or PACE. Both the BSI and ICAO standards require the IS to prefer PACE: BAC is only to be used when PACE is not supported by the eMRTD.

3.5.1 Standard Inspection Procedure

In accordance with [5], the Standard Inspection Procedure is performed if any of the following are true:

- The eMRTD does not support EAC.
- The IS does not support EAC.

That is, the Standard procedure is used whenever an eMRTD is only compliant with the Doc 9303 specification and not the BSI EAC standard. The steps defined for the inspection procedure are:

- Access control (BAC or PACE).
- Start Passive Authentication: read and verify SO_d. If PACE was used verify `EF.CardAccess` against `EF.DG14`.
- Optionally perform Active Authentication, where available.
- Optionally less-sensitive data groups and verify against SO_d (concludes PA).

3.5.2 Advanced Inspection Procedure

The Advanced Inspection Procedure is performed if both of the following are true:

- The eMRTD supports EAC.
- The IS supports EAC.

In other words: the Advanced procedure is only performed when the IS and eMRTD both support it. Additionally, for EAC to succeed the IS must have the access to the appropriate certificate chain.

The steps are:

- Access control (BAC or PACE).
- Perform EAC Chip Authentication (omitted if PACE-CAM is used).
- Start passive authentication: read and verify SO_d. If PACE was used verify `EF.CardAccess` against `EF.DG14`.
- Optionally perform Active Authentication (redundant as we are using CA and therefore typically not done).
- Perform Terminal Authentication.
- Optionally read less-sensitive and sensitive data groups and verify against SO_d (concludes PA).

3.6 Development

Recall form section 2.2 that Doc 9303 evolved from an initial 1980s publication of the standard.

The development is still ongoing and has in the case of eMRTDs left a trail of technical debt in the form of multiple co-existing generations of technology and standards.

The distinction for versions of the ePassport is made as *generations*. At the time of writing the third generation ePassport is the last to be standardized in Doc 9303, and the fourth generation standard is due in early 2016.

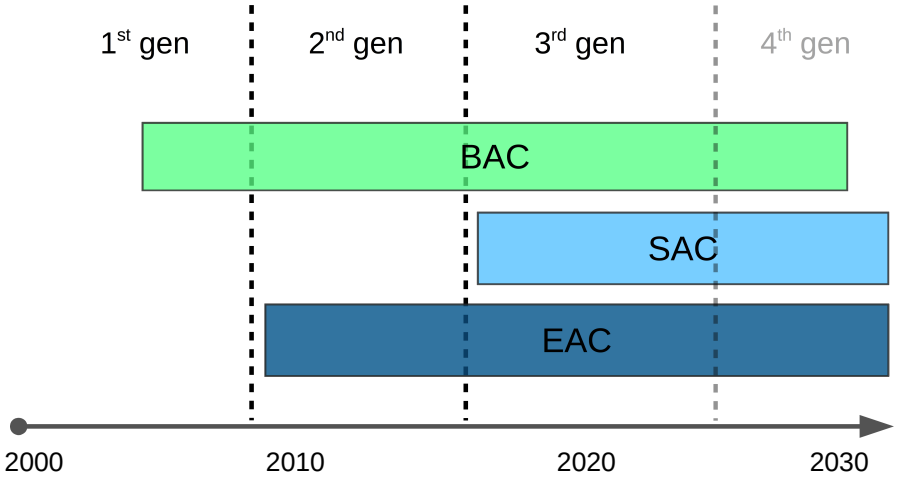


Figure 3.6: The eMRTD generations timeline. The generations are distinguished by the introduction of major technologies (BAC, EAC with sensitive biometrics, SAC) into the standards.

The first generation ePassport standard was adopted by ICAO in 2003 [18] and the first ePassports were rolled out by States starting in the mid-2000s.

This initial generation introduced the contactless IC containing the MRZ, biometrics and signed Security Object.

It supported Basic Access Control for protection against eavesdropping, mandatory Passive Authentication for data integrity and optional Active Authentication for protection against chip substitution.

In 2006 the European Union asked all member States to include a secondary biometric in their passports. Specifically the fingerprint biometric was chosen, and the deadline to start issuing second generation ePassports with fingerprints included was set to mid 2009.

The fingerprint (and iris, for that matter) was considered by the EU to be highly sensitive, more so than the face biometric. From this it was clear that the existing BAC protocol alone was not sufficient to protect access. In part this was due to the general security level of BAC, but also because enforcement of access levels was required: the secondary biometric should

only be accessed by authorized parties.

In answer to this the second generation of ePassports introduced the *Extended Access Control* (EAC, see 3.3.5) protocol to meet these requirements. EAC is built on top of the existing protocol suite (BAC, AA, PA) and introduced Chip Authentication and Terminal Authentication. The latter employs a specialized EAC PKI to enforce access rules for the highly sensitive biometrics.

The third generation of ePassports was brought forth with the introduction of Supplemental Access Control. The EU decided that all member States were to implement PACEv2 as per the SAC specification within the end of 2014 [19]. Notably, for the third generation of ePassports SAC does not replace BAC but is available alongside it. This coexistence was deemed necessary to maintain interoperability and compatibility with inspection systems.

The next evolution of the eMRTD is yet to be standardized at the time of writing, but is expected to revolve around the inclusion of the new *LDS2*. This new revision allows the writing of data post-personalization, intended to support the use cases of digitally signed travel visas and stamps being carried in the ePassport IC.

3.7 Implementations

The Doc 9303 [4] standard contains specifications for a wide range of identity documents, including passports, travel visas and ID cards.

As stated earlier virtually all States issue ePassports today. In addition, many implement the ICAO (and BSI) eMRTD standards in national identity cards and residence permits.

This reuse of standards is obviously advantageous as it permits leveraging existing technical expertise and deployment infrastructure, as well as providing interoperability with existing systems for enrollment and border control.

In the following sections we review a few notable implementations of both ePassports and eIDs.

3.7.1 EU passports and Council Regulation EU/2252/2004

In *Council Regulation No 2252/2004* [19] of December 2014 the EU established that the ePassports of all member States should adhere to the a specification which includes the facial image, fingerprint and appropriate protection thereof, and that these should be implemented in an interoperable format. ICAO Doc 9303 is specifically chosen as the standard for interoperability and all EU passports and supporting infrastructure must therefore comply with it.

The regulation has been amended in later times (2011, 2013) to adopt SAC and EAC [5], making these mechanism mandatory for EU passports issued in recent (and coming) years.

As a result of this the member States of the EU implement largely interoperable ePassports.

3.7.2 The German Identity Card

On November 1st 2010, Germany started issuing their new electronic identity cards (eID) to citizens.

Figure 3.7: The German eID card, from [20].

The German eID is a TD-1 type (credit card sized) Doc 9303-compliant card which includes printed personal data and image of the holder, an MRZ and a contactless chip (IC).

The IC implements three separate applications:

- **ePass:** mandatory. ICAO eMRTD with MRZ data, face image and optional fingerprints. Data is protected by PACE and EAC. The ePass is specifically for governmental use and is valid as a passport substitute for travel (within certain States).
- **eID:** optional. General purpose identification storing personal data, place of residence and so forth. Protected by a PIN an usable by online as well as offline services.
- **eSign:** optional. Stores a signing key and certificate pair for creation of electronic signatures. Allows citizens to sign electronic documents using PIN and card.

This wide range of features is a result of the German vision for the eID as a general purpose authentication token. It is coupled with an online architecture for service providers to take advantage of the eID capabilities.

3.7.3 The Norwegian implementation

The EU decided in [21] that the regulation should encompass the Schengen *Acquis*, and due to this it is also applies to Norway. The Norwegian Parliament sactioned this in 2005.

As a result the first Norwegian ePassport implementation was issued from October 2005. These passports included the bare minimum of the specification, namely DGs 1 and 2 protected with Basic Access Control and Passive Authentication.

In April 2010 Norway introduced the new second generation ePassports which included the fingerprint biometric. The Active Authentication and Extended Access Control mechanisms were also added.

Starting from the 1st of January 2015 all passports were issued with added support for Supplemental Access Control (PACEv2), which marks the third generation of Norwegian ePassports.

A new Norwegian passport is planned for 2017 but does not contain any technical changes from the 2015 generation (it is basically a visual update).

3.7.3.1 Norwegian eID

Work is currently ongoing to standardize the new Norwegian eID[25]. As of now decisions have been made to align the technical specification of eID with the Norwegian ePassport, meaning the eID will stricly follow the standards set forth in the 7th edition of Doc 9303 [4].

This includes support for SAC and EAC. However, the decision on what contents to include has not yet been reached. More specifically the choice of whether the fingerprint biometric is to be included or not is still pending.

Issuance of the eID is due to start in 2017, aligned with the new passports.

[25] *"Nasjonalt ID-kort"*

Chapter 4

Contactless smart cards

This chapter is a review of the smart card technologies most relevant to ICAO eMRTDs. A biew overview of the relevant ISO/IEC standards is given, as well a look at smart card file systems, the data interchange protocol and secure messaging.

4.1 Standards

The ICAO eMRTD specification mainly relies on three separate ISO standards: *ISO/IEC 7816*, *ISO/IEC-1443* and *ISO/IEC 7501*.

4.1.1 ISO/IEC 7816

ISO/IEC 7816 is a fourteen part standard which concerns a host of various smart card types and applications. Part 4 of the standard specifically deals with the message interchange formats and data structures employed by smart cards. The messaging protocols and data structures defined for ICAO eMRTDs are based on this standard.

4.1.2 ISO/IEC 14443

The ISO/IEC 14443 standard specifies the communication interface for contactless, "close proximity" smart cards. This includes the physical RF interface, electrical characteristics and the low lever communications protocols (PHY layer) and anti-collision mechanisms.
Cards complying with the ISO/IEC 14443 standard operate at a radio frequency of 13.56 MHz and are defined to work within a card to reader range of 10 cm. ICAO eMRTDs comply fully with ISO/IEC 14443.

4.1.3 ISO/IEC 7501

ISO/IEC 7501 is essentially a short form ISO endorsement of the ICAO Doc 9303 standards for MRTDs. As such, ICAO eMRTDs also comply with ISO/IEC 7501.

4.2 ISO/IEC 14443 and NFC

The physical communication interface for proximity smart cards is specified in ISO/IEC 14443. It is a four-part standard which establishes the physical characteristics (part 1), RF interface (part 2), initialization and anti-collision (part 3) and transmission protocol (part 4) of compliant systems. ICAO Doc 9303 part 9 specifies that eMRTD contactless ICs should conform to this standard.

There are two variations of ISO/IEC 14443: *Type A* and *Type B*. The main differences between the two are in the coding schemes and modulation methods. The transmission protocol is the exact same for both. The specification allows eMRTDs to support either A or B, meaning inspection systems must support both.

The RF interface for compliant systems is specified to operate at 13.56 MHz and allows for a data rate of 106 kb/s, both ways. The specified physical range of operation is approximately 10 cm between card and reader.

The transmission protocol is a half-duplex block transmission scheme, which is given in part 4 of the standard.

4.2.1 Near Field Communication

Near Field Communication (NFC) is a near-field wireless connectivity technology, standardized by the NFC Forum. In essence, NFC incorporates and extends a range of existing protocols for near-field communication in order to provide a common set of protocols and services.
NFC provides bidirectional communication between devices with three main modes of operation:

- Card emulation - allowing devices to act as contactless smart cards.
- Reader/writer - the device acts as a terminal, able to read contactless smart cards.
- Peer-to-peer - two devices communicate directly with each other.

At its core NFC uses key components of the ISO/IEC 14443 standard, making NFC devices fully compatible with ISO/IEC 14443. Thus eMRTDs are compatible with NFC devices running in reader/writer mode. Also, a compliant eMRTD could be implemented on an NFC device using card emulation.

NFC capability is deployed in a range of consumer devices such a mobile phones, game consoles and laptops. Initially the use cases for NFC in such devices were rather simple, such as reading basic RFID tags and performing an action (e.g. opening a web page or connecting to a network).

Today, however, NFC has evolved to be the de-facto standard for near-field communication in consumer devices and has found more advanced applications such as mobile payment solutions and identification. Both *Apple Pay* and *Android Pay* are examples payment solutions built on top of NFC. Much due to the growth of these services, virtually all modern smart phones are now NFC capable, as well as range of other devices.

4.3 Smart card file systems

ISO/IEC 7816-4 specifies three types of files for smart cards: the *Master File* (MF), *Dedicated Files* (DF) and *Elementary Files* (EF).

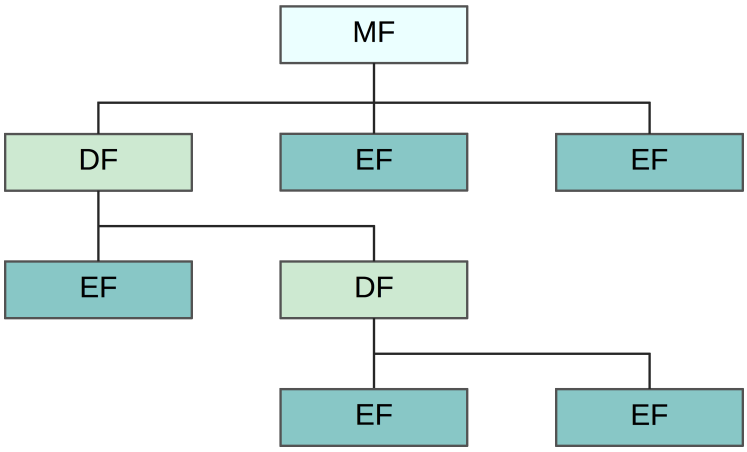


Figure 4.1: ISO/IEC 7816-4 file tree.

The Master File acts as the root directory of the file system. There can only be one MF, and it contains all other directories and files. It is required by the standard to be present in all smart cards.

Dedicated Files act as sub-directories of the MF, which is why they're also known as *Directory Files*. A given DF may contain any number of DFs or EFs, allowing an arbitrary depth file tree [1].

Elementary Files are the base element of the file system, much like the files in a UNIX file system. They are stored directly under a DF or the MF. The layout of a file employs one of four schemes: *transparent*, *fixed record*, *variable record* or *cyclic*. In any of these cases the layout of the actual contents is out of scope of the standard and is application specific.

Two types of EFs exist: working EFs, which are used for data accessible only by the outside world, and internal EFs, which are used solely by the

[1] In practice the file tree depth is limited by the memory of the device and deeply nested file trees are very uncommon.

application/OS itself. The latter might also contain application secrets such as cryptographic keys, and access is therefore protected by the smart card OS.

Referencing a file is done either by a two-byte file identifier (FID), or by a path (concatenation of FIDs). An FID is required to be unique within a DF.
There are also 5-bit *Short FIDs*, which are used for file reference within the context of a command.

4.4 Commands

Communication between a smart card and a terminal is achieved through simple commands known as *Application Protocol Data Units* (APDUs). The structure of an APDU is defined by ISO/IEC 7816-4.
Two types of APDUs exist, the command APDU and the response APDU.

4.4.1 Command APDU

A command APDU is sent by the terminal to the card. It consists of a 4-byte header and a variable size data field (0 to 2^{16} bytes)[2].

Field	Bytes	Description
Header		
CLA	1	Class of instruction
INS	1	Instruction code
P1	1	Parameter 1
P2	1	Parameter 2
Command data		
Lc	1 or 3	Byte length of the data field
Data	Lc	Arbitrary byte string
Le	1 or 3	Maximum expected length of the response data field

Table 4.1: The Command APDU format.

4.4.2 Response APDU

Response APDUs are sent by the card to the terminal. They are always direct responses to command APDUs.
There is no header for the response APDUs, they simply contain a 0 to 2^{16} byte data field[3], followed by two mandatory status bytes, *SW1* and *SW2*.

[2] A standard sized APDU will allow for a maximum of 256 bytes of payload data. The 2^{16} bytes figure requires support for *Extended Length APDUs*.
[3] See footnote 2.

The status bytes indicate the status of the processed command whilst the data field contains the response data, if any. The exact meaning of a status word is specific to the vendor and/or application, though there are common conventions for prefixes indicating certain conditions. The only truly protocol agnostic status word is the *OK* SW: `0x90 0x00`.

Field	Bytes	Description
Response data		
Response data	0-2^{16}	The response
Trailer		
SW1-SW2	2	Command processing status bytes

Table 4.2: The Response APDU format.

4.5 Secure Messaging

ISO/IEC 7816-4 defines a *Secure Messaging* (SM) protocol which provides privacy and integrity of the APDUs exchanged between a card and reader. SM establishes a secure channel by bilateral derivation of session keys $\mathtt{K}_{\mathtt{ENC}}$ and $\mathtt{K}_{\mathtt{MAC}}$.

Once the secure channel is established all subsequent APDUs being exchanged are encapsulated as an SM message. The plain APDU itself is embedded in the SM APDU as an encrypted payload alongside a MAC of the contents.

Optionally, the two parties in SM share a common *Send Sequence Counter* (SSC) which is incremented and prepended to each message. The use of an SSC is a preventive measure against message replay attacks.

The SM protocol defined in ISO/IEC 7816-4 is designed to be very flexible and the actual implementation varies considerably between applications. ICAO Doc 9303 specifies SM using AES or 3DES in CBC mode with 3DES *Retail* MAC or AES-CMAC, respectively. The BAC, PACE and CA protocols are responsible for establishing the SM session, and an SSC is always used.

Chapter 5

Android

Android is an operating system (OS) developed by Google Inc. It is designed first and foremost for mobile, touch-screen devices such as smartphones and tablets, but also supports various other uses such as set-top boxes (Android TV) and cars (Android Auto).

Though internally developed at Google, the source code of Android is available under an open source license, making Android the only major, open source mobile platform. It is notable, however, that most Android devices ship with both open source and proprietary components.

As of the end of 2015 Android held an 80% share [22] of the mobile OS market.

In this chapter we give a brief introduction to Android from an app developer's perspective. We review the OS architecture, look at the fundamental development tools and outline the application model.

5.1 Architecture overview

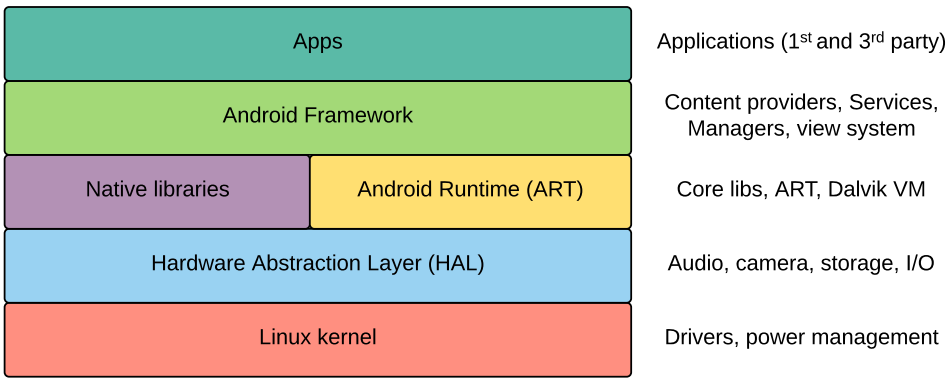


Figure 5.1: The Android software stack.

At its core Android is based on the Linux kernel. On top of this is a Hardware-Abstraction Layer (HAL) which provides a hardware-agnostic interface between the kernel and upper layers of the stack.

Above the HAL is the Android Runtime (ART) and user-space native libraries and middleware (much like a traditional Linux or UNIX system). ART is Android's application runtime which handles ahead-of-time compilation and execution of Java byte code.

The Android Application Framework defines a set of APIs which bridge the gap between Android applications (apps), low-level system services and ART. It comprises the higher level services and managers which all Android apps, from both first- and third-parties, depend on.

5.1.1 Security model

The security model of Android is at the core inherited from Linux, but also extends on this and implements key security mechanisms which don't exist in traditional Linux OS distributions.

Some of these features are:

- **Mandatory app sandboxing:** each app has its own Linux user ID (UID), runs in a separate ART runtime (process) and has its own storage tied to the UID (inaccessible to all other users).
- **Explicitly granted app permissions:** access to system services must be explicitly requested by the app. For security sensitive services such as the camera, network or external storage, permission must also be explicitly granted by the user.

5.2 Android software development

Android application code is written predominately in the Java programming language, but apps can also include native code such as compiled C or C++ libraries. Additionally, third party tools exist to facilitate development of Android apps in a range of other languages such as Java bytecode based languages (Kotlin, Groovy) or web technologies (JavaScript, HTML).

In addition, user interface components (views) are declared in XML files (but can be created programmatically as well), which are transformed to *binary XML* at compile time.

Apps are packaged and distributed as APK files which are essentially portable archive files containing the compiled code, binary XML files and other assets of the app.

As such, Android loosely follows the "Write once, run anywhere" philosophy of Java, though incompatibility between hardware, OS versions and application code is often an issue.

5.2.1 Android SDK

The Android Software Development Kit (SDK) is the official SDK for Android development. It includes the tools needed for compiling, packaging and debugging Android apps on all popular desktop OSs.

In addition to the core SDK libraries the optional Support Library offers a host of APIs and components intended to aid developers in maintaining backwards-compatibility across different releases of the Android SDK for their apps.

5.2.2 Android Studio

Google develops and distributes an official IDE for Android named *Android Studio*. It is based on the *IntelliJ* technology by JetBrains. However, a number of alternatives exist including *Eclipse* and *Netbeans*. Neither are required, though, and only the SDK is strictly needed for Android development.

5.3 Application model

An Android app is made up of four component types:

- **Activities:** a single screen with a user interface. All apps which display a user interface must have at least one activity but typically all self-contained tasks are defined as separate activities.
- **Services:** runs in the background to perform long-running operations or to facilitate communication with remote processes.
- **Content providers:** manages persistent storage and data sharing across apps.
- **Broadcast receivers:** responds to system-wide *broadcasts*, such as system events or broadcasts initiated by apps.

Figure 5.2 shows a simplified overview of the runtime relation of components in a typical app.

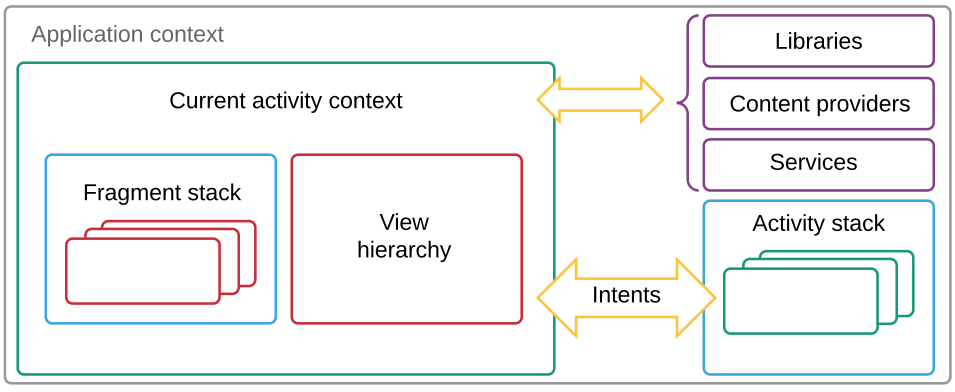


Figure 5.2: A somewhat simplified overview of a typical Android application process.

5.3.1 Life cycles

Mobile devices are subject to completely different use cases than traditional computers. They are frequently put into and awoken from sleep, they have subsystems which run regardless of process state (e.g. networking) and they are very limited in terms of power, memory and computational resources.

The practical implication of this on Android is that an activity must be aware of sudden context changes, and must be designed to be somewhat "pausable" or even suddenly killed when put into the background.

In order to manage this the Android framework has a concept of *life cycles*. Life cycle events are emitted by the system upon an activity launching, pausing, closing, resuming and so on. Activities must therefore implement a series of life cycle callbacks in order to manage themselves across all of these states.

From an application developer's perspective this means developing Android apps is very different from most other systems: the application state must be manually committed to memory and restored on demand and long-running processes must be carefully managed to avoid execution threads leaking.

5.3.2 Storage

The Android framework offers a selection of built-in storage options for apps:

- **Internal Storage:** app-private file storage on the internal device storage.
- **External Storage:** public data storage on the external device memory (typically an SD card).
- **Shared Preferences:** non-relational storage of simple data as key-value pairs. Typically used for storage of app settings. Data is persisted to an XML file in internal storage.
- **SQLite Database:** SQL database for transaction-safe storage of relational data. SQLite databases are stored in internal storage.

Chapter 6

Related work

This chapter reviews a selection of related works in order to contextualize our own work and contributions.

We first give a survey of a small selection of relevant academic works, then look at open source software within the field of eMRTDs and at last we give a brief survey of proprietary products.

6.1 Academic works

Many academic works exist within the field of ePassports, eID and the supporting technologies. However, most of these deal with formal security analysis or biometric authentication techniques and only a select few consider the practical design and implementation of systems.

In this section we present this small selection of academic contributions which we consider to be related or tangential to our work.

In [23] Stein presents a prototype implementation of an NFC smartphone based eID reader. The implementation is focused on using Trusted Execution Environments for security, demonstrating not only that an eID reader on commodity devices is realizable, but also that it can be done with strong security. The prototype presented supports the German eID, which is related to (and has many technologies in common with) EU and ICAO ePassports.

In [24] Terán and Drygajlo explores the implementation of inspection systems (IS) for biometric passports. Their approach is the implementation of a Java based inspection system conforming to the ICAO eMRTD standards, implementing Basic Access Control. They also discuss the implementation of an ICAO ePassport applet on the JavaCard platform, which is used as a tool for evaluation of their system. The resulting IS can thus be used for inspection of emulated as well as real eMRTD smart cards.

Perhaps closest to our project is the Master's thesis of Vošček [25]. It describes the implementation and evaluation of an Android NFC based app for reading biometric passports and performing facial recognition of the holder. The prototype implementation does MRZ scanning, extracts

the face biometric and compares it to a photo of the holder using facial recognition techniques.

6.2 Software

A wide range of both commercial, non-commercial and open source software products exist within the realm of eMRTDs. Most are fairly peripheral to our project, some implement similar functionality and some are in fact integrated into our implementation as software dependencies.

6.2.1 Open source libraries and applications

6.2.1.1 JMRTD

JMRTD is an open source Java library which aims to implement the Doc 9303 and BSI TR-03110 MRTD standards for many applications. It is by far the most complete open source implementation. It consists of:

- A JavaCard *"passport applet"* which allows the user to create their own passports.
- A general purpose host side API for accessing eMRTDs. This includes functionality for the low level communication, performing authentication reading data and verification of documents.
- Supportive libraries for decoding and encoding the image formats (WSQ, JPEG 2000) commonly found in eMRTDs.

The project was initially started in 2006 as part of a research project by the Digital Security Group at Radboud University in Nijmegen, Holland.

Over the course of the next decade many additions and changes have been made. The support for EAC was added in 2009, and support for CBEFF-encoded data groups was added in 2011.

The current version (JMRTD 0.5.6) supports all of the ICAO and BSI specified security protocols, but the implementations of the newer additions such as PACE and the EAC protocols are not yet feature complete.

An initial Android port of JMRTD was made in 2011 along with a (very rudimentary) sample app. In later years the Dutch company Novay created and released a closed source proof-of-concept Android app (2013) which inspects eMRTDs and another company, InnoValor, has been actively developing a proprietary JMRTD based, document verification system (see 6.2.2.2).

JMRTD is licensed under the GNU Lesser General Public License (LGPL) and is available from https://sourceforge.net/projects/jmrtd/.

6.2.1.2 Animamea and Androsmex

Animamea is an open source Java library for EACv2 as defined in [5]. It includes PACE, Chip Authentication and Terminal Authentication implementations. The project is licensed under LGPL and is available from https://github.com/tsenger/animamea.

Androsmex is another project by the Animamea author. It is an Android NFC implementation of PACE which is built on top of the Animamea project. The current version is labelled as *"early alpha"* and claims to only have been tested with the German eID card. It is LPGL licensed and can be downloaded from https://github.com/tsenger/androsmex.

6.2.1.3 The PersoSim project

PersoSim is an open source multi-component simulation framework for the German eID card. The main use case of the project is to provide a simulated eID card for development purposes, as opposed to working with sample cards. A full implementation of BSI TR-03110[5] is provided.

The project also contains tools for setting up virtual eID readers based on the BSI TR-03119[26] specification, which aims to aid development by diminishing the need for costly hardware.

The PersoSim development is led by HJP Consulting GmbH on commission from the German Federal Office for Information Security (BSI), which is the sole sponsor of the project. The BSI hopes that providing the PersoSim tools will aid third-parties in developing services and systems which leverage the eID, as well as having applications for further internal development of the eID infrastructure.

The PersoSim project is licensed under the Gnu General Public License (GPL) and is available from http://www.persosim.de/.

6.2.1.4 OpenPACE

OpenPACE is a project which implements Extended Access Control as specified in BSI TR-03110[5]. It includes *libeac*, which implements the PACE protocol as well as the EACv2 protocols (Chip Authentication and Terminal Authentication). Additionally the project includes tools for working with Card Verifiable Certificates, which are used for the EAC PKI.

The *libeac* library is made to be portable and is written in C. However the project provides API bindings for a host of programming languages such as Python, Ruby, Java and JavaScript.

OpenPACE is licensed under the GPL license and is available from http://frankmorgner.github.io/openpace/.

6.2.1.5 EJBCA

EJBCA is a widely used, open source PKI Certificate Authority (CA) developed by PrimeKey Solutions AB. It is a large, modular, enterprise-grade Java application which offers features allowing implementation of a complete PKI.

Among the many supported PKI technologies, EJBCA includes a complete implementation of the EAC PKI (also known as the EU ePassport PKI) and Card Verifiable Certificates. Thus, EJBCA is the only open source solution for setting up an EAC CA[1].

6.2.2 Commercial and proprietary software

There are many commercial providers offering proprietary software solutions for eMRTDs, from manufacturing of the document itself to PKI and inspection software. Such solutions are built and tailored for a specific implementation and are not usually available to the general public as products.

The following sections present a small selection of notable, proprietary software products.

6.2.2.1 Universal Reader Tool

The Universal Reader Tool (URT) is a desktop application for inspection of eMRTDs and European driving licenses.

For eMRTDs, it supports the third generation ICAO eMRTD standards (PACEv2) and EU ePassports (EAC).

First and foremost intended as a tool for aiding interoperability testing, development of URT is sponsored by several parties including the French government and commercial actors like Gemalto and Oberthur Technologies.

The software can be downloaded free of charge from http://www.keolabs.com/universal_reader.html.

6.2.2.2 InnoValor ReadID

The Dutch company InnoValor develops and licenses the newly announced *ReadID* software. In short, ReadID offers an all-around solution for leveraging electronic identity documents (eMRTDs, eIDs). This includes software APIs for client side inspection of documents (on Android using OCR and NFC, similar to our project) and server side solutions supporting this.

A sample Android app demonstrating the technology is available from Google Play under the name *ReadID - NFC Passport Reader*. ReadID is owned and developed in part by the author of JMRTD.

[1] In case you want to start your own country.

Part III

MRTD Inspector

Chapter 7

Design

In this chapter we present the high-level design of our Android eMRTD inspection system, the MRTD Inspector app.

First we present our system requirements, then the the envisioned user interface and activities of the app, and at last an outline of the system components.

7.1 System requirements

We need to form a set of requirements for our system. The requirements are first and foremost constructed for the purpose of guiding the project towards our goal, and since we are taking an iterative and experimental approach to our process they are not necessarily strictly followed.

7.1.1 Guiding requirements

Our requirements are governed by a small set of guiding requirements which have been given by POD:

1. The prototype implementation should be developed as an Android app.
2. The app should aim to demonstrate the feasibility of an Android based eMRTD inspection system.
3. The app should leverage the built-in NFC functionality of the Android phone to communicate with the eMRTD.

Due to the nature of the project and the software development philosophy of the implementers we define the following additional guiding requirements:

1. Only free and open source software can be used in the implementation.
2. If possible, the prototype should be released under a permissive open source license.

7.1.2 Overview of features

From our review of the ICAO Doc 9303 [4] and BSI TR-03110 [5] in chapters 2 and 3 we recognize two key components to performing eMRTD document inspection:

- Acquiring the static document access keys through OCR or manual input.
- Performing the Standard or Advanced inspection procedure as outlined in 3.5.

Our app should therefore offer these two main features:

- OCR scanning of the MRTD MRZ.
- Inspection of the MRTD over NFC.

Additionally we need the following supportive features:

- The user should be able to configure the app for experimental use cases (i.e. select protocols to use).
- The app should allow the user to install certificates and keys (where applicable) for the ICAO and EAC PKIs.

7.1.3 MRZ OCR reader

The MRZ OCR reader should follow these requirements:

1. The reader uses the back-facing camera of the Android smartphone.
2. OCR techniques are employed to search a continuous stream of images from the camera for a valid two-line (TD3) MRZ.
3. The user should be given visual feedback of the recognition process to aid in capturing optimal images.
4. Upon detecting a valid MRZ the contactless inspection process is launched with the acquired credentials.
5. To aid operation in low-light situations the option to turn on the smartphone torch (using the camera flash) should be given.

7.1.4 Contactless inspection

The contactless inspection should be able to perform both the Standard and Advanced inspection procedures, which means supporting the following protocols:

1. Basic Access Protocol (BAC).

2. Password Authenticated Connection Establishment (PACEv2) as defined by SAC[15].
3. Passive Authentication (PA).
4. Active Authentication (AA).
5. Chip Authentication (EAC-CA).
6. Terminal Authentication (EAC-TA).

The following additional requirements must also be met:

1. Visual feedback should be given to the user during the inspection process.
2. In case of inspection failure the process should be gracefully aborted.
3. The less-sensitive and sensitive (where possible and applicable) data groups should be read.
4. Upon successful inspection the user should be presented with the data read from the MRTD, the features supported by the IC and the result of the verification process.

7.1.5 Configuration

To aid its usefulness as an experimental tool the app should be configurable. The following settings are needed:

1. Adjustable timeout for the NFC communication (milliseconds).
2. Enable/disable forcing BAC authentication for SAC documents.
3. Enable/disable running BAC is PACE fails.
4. Enable/disable Passive Authentication.
5. Enable/disable Extended Access Control (CA and TA).

In addition to these settings the app must support the following:

1. Installing a CSCA Master List for document verification. CSCA Master Lists in the binary format, such as those offered as public downloads by the German BSI, should be supported.
2. Installing CVCA certificate chains and IS private keys for EAC.

7.1.6 Quality requirements

The requirements listed so far are all *functional*. However, functionality and utility in and of itself is only half the picture of building a serviceable IS.

Obviously, the real world usefulness of a mobile IS is dependent on the functionality being in place, but must also perform inspection reliably and quickly. Therefore, we need to consider our requirements for performance and reliability.

It is worth noting that "quality" pertains to a wide range of measures, but because we are implementing a *prototype* many of these are disregarded and we selectively focus on the performance and reliability of performing inspection.

7.1.6.1 Expectations

In order to set our requirements for performance and reliability, we must first determine the expectations. For the case of reliability this is trivial, as the expectation is obviously that the system functions in all reasonable circumstances.

The question of performance, however, is not as straight forward. There is no official "golden standard" determining the duration of the inspection procedure, and information on the subject is scarce.

In order to reach an approximation, however, we have conducted a brief and informal survey of commercial systems to try and deduce reasonable estimates for the following:

- Duration of MRZ scanning.
- Duration of the contactless inspection procedure.

Unfortunately, though there are many vendors in the eMRTD market not many release such specifications to the public, and when they do they are usually rough approximations written in a product sheet. A few of the notable figures we found are:

- The Diletta TDR700 [27] claims MRZ recognition time in 1 second and contactless inspection of BAC + EAC ePassports in 2 seconds.
- The Morpho Ideal Pass^{TM} solution is claimed to perform electronic inspection in 3 seconds (SAC + EAC).
- The German BSI reports that their EasyPASS automated border control system performs the optical + contactless inspection in 5-6 seconds.

It should be recognized that these numbers are highly inaccurate. As pointed out by the vendors themselves: the duration of contactless inspection is dependent on the contactless IC and the also the size of the contained data. We should also assume that the vendors are reporting "best case" figures, as they are taken from product sheets and not scientific publications. As

such, the 5-6 second inspection time reported for the German ABC-solution might be closer to reality.

7.1.6.2 Forming the requirements

Pertaining to the approximate figures found in the previous section, we can reasonably assume such as system to perform the inspection (MRZ and contactless) in the 3-6 second range.

However, we must also consider that these systems are designed for a very different environment and purpose than ours. First, they are made for volume, meaning optimization of the inspection time is paramount in order to reduce queues at border control, governmental offices and so on. Second, they are not mobile systems, and our system is not setting out to directly compete with the performance of stationary border control systems.

We did not find comparable figures for the *mobile* commercial solutions on the market, but one could expect them to perform somewhat slower than their stationary counterparts.

Another aspect to consider is the practical tolerance for the duration of inspection. That is, what would be an acceptable inspection time in a mobile scenario? It might not be reasonable to expect the mobile IS to perform like a stationary one, so there is obviously a trade-off between performance and convenience to consider.

For our specific project no figures have been concretely established by POD for this, but they have given indications. In [28] the following question is posed:

> *"When and how can we conduct a complete ID verification in 10 seconds on a cold winter's night?"*
>
> From [28], translated from Norwegian

Taking this question at face value, considering the precedent set by comparable systems and also considering that we are designing a prototype (which will not strive for perfect optimization), we define the following requirements for performance:

- The MRZ should be scanned and recognized in approximately 2-3 seconds.
- The contactless inspection should be performed in approximately 10 seconds.

7.1.7 Limitation of scope

The overarching goal of the project is to demonstrate the feasibility of an Android app eMRTD IS, and to use this to identify limitations and advantages to such a system. Our chosen approach is the design and

implementation of a *rough* prototype, which means we are selectively disregarding many potential requirements to a real-world implementation of such a system. Specifically:

- As we only have access to a limited set of TD-3 type ePassports, and no other eMRTDs, we create our prototype solely with these in mind. That is, support for non-passport eMRTDs is disregarded.
- Security of the implementation is not a focus.
- Support for multiple models of devices and generations of software (i.e. Android version backwards compatibility) is not regarded.
- The user experience comes second to under-the-hood implementation.

7.2 User interface and activities

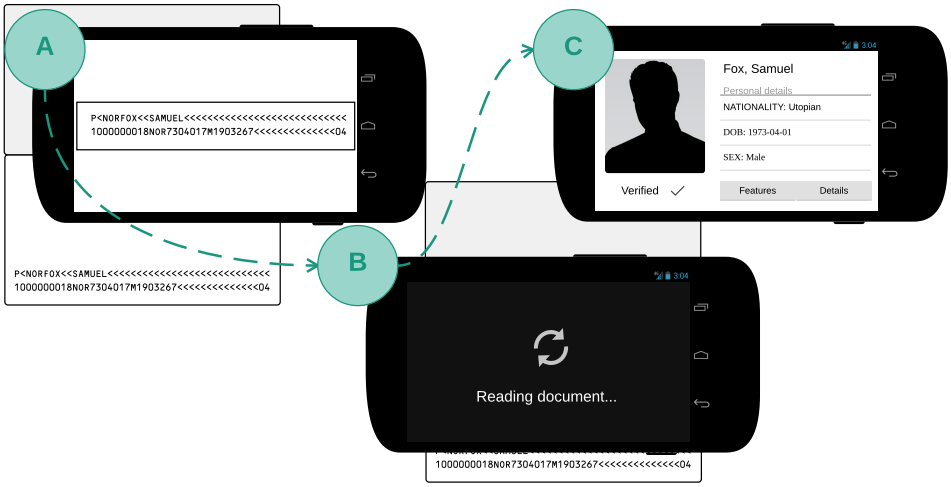


Figure 7.1: Overview of the app's main activity.

The main activity of our app is the inspection of eMRTDs. We define a three-step process for this, as illustrated in Figure 7.1:

A: Capture the MRZ using the phone camera.

B: Perform the contactless inspection.

C: Display the result of the inspection.

For effective inspection the app guides the user through these steps fluidly.

Additionally, we provide a manual entry method for step A, an activity for managing certificates and a configuration screen. Mock-ups of these are shown in Figure 7.2.

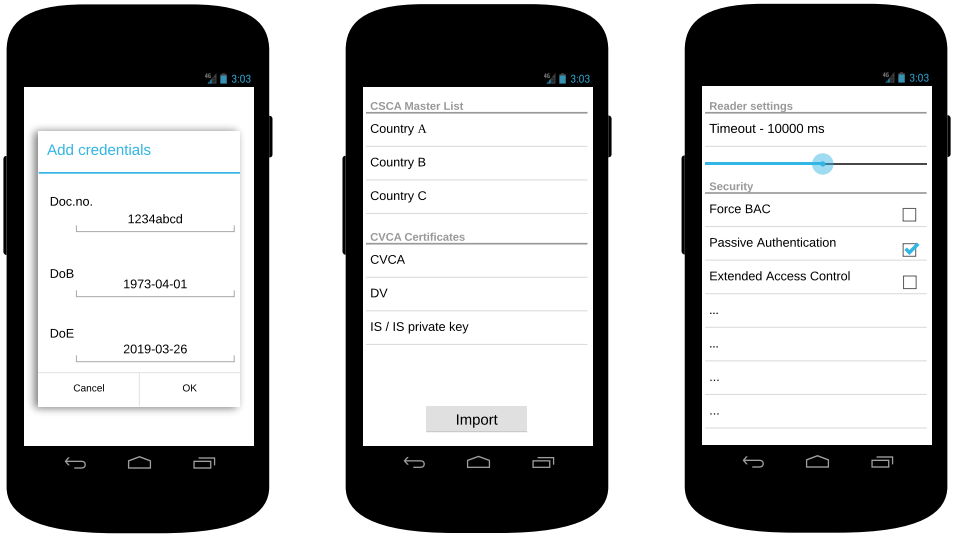


Figure 7.2: Manual credential entry (left), certificate management (middle) and configuration (right).

7.3 System architecture

Our system architecture is comprised of multiple modules, coupled with varying degrees of tightness. We choose this approach with the prospect of creating more scalable and maintainable software. Also, modularization should result in a set of reusable and repurposable components, making our software easier to modify and integrate into other systems (for ourselves and for third parties alike).

Figure 7.3 is a high-level overview of the architecture of our proposed design.

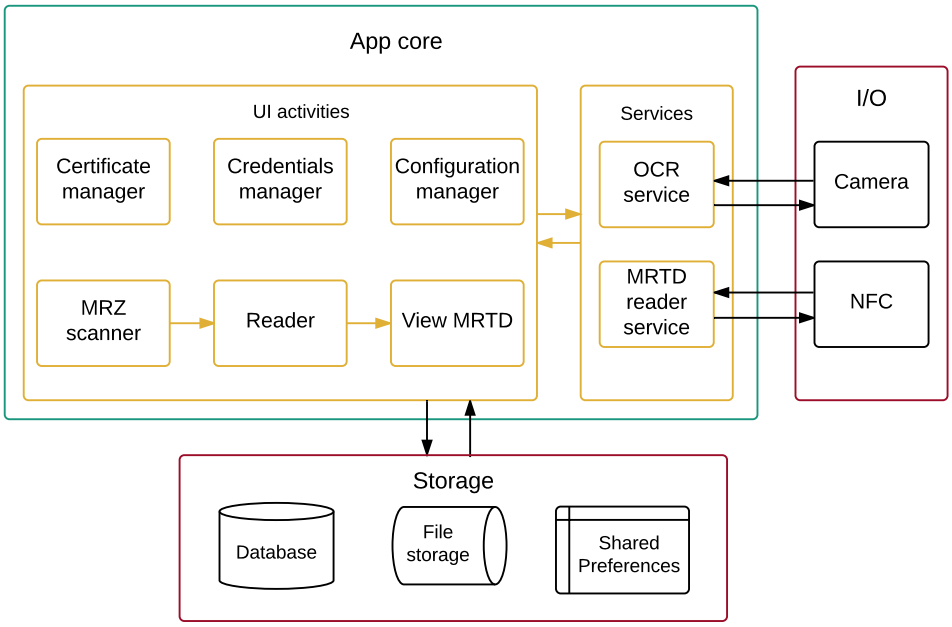


Figure 7.3: Overview of the system architecture and its components.

7.3.1 UI activities

Our user interface as shown in 7.2 consists of a range of single-purpose user activities or *screens*. From an architectural viewpoint these collectively make up the UI of our app. They also reflect the features specified in our requirements.

7.3.2 Application services

We design our core services, the OCR engine (for MRZ recognition) and the MRTD reader service (for NFC communication with the MRTD IC), as singleton components which exist outside the activity life cycle. This is necessary to keep complexity of the activities down and to allow the services to live on through life cycle events.

For example: we cannot have the MRTD reader process be killed due to a screen rotation, but rather want the reader to continue working and the reader activity to reattach itself upon being resumed.

7.3.3 Storage

Various forms of persistent storage is needed for multiple parts of our system. Namely:

- File storage for keeping installed certificates.
- Shared preferences for keeping app configuration.
- Database for keeping records for the credential manager.

As we are not interested in sharing state with external apps and, in fact, wish to keep stored data private we only use the *internal storage* mechanism.

Chapter 8

Implementation

In this chapter we introduce our prototype Android app which we have called *MRTD Inspector*.

The app is a realization of the system requirements and design proposed in Chapter 7 and its implementation constitutes the main body of practical work in this Master's project.

We start the chapter by reviewing and rationalizing our choices of hardware and software components for our prototype.

The next section presents an overview of key components and the choices made in their implementation.

This is followed by two sections giving special attention to parts of the implementation which proved particularly challenging, and also a short summary of specified features which were for various reasons not implemented.

Finally, a short presentation of MRTD Inspector from a user's perspective is given.

Figure 8.1: The MRTD Inspector app icon.

8.1 Hardware and software used

In the subsequent sections we give an overview of the hardware and third-party software components used in the implementation of the app.

8.1.1 Hardware

As stated in our system requirements we consider implementing support for a wide range of devices out of scope for our prototype. Therefore,

the hardware used in our work consists solely of a select few Android smartphone models. These phones have been used as our development targets throughout the project and are also the devices we have used to evaluate our prototype.

The devices are shown in Figure 8.2 and Figure 8.1 is a summary of their respective technical specifications.

Figure 8.2: The development devices. From left: Google Nexus 4, LG G3 and Google Nexus 5X. *(Images sourced from the manufacturers).*

Model	Year	OS	Chipset	NFC chip
LG Google Nexus 4	2012	5.1	Snapdragon S4 Pro	BCM20793
LG G3	2014	5.1	Snapdragon 801	NXP PN547
LG Google Nexus 5X	2015	6.0	Snapdragon 808	NXP PN548

Table 8.1: Technical specifications of the development devices.

From an application development perspective our small selection of devices are not all that different. They all run fairly recent versions of Android, allowing us to avoid considering backwards compatibility in our code base. In this sense, using a single device would actually suffice. This rings especially true when taking into consideration that the Android emulator (which comes with the Android SDK) allows us to spin up a virtual instance of any of a large range of devices on demand.

This is not the case for our app, however, as our core functionality is wholly dependent on the presence of an NFC interface, and we are testing our implementation with real (as opposed to emulated) ePassports.

In fact, having three different physical devices, of three different generations and, in particular, containing three different NFC controllers has shown to be a very useful tool to reveal limitations of our prototype.

Ultimately, the Nexus 5X has been our main development target for the majority of the project duration. This is simply due to our experiences showing that it offers more stable NFC performance than the other devices.

8.1.1.1 Hardware requirements

Though development has targeted the aforementioned devices specifically, Android apps are not device dependent and it should be more than possible to run MRTD Inspector on many other devices.

Table 8.2 lists the minimum technical requirements for running the prototype.

OS version	5.0 (Lollipop)
Display resolution	1280 x 720
Camera resolution	1280 x 720
Required features	Camera2 API, NFC

Table 8.2: Minimum specifications required for MRTD Inspector.

8.1.2 Third-party software

As is the case for most non-trivial software projects we employ a large selection of third-party libraries and components in our implementation. Some are used for peripheral purposes, such as pre-made user interface components and utility libraries, some are essential framework components and some are fundamental to the implementation of MRTD inspection.

The following sections present the third-party libraries which are fundamental to our implementation, what they are used for and how. It is as such not a complete list of third party components.

8.1.2.1 JMRTD

JMRTD is an open source, pure Java library which provides several MRTD related components, including an eMRTD JavaCard applet, libraries for encoding and decoding biometric image formats and a host side API for accessing eMRTDs. A more detailed overview of the JMRTD project and library can be found in 6.2.1.1.

Our entire implementation of eMRTD contactless inspection builds directly on top of the JMRTD 0.5.6 API, and the so-called *host-side* API it is used to handle low level connection, security protocols and reading of data.

In addition to binding to this version of the library we have backported a select few classes relating to certificate handling from version 0.5.0 (which have been removed from more recent releases).

8.1.2.2 Java security, Bouncy Castle and Spongy Castle

The security APIs built into Java (the `java.security` package) are made to be modular and portable across a range of devices and platforms. As

such, it uses the *Provider* framework to facilitate the use of platform- or application-specific implementations.

For example, when wishing to use a specific cipher the code does not directly bind to an implementation, but rather resolves one through the `javax.crypto.Cipher` API, like so:

```
Cipher cipher
    = Cipher.getInstance("AES/CBC/PKCS5Padding");
```

Under the hood of this operation the installed security providers are searched for an implementation of the cipher. If a suitable implementation is found the object is instantiated and returned, and if not an `Exception` is thrown.

As stated, the list of available security providers, and what cryptographic algorithms they support, varies with platforms and devices. Android is no exception to this.

Android ships with a small selection of first party providers, offering a rather limited list of supported algorithms.

For our purposes it is therefore necessary to install a third-party security provider which offer implementations of the algorithms used in PACE, BAC and so on.

In fact, JMRTD already relies on *Bouncy Castle* [29], which is a third-party cryptography library and Java security provider.

However, one of the bundled security providers in Android is, unfortunately, a stripped-down version of Bouncy Castle. The result is a namespace conflict causing JMRTD and Bouncy Castle itself to malfunction in certain circumstances when run on Android.

As the Bouncy Castle namespace conflict is a well-known issue on Android there is also a well-known solution: Spongy Castle [30]. This is a repackaging of Bouncy Castle so as to avoid namespace conflicts by moving packages from `org.bouncycastle.*` to `org.spongycastle.*`. We use the Spongy Castle security provider in our app, as well a selection of the supportive APIs, such as the `org.spongycastle.asn1` package.

Because the compiled JMRTD library distribution (jar-file) already ships with `org.bouncycastle.*`, however, the conflict is not solved by using Spongy Castle alone, and modification[1]. of the JMRTD jar-file is necessary before importing it.

We do so by using the *Jar Jar Links* [31] command-line utility to switch the package name within the compiled library:

[1] Alternatively one could modify the source code and re-compile the library.

```
$ java -jar jarjar.jar process org.bouncycastle.** org.
    spongycastle.@1 jmrtd-0.5.6.jar jmrtd-spongy-0.5.6.jar
```

As a result we use Spongy Castle and so does the JMRTD jar used, meaning we avoid the aforementioned namespace conflict.

8.1.2.3 EJBCA cert-cvc

EJBCA is an open source, enterprise grade Certificate Authority which includes support for the EAC PKI. More details on the project is provided in 6.2.1.5.

One of the components of EJBCA is the `cert-cvc` library. It comprises a full Java implementation of Card Verifiable Certificates (CVC) as well as a selection of supportive CVC utilities.

Our implementation relies on `cert-cvc` for parsing and format conversion of CVCs.

8.1.2.4 Tesseract OCR and tess-two

Tesseract OCR is an open source OCR engine. It was originally developed by Hewlett-Packard in the 1980's to 1990's, open sourced in 2005, and development is now sponsored by Google (since 2006). It is widely considered to be the most accurate open source OCR engine [32].

At the core of Tesseract is `libtesseract`, which is a cross-platform library written i C/C++, but API wrappers are available for many programming languages, including Python, PHP and Java.

Implementing character recognition with Tesseract requires *training* the engine to suit the particular use case (typeface, language, printing media). This is done through a multi-step process which comprises feeding carefully crafted sample data to the training tools, producing a range of *training files* to extract the relevant parameters and ultimately producing what is known as a *traineddata* file.

A large selection of pre-made traineddata files are available from the official Tesseract repository [33], and using tried and true pre-trained data is the recommended approach for recognition of natural language such as English text. That said, if the use case is more specific better results could be achieved through performing the aforementioned training process.

Tesseract is used for recognizing and extracting the MRZ from camera images in our implementation.

However, we do not use `libtesseract` directly, but an Android-specific wrapper library known as *tess-two*[34].

The tess-two package provides an Android-ready OCR setup which includes the native libraries (compiled for the Android platform) and a thin-wrapper Java API. Additionally, the *Leptonica* image processing library [35] is included in the package.

8.1.2.5 RxJava

RxJava is a Java implementation of *Reactive Extensions* (ReactiveX, RX), developed by Netflix.

ReactiveX is dubbed *"an API for asynchronous programming with observable streams"* [36], which in simpler terms means it's a toolkit for writing *reactive* code: a programming paradigm oriented around *data flows* and event propagation.

RxJava in used throughout MRTD Inspector to facilitate an event-driven architecture and to ease the handling of background threads for database access, NFC I/O and other long-running tasks.

8.2 Code overview

The code base of MRTD Inspector consists of four modules:

- `common`: Contains utility classes, generic UI components and helper classes.
- `ocr`: The core OCR implementation. Defines a simple API which encapsulates the initialization of the OCR context, performing repeated image recognition and fetching the results.
- `mrtd`: Comprises everything to do with reading MRTDs, including the `MrtdReader` which implements our contactless inspection procedure, supportive model classes and certificate management code.
- `app`: The actual Android app which contains all user interfaces (Activities), service classes, application specific business logic and models.

The modules depend on each other, and as can be seen in Figure 8.3 the dependencies are one-way only. The result is a code base made to promote component reuse. For example, one could extract the `mrtd` (and `common`) module and use it in another project without changing the code or backporting code from `app`.

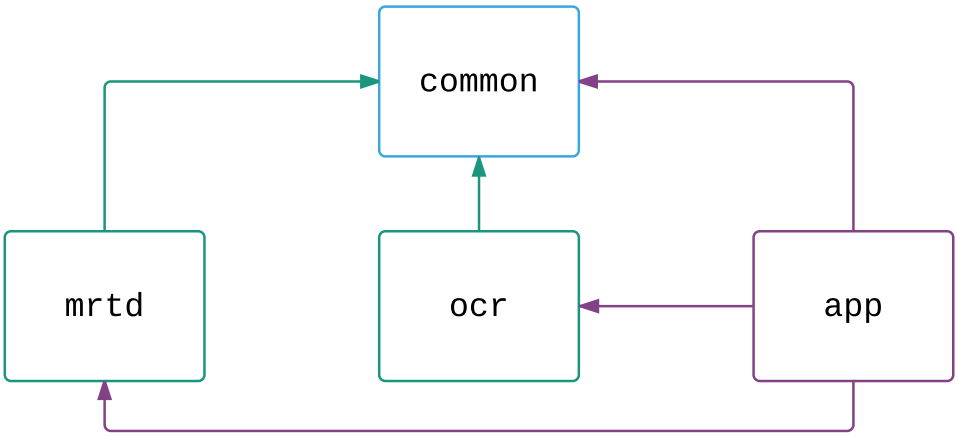


Figure 8.3: Dependency graph of the source code modules.

8.2.1 MVP design pattern

The application code of MRTD Inspector (that is, the `app` itself) is organized using the *Model-View-Presenter* (MVP) design pattern.

MVP is a derivative of *Model-View-Controller* (MVC), and is essentially a design pattern which enforces and promotes *separation of concerns* between the user interface, presentation logic and model. MVP is illustrated in Figure 8.4.

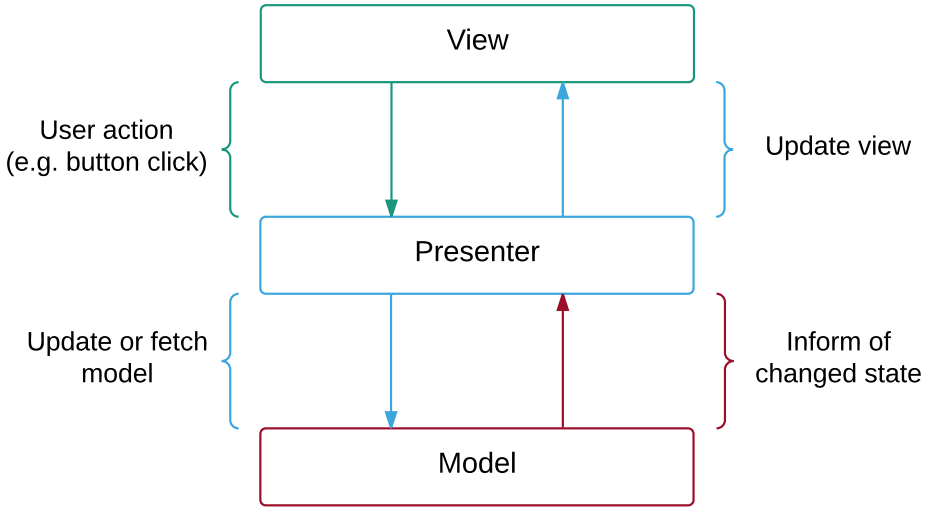


Figure 8.4: The Model-View-Presenter design pattern.

Figure 8.5 shows the structure of our Android application code with the MVP pattern applied. Note that the activities and presenters are volatile (i.e. they will get torn down and re-instated upon certain life cycle events) whilst the models and services are persistent and live for the duration of the application life cycle (from instantiation to app shutdown).

Due to this the presenter must dynamically bind and unbind the models and services. Long-running tasks running in the application services will

live through activity recreation or shutdown, and in order to resume an operation (such as running MRTD inspection) in the user interface the presenter must gracefully rebind services and update the UI state accordingly.

For many parts of the app these tasks are handled using RxJava, which allows weak bindings between the presenters and service layer components, avoiding thread- and memory leaks.

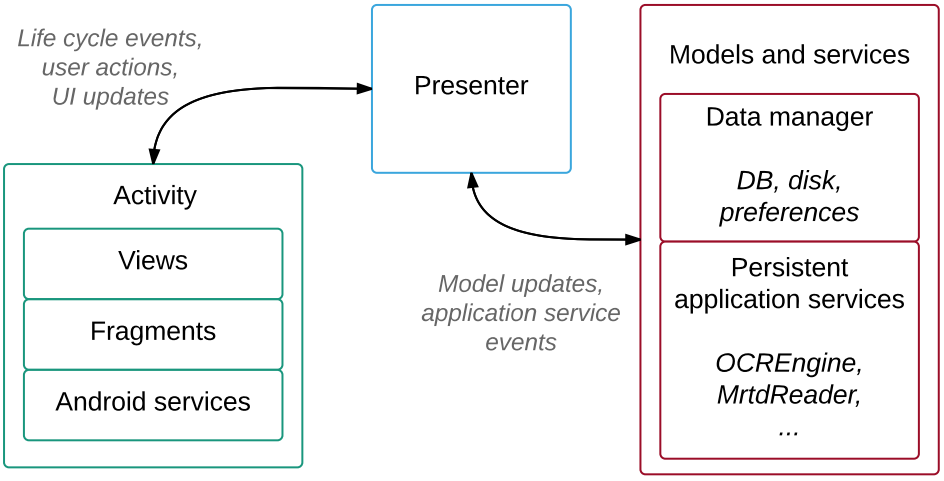


Figure 8.5: MVP in the Android app.

8.3 MRZ scanner

Optically reading the MRZ is one of two core components of the inspection workflow specified in Chapter 7.

In the following sections, key points of the MRZ OCR implementation are surveyed: first we give an overview of the internal API of the `ocr` module, followed by the MRZ OCR procedure and at last a section on tuning the performance of our OCR implementation.

8.3.1 OcrEngine

The `OcrEngine` class in the `ocr` module constitutes the primary API for use by external classes.

It encapsulates configuring and initializing the Tesseract context as well as offering a simple interface for performing the OCR decoding of input images. The public API of `OcrEngine.java` is shown in Figure 8.6.

```
// Constructor. Starts init in background thread on
    construct
public OcrEngine(String tessdataBasePath, String
    traineddataFilename, String traineddataLanguage) {...}

// Stop and tear down the Tesseract context
public void stop(){...}

// Clear the current Tesseract context
public void clear(){...}

// Synchronously do OCR on the input OcrRequest
public IOcrResult
    doOcrDecode(OcrRequest ocrRequest) {...}
```

Figure 8.6: The public API of OcrEngine.java

The `OcrRequest` class encapsulates the input data, including the raw image bytes and a *framing rectangle*, which marks the position of the rectangle shown in the viewfinder and is used to crop out unnecessary image data.

```
public class OcrRequest {
    final Rect framingRect;
    final byte[] data;
    final int width, height;

    public OcrRequest(Rect framingRect, byte[] data, int
        width, int height) {...}
}
```

`IOcrResult` is a marker interface implemented by two concrete classes: `OcrResult` and `OcrResultFailure`.

The first of these is a fairly complex model class which contains the decoded text, the processed image which was used for recognition internally and a selection of other meta-data from the decoding such as the time used and the detected character boxes.

The second is used to represent decode failure (when no text was detected in the image data) and simply contains the time spent on the failed operation.

8.3.2 Continuous OCR decoding

The MRZ scanning is implemented as a continuous process, running in real-time as the user presents the MRTD data page to the camera of the device. Simultaneously, visual feedback allows the user to gauge if the proper image is being captured or if the camera position needs adjustment.

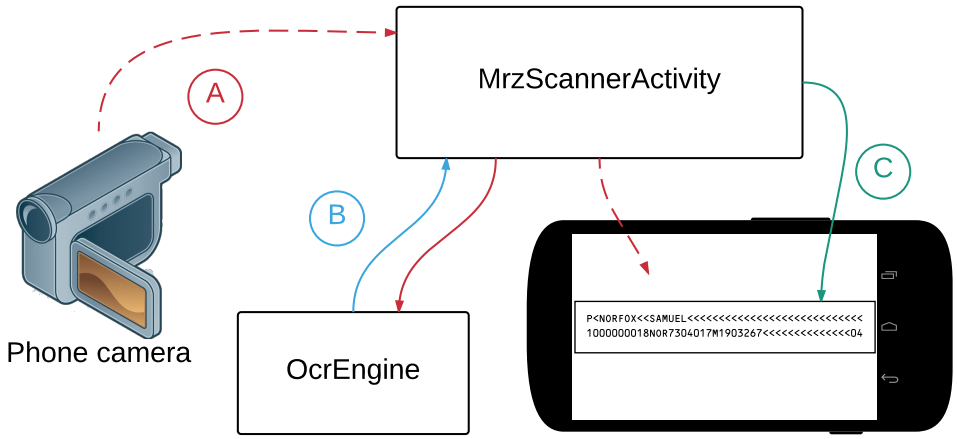


Figure 8.7: The MRZ scanner continuous OCR decoding. The dashed lines denote the live video feed from the camera to the activity, and ultimately the view.

Figure 8.7 shows the continuous process, orchestrated by `MrzScannerActivity`. The labels mark the steps of the process:

A: The camera provides a real-time image stream to the activity. The stream is fed through to the preview view, providing a video preview for the user. This runs independently of **B** and **C**.

B: The activity picks a single frame from the stream, sends it to the `OcrEngine` for OCR decoding on a background thread, blocking until the result is returned.

C: If the result is an `OcrResultFailure` the activity re-starts the process from **B**. However, if the result is an `OcrResult` the activity checks if the returned text is a valid MRZ. If it is, the process is exited and the MRZ is returned. If not, the processed image is extracted from the `OcrResult` and overlaid on the viewfinder, providing the user with an indication of what the OCR engine *sees*. Following this the process loops back to **B** (and continues until a valid MRZ is found or the user exits the activity).

The flowchart in Figure 8.8 depicts this capture-decode-evaluate loop.

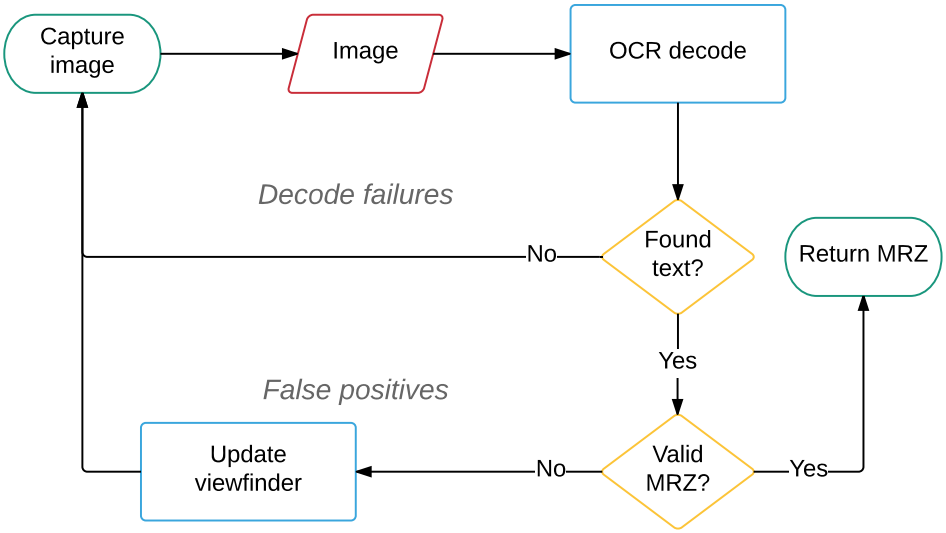


Figure 8.8: Flowchart of the OCR procedure.

8.3.3 Tuning Tesseract

As stated, our OCR implementation is reliant on `libtesseract`, encapsulated within the `OcrEngine` of the `ocr` module.

Tesseract is made to be a flexible API which can be employed for a wide range of OCR uses, and is therefore highly tunable. In fact, though there are tried-and-true ways to use it for the most popular applications, such as recognizing natural language text in a well-known font, it is necessary to tune it specifically for the intended use in order to achieve optimal performance and accuracy. Our requirements for these properties are:

- **Accuracy:** the OCR engine should be tuned to the exact characteristics of the MRZ, avoiding false positives which are "illegal" . This includes searching only for the MRZ character subset printed in the OCR-B typeface, and not looking for structured sentences or words.
- **Performance:** our scanning procedure runs in real-time and the execution time of the OCR decoding must therefore be reasonably short. Limiting the perceived delay between adjustment of the camera position and getting visual feedback will mitigate *lag* and make the scanner easier to use, thus producing better results. Remember also that this is running on a resource-limited mobile device, meaning there are no CPU cycles to waste.

Three steps have been taken in order to tune Tesseract for these properties: training, tuning the configuration and optimizing the input images.

8.3.3.1 Training

Recall that Tesseract is dependent on a *traineddata* file which is more or less tailored to the specific use case.

In the early stages of development we used the pre-trained[2] `eng.tessdata`, but our observations showed that this was not satisfactory for MRZ scanning: the results were inaccurate, recognition time was too high and most recognition attempts returned failure.

As a step to alleviate these issues we elected to create a traineddata file tailored for the task, namely:

- Specifically created for the OCR-B typeface only.
- Trained only for the 37-character subset of the MRZ.
- Trained with MRZ-type data (that is, continuous blocks of text) as opposed to natural language (for which word recognition is desirable).

To do the training we follow the process as described in the official documentation [37]. This is, in short:

- Create test data.
- Run the training tools on the test data.
- Manually correct the output to perfectly reflect the input.
- Manually tune parameters to fit the use case.
- Assemble the `traineddata` file.

Following is a more detailed description of the process used to create the MRZ-specific `traineddata` file.

8.3.3.1.1 Creating `mrz.traineddata`

First, we prepare a text file of MRZ-like data. The file is constructed as one continuous line of text containing MRZ-like data and is exclusively using the MRZ character set. An excerpt is shown in Figure 8.9.

```
WEBER<<HILLARY<<<<<<<<4008143884GBR7709265M1601013<<
AAWWWZZZQQQRRJKLMPASDEWW<<06P<GBRMASTERSON<<FREDERIC
K<<<<<<<<JJJJJJJAADSDSDWJJJ<<9872846363GBR3011227M16
01013<<<<<<<<<<<<<<00P<GBRALLGROVE<<MAUD<<<<<<<<<<05
```

Figure 8.9: Excerpt from mrz_training_text.txt. The file is 2399 characters.

From this we generate a training image by using the `text2image` tool, specifying the OCR-B typeface:

[2] Available for free from the official Tesseract repository at [33].

```
$ text2image --text=mrz_training_text.txt --outputbase=
    mrz.ocrb.exp0 --font='OcrB' --fonts_dir=/fonts/dir
```

The result is a training image, shown in Figure 8.10. The image is made to reflect the characteristics of a scanned piece of paper with the typical OCR image pre-processing applied (aligned, thresholded, binarized). That is, not as clean as a purely digital rendition, but more like an optimized photo.

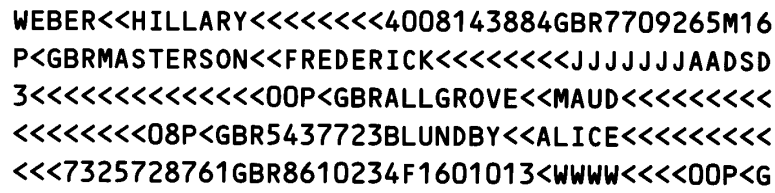


Figure 8.10: A section of the MRZ training image.

This image is then fed into Tesseract to produce a `box file`:

```
$ tesseract mrz.ocrb.exp0.tif mrz.ocrb.exp0 batch.nochop
    makebox
```

The resulting file, `mrz.ocrb.exp0.box`, looks like this:

```
...
2 1365 2721 1388 2758 0
G 1394 2721 1418 2755 0
B 1424 2721 1449 2755 0
R 1456 2721 1478 2755 0
7 1484 2720 1509 2757 0
3 1514 2720 1538 2757 0
0 1544 2720 1569 2757 0
8 1574 2720 1599 2757 0
...
```

It contains one line per recognized character from the training image.

The next and most crucial step in the process is to correct the box file to perfectly reflect the corresponding characters in the input text file. Often similar characters are confused (e.g. `I` and `1`) by Tesseract, and these must be corrected.

This is a tedious process which is most often done with dedicated editor software or through script automation[3].

The next few steps in the process comprise running the corrected box file through the `unicharset_extractor` tool to extract the character set, creating a `font_properties` file with the relevant parameters for the OCR-B typeface and running the `shapeclustering`, `mftraining` and `cntraining` tools.

[3]We used both, in fact.

The last file needed is the `unicharambigs` file which contains a set of structured options to tell Tesseract about any known character ambiguities such as "`I`" and "`1`" or "`O`" and "`0`".

The result of performing all of these steps is the following list of files:

- mrz.inttemp
- mrz.normproto
- mrz.pffmtable
- mrz.shapetable
- mrz.unicharambigs
- mrz.unicharset

These files are combined using the `combine_tessdata` tool, which produces the `mrz.traineddata` file:

```
$ combine_tessdata mrz.
```

The traineddata file is 310 kb, which is considerably smaller than the pre-trained `eng.traineddata` at 20.9 MB.

Also, running Tesseract (on the command line) using the new trained-data on a small selection of MRZ images demonstrates that it is reasonably accurate given the input images are properly pre-processed (thresholded, monochrome)[4].

8.3.3.2 Tuning the configuration

The Tesseract documentation at [38] lists 648 configurable parameters for Tesseract 3.02. However, many do not apply to our needs and we only need to configure a small subset.

Through researching several resources and much trial-and-error we have identified a small set of parameters and values which positively affect the OCR performance of our implementation, they are shown in Table 8.3.

[4]We did this pre-processing manually using the GIMP image editor package in our preliminary testing.

Parameter	Value	Description
`tessedit_char_whitelist`	MRZ subset	Whitelist of characters to recognize
`tessedit_pageseg_mode`	1 (auto + OSD)	Page segmentation mode
`tessedit_unrej_any_wd`	1 (true)	Don't bother with word plausability?
`tessedit_enable_doc_dict`	0 (false)	Add words to the dictionary?
`load_system_dawg`	0 (false)	Load DAWG?
`load_freq_dawg`	0 (false)	Load frequent word DAWG?
`tessedit_tess_adaptation_mode`	0 (disabled)	Adaptation mode.

Table 8.3: The Tesseract configuration. *DAWG* is short for *Directed Acyclic Word Graph*. It is used as a fast-lookup dictionary by Tesseract. The DAWG parameters and values effectively disable the dictionary.

The effects of these settings are that we are:

- Limiting recognition to the MRZ subset.
- Setting the *Page Segmentation Mode* to automatic with script detection, which was shown through experimentation to offer the most accurate results.
- Disabling dictionaries altogether. They are not relevant to our use as we are not detecting natural language and do require substantial resources.
- Disabling adaptation. Our Tesseract context is used once and then cleared, thus adaptation between different runs is only a waste of resources.

There are more than likely more parameters we could tune to optimize performance and accuracy. Unfortunately, the Tesseract documentation is not very deep on this subject matter and considerable effort would have to be put in.

8.3.3.3 Input optimizations

Through early experimentation and learnings from reading the available documentation, it is clear that the quality and characteristics of the input images has great effect on both performance and accuracy.

Two steps need to be taken to optimize OCR conditions for the input data:

- Ensure the original input image is of sufficiently high resolution, is well focused, well lit and well aligned.
- Remove any noise or unnecessary information from the image.

The image resolution is chosen when doing the camera setup in `MrzScannerActivity`. The concrete resolutions which are available is dependent on the device, and we must make sure to pick a suitable one.

This is not necessarily the highest possible resolution, however. Our main development device, the Google Nexus 5X, has a maximum camera resolution of 3840 x 2160 when requesting a 16:9 aspect ratio image. Early testing showed that images of this size severely affects the execution time of the OCR decode procedure, almost tripling it, and causing unacceptable delays and lag. Clearly, there is a trade-off between high quality input and the resources used for capturing and processing the image.

Experimentation with different image resolutions on the aforementioned device revealed that 1920 x 1080 images performed best while still providing reasonably high quality images, whilst lower resolution images gave little performance improvement but severely affected accuracy.

Coincidentally the optimum resolution of 1920 x 1080 is the maximum *guaranteed* to be available on any Android device which supports the *Camera2* API[5].

As stated the alignment, lighting and focus of the image are important as well. These factors are, however, not trivial to mitigate in software and we rely on the user to correctly light the MRZ and align the camera. We do provide a viewfinder in the MRZ scanner to aid in this, as well as an overlay which shows the last decoded image. There is also an option to switch on/off the camera torch (flash) in order to provide a light source in very low-light conditions[6].

Once the image is captured a few steps of preprocessing is done to remove superfluous information, easing the task of the OCR engine.

Tesseract only works with greyscale images, so any color information can safely be removed from the image. The images are captured by `MrzScannerActivity` in the *YUV 420* format. This is a multi-plane *YCbCr* encoding, allowing the luminance (Y) channel to be easily extracted, resulting in a greyscale image.

In addition to this we help Tesseract along by cropping the captured image to exactly reflect the region of the viewfinder rectangle. Doing so ensures that what the OCR engine *sees* is what the user sees, and that no resources are wasted attempting to decode non-MRZ parts of the image. The last step is handled by Tesseract itself. It entails performing *adaptive thresholding* to create a *binarized* image[7]. That is, an image of only binary value pixels.

This simple image processing pipeline is shown in figure 8.11.

[5]For simplicity we support only the newer Camera2 API in our prototype.

[6]Though reflective glare is a potential issue under these circumstances.

[7]The binarized image is also overlaid the viewfinder to guide the user in finding the right camera position.

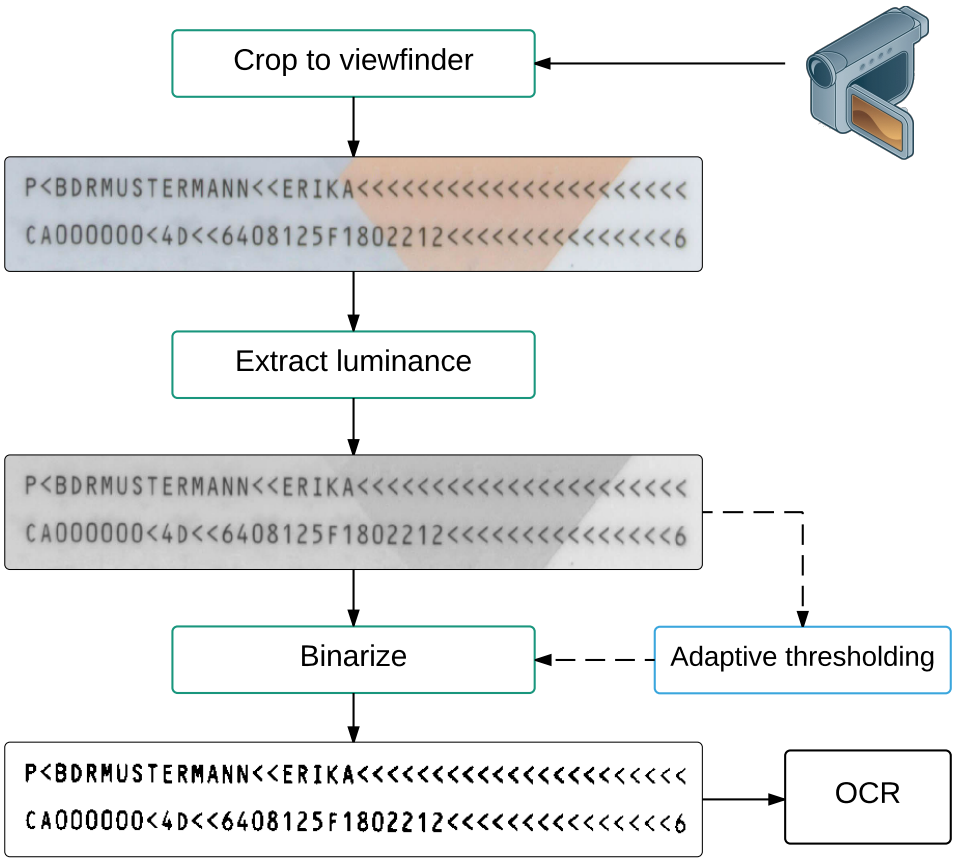


Figure 8.11: The MRZ image processing pipeline.

8.4 Contactless inspection

In the following sections we review the implementation of contactless inspection.

First we look at the APIs provided in the `mrtd` package and in particular the `MrtdReader` and its supportive classes.

Next, we go into the details of the actual eMRTD inspection procedure implementation, including authentication, reading of data and validation.

8.4.1 MrtdReader

The implementation of contactless inspection resides in the `mrtd` module. It comprises all inspection logic as well as supportive classes such as MRTD-specific models and utility classes.

`MrtdReader` is the central component of the module and contains all the core logic for performing contactless inspection.

It exposes a very simple API, seen in Figure 8.12:

```
// Public constructor
public MrtdReader(
    MrtdReaderListener listener,
    TerminalCertificateStore terminalCertificateStore,
    TerminalKeyStoreWrapper keyStoreWrapper,
    KeyStore cscsKeyStore) {...}

// Connect to tag and start reading
public void read(Tag tag, MrtdCredential credential,
    MrtdReaderConfiguration configuration) {...}

// True if reading is in progress, false otherwise
public boolean isReading() {...}

// Force close the reader thread
public void abortRead() {...}

// Checks if tag is currently connectable (physically
    present at the NFC interface)
public boolean hasConnectivity(Tag tag) {
```

Figure 8.12: The public API of MrtdReader.

The constructor requires an instance of `MrtdReaderListener`, which defines a callback API for reader events. An excerpt is shown in Figure 8.13.

```
void onReadStart();
void onReadFinished(Mrtd mrtd);
void onError(Throwable error);

void onBeforeBac();
void onBacResult(AuthenticationResult
    authenticationResult);

void onBeforePace();
void onPaceResult(AuthenticationResult
    authenticationResult);

void onBeforeEac();
void onEacResult(AuthenticationResult
    authenticationResult);

void onBeforeReadDg1();
void onReadDg1();

void onBeforeReadDg3();
void onReadDg3();
```

Figure 8.13: Excerpt from the MrtdReaderListener interface.

On each of the reader events, such as performing an authentication pro-

tocol, reading a data group or the read process failing, the corresponding callback method is called on the supplied `MrtdReaderListener` implementation.

Due to this event oriented design, the consumer of the `MrtdReader` API can selectively monitor the different events of the inspection process, handling them in a manner appropriate for the specific application. For instance, one might want to use the reader API for a back-end process which only cares about the end result, and could thus implement a listener handling the `onReadFinished(..)` callback only. In the case of a UI application, on the other hand, the events could be bound to UI updates in order to keep the user informed of the reader status.

The latter is the case for MRTD Inspector, where the `ReadProgessPresenter` listens to the events, updating the UI in `ReadProgessActivity` as the inspection progresses.

The `read(..)` method of `MrtdReader` is the entry point for performing contactless inspection. It is called with three arguments:

- **Tag:** the Android abstraction for an NFC tag which has been discovered.
- **MrtdCredential:** a model object which encapsulates the MRZ-derived credentials needed to form the BAC or PACE static keys: document number, date of birth and date of expiry.
- **MrtdReaderConfiguration:** a model object which contains values for all of the configurable parameters of the reader. This includes the reader timeout, a list of the DGs to read and settings for which protocols to use or exclude.

Once `read(..)` is called, `MrtdReader` spawns an internal worker thread, executes the procedure as dictated by the supplied configuration, calls the appropriate callback methods whilst doing so and at last returns an instance of the `Mrtd` object in the final `onReadFinished(..)` callback. This architecture is illustrated in Figure 8.14.

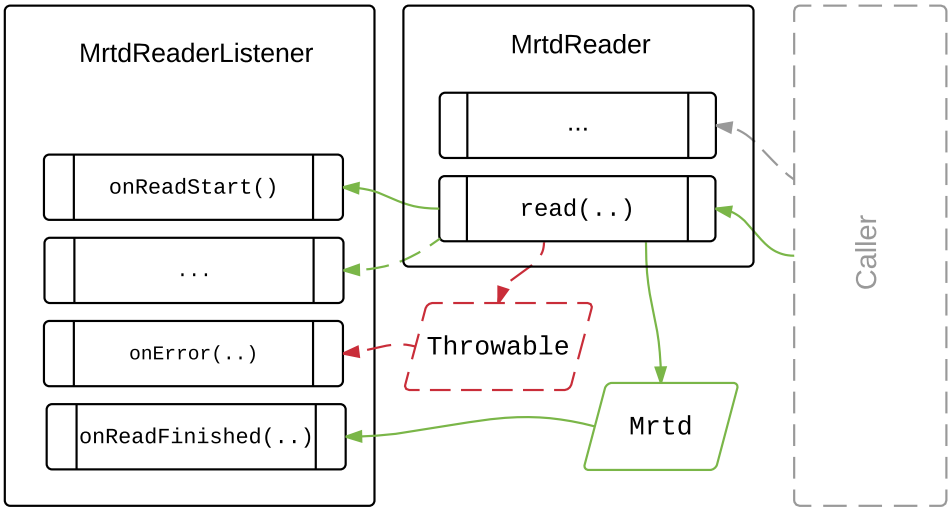


Figure 8.14: The MrtdReader callback architecture.

8.4.2 Mrtd model class

The underlying library we use to perform the protocols and operations of the inspection, JMRTD, defines a single encapsulation of the eMRTD Logical Data Structure in the `LDS` class.

This class is heavily integrated into the JMRTD `PassportService` API, and a single, global and mutable instance is more or less required to perform most parts of the inspection. This because the object is used as the base holder of all data read, and also holds references to the underlying `InputStream` instances (which bind to the NFC subsystem).

Due to this rather inelegant and inflexible design, an instance of the `LDS` class is a potential minefield of memory, stream and thread leaks. Also, it cannot be serialized and thus cannot be passed through our activities[8]. In short: it simply doesn't fit well with out application architecture. Also, it's not good practice to expose third-party packages in APIs, potentially making our *mrtd* harder to use.

Answering to these shortcomings of JMRTD we have elected to implement our own LDS abstraction in the `Mrtd` class.

It is not a complete implementation of the LDS by any means, as that is strictly not needed. It is rather a simplified container for the data (DGs, security data) we are extracting and wish to expose in the user interface.

It is implemented as an immutable object, meaning its state is guaranteed after construction. This is good practice for static data: once the object has been returned from the reader it cannot and will not change. It also implements the Android `Parcelable` interface, meaning it can be easily and efficiently serialized using the native Android serialization format.

A class diagram is shown in Figure 8.15.

[8]Passing application data between activities is tricky due to the separation between them and requires objects to be serializable or *Parcelable*. Alternatively one could pass objects in shared memory, but it is considered bad practice.

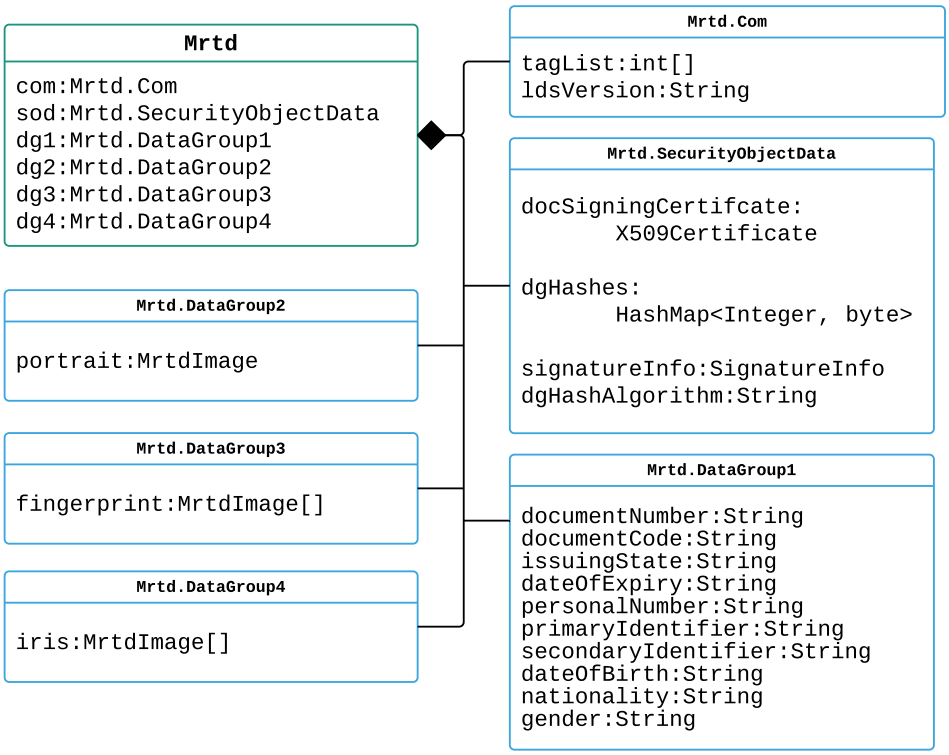


Figure 8.15: The `Mrtd` class. All of the contained data group objects are defined as public inner classes of `Mrtd`.

8.4.3 Inspection

As stated, the `MrtdReader` relies on JMRTD to perform the various parts of the inspection procedure. It does so by using an instance of the JMRTD-provided class `PassportService`. This object encapsulates the connection to the eMRTD tag and exposes methods to perform authentication, read data and manage the connection.

The `internalDoRead(..)` method of `MrtdReader` initializes the `PassportService` and Tag[9] connection as seen in Figure 8.16.

[9] *Tag* is the Android encapsulation of a connected NFC tag.

```
// IsoDep: encapsulation of ISO-14443 tag
IsoDep isoDep = IsoDep.get(tag);
isoDep.setTimeout(configuration.getReaderTimeout());

try {
  IsoDepCardService idcService
      = new IsoDepCardService(isoDep);
  mPassportService = new PassportService(idcService);
  mPassportService.open();
} catch (CardServiceException e) {
  /* Throw exception and exit */
}
```

Figure 8.16: Initialization of PassportService. Simplified for brevity.

8.4.3.1 Authentication

Once the `PassportService` is set up, the next step is to start the initial authentication procedure. This comprises two initial steps:

- Checking if PACE is supported and that the app configuration allows using it (remember, there is a toggle for forcing BAC).
- Performing PACE or continuing without.

If PACE is performed, `onPaceResult(..)` is called on the registered `MrtdReaderListener`, containing the status of the authentication:

```
try {
  doPaceAuth(credential, paceInfo);
} catch (MrtdReaderException e) { /* PACE failed */
  mListener.onPaceResult(   /* Report and exit */
    new FailedAuthenticationResult(
        AuthMethod.PACE, e, true
    );
    return;
  );
}
```

Once either PACE has been performed or it has been established to use BAC the corresponding eMRTD application can be selected:

```
// isPaceAuth == true  ==> PACE applet
//               false ==> BAC applet
mPassportService.sendSelectApplet(isPaceAuth);
```

If the BAC applet was selected, BAC authentication happens next (in a similar fashion to PACE). If the PACE applet was selected, however, authentication is now finished.

Once either BAC or PACE has been completed, `PassportService` switches to Secure Messaging using the derived session keys and the less sensitive data groups are now accessible.

8.4.3.2 Reading the less sensitive data

The read operation is similar for all DGs. The internal method `readLDSFileOrThrow(..)` is a generic method used internally for reading all LDS DGs:

```
private
<T extends LDSFile> T readLDSFileOrThrow(
    LDS lds, final short fid, Class<T> clazz
    ) throws MrtdReaderException {
  T dgFile;

  try (
    CardFileInputStream in = mPassportService.
        getInputStream(fid)
  ) {
    lds.add(fid, in, in.getLength());
    dgFile = clazz.cast(lds.getFile(fid));
  } catch (CardServiceException | IOException e) {
  throw new MrtdReaderException(
    "Failed to add and read DataGroup with FID " +
      fid + " of Class " + clazz.getName(), e);
  }
  return dgFile;
}
```

As can be seen, the `InputStream` for the DG is first resolved using the `PassportService`. It is then added to the `LDS` object, which internally performs the underlying `READ_BINARY` operation with the contactless IC and stores the data. The returned `LDSFile` subclass object is a Java representation of the DG, populated by said data.

First, the `EF.COM` and $\texttt{EF.SO}_\texttt{d}$ DGs are read, followed by the mandatory DGs 1 and 2.

Reading DG2 is a bit more complicated as it comprises extracting the encoded face image. The aforementioned generic method is still used to fetch the `DG2File` object, but a few more steps must be taken to get the actual image from the contained CBEFF-compliant data structure. The `readDG2(..)` method does the following:

```
List<FaceInfo> faceInfos = dg2File.getFaceInfos();

if (faceInfos == null || faceInfos.isEmpty()) {
  throw new MrtdReaderException(/* .. */);
}

FaceInfo faceInfo = faceInfos.get(0);
List<FaceImageInfo> faceImageInfos = faceInfo.
    getFaceImageInfos();

if (faceImageInfos == null || faceImageInfos.isEmpty()) {
  throw new MrtdReaderException(/* .. */);
}

FaceImageInfo faceImageInfo = faceImageInfos.get(0);

MrtdImage image;

try {
  image = extractImage(faceImageInfo);
} catch (IOException e) {
  throw new MrtdReaderException(/* .. */);
}
```

As can be seen the image data is resolved from the nested `FaceInfo` and `FaceImageInfo` objects. The `MrtdImage` class is a simple wrapper for the raw data and MIME type[10] of the image and decoding the image data into bitmap format is handled elsewhere (UI layer).

8.4.3.3 EAC and sensitive data groups

Next, the procedure conditionally performs Extended Access Control. The following conditions must be met for EAC to be executed:

- The `EF.COM` `tagList` contains `EF.DG14`[11], `EF.CVCA` and at least one of the sensitive data groups (3, 4).
- The `TerminalCertificateStore` contains a certificate chain and IS private key which matches the CVCA.
- EAC is not disabled by the user.

First, Chip Authentication is performed by the `doChipAuthentication(..)` method. It first resolves a suitable CA cipher suite to use by performing the following steps:

1. Resolve the `ChipAuthenticationPublicKeyInfos` from `EF.DG14`.

[10] Two-part media identifier in the *type/encoding* format. Example: *image/jpeg*.
[11] Holds the Chip Authentication public key.

2. Select a CA public key from `ChipAuthenticationPublicKeyInfos` for an algorithm suite which is supported[12].

Once a suitable CA public key is select, the `doCA(..)` method of `PassportService` is executed with the selected key OID[13] and public key.

If successful, Secure Messaging is restarted with the selected cipher parameters and execution of Terminal Authentication starts in the `doTerminalAuthentication(..)` method.

Before TA can be initiated with the IC, several parameters are resolved.

First the `EF.CVCA` is read and the `CAReference` is extracted. This is then used to search the `TerminalCertificateStore` for a matching chain of Card Verifiable Certificates.

If no match is found we cannot authenticate with TA and execution stops. If a matching chain is found, however, we are in possession of the correct credentials (as far as is known at this stage) and execution resumes.

Additionally, the matching IS private key is resolved from the `TerminalKeyStoreWrapper`, which is the app-internal store of private keys.

At this stage everything is in place and TA is executed by calling `doTA(..)` on the `PassportService` with the resolved chain of CVCs, IS private key, compressed public key from CA and MRZ-derived credentials.

If TA fails the inspection resumes as usual, but if it succeeds access to the sensitive DGs should now be granted (depending on the privileges of the particular IS certificate used). In the latter case DGs 3 and/or 4 are read, which is a process very similar to reading DG2 but with the added step of extracting multiple biometrics (if they are present).

8.4.3.4 Validation

The last step of the inspection procedure is document validation. In more technical terms, the procedure performs Passive Authentication.

First, the signature of SO_d is checked in `checkSodSignature(..)`. The following preliminary steps are taken:

- Get the `eContent` and encrypted digest from the SO_d SignedInfo structure.
- Resolve which digest algorithms are used for the DG hashes and for the encrypted digest.

Once these have been performed the next step is to do the signature verification itself:

[12]Due to a bug in the current version of JMRTD only 3DES is supported for Secure Messaging post-CA.

[13]Object Identifier, in this case identifying a specific suite of ciphers for CA and Secure Messaging.

```
Signature signature = null;

try {
  signature = Signature.getInstance(signatureAlgorithm);
} catch (NoSuchAlgorithmException e) {
  return false;
}

try {
  signature.initVerify(sod.docSigningCertificate());
  signature.update(signatureInfo.eContent());
  return signature.verify(sod.signatureInfo().signature()
      );
} catch (InvalidKeyException | SignatureException e) {
  return false;
}
```

Next, to ensure the signature is genuine the Document Signer Certificate C_{DS} in SO_d is validated by the method `checkDocSignerValidity(..)`.

It employs the *Certificate Path Validation* algorithm defined in RFC 5280 [39] to try and resolve and validate a certificate chain for C_{DS} from the CSCA certificates installed in the `CscaKeyStore`.

```
...
CertPathBuilder builder
   = CertPathBuilder.getInstance("PKIX");
PKIXBuilderParameters params =
   new PKIXBuilderParameters(cscaTrustAnchors, selector);

params.addCertStore(docStore);
for (CertStore trustStore : cscaStores)
   params.addCertStore(trustStore);

// We do not support CRLs at this time.
params.setRevocationEnabled(false);

PKIXCertPathBuilderResult pkixResult = null;

try {
   pkixResult = (PKIXCertPathBuilderResult) builder.build
       (params);
} catch (CertPathBuilderException e) { /* Failed! */ }
/* Succeeded! */
...
```

Figure 8.17: Excerpt from MrtdReader: Certificate Path Validation

Figure 8.17 shows an excerpt from the internal method `doPkixForChain(..)` which performs the aforementioned certificate path validation.

At this point the SO_d has been fully authenticated against a known CA in our store of CSCA certificates. The next and final step of document

validation is therefore to check that the now authenticated data group hashes in SO_d match the data that has been read. The steps done for this are:

1. Resolve the hashing algorithm used to create the hashes (from the `SignedInfo` in SO_d).
2. Instantiate the hashing algorithm and use it to compute a list of hashes over the data groups read during the inspection.
3. Get the list of stored data group hashes from SO_d and compare them to the ones computed in the previous step.

Figure 8.18 is an excerpt from `doPA(..)` which performs this procedure.

```
for (Map.Entry<Integer, ? extends DataGroup> entry
        : dataGroups.entrySet()) {
  int dgNum = entry.getKey();
  DataGroup dataGroup = entry.getValue();

  byte[] encoded = dataGroup.getEncoded();
  if (encoded == null || encoded.length == 0) {
    continue; // This DG has not been read
  }

  byte[] computedHash = digest.digest(encoded);
  computedDataGroupHashes.put(dgNum, computedHash);

  byte[] sodHash = sod.dataGroupHashes().get(dgNum);
  if (sodHash != null) { /* Check if hashes match */
    dataGroupHashValidationResults.put(
      dgNum, Arrays.equals(sodHash, computedHash)
    );
  }
}
```

Figure 8.18: Excerpt from MrtdReader: computing and comparing DG hashes.

8.5 Known flaws and shortcomings

In the following sections we list known flaws and shortcomings pertaining to our implementation of the ICAO and EAC inspection procedures.

Active Authentication support

Support for Active Authentication is specified in the system requirements (7.1) but is at this point not implemented.

That said, and implementation is available in JMRTD, and we have tested it briefly (it works), so adding support should be trivial.

Passive Authentication deviations

The current implementation of Passive Authentication in `MrtdReader` is not fully compliant with the specification given in [4].

First, when the Document Signer certificate validation is performed, the relevant revocation lists are not checked for the resolved CSCA certificate.

This is obviously an important part of the validation procedure and not performing it essentially breaks the integrity of the system.

Second, the sequence in which the steps (validating C_{DS}, CSCA, SO_d and the hashes) are performed is not entirely in line with the procedure specified in Doc 9303, though all steps (bar the CRL checks) are performed.

Limited Secure Messaging cipher support after CA

When performing Chip Authentication a suite of algorithms is chosen from the ones offered by the eMRTD. When selection a suite which employs the AES cipher the subsequent Secure Messaging setup fails.

The reason for this is missing support in the underlying JMRTD implementation of CA.

Currently, a workaround is implemented which forces the selection of a 3DES cipher. It works for our single EAC ePassport sample, but will fail to perform CA in cases where a 3DES option is not available.

PACE shortcomings and instability

The implementation of PACE has been a challenge for the entirety of development.

Though some show-stopping bugs were solved in early stages, a few both major and minor issues, defects and frailties remain.

First, only a subset of the PACEv2 protocol is supported. In particular there is no implemented support for PACE-CAM[14], and also no support for using a Card Access Number (CAN)[15]. Both of these missing features are due to the absence of support in JMRTD.

For the same reason only the Generic Mapping is supported at this time, meaning documents which only support the Integrated Mapping cannot be read using PACE. [15] specifies that supporting both GM and IM is mandatory for SAC inspection systems.

The main issue with the PACE implementation is not these missing features, however, but rather a very prominent instability problem.

Simply put, PACE authentication will very often fail for no known reason. The failure is always at the IC side, but the phase of PACE at which it occurs does vary some, with the majority of cases (that we have observed) being in the fourth and final step (mutual authentication). An

[14]Chip Authentication Mapping

[15]The specification in [15] does state that CAN support is optional, though

error code is returned, but it is only the generic `0x6300`, simply translating to "authentication failed".

In addition to this, when PACE does succeed the performance seems to be very unstable, ranging from a few short seconds to the 10-20 second range (which is obviously way beyond unacceptable).

A key observation relating to this is that the positioning of the device in relation to the MRTD seems to affect both stability and performance. This also strengthens the theory that these issues are linked and are related to insufficient power delivery from the device NFC antenna to the eMRTD.

It could also be argued that the fact that PACE has these problems, and not other protocols is indicative of this: it is undoubtedly the most complex and resource demanding of them all. The step which tends to fail (the fourth) is also the most taxing operation of the protocol: exchanging and validating the authentication tokens.

This said, we only have access to a single PACE-supporting MRTD. More accurate indications of the root of the issues could be found by testing our implementation with a larger selection of MRTDs, or even emulated ones.

8.6 Some obstacles

During the course of development we have met many obstacles, both small and large. A couple of them have taken so much effort to solve, however, that they require special attention.

Both were eventually tracked to bugs in the JMRTD library, and have required debugging and modification of it in order to find a solution. Both of these solutions have been submitted as patches to the JMRTD developers and have later been merged into the JMRTD source code.

In the interest of learning the next sections explain these bugs and the solutions which ultimately resolved them.

Wrongful IV size assumption for PACE with AES

In initial versions of the app, PACE would simply not work. It consistently failed after the first step of the protocol (transceiving the encrypted nonces).

Debugging of the JMRTD source code eventually pointed to an issue with the IV[16] of the static cipher used, and it was finally discovered that the IV size was wrongly assumed in the source code to be equal to the size of the encrypted nonce.

The nonce is 16 bytes, yet the IV length of a block cipher is always equal to the key length. Thus, for 3DES a 16 byte IV worked fine since 3DES uses a 16 byte (128 bit) key. Yet for AES-256 (which is the only algorithm supported by our sample MRTD) it failed as the IV length should have been 32 bytes (256 bits).

Once identified, the issue was easily resolved by dynamically setting the IV length to the length of the cipher key.

[16] Initialization Vector

Block misalignment with AES Secure Messaging

Once the aforementioned IV bug was fixed, PACE would work[17], but a another show-stopping issue appeared shortly thereafter.

We discovered that when reading data groups after PACE authentication the returned data would occasionally be corrupted. That is, JMRTD and the underlying Bouncy Castle ASN.1 parser would fail to unmarshall the data, making it unusable.

From much experimentation and debugging a couple of facts were apparent: the issue was consistent, but only arose when reading the "large" data groups such as SO_d and DG2 (face image). Also, the issue was not reproducible with BAC authentication.

Our PACE sample ePassport uses AES for Secure Messaging after PACE and 3DES after BAC. Thus, we theorized that the issue might be related to AES SM.

We went on to examine the raw response APDUs returned from AES SM and comparing it to those from 3DES SM. From this we observed that the data was corrupted by single bytes being replaced by a null-byte at the end of certain blocks.

In the end the cause was traced to the fact that AES allows a one-byte padding (in fact, zero padding is forbidden), which would occasionally happen when requesting blocks of certain sizes from the eMRTD. However, the handling of requesting and receiving blocks in JMRTD did not take this into consideration, always assuming the returned data was the size of the request. Example:

- We request bytes 224—448 (224 bytes) from the eMRTD IC.
- The IC returns a block of size 224, containing 223 bytes plus `0x00`.
- We strip the trailing `0x00`, yet still assume the block was 224 bytes, essentially re-introducing the null-byte when copying it to a 224 byte buffer.
- The next block is read from byte 448, whilst we in actuality should have read from 447, causing a null-byte "hole" in the final data.

The issue was finally resolved by re-ordering the logic in JMRTDs `MRTDFileSystem` class which keeps track of the returned block length, making sure it is adjusted correctly when receiving 1-byte trailers.

8.7 Presentation of MRTD Inspector

In this section we give a brief tour of the MRTD Inspector app, showing its various features and the MRTD inspection workflow.

[17] Though in an unstable manner, as described in 8.5.

8.7.1 Configuration and utility features

Figure 8.19 shows the app menu, the credentials manager and the settings screen.

The credentials manager allows adding, deleting and editing MRZ credential sets to the internal database. This is mostly a convenience feature for testing purposes.

The settings screen contains all of the configurable options of the app, including the NFC timeout and settings for the various supported protocols, such as forcing BAC authentication only, disabling EAC and so forth.

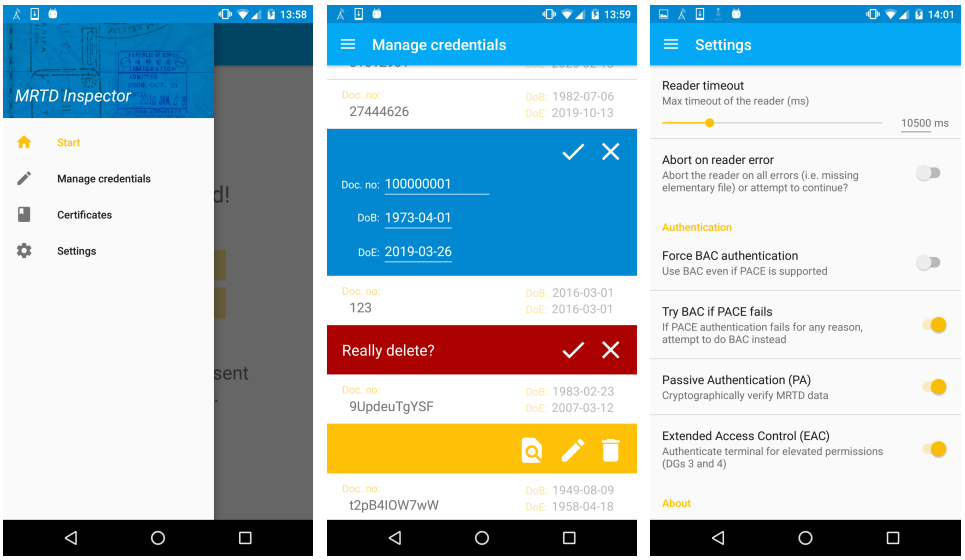


Figure 8.19: The menu (left), credentials manager (middle) and settings screen (right).

Figure 8.20 shows the certificate management features of the app.

It allows browsing the installed CSCA certificates, which can also be viewed in detail. The menu in the top-right corner contains options for deleting the current list or importing a new one. Currently only LDIF-format CSCA Master Lists are supported for import, and they are selected and fetched from the device file system.

There is also a feature which allows installing CVCA certificates for EAC. As with the CSCA Master Lists, these are also selected by the user and fetched from the file system.

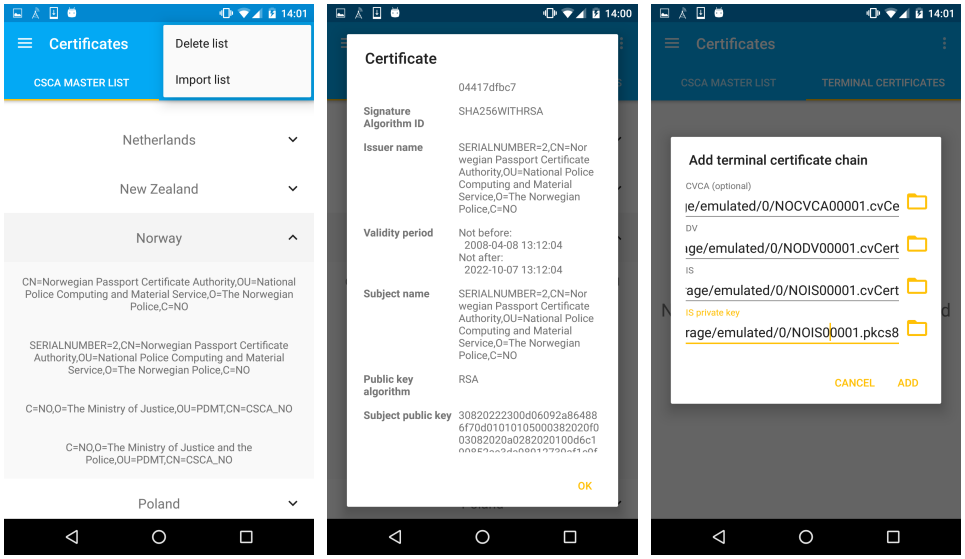


Figure 8.20: The certificate management screens. Installed CSCA Certficates (left), detailed certificate view (middle) and installing CVCA certificates (right).

8.7.2 Inspection workflow

The main screen, seen in Figure 8.21 is the home screen of the app and the default entry point for MRTD inspection.

Holding an eMRTD to the back of the device or touching the ePassport icon will launch the inspection procedure.

Also, the inspection is launched directly from any other part of the app if an eMRTD is detected.

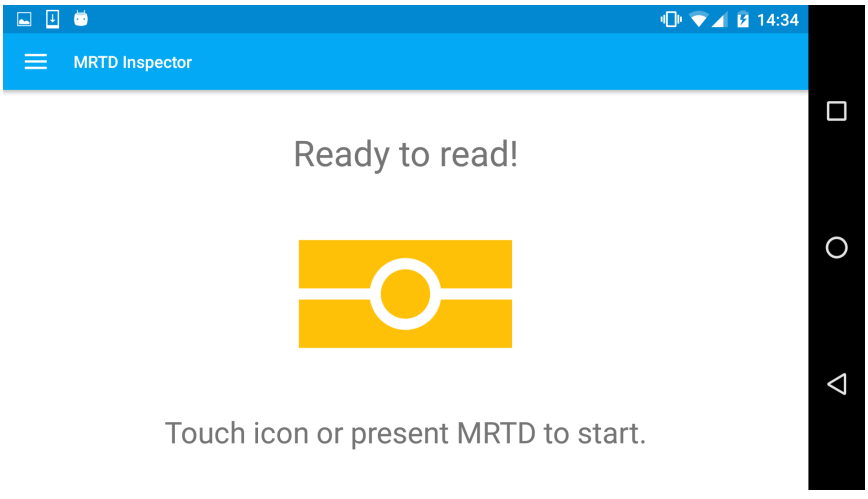


Figure 8.21: The main screen.

The first step of the inspection is to acquire the credentials for BAC/PACE, which is handled in the credentials selection screen seen in Figure 8.22.

As we know, this is done by either manual entry or scanning the MRZ. Manual entry is handled in the aforementioned credential manager, and the user can select from that list or one of the recently used credentials. The last option is, of course, to scan the MRZ.

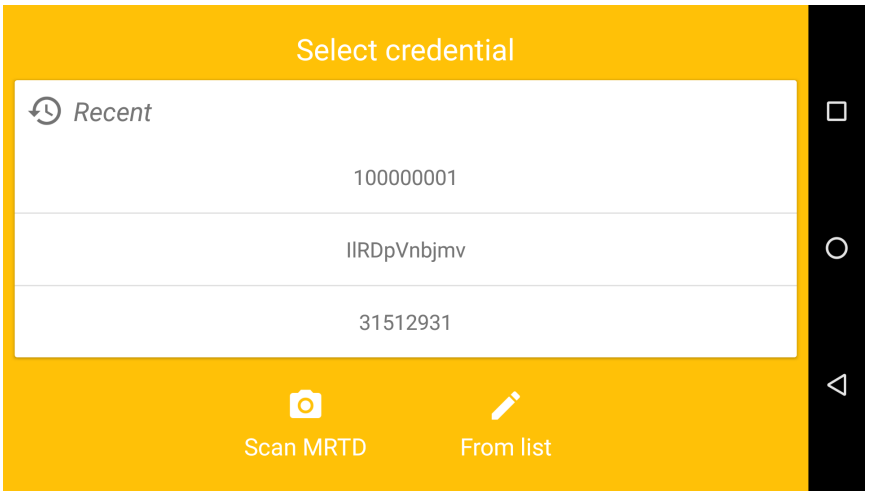


Figure 8.22: The credentials selection screen.

The MRZ scanner is seen in Figure 8.23. It features a viewfinder to help the user in finding a good position for the camera. The viewfinder also overlays the image used for the last scan, showing the user what the scanner is "seeing".

There is also a toggle button in the bottom left corner which allows the user to switch on/off the device torch for use in very low-light conditions.

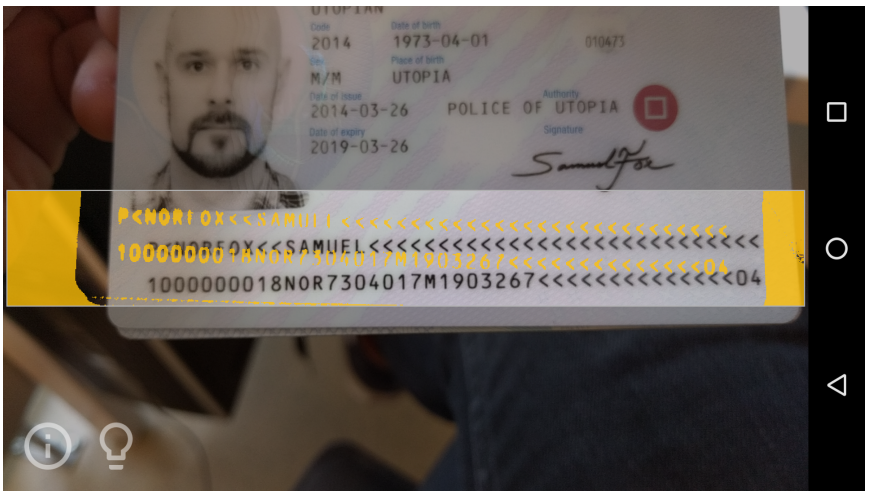


Figure 8.23: The MRZ scanner.

Once credentials have been selected or acquired the contactless inspection starts.

First, the user is presented with the "Present MRTD" screen, shown in Figure 8.24.

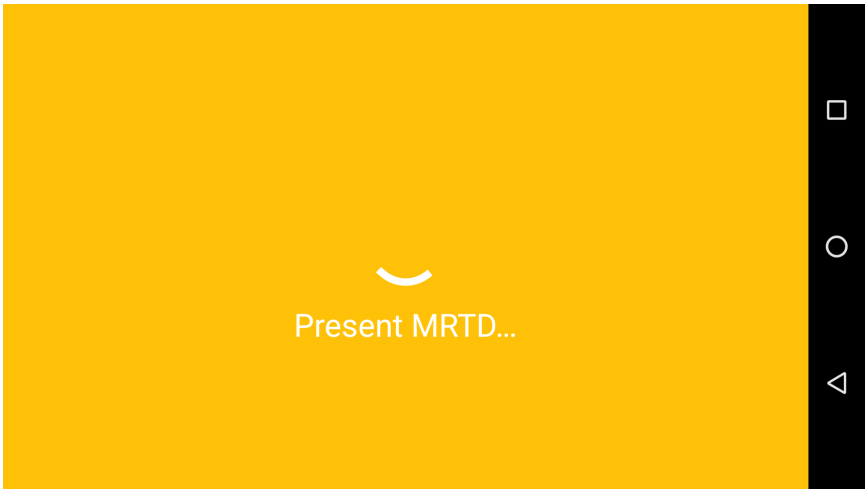


Figure 8.24: The “Present MRTD” screen.

As soon as an MRTD is detected the inspection procedure starts, giving progress updates while running as seen in Figure 8.25. In case of an error the procedure stops and displays an error message to the user.

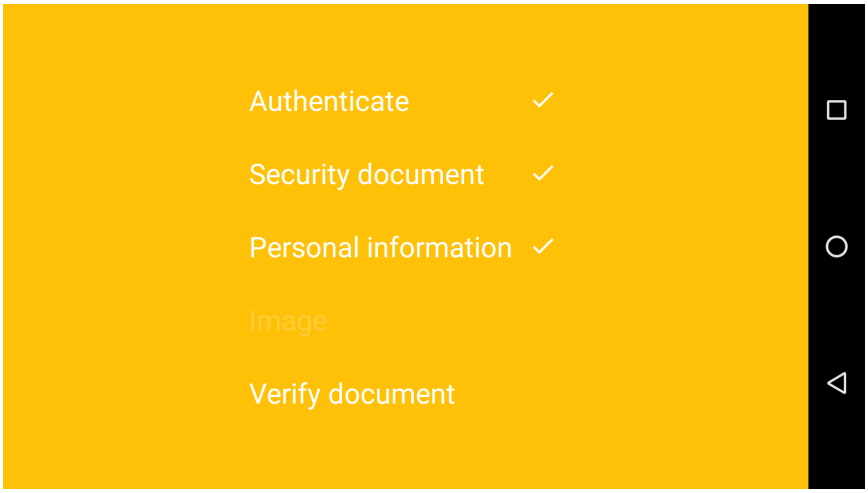


Figure 8.25: The progress screen.

Upon inspection finishing the MRTD viewer screen, seen in Figure 8.26, is launched.

The details of the document is displayed, also allowing the user to view the portrait and fingerprint (see Figure 8.27) in full resolution.

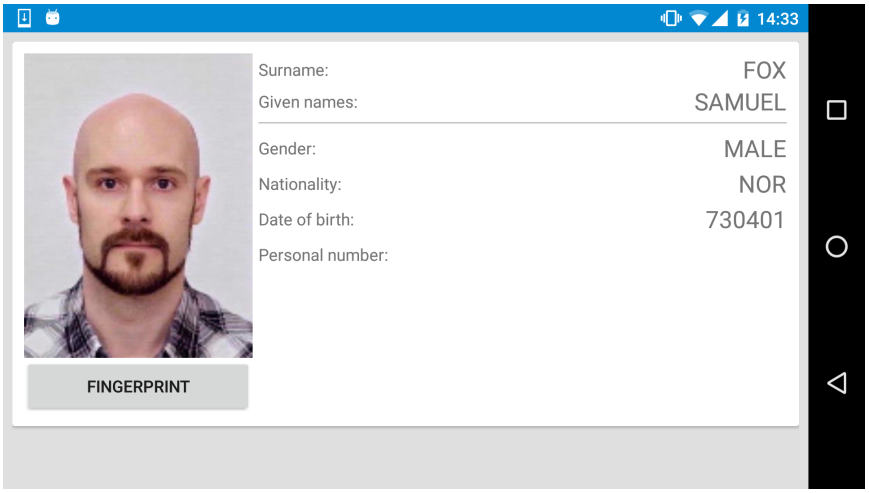


Figure 8.26: The view MRTD screen.

Figure 8.27: The fingerprint shown in full resolution. The image is zoomable (pinch-to-zoom).

Chapter 9

Evaluation

In this chapter we present our evaluation of MRTD Inspector. We do so in order to gauge to what degree the app meets the requirements given in 7.1, and in extension we use these results as a basis for discussion.

In direct evaluation of our prototype we are most interested in the performance and stability when performing inspection. Answering to this we've devised and executed a series of experiments which assess key parts of the procedure in isolation, providing discrete measurements of execution times, accuracy and stability. For each experiment we present the immediate results along with a short analysis. The broader perspective is handled in the next section, however, which summarizes and discusses the results in relation to each other and the wider context.

We conclude the chapter by presenting our main findings from the experiments.

9.1 Experiments and measurements

In Chapter 7 we identified two key aspects of the eMRTD inspection procedure:

1. Obtaining the BAC/SAC keys through scanning the MRZ and using OCR.

2. Performing the inspection procedure(s) with the contactless IC (security protocols and reading contents).

These are the core mechanisms which make up all eMRTD inspection use cases. Therefore our experiments are devised to measure the performance of these mechanisms in our implementation.

9.1.1 Method

The following sections establish and discuss the methods and tools used for the experiments.

9.1.1.1 Measurements

All of our defined measurements are execution times, and therefore simple time intervals, or discrete values (counts). We take measurements by slightly modifying the source code of MRTD Inspector to record and output the values at runtime (in the debug log).

This is by no means an exact method, and execution time can be affected by much more than the execution thread itself, but fortunately we are not dependent on micro-measurements, and are in any case only interested in the real-life performance of our app.

9.1.1.2 Test subjects

We have access to three test subjects of which two are 1st generation Norwegian ePassports and one is a Gemalto specimen ePassport. The specimen passport S_3 is technologically identical to the 3rd generation Norwegian ePassports (issued from 2015). Contrary to the case for officially issued documents, however, we have access to the entire EAC PKI for S_3, allowing us to outfit the MRTD Inspector app with a valid EAC certificate chain and private key.

An overview of the ePassport subjects and their supported features is given in Table 9.1, and Table 9.2 shows the sizes of the images in data groups 2 and 3.

Subject	Issuer	DGs	BAC/PA	PACE	AA	EAC
S_1	Norway (2007)	1, 2	✓	✗	✗	✗
S_2	Norway (2009)	1, 2	✓	✗	✗	✗
S_3	Gemalto (2014)	1, 2, 3	✓	✓	✓	✓

Table 9.1: The table shows our three test subjects and what features they support. They are all ICAO TD3-type ePassports.

Subject	DG2 image size	DG3 image size
S_1	17786 bytes	—
S_2	16344 bytes	—
S_3	18237 bytes	9792 bytes

Table 9.2: DG2 and DG3 image sizes.

9.1.1.3 Physical vs. virtual

Due to the time- and resource-limited nature of this Master's project we have elected to only use these three physical eMRTDs for our testing, and not virtual or emulated implementations. The latter would allow for more diversity in the available features of the subjects, and would also enable testing the software in separation of the hardware (no environmental factors such as radio noise and physical position coming into play).

However, setting up a test bed which integrates with virtual eMRTDs would be a considerable undertaking. The available software to do so is complicated, and a lot of ground would need to be broken in order to interface with our app. That is: it could be done, but would take considerable effort.

In any case our main goal is to evaluate using an implementation like ours in real world scenarios, making them our priority for testing.

9.1.1.4 Hardware

For the experiments we use only our main development device, namely the LG Google Nexus 5X.

9.1.1.5 A note on statistical method

All of our measurements are discrete samples of time or quantity, and are presumed to be independent. From the raw samples collected we present the following calculated values:

- The range (min, max) of the samples.
- The sample mean.
- the Standard Error of the Mean (SEM).

For a sample group of size n we calculate the sample mean $\overline{x}$ like so:

$$\overline{x} = \frac{1}{n}(X_1 + X_2 + ... + X_n)$$

The Standard Error of the Mean $SE_{\overline{x}}$ is defined as:

$$SE_{\overline{x}} = \frac{s}{\sqrt{n}}$$

Where s is the Standard Deviation, for which we use the Population Standard Deviation σ, defined as:

$$\sigma = \sqrt{\frac{\sum(X - \mu)^2}{n}}$$

Where X is the sample, μ is the mean and n is the sample size.

We choose the Population Standard Deviation because we do not wish to generalize our findings, but deem them as a self-contained group.

9.1.1.6 Raw experiment data

The raw data captured during the experiments are available in Appendix C.1.

9.1.2 Experiment A: MRZ recognition

Our implementation uses the built in, rear-facing camera of the smartphone to perform the MRZ scanning procedure. This obviously contrasts a purpose-built inspection system, which typically uses a "scanner-like" system, tuned and constructed for the task at hand.

With the phone camera we are faced with challenges such as mobility of the device and eMRTD, poor lighting and sub-optimal angles. In order to evaluate performance of the MRZ recognition we take the following measurements:

Measurement	Unit	Description
A.a	Count	Number of failed OCR recognitions until a valid MRZ is found.
A.b	Count	Number of false positive OCR recognitions until a valid MRZ is found.
A.c	Milliseconds	Time from scanner is started until a valid MRZ is found.

Table 9.3: Measurements for experiment A.

The descriptions of A.a and A.b might need some elaboration:

- *"Failed OCR recognitions"* refers to the case where the OCR engine attempts to decode an image but does not find any recognizable data at all, thus returning a failure.
- *"False positive OCR recognitions"* are results returned from the OCR engine where text has been found, but is not a valid MRZ.

The timer for A.c starts once the MRZ scanner is launched and stops as soon as an MRZ is found.

To learn about the limitations of our implementation using the phone camera we perform the tests with variations of the following environmental factors:

- Lighting: natural daylight or dark room (shutters closed, no direct light source).
- Stability: eMRTD held in the hand of the user or laying still (phone is hand held).

The combinations of these produce the following four test cases:

Test case	Lighting	eMRTD stability
A.1	Low light	Stable
A.2	Low light	Unstable
A.3	Daylight	Stable
A.4	Daylight	Unstable

Table 9.4: The table shows the four test cases for experiment A.

For each of the three test subjects in Table 9.1 we perform each test case from Table 9.4 10 times, yielding a total of 120 tests executed. For each execution we record data for each of the three measurements given in Table 9.3.

It should be noted that the results are dependent on a range of factors, not only those specifically tested for. Also, the experiment is performed with the operator trying to achieve the optimal (straight) angle, which is most likely learned and somewhat adjusted for even before the MRZ scanner is started.

9.1.2.1 Results

The results from Experiment A are given in Table 9.5.

Test case	Measurement	Range	Mean	SEM
A.1: low light, stable	A.a	0—11	1.6	0.2
	A.b	0—11	3.2	0.5
	A.c	1708—10818	4341	332
A.2: low light, unstable	A.a	0—16	0.9	0.2
	A.b	1—16	3.7	0.8
	A.c	1848—21727	5498	842
A.3: daylight, stable	A.a	0—8	1.8	0.3
	A.b	0—4	1.6	0.2
	A.c	843—4306	2105	134
A.4: daylight, unstable	A.a	0—18	0.6	0.1
	A.b	0—18	3.0	0.8
	A.c	726—11387	2548	352

Table 9.5: Results from experiment A.

Execution time of MRZ scanning

Breaking the number from Table 9.5 down we get the following figures for the execution time of the test cases:

- **Low light, stable:** 4341$\pm$332 ms
- **Low light, unstable:** 5498$\pm$842 ms
- **Daylight, stable:** 2105$\pm$134 ms
- **Daylight, unstable:** 2548$\pm$352 ms

The results indicate that the lighting conditions are the most significant of the environmental factors we have examined. A well-lit environment yielded MRZ recognition times in the range of two to three seconds for both the *stable* and *unstable* test cases.

In the low light environment, however, the values are in the four to six seconds range, which is considerably slower.

For all cases we do see that stability has an effect on the recognition times, but not greatly so. That said, we observe that the unstable test cases display a significantly larger range (and therefore SEM).

This is in line with our own observations while executing the test cases: sometimes you "get lucky" and get a correct recognition right away, whilst next time you might end up having to readjust the camera position multiple times, prolonging the recognition time. Of course, the unstable scenario is more sensitive to this as finding a "good" camera position is easier when the MRTD is stationary.

Failures and false positives

The amount of failed recognitions are:

- **Low light, stable:** 1.6 ± 0.2 failures
- **Low light, unstable:** 0.9 ± 0.2 failures
- **Daylight, stable:** 1.8 ± 0.3 failures
- **Daylight, unstable:** 0.6 ± 0.1 failures

We observe that for our experiment the "failure" condition is a fairly seldom occurrence. This is likely related to the fact that the experiments were performed in a very controlled manner, with a "trained" operator attempting to optimize the conditions.

This is reinforced by our observations when running the MRZ scanner: when the captured image does not contain any MRZ-like information, for example when pointed at the other parts of the data page, the failures are fast and many. When the image does contain MRZ data, however, the process most often recognizes at least some text.

The amount of false positives are:

- **Low light, stable:** 3.2 ± 0.5 false positives
- **Low light, unstable:** 3.7 ± 0.8 false positives
- **Daylight, stable:** 1.6 ± 0.2 false positives
- **Daylight, unstable:** 3.0 ± 0.8 false positives

The "false positive" condition happens when there is text data recognized, but it does not contain a valid MRZ. It could contain parts of the correct MRZ data, or it could contain only wrong information.

We observe that this condition happens, in most cases, significantly more often than the "failure" condition, which supports the argument that the experiment is somewhat biased in terms of chosen camera angle and cropping region of the image.

The only exception to this is the "best case" daylight/stable scenario, where both values are comparatively low. In this case the input data is

of very high quality and recognition is accurate enough so that very few attempts are needed to find the correct MRZ.

9.1.3 Experiment B: Standard Inspection Procedure

In this experiment we wish to measure the performance and reliability when executing the Standard Inspection Procedure as defined by BSI TR-03110 [5]. It entails access control (BAC or PACE), reading the less-sensitive data groups and Passive Authentication. The procedure is presented in 3.5.1.

Though [5] states that PACE should be preferred over BAC we have elected to run the experiment with only BAC. The reason is that PACE is not supported by test subjects S_1 and S_2, and that our implementation of PACE is fairly unreliable (an isolated experiment is devised in 9.1.5 to test our PACE implementation).

BAC and PA together make up the entire protocol suite supported by test subjects S_1 and S_2, meaning this experiment equates to the common feature set of all our available ePassports.

In order to gauge the performance and stability of our implementation we have defined the following measurements for this experiment:

Measurement	Unit	Description
B.a	Milliseconds	Execution time of BAC.
B.b	Milliseconds	Execution time of reading DG2.
B.c	Milliseconds	Execution time of entire procedure.

Table 9.6: Measurements for experiment B.

We have chosen to not explicitly record the execution time of reading DG1 or doing Passive Authentication as our preliminary observations are that these are performed in negligible time (due to small amounts of I/O required). That is: we prioritize studying the known bottlenecks.

Throughout development we have occasionally experienced that the inspection *just fails*. It is most often due to losing the connection with the eMRTD (as reported by the Android NFC subsystem), even in cases where there is no movement during inspection.

Should failure happen during our experiment we do the following:

- Record the point of and reason for failure (to the best of our ability) into a separate data set.
- Re-run the test.

This gives us the ability to learn from failures should they happen, yet avoid diminishing the main data set.

Test subjects S_1 and S_2 are of a different generation and manufacture than S_3, which reflects the current ePassports issued in Norway. Therefore, we wish to test these as separate cases, giving us two test cases:

Test case	Test subject
B.1	S_1
B.2	S_3

Table 9.7: Test cases for experiment B.

As we have observed that our system is sensitive to the positioning of the eMRTD in relation to the smartphone we choose the best possible position and use it for every run.

We run each of our test cases 15 times. Any recorded failures come in addition.

9.1.3.1 Results

Test case	Measurement	Range	Mean	SEM
B.1	B.a	195—233	205	3
	B.b	6072—7778	6881	101
	B.c	7522—9293	8350	107
B.2	B.a	74—110	82	3
	B.b	5801—6424	6076	46
	B.c	6561—7269	6854	52

Table 9.8: Results from experiment B.

For test case B.1 there were three instances of failure, once due to the NFC subsystem losing the tag and twice from a `READ_BINARY` failure on the IC side. B.2 had no failures.

Looking at the results a few characteristics are apparent:

- The performance of the Standard procedure is very stable, with both subjects displaying a tight grouping of results.
- The long read operations, like reading DG2, are by far the most expensive.
- The implementation of the eMRTD itself plays a role in performance, where the newer generation ePassport performed the procedure 1 to 2 seconds faster on average. In particular the BAC execution was over twice as fast on the newer ePassport.

Generalizing the results we see that the Standard procedure is performed in the range of 6 to 8 seconds (approximate sum of means from B.1 and B.2 including SEM).

Data transfer rate

The DG2 read operation is largely comprised of reading the image data. Knowing this we calculate a rough approximation of the effective transfer

rate, using the mean of the DG2 read duration and the known image sizes (Table 9.2). The results are shown in Table 9.9).

Test case	Image size	Duration	Transfer rate
B.1	17786 bytes	6.8 s	20.9 kbit/s
B.2	18237 bytes	6.0 s	24.3 kbit/s

Table 9.9: The approximate transfer rates of DG2 data.

As stated, these calculated transfer rates are approximations and should not be considered as accurate. They do, however, underline that the effective rate of transfer is far from the ISO/IEC 14443-4 minimum specified rate, which is 106 kbit/s.

9.1.4 Experiment C: Advanced Inspection Procedure

With this experiment we wish to measure the performance and reliability when executing the Advanced inspection procedure as defined by BSI TR-03110 [5]. That is: access control (BAC or PACE), Chip Authentication, Terminal Authentication, reading of sensitive and less-sensitive data groups and Passive Authentication. We are not using Active Authentication. The procedure is presented in 3.5.2.

As is done in Experiment B we use only BAC for the initial access control, which avoids the PACE instability issues affecting our results. Note that this is not in keeping with the ICAO Doc 9303 and BSI TR-03110 standard as they specify that PACE should always be preferred where available.

Only S_3 supports the Advanced procedure, thus it is the only subject used.

Table 9.10 shows the measurements defined for this experiment:

Measurement	Unit	Description
C.a	Milliseconds	Execution time of reading DG2.
C.b	Milliseconds	Execution time of EAC (CA and TA).
C.c	Milliseconds	Execution time of reading DG3.
C.d	Milliseconds	Execution time of entire procedure.

Table 9.10: Measurements for experiment C.

As for experiment B, in case of failure we record this in a separate data set, along with an explanation of the point of failure (if any), and re-run the experiment.

We have only the single test case and a single subject (S_3). The experiment is run 20 times.

9.1.4.1 Results

The results are given in Table 9.11.

Measurement	Range	Mean	SEM
C.a	5616—7483	6149	96
C.b	1140—3190	1444	129
C.c	3064—3876	3288	40
C.d	14024—19445	15125	326

Table 9.11: Results from experiment C.

There were no failed executions of this experiments.

Our main observations are:

- The mean of the execution time for the Advanced procedure is 15125±326 ms, suggesting a 15 second execution on average, but we did see outliers up to the 20 seconds range.
- Performing the EAC protocol itself adds roughly 1.5 seconds to the procedure, which is a fairly small proportion yet still significant.
- As anticipated the reading of large data groups (image and fingerprint) is the definite bottleneck of the procedure.

9.1.5 Experiment D: PACE execution

Throughout our design and implementation process, achieving a stable and satisfactory implementation of PACE has been an ongoing challenge. Though we have been able to get the implementation to a somewhat functional state there are major instability issues which are still unsolved.

We observe that PACE will run successfully on occasion, and fatally fail in various stages of the protocol otherwise. We also observe that the relation between the physical position of the device and the eMRTD seems to be of significance. In addition, PACE runs fairly slowly[1].

This experiment is designed as a reaction to these issues, and attempts to give indicative answers to the following questions:

- How often does PACE fail? That is: what is the failure rate?
- If it fails, what are the points of failure?
- If it succeeds, how long does it take?

We only take a single measurement:

Measurement	Unit	Description
D.a	Milliseconds	Execution time of PACE.

Table 9.12: Measurements for experiment D.

[1] In comparison to BAC, EAC

In the case of PACE failure we record it in a separate data set, along with the reason (if we are able to deduct one) and re-run the experiment.

As stated we only have access to one PACE-capable test subject (S_3). We therefore run the experiment only with S_3, and until we have 15 successful executions.

We know from observation that the execution of PACE is highly sensitive to the physical placement of the eMRTD in relation to the handset. However, measuring and examining the physical properties at play is out of scope of this project. Therefore we attempt to select the best possible positioning of the passport for this experiment. In other words, the results will be best-case (as far as we know) for our implementation.

9.1.5.1 Results

Measurement	Range	Mean	SEM
D.a	2265—8781	3517	497

Failures	24
Successes	15
Failure rate	61.5%

Table 9.13: Results from experiment D. Note that the failure rate is not given in the statistical sense of the word but is rather an observation of the failures vs. successes for this small experiment.

We observe that 24 out of 39 total attempts at performing PACE failed and that 21 out of 24 failures happened in the fourth step of PACE as specified in [5]. This particular step is the *Mutual Authentication* step, and the error is indicated to be on the IC side, which returns the generic "authentication failed" APDU error code `0x6300`.

Though we cannot surely rule out that this is due to an IS side flaw in performing the procedure, we do know that PACE works on occasion with the exact same parameters.

Execution time

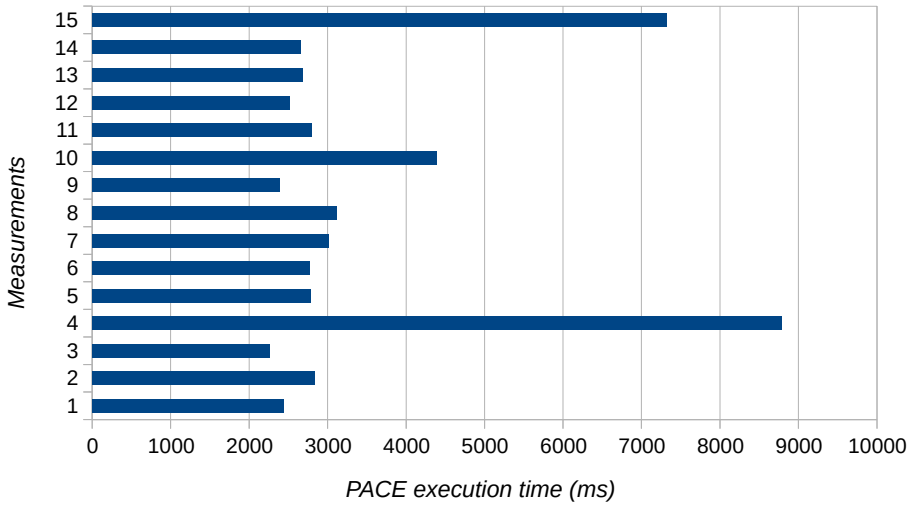


Figure 9.1: The execution times for the successful runs of PACE.

Figure 9.1 presents the successful measurements of PACE execution time taken during the experiment.

It clearly demonstrates the recorded mean execution time of 3.5 ± 0.5 seconds, and also highlights the outliers which in a couple of cases are very significant.

Thus, the experiment results indicatively confirm our observation that our implementation of PACE is highly unstable in two senses: it fails more than it succeeds, and when it does succeed the execution is both slow and somewhat unpredictable.

9.2 Discussion of the results

In this section we summarize and discuss the results from the four experiments in 9.1. The more detailed results (including the calculated means) are given and discussed in the respective experiment sections.

9.2.1 MRZ scanning

Experiment A 9.1.2 has been devised and conducted in order to gauge the performance of our MRZ scanner.

Through our preliminary observations we identified two environmental factors which seemed to affect the recognition time: light conditions and stability. Therefore our experiment was split into four test cases, allowing us to compare the performance of the recognition under the variations of these conditions and giving a more comprehensive picture of the performance.

A comparative chart is given in Figure 9.2, showing the mean recognition time (and SEM) of the test cases.

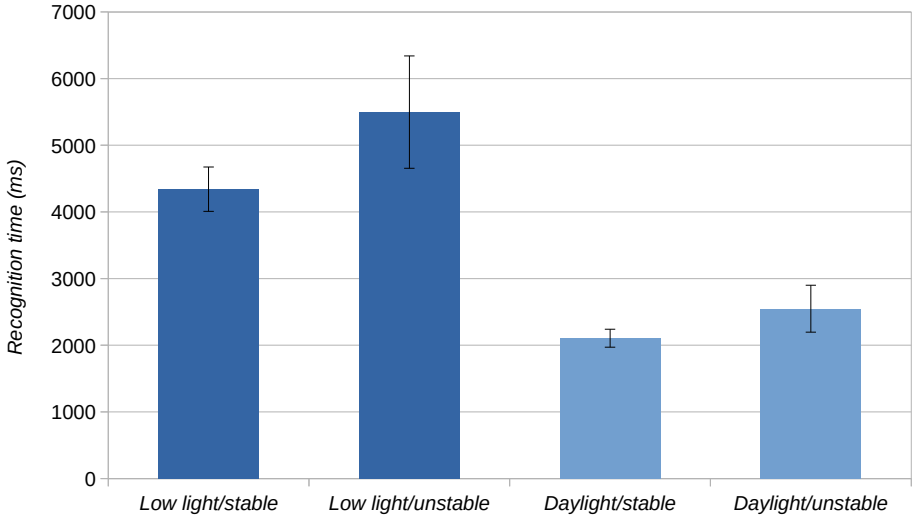


Figure 9.2: Comparison of the MRZ recognition test cases.

Comparing the four test cases we make a few observations. First, lighting is the most significant of the environmental conditions we have varied.

Instability does negatively affect performance but not consistently. That is, instability decreases the chances of capturing a "correct" image, but this effect is not always severe. For instance the "daylight/stable" and "daylight/unstable" test cases yielded reasonably similar results, reaching into the same ranges (when the SEM is considered). The same is true for both "low light" test cases.

The mean recognition times found for the four cases ultimately showed that:

- Under the "best case" conditions the recognition time is quite low at around 2 seconds.
- With the "worst case" conditions the recognition time is considerably higher at around 5-6 seconds, but does vary more.

We do recognize, however, that there is some degree of "happy-path" testing in the experiment as the test cases are very controlled.

In a real world scenario many other factors would come into play, including variation in lighting happening *during* the recognition process or reflective glare in the (plastic) data page in direct sunlight (or when using a directed light source at night time).

Also, it should be noted that the operator of the experiment performed the process many times in a row, which gives some degree of intuition for the optimal positioning of the camera under the specific conditions.

Additionally, we have confirmed through this experiment what we anticipated during the implementation of the MRZ scanner: the quality of the input image has a big impact on performance.

Therefore, we can conclude that a very high quality camera is desirable for a device which is to run an app like ours. In particular, a camera which operates well under low light conditions and that outputs fairly high resolution images. Notably, the camera in our test device (the Nexus 5X) is deemed to be in the upper tiers, but is in no way the best on the market, suggesting a potential for even faster and more accurate recognition on other devices.

9.2.2 Contactless inspection

Two experiments were devised and conducted to examine the duration of inspection. The first measured the execution of the Standard procedure and the second measured the Advanced procedure.

The results showed that the standard procedure was performed in approximately 7.5 seconds, whilst the Advanced procedure was approximately twice that, at around 15 seconds.

This confirms the preliminary observation that the definite bottleneck of contactless inspection is the reading of the larger data groups (2, 3 and 4).

Figures 9.3 and 9.4 illustrates these results for the Standard and Advanced procedures respectively.

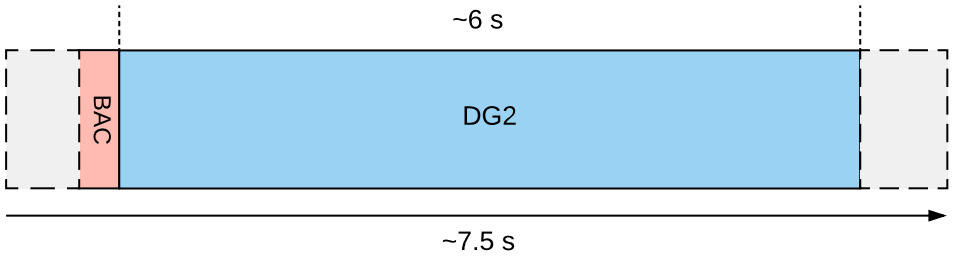


Figure 9.3: Duration of the Standard inspection procedure.

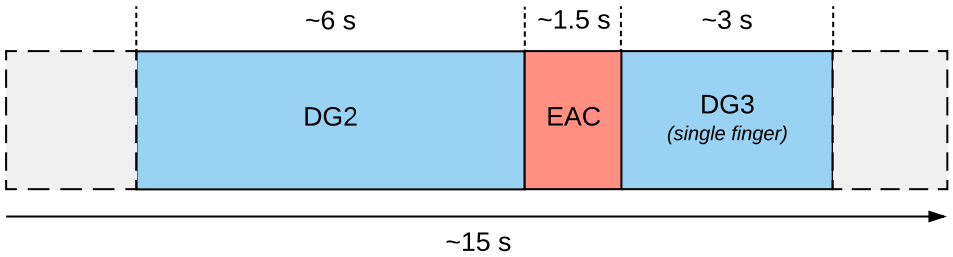


Figure 9.4: Duration of the Advanced inspection procedure.

For the Advanced procedure we also observed that the duration of EAC execution is significant, though not when compared to the large DG reads.

Figure 9.5 is a summary of the entire inspection procedure. It shows the aforementioned approximate duration of the two contactless procedures and also the duration of the MRZ recognition.

The selected MRZ recognition time is based on the "best case" results, and the procedures are based on the calculated means. As such, this illustration does not paint the whole picture, but gives an indication based on our results.

As can be seen from these indications: for the Standard procedure a total inspection time of around 10 seconds is achievable, whilst for the Advanced, however, 18 to 20 seconds is more likely.

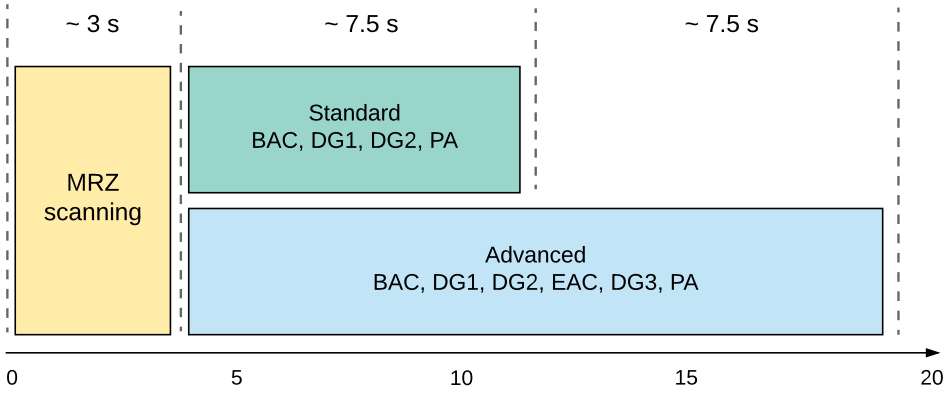


Figure 9.5: Approximate duration of inspection. The MRZ scanning time chosen is based on the results from the "best case" scenario.

Wishing to further investigate the apparent bottleneck of performing large data reads we calculated estimations for the transfer rate of DG2 in experiment B.

Not surprisingly these numbers, though inaccurate, revealed that the rate is much lower than the theoretical limit of the technology.

ISO/IEC 14443-4 specifies transfer rates of 106 kbit/s, 212 kbit/s and 424 kbit/s. Unfortunately we do not know which one was used for our experiments, and Android does not expose an API which allows us to investigate this. What we do know, however, is that our real life estimated rates of 20-24 kbit/s is very low in comparison.

Bear in mind, however, that the theoretical rates are pure transfer rates, and we are performing a complex protocol which does reads and writes in both directions. This means one should not expect near-limit performance in any case, but there is definitely room for improvement.

9.2.3 PACE execution

Experiment D was devised to shed light on the already known problems with the PACE protocol for our prototype: it is slow and very unreliable.

The results of the experiment confirmed this, with 24 of 39 attempts at PACE failing, most on the same (and last) part of the protocol.

Notably, this happened even under the conditions where we knew and attempted to use the "optimal position" of the passport in relation to the phone.

For our experiment the mean PACE duration (of the successful executions) was 3.5±0.5 seconds, demonstrating that even when it does work the performance is below reasonable.

Summarizing these results: PACE is (barely) functional in our prototype and more ground needs breaking to reach a serviceable implementation.

9.3 Main findings

Taking into consideration our experiment results in 9.1, the discussion of said results in 9.2 and the requirements set forth in 7.1, we present our main findings from the experiments.

- Our implementation of MRZ recognition meets the approximated requirement of a 2-3 second execution under certain conditions. Yet, in low light or with an unstable camera the recognition time is higher.
- The approximate requirement of a 10 second execution of the contactless inspection procedure is surpassed for early generation documents (BAC, PA), but not met for the current generation (BAC/SAC, EAC). The bottleneck is most likely the low data transfer rate.
- Our prototype performs inspection with reasonable stability under the right conditions. With the exception of PACE, the inspection procedure very rarely fails. We do recognize, however, that the system is very sensitive to movement and the positioning of the device in relation to the document. We assume this indicates that the coupling between the device and eMRTD antennas is sub-optimal.
- PACE will run under certain conditions, but most often fails for unknown reasons. This could be related to the same factors which cause sensitivity for other parts of the procedure, but a IS side software problem is also likely.

Part IV

Conclusion

Chapter 10

Discussion

The objective for this Master's project was to investigate and implement a prototype eMRTD inspection system for Android.

In doing so we aimed to show how such a system could be realized, and to uncover the inherent challenges and limitations imposed on it.

10.1 Results

The primary results of the project lie in our proposed solution.

We have researched, designed, implemented and evaluated a prototype which incorporates a wide range of technologies and techniques in order to perform inspection in an efficient manner.

It employs the camera of the device and optical character recognition to rapidly read the MRZ, performs the entire ICAO and BSI suites of protocols during contactless inspection and last but not least wraps everything into an easy to use interface which provides ample feedback to the user.

That said, the solution is in no way perfect, and we have identified several challenges throughout the project. Some we have solved, some we have attempted to mitigate, whilst some remain unsolved.

The nature of these vary, and so does our knowledge of their underlying reasons. In the next sections we discuss the major challenges we have met.

10.2 Software availability and quality

Given the scope of this Master's project, the implementation of a complete solution for eMRTD inspection has been a considerable endeavour. Therefore, we have been completely reliant on third party, open source and free software components to provide abstractions for the some of the low-level complexities of smart card communication and cryptographic protocols.

The paradoxical nature of this is, however, that when such abstractions fail the benefits are often outweighed by the time consuming task of identifying and addressing issues in third party code.

This double-edged sword of open source software has been a theme throughout the development of our prototype.

In particular, we have built our contactless inspection procedure on top of what is in reality the only freely available and comprehensive implementation of the ICAO and BSI standards[1]. We could not have built such an extensive application without it (given the resources at our disposal), yet we have also spent considerable time analysing and fixing flaws in it.

This brings us to an important recognition: in the process of implementing our prototype the lack of tools, documentation and a proper reference implementation for the eMRTD standards has been perhaps the most prominent challenge.

The argument could also be made that this is a generic problem for the implementation eMRTD software. Though the standards are open, the implementations are closed, making the utilization of the technologies a challenging effort without the resources of a vendor.

10.3 Contactless performance and reliability

Though the NFC interface in an Android smartphone is technically compatible with eMRTDs, this is clearly not the intended use.

NFC in phones and other consumer grade devices is first and foremost used for reading simple tags or for payment applications, and seldom for complex communication and transfer of large data.

The unsatisfactory robustness and performance of the contactless connection became apparent already in early stages of development, and has been an issue throughout.

The system is very sensitive to the exact positioning of the device in relation to the eMRTD, and movement during inspection or picking the wrong positioning often causes the connection to fail.

Also, as our experiments have indicated this does not only affect reliability but also imposes a significant performance hit. We have identified the reading of the large data groups as being the main bottleneck, and approximation of the achieved transfer rate was significantly lower than what the standard specifies, suggesting there is definite room for improvement.

Of course, the potential for improvement is also suggested by the comparatively much faster proprietary inspection systems on the market, which also highlights an important question: there might or might not be a physical limitation at play.

A natural assumption to make is, as stated earlier, that the NFC interfaces of these consumer grade devices are simply not made for this use, and thus unsuitable when higher performance is required. The NFC controllers might not be able to deliver the required power, and the device antennas might offer insufficient inductive coupling with those of the eMRTDs.

[1] JMRTD

That said, we could also make the assumption that future performance improvements in consumer grade NFC will help mitigate the issue. New handsets are released continuously and we have already seen the quality and performance of NFC interfaces improve significantly over the past few years. That this development will continue is not an unreasonable assumption to make.

Also, there are promising developments on the horizon of NFC. The *Very High Bit Rate* (VHBR) [40] technology enhances the ISO/IEC 14443 contactless interface and promises significant increases in data transfer rate with a theoretical 6.8 Mbit/s, as opposed to the 848 kbit/s maximum offered by the current standards.

Furthermore, the VHBR technology is already being envisioned by major vendors such as Gemalto and Infineon to make its way into the fourth generation of ePassports and eIDs.

The prospect of fast data transfers over NFC is likely to make the technology attractive also for consumer grade devices, which makes VHBR NFC a very promising future proposition for a system like ours.

10.4 MRZ recognition on a mobile device

As we know, the MRZ is designed to be machine-readable. Therefore, in the realm of OCR, MRZ recognition is a trivial task. However, the machines performing MRZ recognition were never thought to be mobile, and as such the characteristics of the MRZ are not necessarily optimized for mobile recognition (as opposed to QR codes, for instance).

In our work we have identified that the mobile use case imposes a new set of requirement on MRZ recognition. The stability of the camera and the lighting conditions are among the variable factors which must be considered. In addition, running a resource demanding task such as OCR on a mobile device means the balance between performance and accuracy must be found.

Our implementation demonstrates an approach to solving these challenges, and we have shown through experiments that it is sound. It is capable of performing recognition within our requirements, given that the conditions are good enough. In worse conditions, however, recognition was showed to be slower. In particular, we identified the lighting conditions to make the biggest impact on performance for the experiment.

That said, we have also made the important acknowledgement that the experiment is somewhat biased: the operator repeated the experiment many times in a row, learning the "correct" angle and distance of the camera. This makes the experiment useful in judging the best case scenario, but it might not be indicative of the real world performance.

As mentioned we optimized the camera angle and distance during the experiments, and capturing the "correct" image is admittedly the crutch of our implementation. Though we provide a few tools for the user to do so, capturing a somewhat straight-angled and correctly justified image of the MRZ can be a challenge, especially in movement.

In [41], Hartl et al. proposes a solution to this very problem. Their real-time algorithm for MRZ recognition is devised specifically to allow detection of the MRZ using a mobile device and with sub-optimal viewing angles. As such, it could serve to greatly increase the reliability of the MRZ scanner in our prototype.

10.5 PACE uncertainties

In order to be compliant with ICAO standards, not only should an inspection system support PACE, but it should also never prefer BAC. That is, in the case of PACE being present on the eMRTD, it should always be used.

For the current state of our prototype this standard is more or less unattainable as PACE is both unstable and significantly slower than what is acceptable. This unserviceable state of our PACE implementation was also confirmed by our experiments, in which we had more failed runs than successes, and a mean execution time of 3-4 seconds.

In our system requirements we identified compliance with ICAO standards a key qualification for an inspection system, making the lack of proper support an unfortunate deviation that should be addressed.

At this point we do not know the exact reasons for the relatively poor performance of PACE in our prototype. We do, however, have a few theories and observations which are worth noting.

First, the instability and slowness issues previously mentioned are, of course, also present for PACE. The protocol does not perform a lot of I/O, but does do considerable computation. There is a possibility that low power delivery, paired with high power requirements of the calculations plays a role.

On the contrary, it is also possible that the underlying libraries are exceedingly slow when performing the elliptic curve operations, making the eMRTD time out.

At this point the relative weight of the factors affecting performance cannot be determined exactly, and it is clear that further investigation is needed in order to determine the cause.

Chapter 11

Conclusion

In this Master's project we have researched, designed and implemented a prototype eMRTD inspection app for Android.

The app performs optical recognition of the MRZ using the device camera and contactless inspection as per the ICAO and BSI standards over NFC, making it a complete and functional solution for reading and verifying ePassports in a mobile environment.

We have shown through experiments that our solution performs well under certain conditions, but leaves room for improvement in others.

In particular we have identified the data transfer rate of the contactless interface to be a major bottleneck for efficient operation, causing the duration of inspection to be higher than desired. Also, we have shown that the system is highly sensitive to the relative positioning of the device and ePassport.

From this we draw the conclusion that the current state-of-the-art smartphone is unsuitable for the purpose of an eMRTD inspection system. However, great promise is shown in the next generation of eMRTDs, the advent of higher data rate NFC technologies and the inevitable development of Android devices supporting these.

11.1 Further work

There are many important and interesting aspects of mobile eMRTD inspection which we have not discussed. In the following sections we present a few of these, in addition to a recommendation for further investigation of the known issues of our prototype.

11.1.1 The prospect of biometric authentication

Due to the scope of this Master's project, we have selectively disregarded the possibility for biometric authentication.

Naturally, reading the fingerprints from a biometric passports has no practical use if they cannot be verified. Also, though the face image is displayed to the operator upon inspection, there is also the possibility of performing facial recognition on the mobile device itself.

For the case of the fingerprint authentication, integrated fingerprint readers are quickly becoming a common addition in smartphones. They are more or less exclusively used for authentication of the device user herself at this point, but research could be put into uncovering whether or not it is possible to expand their use for authentication of ePassport fingerprints.

We do know that this is tricky at the current time, as Android does not offer an API for raw fingerprint recognition. The fingerprint sensor is used only for direct user authentication, comparing against a fingerprint template stored in the embedded Secure Element.

Another possible avenue to explore would be the use of an external fingerprint reader connected over the USB-OTG[1] interface.

In addition, there are specialized devices available which contain an integrated fingerprint reader that is not directly governed by the Android system.

11.1.2 Security

For this project, the security of our system has not been regarded. Yet, we should recognize that the security implications of using an Android device for eMRTD inspection needs investigation.

There are many possible security concerns for such a system, and clearly it must be hardened sufficiently to safely handle sensitive personal information.

The primary concern, however, is the storage and handling of highly sensitive certificates and keys. For the ICAO PKI (CSCA certificates) this is a matter of being able to obtain or store public certificates in a trusted manner.

For the EAC PKI, on the other hand, the implications are much more severe as the inspection system is required to access and handle private keys, which in turn give access to sensitive biometric data. Naturally, the theft of a device containing such a key is a serious threat, meaning the requirements for storage and access are very strict.

As an example, a solution utilizing the hardware backed Android keystore could be explored.

[1] USB On-The-Go. Interface which allows the Android device to operate as a host for USB devices.

Appendices

Appendix A

POD project description

Masteroppgave – MRTD

G3kko-prosjektet, som er en del av IDeALT-programmet i **Politidirektoratet,** kan tilby en masteroppgave. Oppgaven er én av tre oppgaver som inngår i samme fagmiljø og med tilstøtende problemstillinger.

Pass fra alle land, (det nye) Nasjonalt ID-kort, oppholdskort som utstedes av UDI, grenseboerbevis og mange andre identitets- og reisedokumenter inneholder en brikke og støtte for NFC. I brikken finnes i tillegg til informasjonen som er trykket i klartekst på dokumentet også biometriske opplysninger (fingeravtrykk, og annet). Informasjonen er beskyttet med kryptografi på forskjellige måter. Teknologien er i stor detalj beskrevet i retningslinjene gitt av FNs luftfartsorganisasjon (ICAO), i deres håndbok nummer 9303, og i en (lang) rekke andre internasjonale standarder. Fellesnevneren er «Machine Readable Travel Documents», eller bare MRTD.

Politiet ønsker å teste ut bruken av standard «mobiltelefon» som leseutstyr for MRTD. Oppgaven går helt kort ut på å lage en «app» for Android som viser at dette er en funksjonelt fullgod løsning.

Hovedutfordringene i oppgaven er:

- Kommunikasjonen skjer over ikke-trivielle protokoller (på toppen av NFC);
- De relevante opplysningene er beskyttet med kryptografi, både symmetrisk og asymmetrisk;
- Politiet opererer under strenge krav til internkontroll (logging, sporbarhet, privathet).

For på kunne løse oppgaven må man kunne lage en «app» på Android som benytter NFC, ha kjennskap til kryptografiske prinsipper og mekanismer, samt kunne bruke standarder.

Det vil trolig bli nødvendig med møter både i Oslo og i Toscana, Italia.

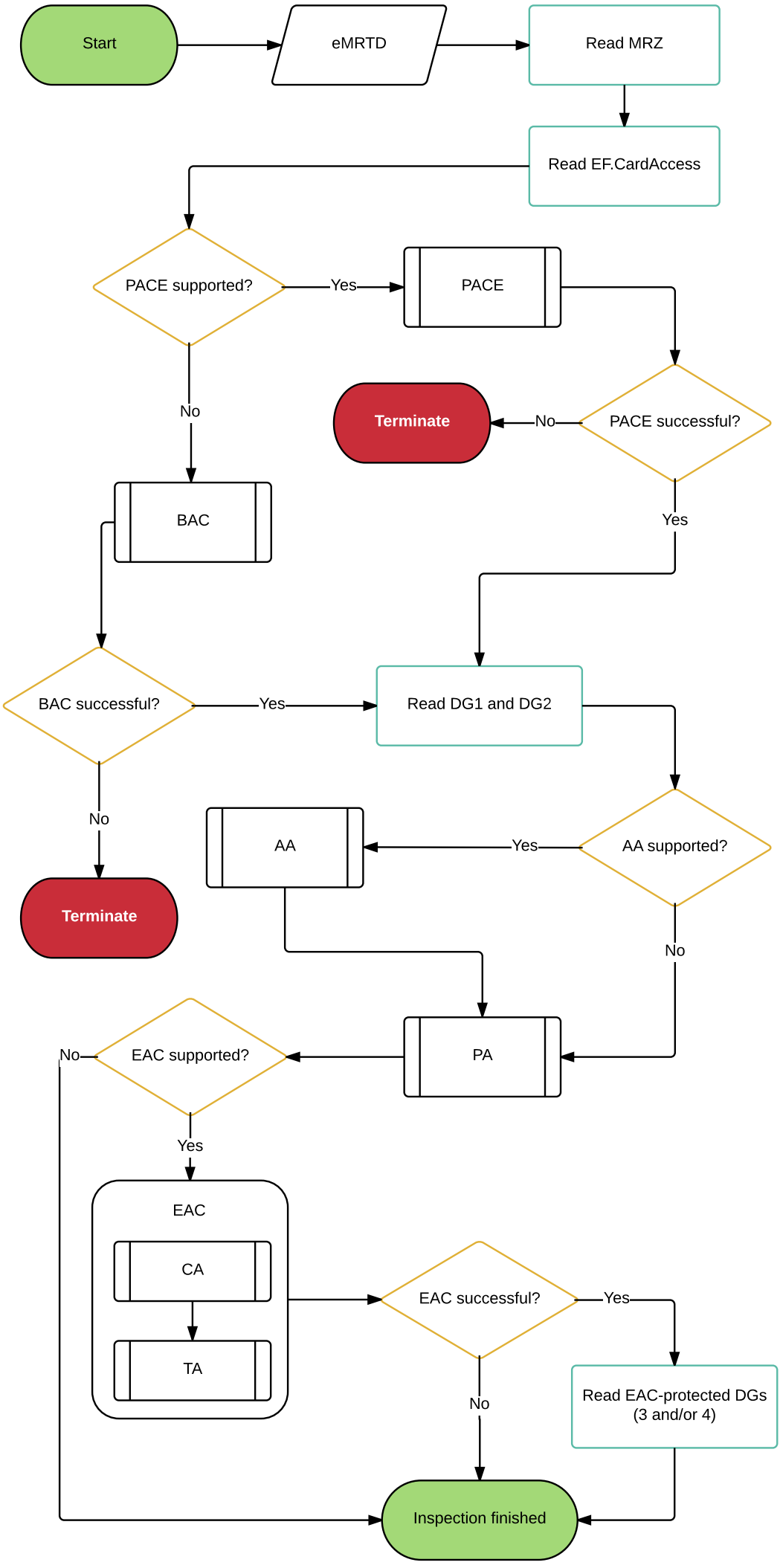
Spørsmål?
Send dem umiddelbart til:

- Førsteamanuensis Anders Andersen (anders.andersen@uit.no) som vet alt om å være masterstudent;
- Politiinspektør Håvard Nordbø (havard.nordbo@politiet.no) som vet alt om G3kko-prosjektet og IDeALT-programmet, og
- Seniorrådgiver Øyvind Næss PhD (oivind.naess@politiet.no) og Dr. Tage Stabell-Kulø (tage.stabell-kulo@politiet.no) som vet alt om selve oppgaven. Begge har tidligere undervist på universitet og veiledet studenter.

20. mai 2014

Appendix B

Inspection procedure flowchart


Start
eMRTD
Read MRZ
Read EF.CardAccess
PACE supported?
Yes
PACE
No
Terminate
No
PACE successful?
BAC
Yes
BAC successful?
Yes
Read DG1 and DG2
No
AA
Yes
AA supported?
Terminate
No
No
EAC supported?
PA
Yes
EAC
CA
TA
EAC successful?
Yes
No
Read EAC-protected DGs
(3 and/or 4)
Inspection finished

Appendix C

Downloadable content

C.1 Experiments raw data

http://folk.uio.no/halvdang/mrtd-inspector-experiments.ods

Bibliography

[1] International Civil Aviation Organization. 'MRTD REPORT: The New eUNLP: Vol. 8, No. 1'. In: MRTD Report (2013).

[2] Camilla Flaatten. Image from online news article: *"Har du prøvd disse noen gang?"* 4th Sept. 2014. URL: http://www.aftenposten.no/reise/Har-du-provd-disse-noen-gang-71925.html (visited on 14/04/2016).

[3] Gemalto. *Electronic travel document programs*. URL: http://www.gemalto.com/brochures/download/gov-etravel-doc.pdf (visited on 14/04/2016).

[4] International Civil Aviation Organization. *Doc 9303, Machine Readable Travel Documents*. Seventh edition. International Civil Aviation Organization, 2015.

[5] German Federal Office for Information Security (BSI). *Advanced Security Mechanisms for Machine Readable Travel Documents v. 2.10*. Tech. rep. 2008. URL: https://www.bsi.bund.de/EN/Publications/TechnicalGuidelines/TR03110/BSITR03110.html.

[6] International Civil Aviation Organization. 'Part 1: Introduction'. In: *Doc 9303, Machine Readable Travel Documents*. Seventh edition. International Civil Aviation Organization, 2015.

[7] International Civil Aviation Organization. 'Part 3: Specifications Common to all MRTDs'. In: *Doc 9303, Machine Readable Travel Documents*. Seventh edition. International Civil Aviation Organization, 2015.

[8] International Civil Aviation Organization. 'Part 4: Specifications for Machine Readable Passports (MRPs) and other TD3 Size MRTDs'. In: *Doc 9303, Machine Readable Travel Documents*. Seventh edition. International Civil Aviation Organization, 2015.

[9] International Civil Aviation Organization. 'Part 9: Deployment of Biometric Identification and Electronic Storage of Data in eMRTDs'. In: *Doc 9303, Machine Readable Travel Documents*. Seventh edition. International Civil Aviation Organization, 2015.

[10] International Civil Aviation Organization. 'Part 10: Logical Data Structure (LDS) for Storage of Biometrics and Other Data in the Contactless Integrated Circuit (IC)'. In: *Doc 9303, Machine Readable Travel Documents*. Seventh edition. International Civil Aviation Organization, 2015.

[11] International Civil Aviation Organization. 'Part 11: Security Mechanisms for MRTDs'. In: *Doc 9303, Machine Readable Travel Documents*. Seventh edition. International Civil Aviation Organization, 2015.

[12] International Civil Aviation Organization. 'Part 12: Public Key Infrastructure for MRTDs'. In: *Doc 9303, Machine Readable Travel Documents*. Seventh edition. International Civil Aviation Organization, 2015.

[13] Gildas Avoine, Kassem Kalach and Jean-Jacques Quisquater. 'Financial Cryptography and Data Security'. In: ed. by Gene Tsudik. Berlin, Heidelberg: Springer-Verlag, 2008. Chap. ePassport: Securing International Contacts with Contactless Chips, pp. 141–155. ISBN: 978-3-540-85229-2. DOI: 10.1007/978-3-540-85230-8_11. URL: http://dx.doi.org/10.1007/978-3-540-85230-8_11.

[14] Tom Chothia and Vitaliy Smirnov. 'A Traceability Attack Against e-Passports'. In: *Proceedings of the 14th International Conference on Financial Cryptography and Data Security*. FC'10. Tenerife, Spain: Springer-Verlag, 2010, pp. 20–34. ISBN: 3-642-14576-0, 978-3-642-14576-6. DOI: 10.1007/978-3-642-14577-3_5. URL: http://dx.doi.org/10.1007/978-3-642-14577-3_5.

[15] International Civil Aviation Organization. *Supplemental Access Control for Machine Readable Travel Documents v. 1.1*. Tech. rep. 15th Apr. 2014. URL: http://www.icao.int/Security/mrtd/Downloads/Technical%20Reports/NEW%20TRs%20post%20TAG%2022/TR%20-%20Supplemental%20Access%20Control%20V1.1.pdf.

[16] International Civil Aviation Organization. *PKD Participants*. URL: http://www.icao.int/Security/mrtd/Pages/PKD-Participants.aspx (visited on 11/04/2016).

[17] International Civil Aviation Organization. *ICAO PKD data download*. URL: https://pkddownloadsg.icao.int/ (visited on 11/04/2016).

[18] Roderick Heitmeyer. 'ICAO Civil Aviation and MRTD Standards'. In: Keesing Journal of Documents and Identity 31 (2010).

[19] The European Commission. *Commission Decision of 4.8.2011*. 4th Aug. 2011. URL: http://ec.europa.eu/dgs/home-affairs/e-library/docs/comm_native_c_2011_5499_f_en.pdf.

[20] The German Federal Ministry of The Interior. *Image of specimen German eID card*. URL: http://www.personalausweisportal.de/DE/Service/Presse/Bildmaterial/bildmaterial_node.html (visited on 13/04/2016).

[21] The Council of the European Union. *Council Regulation (EC) No 2252/2004 of 13 December 2004 on standards for security features and biometrics in passports and travel documents issued by Member States*. 13th Dec. 2004. URL: http://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX:32004R2252.

[22] Gartner. *Press release: "Gartner Says Worldwide Smartphone Sales Grew 9.7 Percent in Fourth Quarter of 2015"*. 18th Feb. 2016. URL: http://www.gartner.com/newsroom/id/3215217 (visited on 19/04/2016).

[23] Maximilian Stein. 'Mobile devices as secure eID reader using trusted execution environments'. In: *Lecture Notes in Informatics. Open Identity Summit 2013*. Gesellschaft für Informatik, Bonn, pp. 11–19.

[24] Luis Terán and Andrzej Drygajlo. 'On Development of Inspection System for Biometric Passports using Java'. In: *Biometric ID Management and Multimodal Communication*. Springer, 2009, pp. 260–267.

[25] Ján Vošček. 'Identity verification based on ePassports'. Master's thesis. Masaryk University, Faculty of Informatics, Brno, 2014. URL: http://is.muni.cz/th/325238/fi_m/.

[26] German Federal Office for Information Security (BSI). *Requirements for Smart Card Readers Supporting eID and eSign Based on Extended Access Control*. Tech. rep. 2013. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TR03119/BSI-TR-03119_V1_pdf.pdf.

[27] *DILETTA TDR700 - ePassport Reader with UV-Colour Feature*. URL: http://www.diletta.com/DE/Passport_Reader_TDR700.htm (visited on 01/05/2016).

[28] Annar Bohlin-Hansen and Jørn Anders Jenssen. *Presentation: IDeALT-programmet (in Norwegian)*. 13th Nov. 2013. URL: https://wiki.uio.no/mn/ifi/AFSecurity/images/f/f4/AFSec20131113-Bohlin-Hansen-IDeALT.pdf (visited on 01/05/2016).

[29] *Legion of The Bouncy Castle website*. URL: https://www.bouncycastle.org/.

[30] *Spongy Castle website*. URL: https://rtyley.github.io/spongycastle/.

[31] *Jar Jar Links website*. URL: https://github.com/shevek/jarjar.

[32] *Google's Tesseract OCR engine is a quantum leap forward*. 28th Sept. 2006. URL: https://www.linux.com/news/googles-tesseract-ocr-engine-quantum-leap-forward (visited on 20/04/2016).

[33] *Tesseract code repository*. URL: https://github.com/tesseract-ocr/.

[34] *tess-two code repository*. URL: https://github.com/rmtheis/tess-two.

[35] *The Leptonica Image Processing Library*. URL: http://www.leptonica.com/.

[36] *ReactiveX website*. URL: http://reactivex.io/.

[37] *How to use the tools provided to train Tesseract 3.0x for a new language*. 26th Mar. 2016. URL: https://github.com/tesseract-ocr/tesseract/wiki/TrainingTesseract#training-procedure (visited on 29/03/2016).

[38] *tesseract-ocr parameters in 3.02 version*. 29th Dec. 2012. URL: http://www.sk-spell.sk.cx/tesseract-ocr-parameters-in-302-version (visited on 10/02/2016).

[39] D. Cooper et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. http://www.rfc-editor.org/rfc/rfc5280.txt. RFC Editor, May 2008. URL: http://www.rfc-editor.org/rfc/rfc5280.txt.

[40] Christian Saminger et al. 'Introduction of very high bit rates for NFC and RFID'. In: *e & i Elektrotechnik und Informationstechnik* 130.7 (2013), pp. 218–223. ISSN: 1613-7620. DOI: 10.1007/s00502-013-0154-0. URL: http://dx.doi.org/10.1007/s00502-013-0154-0.

[41] Andreas Hartl, Clemens Arth and Dieter Schmalstieg. 'Real-time Detection and Recognition of Machine-Readable Zones with Mobile Devices'. In: *Proceedings of the International Conference on Computer Vision Theory and Applications*. URL: https://pdfs.semanticscholar.org/e4d6/0075b43f4e3d881482b5d744a85ca0142967.pdf (visited on 04/30/2016).