Radboud University Nijmegen

Faculty of Science

Master's Thesis Computer Science

Kerckhoff's Institute

IRMA Verified Assurer

Securely storing identity document chip data onto IRMA cards

Author:
G.A. Smelt

Supervisor:
prof. dr. B.P.F. Jacobs

Reader:
dr.ir. E. Poll

28th October 2015

Abstract

Selectively disclosing personal data in order to gain access to some type of product or service can be achieved by covering the non-relevant parts of one's identity document. However the IRMA implementation is much more secure and privacy-friendly. This requires the person to have previously obtained an identifying set of digital characteristics in order to pass verification by a relying party. IRMA Verified Assurer is an ideal method for new adopters of IRMA technology to obtain an initial set of such attribute-based credentials by leveraging one's identity document, such as a passport. Due to the nature of IRMA such an initialization is required to adhere to strict cryptographic rules. In this paper we describe the theory behind IRMA and passport security. Furthermore we specify guidelines which such a protocol has to follow and we design the Assurer protocol. We prove that the protocol is cryptographically secure by translating it into a model written in the Pi Calculus and proving this model using the ProVerif cryptographic protocol verifier.

Contents

List of Figures

Chapter 1

Introduction

Consider a scenario where you have to legitimize yourself in order to purchase a particular item such as a pack of cigarettes. The cashier is legally required to check your identification to ensure you are not underage before allowing you to purchase the items. Generally the customer presents some form of identification to the cashier, such as a passport. The cashier then checks your date of birth to ensure you are of legal age. In this process the cashier may also learn your name, location of birth and the document number, among other things. To some this feels like a breach of privacy and these individuals may wish to share only their date of birth, but ideally just the property 'of legal age' with the cashier. This is what the IRMA card is designed to do. The name is an acronym of I Reveal My Attributes and it is designed in such a way that you can selectively disclose properties of yourself to select individuals. Before you can use such a card you would need to add properties about your person to it, called attributes, which can then be selectively disclosed to gain various privileges, turning the attributes into attribute based credentials or ABCs. A particularly good set of starting properties to use for generating ABCs are the data stored in a person's passport. For this reason, we have designed the Assurer protocol.

The IRMA card has a lot of potential. There are many use cases the card could improve upon. Assurer could potentially allow for a much quicker adaptation of the IRMA card. The reason for this is that most people already have an identity card or a passport that contains all their characteristics in one place. Leveraging this in order to quickly gather many ABCs at once would be an ideal way to allow newcomers to get started. The passport or identity card generally holds the most generic characteristics, such as name, birthday, height and sex, making its conversion into attributes have immediate visual result.

Assurer is a protocol designed to facilitate transfer these passport characteristics to IRMA cards. In this paper we describe the steps taken during the design process and provide proof it satisfies the goals set out for it. In this protocol

we identify three actors: the assurer, the verification and issuing server, and the IRMA card. We will describe these further in chapter 2. The protocol is used to convert personally identifiable information stored on a passport into attribute based credentials that can be stored on an IRMA card. This is done by first verifying the passport both locally and remotely, upon which a set of attribute based credentials is generated for storing on the IRMA card. We will explain the protocol in much more detail in chapter 2.3.

Over the course of this paper we use many terms the reader may not be familiar with. Furthermore, abstraction is sometimes favorable, thus we now describe the meaning of certain terms that are used throughout this paper. The IRMA Verified Assurer protocol, denoted 'Assurer' from this point onwards, makes use of tablets for the clients that connect to the server. We will simply identify these tablets as 'clients' and similarly will refer to the IRMA card simply as 'card.' The clients connect to a central server which provides the clients with cryptographically signed attribute data, which we will denote as attribute-based credentials (ABCs). There are many forms of electronic identity documents (eIDs), such as a driver's license, ID card or a passport. For simplicity we will use the term 'passport' for all of these variations. Furthermore, authentication with respect to IRMA cards means the card is presented to a terminal (e.g. in a supermarket) which verifies the card user has the required attributes (e.g. age $\geq$ 18) for purchasing items (e.g. alcohol).

The paper is structured as follows. We start with the theory behind the technologies of IRMA and passports in chapter 2. Following that, in chapter 3 we state the goals that the protocol must satisfy and describe several attack scenarios. Afterwards we delve into the specifics of the TLS protocol that the Assurer protocol depends on, followed by the cryptographic assumptions we made with Assurer in mind, and finally the protocol itself. This is found in chapter 4. Next, in chapter 5 we describe the formalisation of the protocol as a model in the Pi Calculus for verification using ProVerif and explain the various implementation choices. Finally we discuss our findings and shortcomings and provide a basis for future work in chapter 6. The model is listed in appendix A.

Chapter 2

Theory

This chapter describes the theory behind the two main actors of the protocol that Assurer aims to connect. We start by providing insight into the IRMA ecosystem, followed by an overview of the electronic passport's security measures. Afterwards we sketch a typical run of the IRMA card application process.

2.1 IRMA

Here we explain the background of the IRMA proofs. IRMA stands for I Reveal My Attributes, which is a reference to attribute-based credentials (ABC) as described by [3, 9, 35]. ABCs (sometimes also referred to as anonymous credentials) are a way for people to share a part of their digital identity without disclosing any other information irrelevant to the goal of authentication. A digital identity is generally considered to be a set of characteristics describing particular properties about an individual. A distinction is made between identifying and non-identifying ABCs. Identifying ABCs are typically date of birth, name and social security number, whereas non-identifying ABCs may include hair color or a favorite dish. Furthermore the set of ABCs is context-dependent and dynamic [23]. For the IRMA protocol only the identifying ABCs are relevant and use of the term ABC in the remainder of this document will refer only to this type.

In addition to the property it describes, any ABC is required to contain two additional basic attributes. First, an expiry date has to be determined at issuance, and it is included as an attribute applying to the whole credential. When the credential is verified, the expiry date can be revealed to confirm validity. Second, each user has a master secret key, stored in the smart card's secure storage, which is also incorporated – technically, like an attribute – in all credentials [3].

Before it is possible to make use of ABCs for identification these first have

to be provided by a third party, called an issuer. These issuers are parties who are allowed to give out ABCs. Any issuer cannot simply give out any ABC however. For example a bank may issue an ABC called 'customer ID' for people who are customers with that particular bank, but may not issue credentials such as 'date of birth' for the same people, or any other people for that matter. There are certain roles that issuers may assume and those they may not. Issuers determine the particular attributes they give out, but are verified by a scheme manager. Such a scheme manager effectively facilitates purpose limitation and data minimization.

To summarize, the ecosystem distinguishes four roles:

1. **Users** are people who own a smart card that holds valid ABCs; validity means that the ABCs on the card are valid for the card holder (and are not expired).

2. **Issuers** are the authorities that sign credentials with attributes and provide them to users. For instance, citizen registration authorities are the obvious issuers of 'over 18' ABCs (and of many other ABCs as well) and banks are authoritative issuers of bank account number ABCs.

3. **Verifiers** (also called **relying parties**) are the parties that verify a subset of the available ABCs on a card in order to authorize a transaction. An example verifier is a website that wants to verify the ABC 'over 18' before it allows viewing of a certain video online.

4. The **scheme manager** is an independent, non-profit organization that sets the rules for the different parties (users, issuers and verifiers) and is responsible for the software and smart card management.

Effectively an ABC can be seen as a secure container in which a characteristic from one's digital identity is stored. The attribute values are verified by issuer to ensure they match the individual's characteristics. Once a characteristic is verified the issuer will cryptographically sign the corresponding attribute and store it in the container. A municipality may for instance issue ABCs that state your place of birth and date of birth, which can then be checked by relying parties that require access to this information.

The benefit of using ABCs in favor of regular identification via say passports is the fact that it allows for selective disclosure of characteristics. This enhances the privacy of your (digital) identity by not revealing anything that is not strictly necessary. Consider the following example. You wish to rent a movie that contains violence. These movies typically require a minimum age. In the Netherlands such movies require you to be at least 18 years of age. When you wish to rent such a movie, the cashier asks you to prove you are at least 18

years old. ABCs allow you to digitally tell the cashier that you are 'at least 18 years old' and are allowed to rent the movie. Note that the cashier never learns your actual age.

As stated above the IRMA project makes use of these ABCs, but mainly focuses on efficiency and practical issues when employing them [23]. All applicants receive a smart card that only features a photo of the card holder on the front. No further personally identifiable information is visible on the card.

An IRMA (smart) card is the physical container of attribute-based credentials. As you can see below, the card shows only a picture of its holder but no other personal data. This is important for privacy and security reasons.

(a) Front (b) Back

Figure 2.1: IRMA card

1. Your photo enables others to verify that the card belongs to you.
2. In many situations it is not necessary to reveal your name or date of birth. That's why they are not printed on the card. However, the card may contain these values digitally, stored as ABCs.
3. On the back of the card there is a card number. This number is only used for card administration. For instance, when a new card is handed over to a user, it is easy to find it based on this number.
4. The card number is not visible, but stored digitally inside the card.

2.2 Passport

The passport is a travel document issued by a country's government that certifies the identity and nationality of its holder for the purpose of international travel [10]. A biometric passport is an upgraded version that contains biometric information that can be used to authenticate the identity of travelers. This information is stored on an embedded chip that can be read using contactless smart card technology. Passports that feature such a chip generally feature

the symbol shown in figure 2.2 on the cover. The data stored on the passport chip needs to be protected from modification, cloning, eavesdropping, etc. For this purpose several protection mechanisms have been implemented. Each of these mechanisms exists alongside each other and protects against different types of attacks. This section provides a brief overview of the various security protocols supported by passports, i.e. PA, BAC, PACE, SM and AA as specified by ICAO as well as EAC as specified in by BSI [8, 20].

Passive Authentication

Passive Authentication (PA) is not actually a protocol. It simply indicates that the chip makes use of digital signatures of its data. PA involves the terminal reading the data and verifying both its hash and signature. PA is the only 'protocol' that is ICAO mandatory; all other protocols are optional [27].

Basic Access Control

Figure 2.2: Symbol to indicate a biometric passport

Consider the case where a passport does not feature an RF chip. In this scenario privacy of the passport data is achieved by being able to keep the passport closed so nobody can read its data. With the addition of the RF chip this would no longer hold. Anyone with an RF reader (often called a terminal) is able to read the data on the chip if it is in close proximity. The solution to this problem is to protect the data with a key that the reader needs to know before being allowed to read the data. This is how the Basic Access Control (BAC) protocol works. BAC is essentially not required to be used by ePassports, but is strongly recommended. In the European Union however the use of BAC is mandatory [20]. The key used for BAC is derived from three properties of the passport.

1. The document number (usually 9 digits)
2. The date of birth (formatted YYMMDD in Dutch passports)
3. The document expiry date (formatted YYMMDD in Dutch passports)

These three properties are part of the Machine Readable Zone (MRZ) that is printed in monospace at the bottom of the passport (see figure 2.3). Also part of the MRZ is the Burger Service Nummer (BSN), which is equivalent to the social security number and uniquely identifies a Dutch citizen, but this number is not used for the key derivation for BAC. Combining the aforementioned properties eventually results in the key with which the reader may access the chip's data. Keep in mind the MRZ is not visible while the passport is closed

and cannot be obtained without opening the passport [21]. Strictly speaking the MRZ could be obtained from the citizen database, but additional security checks are in place before it can be accessed. The only method of transferring the MRZ to the reader is for the reader to either utilize optical character recognition (OCR) software or for the passport holder to manually enter the information into the reader. In either case the MRZ is transmitted via an out of band channel. After the key is used for authenticating the reader to the passport, all further communication is performed via an encrypted channel using a session key [20].

In some countries, e.g. the US, the chip is shielded by a very thin metal mesh that is integrated into the cover of the passport [22]. This prevents readers from accessing the chip without having the passport holder open his or her passport first.

Supplemental Access Control

Supplemental Access Control (SAC) was introduced by ICAO in 2009 for addressing BAC weaknesses. It was introduced as a supplement to BAC (for keeping compatibility), but will replace it in the future. In principle it is a set of security features, which specifies the Password Authenticated Connection Establishment (PACE) protocol [20]. PACE is preferred over BAC if it is implemented by a passport. It also derives the session key from the MRZ, but also allows key derivation from the Card Access Number (CAN) that is also present on the front of passports. The protocol uses a weak password (possibly of low entropy), verifies the password, and generates cryptographically strong session keys. It is mandatory from December 2014 onwards [16].

The PACE protocol comprises four steps:

1. The chip randomly chooses a random number, encrypts it with a key derived from the password and sends the encrypted random number to the terminal, where it is recovered.

2. Both the chip and the terminal use a mapping function to map the random number to parameters for asymmetric cryptography.

3. The chip and the terminal perform a Diffie-Hellman protocol based on the parameters generated during step 2.

4. The chip and terminal derive session keys, which are confirmed by exchanging and checking the authentication tokens.

Figure 2.3: Dutch passport with a Machine Readable Zone

Secure Messaging

The Secure Messaging (SM) protocol is used to protect the integrity and confidentiality of the communication between terminal and chip. Essentially the protocol sets up a secure channel, the key for which is agreed upon during the BAC or PACE protocol. When also performing EAC, the CA part will refresh the key (see below).

Extended Access Control

Starting in 2006 many countries added their citizens' finger prints and iris scans to the chip, by definition turning it into a biometric passport. In the Netherlands finger prints used to be stored for identity cards as well, but this decision was reverted and now only the passport holds the finger prints of Dutch citizens [34]. The biometric data is a lot more sensitive than the data protected by BAC, meaning it needs to be protected by stronger cryptography. For this the Extended Access Control (EAC) protocol was developed and added to the second generation of ePassports in 2009 [16]. Any reader that has successfully performed BAC may subsequently perform EAC in order to obtain access to the finger print and iris of the passport holder. EAC is implemented in all new generation EU passports, but nowhere else.

EAC consists of two protocols, Chip Authentication (CA) and Terminal Authentication (TA), both relying on a public key infrastructure in which certificates are issued to the passport as well as other governments for verification purposes. The CA protocol is used for the terminal to authenticate the chip. It is based on a secret key agreement and a successful run of the protocol provides both parties with a new key for SM. The TA protocol is used for authenticating the terminal to the chip and possibly increase access rights. It actively verifies certificates. After both protocols have finished, mutual authentication is achieved and a new secure channel is created.

Active Authentication

Active Authentication (AA) is a challenge-response protocol that proves the authenticity of the chip, verifying the chip has not been cloned. The chip contains a private key, of which the chip proves knowledge during AA, and a certificate for this key as signed by the passport issuing country. This protocol is redundant when the passport also supports EAC, since the CA protocol also (implicitly) proves knowledge of this private key [18]. This means that AA is only useful in cases where passports do not support EAC, but still wish to verify knowledge of the private key.

An example scenario of AA would be a passport that has a private key securely embedded in its chip. The public counterpart of this key is then signed by the passport's producer, Morpho[1] in the Netherlands. Morpho's public key is subsequently signed by the Dutch government, creating a certificate chain. Sending a challenge to the chip, encrypted with its public key would allow the chip to solve the challenge by decrypting it using its private key and thus proving its identity to the terminal.

2.3 Usage scenario

This section describes the basic course of events when a citizen requests a personal IRMA card and would like it to be initialized with his or her passport information. Here we only mention assumptions of operational nature. Goals and assumptions of cryptographic nature are discussed in chapter 4.2.

First of all we assume the citizen in question is an inhabitant of the Netherlands and is in possession of an electronic passport. The citizen will fill out an online application form and enclose a photo in order to request a new IRMA card. Once the application is received by the IRMA card manufacturer a new IRMA card will be created. This card is blank, i.e. there are no ABCs on the card. The application for this card, as well as the blank card itself, will subsequently be forwarded to one of several dozens of assurers.

These assurers may be situated in a government building, for example in a town hall, or they could be notaries, they might be something else entirely as long as they are public figures who either directly or indirectly work for the government. Assurers are in possession of a tablet device that contains a Near-Field Communication (NFC) chip that allows for contactless communication with the IRMA card. This tablet will be kept under lock and key in a safe and is protected with a PIN in order to prevent unauthorized use. At each location only one person (a few at most) will be in posession of this key and PIN. The citizen will go to one of these assurers and present his or her passport to the

[1] `http://www.morpho.com/`

tablet if the assurer confirms that the uploaded photo, which is printed on the front of the IRMA card, matches the citizen.

The data on the passport is verified by the tablet and if confirmed to be valid will be sent to a (the only) central server. The server repeats these checks and also performs several additional checks to definitively verify the passport. Such additional checks may include absence of this particular passport in the database of stolen and lost passports. The server will then proceed to convert the passport's data into ABCs. These ABCs are cryptographically signed by the server, who is the only party in posession of the private part of the only attribute signing key pair, i.e. the issuer key pair. Once the passport data is converted into ABCs the server sends them back to the assurer's tablet. The server keeps a log of the passport number plus the time of the request, but deletes all other traces of the data (both passport and attribute) once it has been received by the assurer's tablet.

The ABCs are received by the tablet and the assurer asks the citizen to present his or her IRMA card. The ABCs are subsequently written to the IRMA card by the tablet. Upon successful transfer of the ABCs, all traces of the entire transaction are deleted from the tablet. The citizen has now successfully added the data from his or her passport to the newly created IRMA card.

Chapter 3

Goals

This chapter focuses on the goals of Assurer. We will describe the goals that should be met by the protocol and are going to be verified by means of a protocol prover. First and foremost, the protocol should ensure the clients will be authenticated to the server and vice versa. These properties should prevent susceptibility to man-in-the-middle attacks. Furthermore, both the all application data sent between client and server should be secret and trustworthy. Moreover, the protocol should be resistant against replay attacks targeting ABCs. Finally the protocol should be designed in such a way that the connection to the server will not cause a denial of service.

To summarize the protocol should offer:

- authentication of client to server;
- authentication of server to client;
- integrity of passport data sent between server and client;
- confidentiality of passport data sent between server and client;
- integrity of ABCs sent between server, client and card;
- confidentiality of ABCs sent between server, client and card;
- resistance against replay attacks.

In addition, we wish to achieve perfect forward secrecy, meaning an attacker cannot derive previous session keys even if private keys are obtained (see chapter 3.1). This means we have to use cryptographically strong keys, which also should be ephemeral.

Note that replay attacks are partly mitigated by policies surrounding the system. For example, replaying of ABCs is only possible should an attacker have

access to the assurer's tablet, which is kept in a safe and is inaccessible. To improve upon this, implementations of the protocol should ensure it does not allow or facilitate storing ABCs internally for repeated use. All ABCs should be stored on the corresponding IRMA card and securely deleted immediately after.

3.1 Attack scenarios

This section describes possible attack scenarios with respect to the goals described above. These are attacks the protocol should be resistant against. The possible impact of such an attack is described directly after the attack itself, along with the likelihood of such an attack occurring.

- An attacker reads the passport data sent between client and server. This attack causes personally identifiable information to be learned, leading to a breach in confidentiality. This may be likely in the event weak cryptography is used or if either party loses its key. However, because of the requirement of perfect forward secrecy this issue is partly mitigated (see section 3.1).
- An attacker modifies the passport data sent between client and server. This attack causes the server to receive data inconsistent with the passport data sent by the client, at worst leading to the issuance of ABCs not corresponding to the passport. This can be mitigated, like the previous scenario, with the use of sufficiently strong cryptography, but is still somewhat likely to occur with a powerful adversary.
- An attacker modifies signed ABCs. This attack causes the client to receive incorrect ABCs, at worst leading to the issuance of ABCs not corresponding to the passport. This is highly unlikely as it would require the attacker to be able to sign the ABCs using the issuer's private key to keep relying parties from detecting the fraud.
- An attacker intercepts ABCs and stores it onto his own card. This attack causes false issuance of ABCs, in turn leading to possible fraud. This may be a likely scenario if the attacker controls the network and is actively attacking the protocol. A mitigating factor is the fact the attacker must first obtain access to the tablet, which as described is kept under lock and key.
- An attacker submits forged passport data to the server. This attack causes the server to possibly provide ABCs for fictitous people, in turn leading to possible fraud. Because the server verifies all passport data before providing ABCs, it may also be possible to query the server, guessing correct personally identifiable information.

- An attacker intercepts and later reuses ABCs already stored onto a card for storing on another card. This attack causes the attacker to be able to commit fraud. This is likely in the case where ABCs are not protected against replay attacks. In Assurer we make use of nonces to ensure timeliness, which should mitigate this issue.

- An attacker replays the sending of passport data to obtain another identical set of ABCs. This attack effectively facilitates the cloning of cards. Since the server is keeping a log of all of the ABCs that are given out a simple lookup would reveal the replay attack, making this an unlikely scenario.

- An attacker uses different ABCs from different people to authenticate. This attack allows combining ABCs to achieve the required combination for authentication. This is hardly an issue, due to the fact the IRMA card features a photo that should resemble the authenticating person. Adding to that the fact that switching cards midway would raise eyebrows leads us to think this attack is unlikely to occur.

- An attacker causes a denial of service on the client. This may cause clients to not receive the requested ABCs, causing them to restart the process. This does not lead to a security risk.

- An attacker causes a denial of service on the server. This halts all activity surrounding the issuance of ABCs, but does not lead to a security risk.

- An attacker shows up with someone else's passport. At worst this could cause incorrect ABCs to be placed onto an IRMA card, however this is mitigated by verifying the photograph printed on the passport. In order to pass this test the attacker has to forge the passport, which is detected upon verification by either the client or the server. This scenario is therefore unlikely.

- An attacker switches out the non-initialized IRMA cards before they reach the assurer's office. This also is a non-issue, because these cards only contain a unique ID at this point in the process. This ID is not used by the relying parties for verification of ABCs and therefore has no impact on the security and privacy of the system.

Perfect Forward Secrecy

In key exchange protocols there is a property called Perfect Forward Secrecy (PFS) that provides a more secure way of encryption with respect to everyday session encryption, because the key is deleted immediately after use and therefore cannot be stolen by an attacker or forced to be handed over by

the government in an attempt to decrypt intercepted traffic [11]. More specifically, the exposure of long-term keying material, used in the protocol to negotiate session keys, does not compromise the secrecy of session keys established before the exposure. This property is especially relevant to scenarios in which exchanged session keys require secrecy protection beyond their lifetime, such as in the case of session keys used for data encryption. This means it is relevant in Assurer, since a lot of sensitive information is being transfered and therefore needs to be kept secret.

The most common way to achieve PFS in a key-exchange protocol is by using the Diffie-Hellman key agreement with ephemeral exponents to establish the value of a session key, while confining the use of the longterm keys (such as private signature keys) to the purpose of authenticating the exchange (see authentication). One essential element for achieving PFS with the Diffie-Hellman exchange is the use of ephemeral exponents which are erased from memory as soon as the exchange is complete. This should include the erasure of any other information from which the value of these exponents can be derived such as the state of a pseudo-random generator used to compute these exponents [24].

3.2 Cryptographic properties

To summarize the previous sections, the protocol should satisfy the following cryptographic properties.

- Authenticity
- Accountability
- Confidentiality
- Integrity
- Availability

Chapter 4

Protocols

Assurer is an application-level protocol. It is designed to run on top of TLS 1.2. Only the latest version of the TLS standard is considered cryptographically secure and therefore is the only version Assurer supports. A second argument for the use of version 1.2 is the possibility to achieve Perfect Forward Secrecy (PFS), meaning that in case the long term key is ever compromised, then the session keys derived from it before compromise are still secure [24]. This chapter will first describe the TLS protocol, explaining the chosen parameters. Assurer follows directly after.

4.1 Transport Layer Security

The Transport Layer Security (TLS) protocol is a protocol operating on the presentation layer of the OSI Reference Model [12]. It is a protocol that secures the connection between two parties across an insecure channel, ensuring secrecy [13]. It also has the possibility for authentication based on certificates. TLS is commonly used in web browsers to encrypt HTTP into HTTPS traffic, but it is capable of protecting any TCP connection [4].

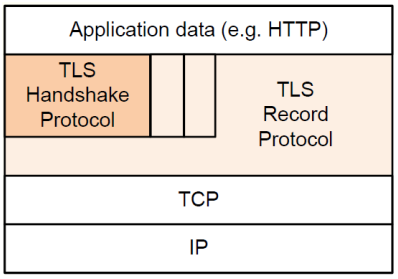

Figure 4.1: TLS Protocol Hierarchy

TLS 1.2 was defined in RFC 5246 in August 2008. It is based on the earlier TLS 1.1 specification. It was further refined in RFC 6176 in March 2011 removing their backward compatibility with SSL such that TLS sessions will never negotiate the use of Secure Sockets Layer (SSL) version 2.0.

The protocol consists of two major parts, TLS Handshake and TLS Record.

Handshake is used for setting up connections between two parties, with optional authentication, while Record is used for ordering, encrypting and sending of the data. Handshake makes use of Record, so they work both alongside as well as on top of each other (see figure 4.1).

TLS handshake

Figure 4.2 shows a schematic overview of the TLS handshake, copied from RFC 5246 [13]. An asterisk indicates a step necessary for client-to-server authentication. The handshake begins when a client connects to a TLS-enabled server requesting a secure connection and presents a list of supported cipher suites (ciphers and hash functions). From this list, the server picks a cipher and hash function that it also supports and notifies the client of the decision. The server usually then sends back its identification in the form of a digital certificate. The certificate usually contains the server name, the trusted certificate authority (CA) and the server's public encryption key. The client may contact the server that issued the certificate (the trusted CA) and confirm the validity of the certificate before proceeding. In order to generate the session keys used for the secure connection, the client encrypts a random number with the server's public key and sends the result to the server. Only the server should be able to decrypt it, with its private key. From the random number, both parties generate a 'master secret' and then negotiate a session key for encryption and decryption.

The details about the underlying cryptographic functions selected for the IRMA Assurer protocol, such as encryption and hashing functions will be discussed later in chapter 4.2.

TLS record

The record protocol handles the sending and receiving of TLS related messages. It forms the basis of the TLS protocol. When sending, it will split the data in blocks, optionally compress it, apply a MAC, encrypt the data, add a fragment header and finally send the data over TCP port 443. When receiving, it will decrypt the data, verify the MAC, optionally decompress, defragment and finally deliver the data to the upper layer. Essentially this protocol forms the secure channel between client and server [13].

Extensions

TLS also supports extensions. These are additional supported features, each with its own specification [15]. These extensions can be used in either party's Hello message to indicate special wishes. This is useful to upgrade the TLS connection to a more secure version by increasing the minimum recommended cryptographic parameters. For example, TLS in its basic form is vulnerable

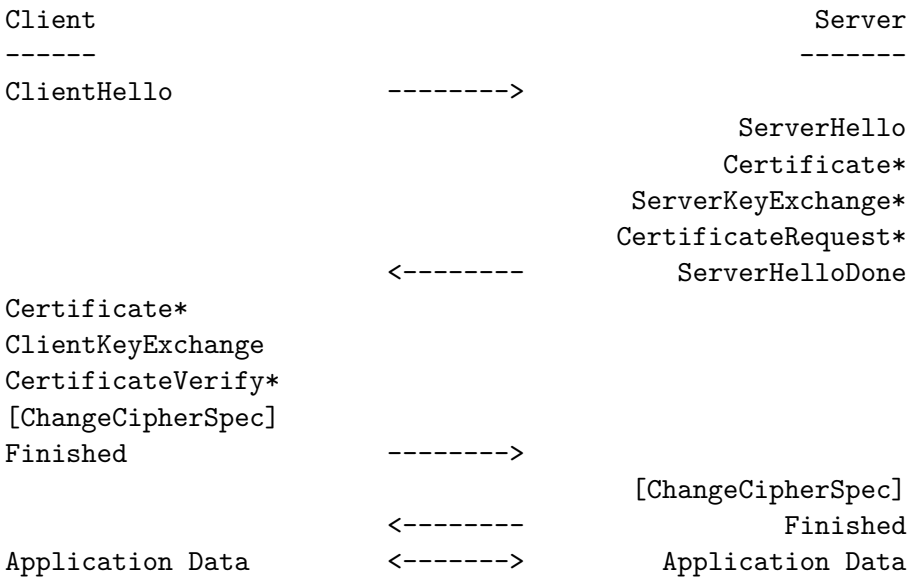

Figure 4.2: TLS Handshake

to a handshake renegotiation attack [4]. In essence an attacker may send arbitrary data followed by a renegotiation intercepted from a regular client to trick a server into believing the arbitrary data was sent by this client instead. An extension has been developed to combat this, which is described in RFC 5746 [33]. We assume this extension to be used on every handshake performed during runs of Assurer and do not explicitly mention it in the following sections.

4.2 IRMA Assurer

Assurer makes use of TLS 1.2 as described above. This means both the server and the tablets will verify each other's certificates and agree upon a session key. Once a secure channel has been successfully set up, we have achieved privacy and data integrity of all communication over this channel [13].

After the secure channel is established the application data is transferred between client and server. This entails the sending of the passport data from the client to the server, followed by the sending of ABCs from the server to the client. Before passport data can be sent however, the client needs to read the chip that is present in the passport using special software and hardware. For the purpose of this protocol we assume the client has already performed this action and has knowledge of the passport data.

Shown in figure 4.3 is the Assurer protocol using the informal Alice-Bob notation. Lines that are not numbered indicate an action outside of the scope of

the protocol or an assumption that needs to hold before the next line can be executed. In this notation we denote the client by `A` and the server by `B`. Note that the Active Authentication steps do not involve the client as it only forwards communication to and from the passport, denoted by `P`. Furthermore, `skP` indicates the private key stored inside the passport and `kAB` is the key that is agreed upon during the handshake.

```
   [Citizen presents passport]
   [Client performs PA]
1. A --> B: ClientHello
2. B --> A: ServerHello, Certificate, ServerKeyExchange,
            CertificateRequest, ServerHelloDone
3. A --> B: Certificate, ClientKeyExchange, CertificateVerify,
            ChangeCipherSpec, {Finished}kAB
4. B --> A: ChangeCipherSpec, {Finished}kAB
5. A --> B: {{Passport, A, B, Na}kAB, #{Passport}kAB}kAB
   [Server performs PA]
6. B --> A --> P: {Nb}kAB
7. P --> A --> B: {{Nb}skP}kAB
   [Server verifies response to AA challenge]
8. B --> A: {{ABCs, A, B, Na}kAB, #{ABCs}kAB}kAB
   [Client stores ABCs on IRMA card]
```

Figure 4.3: IRMA Assurer including TLS handshake

Assumptions

In this section we discuss the assumptions made with regard to Assurer. The section is divided into two parts. The first section discusses assumptions on the operational level and the second describes assumptions with respect to cryptography.

Operational

Operational assumptions mostly focus on procedures that have to be followed for the Assurer protocol to work. We reiterate some parts of the basic course of events described in chapter 2.3 to make them explicit as assumptions.

The entire ecosystem in which Assurer operates can be described as a star structure. At the center of the star will be the (only) server. The edges (points) of the star are formed by the clients. Clients do not communicate with each other, but only with the server. Each of these clients has their own NFC-enabled tablet, which is locked up in a safe and PIN code protected to prevent malicious use. By using an Android app installed on these tablets assurers may access data on passport chips and IRMA cards. Upon client initialization

this app is installed on the tablet, as well as the fully qualified domain name (FQDN) of the server. By using the FQDN the server has the option to switch IP addresses without breaking the system. Also installed on the tablet are a client certificate (signed by the server) required for authentication and its corresponding private key. The server checks the certificate for validity and also checks if it has not been revoked by accessing Certificate Revocation Lists (CRL), which mitigates issues with stolen tablets or misuse. Finally the public key of the server is installed for easy access.

Although subject to change, at the time of writing the communication between IRMA card and terminal, i.e. tablet, is still being sent in the clear. Upon reading data from a passport chip the integrity is verified by the client by executing BAC and PA, and sent to the server for further analysis. The server performs the same checks, plus AA, and looks up the person corresponding to the data in a central database. If everything is proven to be valid the server will create digitally signed attribute-based credentials that correspond to the passport data. The server will then send these ABCs to the client, who will in turn install them on the person's IRMA card. Upon successful installation onto the IRMA card, the ABCs are securely deleted from both server and client. Furthermore, the passport data is securely deleted from the server and client. However, the server does keep a log of the passport number that was part of the attribute signing request sent by the client, along with a timestamp. This facilitates traceability in case of anomalies. Finally, we do not support TLS session resuming. This property allows clients to send a stored session identifier to the server and then pick up where left off. Supporting this weakens the strength of the TLS connection, in particular the PFS property of TLS [37]. Since we wish to have PFS we do not allow session resuming.

Cryptographic

Before going into the cryptographic choices regarding Assurer it is important to take a quick look at the transport layer used below. There are two options, the User Datagram Protocol (UDP) and the Transport Control Protocol (TCP). We have chosen to make use of TCP for the transport layer protocol. The main reason for this is that we require the packets to be delivered without packet loss. In other words, we favor TCP over UDP for its high reliability [19]. Furthermore, it is best compatible with the TLS protocol which we aim to use. Other arguments for choosing TCP include its data flow control and packet reordering capabilities.

In order to achieve the goals described in chapter 3 several choices have been made regarding the cryptography. What follows is an argumentation for the decisions made for Assurer.

The guideline for choosing supported cipher suites is to offer perfect forward secrecy. Any cipher suites that offer PFS capabilities have been selected, while suites that do not are not selected. In principle, any public key encryption scheme can be used to build a key exchange with PFS by using the encryption scheme with ephemeral public and private keys [24]. This means we have the option to use any public key infrastructure as long as we throw away the keys when we are done using them. Furthermore, we can either choose elliptic curve cryptography or conventional cryptography. Because we wish to achieve mutual authentication both parties are required to exchange certificates. These certificates must be of type X.509v3 in accordance with the TLS specification [13]. The type of cryptography used for these certificates may either be RSA or DSA. Generally, in terms of performance, neither is significantly better than the other, meaning we support both.

As explained above we have chosen TLS 1.2 as basis for Assurer for its resistance against publicly known feasible attacks, as well as for its Perfect Forward Secrecy capabilities. This requires that the key generation for the encryption scheme must be fast enough. For most applications this disqualifies, for example, the use of ephemeral RSA public key encryption for achieving PFS, since the latter requires the generation of two long prime numbers for each exchange, a relatively costly operation [24]. For this reason we have selected the Diffie-Hellman Key Exchange protocol. Furthermore because of the faster key generation, better performance and shorter key length while still achieving the same level of security we have chosen to use elliptic curves [30].

Since Assurer is to be used only on newly developed hardware and software it is safe to use the newest cryptography; there should not be any compatibility issues. Mozilla lists the following as the best choice for modern clients in their documentation on server-side TLS [28]. The list is ordered from most recommended to least recommended. An exclamation mark indicates a technique which is forbidden.

Ciphersuite `ECDHE-RSA-AES128-GCM-SHA256, ECDHE-ECDSA-AES128-GCM-SHA256, ECDHE-RSA-AES256-GCM-SHA384, ECDHE-ECDSA-AES256-GCM-SHA384, DHE-RSA-AES128-GCM-SHA256, DHE-DSS-AES128-GCM-SHA-256, kEDH+AESGCM, ECDHE-RSA-AES128-SHA256, ECDHE-ECDSA-AES-128-SHA256, ECDHE-RSA-AES128-SHA, ECDHE-ECDSA-AES128-SHA, ECDHE-RSA-AES256-SHA384, ECDHE-ECDSA-AES256-SHA384, ECDHE-RSA-AES256-SHA, ECDHE-ECDSA-AES256-SHA, DHE-RSA-AES128-SHA-256, DHE-RSA-AES128-SHA, DHE-DSS-AES128-SHA256,DHE-RSA-AES-256-SHA256, DHE-DSS-AES256-SHA, DHE-RSA-AES256-SHA, !aNULL, !eNULL, !EXPORT, !DES, !RC4, !3DES, !MD5, !PSK`

Versions `TLSv1.1, TLSv1.2`

RSA key size `2048`

DH Parameter size `2048`

Elliptic curves `secp256r1, secp384r1, secp521r1` (at a minimum)

Certificate signature `SHA-256`

HSTS `max-age=15724800`

Note that this is the optimal configuration for Mozilla's own servers providing HTTPS connections, which explains the mention of HTTP Strict Transport Security (HSTS). For the Assurer protocol we are only interested in cipher suites supported in TLS 1.2, while making use of elliptic curves and Diffie-Hellman, meaning we can safely ignore anything that is unrelated. Cross-referencing this list of cipher suites with the list of elliptic curve and TLS 1.2 cipher suites described by the OpenSSL documentation yields the following list of cipher suites supported by the Assurer, in descending order of priority [31]. These are named following IANA guidelines, which differs from the naming convention used by Mozilla who use OpenSSL guidelines, but is in correspondence with the RFC documents on TLS [32]. The blank lines indicate Mozilla favors either one or more non-elliptic curves or cipher suites incompatible with TLS 1.2 over the ones that follow.

```
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384

TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256

TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
```

There are many reasons for this particular ordering. We highlight the main reasons below.

- Ciphers making use of ECDHE featuring AES in Galois Counter Mode (GCM) are selected first. These are TLS 1.2 ciphers and not widely supported at the moment. No known attack currently targets these ciphers.
- PFS ciphersuites are preferred, with ECDHE first, then DHE.

- ECDHE provides faster handshakes than DHE [5, 25].
- AES128 is preferred to AES256, because it provides good security, is really fast, and seems to be more resistant to timing attacks.
- SHA256 is favored over SHA384. This appears to be mainly for interoperability purposes.

Galois Counter Mode (GCM) is an Authenticated Encryption (AE) algorithm. AE algorithms are designed to provide both data authenticity (integrity) and confidentiality. Since both are goals of Assurer we favor GCM over CBC, but do not require it explicitly.

For more information about the reasons behind this ordering please see the Mozilla Wiki page on this subject [28]. In August 2015 the NSA has issued a new version of their Suite B Cryptography document, in which they withdraw P-256, SHA-256, and AES-128 in order to start the transition to quantum resistant algorithms [29]. Mozilla however does not yet have an updated document.

Diffie-Hellman parameters

Unfortunately, some widely used clients lack support for ECDHE and must then rely on DHE to provide perfect forward secrecy. This is true for Android $< 3.0.0$, Java < 7 and OpenSSL $< 1.0.0$, among others. Nowadays many, if not all, devices run Android versions higher than 3.0.0, so we don't expect to see issues in this area. However the Java requirement could lead to issues when implementing Assurer as an Android app, mainly because of the many IRMA-related dependencies that such an app has to deal with. These dependencies may not support Java 7 or 8 yet. Adding to that, Java 6 and 7 do not support Diffie-Hellman parameters larger than 1024 bits. This has consequences for the PFS requirement of Assurer. The only way a secure connection can be achieved from a Java 6 client is to use DHE cipher suites and use 1024-bit groups.

Use of the most widely used 1024-bit pre-computed Oakley group 2, standardized by the IETF [17], is considered unsafe, mainly because it is very likely that a state-level adversary may have broken it. In any case it is recommended to generate a random DH group instead of using a standardized one when setting up a new server [1].

CBC ciphers can be attacked with the Lucky Thirteen attack if the library is not written carefully to eliminate timing side channels. This attack requires multiple sessions and is possibly detectable due to the low volume of traffic in Assurer. The attack is mitigated by use of a reasonably up-to-date cryptographic library, i.e. a library released anytime after March 2013. Most of the

industry's libraries have released patches throughout February 2013, which probably makes this a non-issue for Assurer [2].

Logjam When choosing this list of supported cipher suites we have chosen the "modern" preset described by Mozilla. This is especially important, because of the recently discovered Logjam attack on the Diffie-Hellman key agreement, in particular within TLS, SSH and VPN connections [1]. Essentially Logjam is a type of attack that allows an attacker to downgrade the security of the connection to DHE export cipher suites. It is also possible to attack weak ($\leq$ 1024-bit) Diffie-Hellman groups by precomputing the group and then using it for lookups. The short term solution for mitigating Logjam attacks is for servers to disable export ciphers and use freshly generated groups of 2048 bits or larger. Clients should no longer accept groups of sizes lower than 1024 bits. In the long term however it is preferable to switch to elliptic curve cryptography (ECC) and not allow any other cipher suites. This is because none of the attacks presented work against ECC. As described above the Assurer protocol only makes use of a subset of the "modern" preset, which in turn only makes use of ECC cipher suites, and therefore is not susceptible to the Logjam attack.

Elliptic curves NIST has defined 15 standard curves [14]. However, in practice, many implementations only support two of them, P-256 and P-384, because that is what the NSA recommends. As seen above Mozilla recommends to use at the very least P-256, P-384 and P-521. For Assurer we will follow that recommendation.

Certificates

Each client posesses one certificate, signed by the server, with which it may authenticate to the server. This occurs during the run of the TLS handshake protocol. Furthermore each passport contains one certificate with which the authenticity may be verified (see chapter 2.2).

Keys

The server owns two keypairs. The first keypair is used for signing and verification of certificates while the second one is used for the issuing and verification of attribute-based credentials. A client only owns one keypair, which is used for signing and verification of certificates. The server's public key, used for signing and verification, is pre-loaded onto the clients during their initialization, while the clients' public keys are stored within the pre-loaded certificates.

The use of TLS 1.2 with the aforementioned cryptographic options for the handshake will result in a symmetric session key held by both parties. From

this point onwards we therefore have a secure channel to use for application data transfer.

Application data

All application data is protected with the session key as generated by the TLS handshake protocol. This ensures the passport and ABCs are sent over a secure channel. Once this channel has been established the client sends the passport data, including both parties' name and a fresh nonce, encrypted with the session key. Note that this is the same session key negotiated by the TLS handshake, meaning the passport data is essentially encrypted twice. Passive Authentication is then performed by the server. This is a double-check and consequently serves as verification that the client is not malicious. As described in chapter 2.2 PA is ICAO mandatory and involves verification of the hash stored in the SOD file to ensure data integrity. After the integrity of the passport is verified the server continues with Active Authentication. For this it sends a challenge to the passport via the client, which has to be solved by the passport. The passport proves knowledge of a private key by signing the AA challenge using this key, which can in turn be verified by the server using the public key that was included with the passport data sent to the server during PA.

After both PA and AA have been performed the server will proceed with generating attribute-based credentials that resemble the passport data. These ABCs are signed using the issuer key and combined with the names and nonce received during PA. This data is then encrypted and an HMAC is computed, both once again using the same session key, following which the data is sent to the client. The client checks the HMAC and decrypts the data. If the nonce and both parties' name matches the client will store the ABCs on the IRMA card. If the nonce does not match this may indicate a replay attack. If either party's name is incorrect this may indicate a man-in-the-middle attack. In both cases no ABCs will be stored on the IRMA card and the protocol will halt.

Chapter 5

Formalisation

In this chapter we describe the formalisation of the assumptions and use cases into a model written in the Pi Calculus [26]. This model can be proven to be cryptographically secure using ProVerif. We have chosen ProVerif for its ability to automatically analyze the security of cryptographic protocols. ProVerif is capable of proving reachability properties, correspondence assertions, and observational equivalence. These capabilities are particularly useful to the computer security domain since they permit the analysis of secrecy and authentication properties. Moreover, emerging properties such as privacy, traceability, and verifiability can also be considered. Protocol analysis is considered with respect to an unbounded number of sessions and an unbounded message space. Moreover, the tool is capable of attack reconstruction: when a property cannot be proved, ProVerif tries to reconstruct an execution trace that falsifies the desired property [7].

The model we have constructed for Assurer is listed in appendix A. This model is based on the TLS handshake model by Tankink and Vullers [36]. Their model makes use of RSA for the key exchange, but our model uses Elliptic Curve Diffie-Hellman. Similarly to their model we make use of message tagging, as is generally considered good practice. Furthermore, ProVerif has trouble finding attacks on variables that are being computed instead of being declared. Tankink and Vullers solve this by creating a new flag variable and outputting this on the supposed-to-be secret. This way when an attacker is able to obtain the flag, then he must have knowledge of the secret and an attack trace can be found. Furthermore, we have added dead code checks to the model. This helps determine if the model fails midway. Such a failure is indicated by the fact that no falsification is found by ProVerif. Finally, ProVerif appears to have a fixed limit on the amount of RAM that it may use. It cannot allocate more than 2GB of RAM, even though plenty is still available, and will consequently report a fatal error. To work around this it is necessary to do all initialization of fresh names (denoted by `new var` in the

model) at the beginning of both the client and server processes.

The model consists of three distinct phases. During the first phase the TLS handshake protocol is executed. As explained before we make use of ephemeral Elliptic-Curve Diffie-Hellman (ECDHE) key agreement. This means that in this protocol the only use for both parties' keypair is the signing and verification of messages sent during the key agreement. Because of the nature of Diffie-Hellman, i.e. the Discrete Logarith Problem, it is not required to encrypt the protocol parameters as they can be sent in the clear without risk of an eavesdropper learning the agreed upon key. For authentication purposes however, we do require these protocol parameters to be signed by the sending party. This allows the receiving party to verify the identity of the sender before starting an encrypted session [6, 13].

The second phase starts the actual Assurer application protocol, which takes place over the encrypted channel for which the key was negotiated during the TLS handshake of phase one. During this phase the passport is verified on the client's side by using passive authentication. This is done by checking the hash stored in the Security Object file of all the data groups contained within the passport. Once the client is confident that the integrity of the passport holds it proceeds by sending the passport data, plus a fresh nonce, to the server via the encrypted channel. The server will then also perform passive authentication as a double-check. If the server agrees with the client on the integrity of the passport, active authentication is performed. To do this, the server sends a challenge to the passport, through the client, that the passport has to solve. The passport has to sign this challenge in order to prove it has knowledge of the embedded private key. During passive authentication the server has obtained the passport's public key, which is used to verify the response to the challenge.

The third and final phase of the protocol is where the issuing of the attribute-based credentials takes place. As mentioned before the server is the only party that posesses the issuer private key. Every relying party, i.e. a terminal, for example at a supermarket has knowledge of the issuer's public key and is therefore able to verify the signature placed on the ABCs. The issuer's public key is not used in this model, as attribute verification is not included in its scope. The server creates ABCs and encrypts these, along with the nonce sent by the client during phase two as well as both parties' names, using the session key and computes an HMAC over the result. The encrypted data and its HMAC are sent to the client, which in turn verifies the HMAC and decrypts the data. If the decrypted data is found to match the nonce and the names of both parties, the client accepts the ABCs and stores them onto the IRMA card and closes the session.

5.1 Discussion

Here we reflect on the details of the model. This reflection is split into two parts: the handshake part and the Assurer part. For each part we first describe the different functions we use throughout the model, followed by a description of the queries and finally the parties involved. Should the reader not have experience with ProVerif, we would recommend familiarizing oneself with the basics in order to better understand this section [7].

The functions we use for this model are fairly straightforward, apart from the key derivation functions. First of all is the `hash` function which used for both verification of the client's signature and integrity of the `finished` messages. The `hmac` function is used to ensure integrity of both passport data and ABCs. We also have `encrypt` and `decrypt` functions, which are symmetric and use the generated session key. This model does not use assymmetric cryptography for encryption and decryption. It does however use key pairs and certificates, generated by the `keypair` and `cert` functions respectively, but these are used for signing and verification of the Diffie-Hellman parameters sent by the server. The public and private parts of a key pair are obtained using the `pk` and `sk` functions respectively. It is also important to note that the `verify` function does not actually verify a message that was signed, but rather retrieves the public key that is included in the certificate. For actually removing the signature and verifying the message the `unsign` function is used.

Furthermore. we have a pseudo-random number function `PRF` which results in an initialization vector when provided with the master secret. With this initialization vector and the functions `clientK` and `serverK` the client and server derive the session key respectively.

Finally we have the Diffie-Hellman functions `G` and `sm`. The `G` function is essentially not a function, but a constant instead and represents the generator point of the finite cyclic group of points on the elliptic curve. The `sm` function represents scalar multiplication. The modulus is abstracted from in this computation, as it proves to be a large challenge in Pi Calculus, and can be left implied without consequences for the correctness of the model. The equation defines the relation between the generator point of the group and the coefficients, and results in a usable key agreement scheme.

The query section describes the goals that must be met after checking it with ProVerif. These directly reflect the goals we set in chapter 3. First we state that an attacker may not learn `Sa` and `Sb`, which are flags that both `A` and `B` output on the newly generated secure channel `s`, which in turn is encrypted using the key agreed upon during the execution of the handshake.

The next set of queries state that an attacker may not learn `PMSa`, `PMSb`, `MSa` and `MSb`. These are four flags, half of which are output by both `A` and `B` within

the model to allow ProVerif to check that the attacker cannot learn the Pre-Master Secret (`PMS`) and the Master Secret (`M`), respectively. These constraints ensure an attacker cannot decrypt the traffic sent over the encrypted channel.

For the application data we have two more queries to be proved by ProVerif. We add two new flags, `passportFlag` and `abcFlag`, which are used to prove secrecy of the sent passport data and ABCs respectively.

Furthermore we check for the secrecy of the `Finished` messages sent at the end of the handshake by both parties. These messages are already encrypted with the generated session key and for an attacker to have knowledge of the plaintext would mean the channel is not secure.

Next we check the authentication status for both client and server, as we require mutual authentication. This is achieved by ensuring the events `endServerAuth` and `endClientAuth` must always are being preceded by the events `beginServerAuth` and `beginClientAuth` respectively. These are injective queries, which means that all `end`-events must be preceded by exactly one `begin`-event, but it is not required that all `begin`-events lead to an `end`-event. For the queries in our model this means that whenever we observe an `end`-event, and thus assume a party to have authenticated to another, then there must have been a session in which the other party has generated a `begin`-event. As stated previously, this satisfies mutual authentication between both parties when both queries hold.

In addition to the mutual authentication we also wish to check Passive Authentication and Active Authentication. For this reason we have added two more of such injective queries. Finally the last injective query (`beginTransaction` and `endTransaction`) ensures that no ABCs can ever be sent before a client requests them.

Finally, we use two queries to ensure the entire model correctly executes. Essentially the output of `serverFinished` and `clientFinished` on a public channel serves as a dead code check. This query must hold, for if it does not then none of the other queries can be considered to have been proved succesfully, since the model has not been verified fully.

Both the `server` and the `client` process have been designed to closely resemble the specifications of RFC 5246 [13]. Differences include message tagging, as mentioned before, for easier reading and ensuring correct execution of the model. The `server` process also features replication, which allows for multiple parallel runs of the process, in turn allowing the server to accept sessions from multiple clients. This represents the star-architecture of the client-server connections. Note that a `client` process will always initiate the TLS handshake, since it is the client who requests passport data to be turned into ABCs. Another important difference from the specification is that at the

end both processes output the flags mentioned above onto a public channel to allow for secrecy and dead code checks.

The system shows the `client` process creates a passport object from a DataGroup, which is the internal representation of a passport's data. There are 16 of these DataGroups in total, one of which (DG15) is used for storing the public key of the passport. In our model we have simplified this to a single DataGroup and have created a separate variable to contain the keypair, the public part of which is otherwise stored in DG15. These abstractions are simply to allow for less variables and thus faster verification, without impacting the security proofs. The passport sent to the server therefore contains the DataGroups, a hash of these DataGroups called the Security Object (SOD file) and the public key. Also sent along are both parties' names and a nonce. This passport is protected by an HMAC generated using the session key.

Upon receiving the passport the server checks the hash and thus performs Passive Authentication (PA). Only after PA is performed will the server proceed with Active Authentication. This simply involves the server generating a nonce (challenge) the passport has to sign (response) to prove it has the secret key corresponding to the public key within the passport.

After both PA and AA are complete the server proceeds by creating ABCs. In the model it creates a single characteristic named `Char` and signs it using the issuer key. The attribute data is then encrypted along with both parties' names and the previously received nonce. This is then sent to the client along with another HMAC generated using the session key. Upon receiving this the client checks the nonce and the names. At the end of both processes the parties output another flag to verify an attacker cannot obtain the passport data or the ABCs, as well as output the dead code check. Once again, if the dead code check does not pass, the proof of all other queries does not hold.

Apart from the aforementioned two processes there is also the `initializer` process that is not part of the TLS specification. This process handles the task of setting up keypairs and certificates for both client and server. This initialization takes place on separate (secure) channels, to mimic the preloading of certificates and keypairs that takes place when a person applies for a new IRMA card as well as the first (and only) time the server is being set up. It also initializes the `server` process with a secondary key pair used for issuing of ABCs. This issuer key pair is sent over yet another private channel to make sure the `server` process cannot mistake one for the other. To ensure ProVerif has all available non-secret parameters we let the `initializer` process publish those to a public channel, otherwise some attacks may not be discovered.

Chapter 6

Conclusion

We have developed Assurer, a protocol for migrating personally identifiable information from identity documents to attribute-based credentials on IRMA cards. For this we have set several cryptographic goals and made assumptions regarding the use of the protocol. The protocol should satisfy authenticity, accountability, confidentiality, integrity and availability.

We have set out to prove these goals hold under the assumptions made, a task for which it is required to create a model of our protocol in the Pi Calculus and subsequently verify it using the ProVerif cryptographic protocol verifier. The creation of such a model has proven to be an error-prone task, mainly due to the limited amount of available literature on the subject, but also due to memory limitations of the software used. Thankfully, with the help of Jerry den Hartog, assistant professor in the security group at Eindhoven University of Technology, as well as Bruno Blanchet, head of research at the INRIA research institution, it was possible to overcome these issues.

Using ProVerif we have provided proof that all of our goals, save availability, are met by the protocol. Since availability in our case constitutes only of the rule that clients must not cause an amount of traffic sufficiently high as to cause a denial of service on the server side. Unlike the other goals this is not really a cryptographic requirement, but instead more of a usability requirement, meaning we cannot use ProVerif for proof that this goal is satisfied.

6.1 Future work

The client in this protocol performs the task of storing the ABCs received from the server on a citizen's IRMA card. While several checks are performed to ensure honesty of clients this may still be undesirable. One would probably wish for less possible interference with the ABC data. Future work on this protocol might focus on making the transaction process of ABCs more opaque to the client, who would then only serve as a non-transparent tunnel through

which the data is flowing. This in turn would mean the IRMA logic has to be moved from the client to the server, undoubtedly presenting new challenges.

Furthermore, at the time of writing the communication between IRMA card and the terminal is still being sent in the clear, i.e. without any cryptography. A valueable addition obviously would be to improve upon this area by creating a secure channel, which in fact would be a requirement for making the transaction process opaque as described above.

Finally one might wish to use this protocol for a software implementation. Due to the nature of IRMA and its implementations it is advisable to use Java for this. I have developed a prototype implementation of the protocol in order to get a good feel for the issues at hand. This implementation however remains unfinished, because it is now partly obsolete with the availability of IRMA's self-enrollment option via smart phones.

Appendix A

Model

A.1 Definitions

```
(* A public channel *)
free net.

(* Message tags *)
free ClientHello, ClientCertificateRequest, ClientCertificate,
     ClientKeyExchange, CertificateVerify, ClientChangeCipher-
     Spec, ClientFinished.
free ServerHello, ServerCertificate, ServerKeyExchange, Server-
     HelloDone, ServerChangeCipherSpec, ServerFinished.
free ClientPassport, ActiveAuthenticationChallenge,
     ActiveAuthenticationReponse.

(* Agent initialization is done over a private channel *)
private free clientInit, serverInit, initChannel.

(** Initialization functions **)
(* Generating certificates for agents *)
private fun cert/2.
(* Generating assymmetric keypairs for agents *)
private fun keypair/1.

(** The cryptographic constructors **)
fun hash/1.          (* Hashing *)
fun hmac/2.          (* Keyed-hash message authentication code *)
fun encrypt/2.       (* Symmetric key encryption *)
fun sign/2.          (* Public key signing *)
fun sk/1.            (* Extracts secret key of a keypair *)
fun pk/1.            (* Extracts public part of a keypair *)
```

```
(** The cryptographic destructors **)
(* symmetric key decryption *)
reduc decrypt(encrypt(x, y), y) = x.
(* signature verification *)
reduc unsign(sign(x, sk(y)), pk(y)) = x.
(* verification of the agent as owner of the
   key and retrieving the key from the certificate *)
reduc verify(cert(x, y), x) = y.

(* Pseudo-random-number function for generating TLS session key
   randomness *)
fun PRF/1.

(* Symmetric key construction *)
fun clientK/3.
fun serverK/3.

(* Diffie-Hellman computations *)
fun G/0.                (* Generator point of the group *)
fun sm/2.               (* Scalar multiplication *)
(* Equality property: x × yG = y × xG *)
equation sm(y, sm(x, G)) = sm(x, sm(y, G)).
```

A.2 Queries

```
(* secrecy secure channel *)
private free Sa.
private free Sb.
query attacker: Sa.
query attacker: Sb.

(* secrecy passport *)
private free passportFlag.
query attacker: passportFlag.

(* secrecy ABCs *)
private free abcFlag.
query attacker: abcFlag.

(* secrecy Pre Master secret *)
private free PMSa.
private free PMSb.
```

```
query attacker: PMSa.
query attacker: PMSb.

(* secrecy Master secret *)
private free MSa.
private free MSb.
query attacker: MSa.
query attacker: MSb.

(* secrecy Finished messages *)
private free FinishedAFlag.
query attacker: FinishedAFlag.
private free FinishedBFlag.
query attacker: FinishedBFlag.

(* authenticity of the server *)
query evinj: endServerAuth(x, y, z) ⟹ evinj: beginServer-
Auth(x, y, z).

(* authenticity of the client *)
query evinj: endClientAuth(x, y, z) ⟹ evinj: beginClient-
Auth(x, y, z).

(* Passport checks *)
query evinj: endPassiveAuth(x, y, z) ⟹ evinj: beginPassive-
Auth(x, y, z).
query evinj: endActiveAuth(x, y, z) ⟹ evinj: beginActive-
Auth(x, y, z).

(* ABC transaction check *)
query evinj: endTransaction(x, y, z) ⟹ evinj: beginTrans-
action(x, y, z).

(* Dead code check *)
private free clientFinished.
private free serverFinished.
query attacker: clientFinished.
query attacker: serverFinished.
```

A.3 Server process

```
let Server =
(** Start of initialization **)
```

```
(* B receives initial agent data over a trusted channel *)
in(serverInit, (B, serverKeypair, serverCert));

(* B receives the issuer keypair over another trusted channel,
   double checking to make sure it is in fact intended for B *)
in(initChannel, (=B, issuerKeypair, issuerCert));

(* B retrieves the secret keys from both keypairs *)
(* Secret key used for communication *)
let SKs = sk(serverKeypair) in
(* Secret key used for signing of attributes-based credentials *)
let SKi = sk(issuerKeypair) in

(** End of initialization **)

(** Start of TLS handshake **)

(* Replication to model arbitrary sessions *)
!

(* B receives ClientHello from A *)
in(net, CH); let (=ClientHello, A, Na, SupportedOptions) = CH in

(* B generates fresh nonce Nb *)
new Nb;

(* B creates a new characteristic, such as "student" or "male" *)
new Char;

(* B picks a cipher suite and compression method from
   SupportedOptions received from A *)
new SelectedOptions;

(* B sends ServerHello to A *)
let SH = (ServerHello, B, Nb, SelectedOptions) in out(net, SH);

(* B sends ServerCertificate to A *)
let SC = (ServerCertificate, serverCert) in out(net, SC);

(* B generates Diffie-Hellman Key Exchange parameters *)
new p; (* Prime modulus *)
new n; (* Order of generator point G *)
new h; (* Cofactor *)
```

```
new a; (* Elliptic curve parameter 1 *)
new b; (* Elliptic curve parameter 2 *)
new beta; (* Server's secret multiplier *)

(* B sends ServerKeyExchange to A, featuring all public ECDHE
   parameters *)
let SKE = (ServerKeyExchange, p, a, b, G, n, h, sm(beta, G),
sign((Na, Nb, p, a, b, G, n, h, sm(beta, G)), SKs)) in out(net,
SKE);

(* B creates a list of acceptable certificate types and CAs *)
new Acceptable_certificate_types;
new Acceptable_certificate_authorities;

(* B sends ClientCertificateRequest to A *)
let CCR = (ClientCertificateRequest, Acceptable_certificate_
types, Acceptable_certificate_authorities) in out(net, CCR);

(* B sends ServerHelloDone to A *)
let SHD = ServerHelloDone in out(net, SHD);

(* B receives ClientCertificate from A *)
in(net, CC); let ( = ClientCertificate, clientCert) = CC in

(* B receives ClientKeyExchange from A, containing the remaining
   ECDHE parameters *)
in(net, CKE); let( = ClientKeyExchange, AG) = CKE in

(* B receives CertificateVerify from A *)
let unsignKey = verify(clientCert, A) in
in(net, CV); let (=CertificateVerify, cvHash) = unsign(CV, un-
signKey) in

(* B verifies client signature *)
let = cvHash = hash((CH, SH, SC, SKE, CCR, SHD, CC, CKE)) in

(** End of Client authentication **)
event endClientAuth(A, B, cvHash);

(* B calculates the Pre-Master Secret *)
let PMS = sm(beta, AG) in

(* B receives ClientChangeCipherSpec from A *)
in(net, CCCS); let = ClientChangeCipherSpec = CCCS in
```

```
(* B receives Finished from A *)
in(net, FA);

(* B calculates the Master secret *)
let M = PRF((PMS, Na, Nb)) in

(* B calculates Finished *)
let Finished = hash((CH, SH, SC, SKE, CCR, SHD, CC, CKE, CV,
CCCS, M)) in

(** Start of Server authentication **)
event beginServerAuth(A, B, Finished);

(* B sends ServerChangeCipherSpec to A, indicating intention to
   switch to the encryption negotiated above *)
let SCCS = ServerChangeCipherSpec in out(net, SCCS);

(* B sends Finished to A *)
out(net, encrypt(Finished, serverK(Na, Nb, M)));

(* B verifies received Finished *)
let = Finished = decrypt(FA, clientK(Na, Nb, M)) in

(** End of TLS handshake **)

(** Start of application data **)

(* B creates a new channel secured with the newly agreed upon
   session key *)
new s;

(* B sends the channel to A after encrypting *)
out(net, encrypt(s, serverK(Na, Nb, M)));

(** Start of application data **)

(* B receives the encrypted passport and HMAC from A *)
in(s, CP); let (=ClientPassport, EP, HP) = CP in
(* B decrypts the passport data *)
let DP = decrypt(EP, clientK(Na, Nb, M)) in
(* B verifies the HMAC *)
let = HP = hmac(EP, clientK(Na, Nb, M)) in
(* B verifies the participants and stores the nonce *)
```

```
let (Passport,  = A,  = B, Nonce) = DP in
(* B stores passport data, public key and hash *)
(* Note: If BAC is required, everything is inaccessible until BAC
   is performed *)
(* We assume BAC is performed once we get to this part *)
let (DG, SOD, PKp) = Passport in

(* B performs Passive Authentication *)
let = SOD = hash(DG) in

event endPassiveAuth(A, B, Passport);

(** Start of Active Authentication **)

(* B creates a challenge *)
new AAC;

event beginActiveAuth(A, B, AAC);

(* B sends ActiveAuthenticationChallenge to A *)
out(s, (ActiveAuthenticationChallenge, AAC));

(* B receives ActiveAuthenticationResponse from A *)
in(s, AAR); let (=ActiveAuthenticationReponse, AAResp) = AAR in

(* B verifies that it is in fact the correct solution to the
   challenge *)
let = AAC = unsign(AAResp, PKp) in

(** End of Active Authentication **)
event endActiveAuth(A, B, AAC);

(** Start of ABC logic **)

(* B signs the characteristic, turning it into an ABC,
   using the issuer's private key *)
let ABC = sign(Char, SKi) in

event beginTransaction(A, B, ABC);

(* B creates attribute data (AD) containing the credential,
   participants and the previously received nonce *)
let AD = (ABC, A, B, Nonce) in
```

```
(* B encrypts the attribute data (EA) *)
let EA = encrypt(AD, serverK(Na, Nb, M)) in

(* B generates the HMAC of the encrypted attributes (HA) *)
let HA = hmac(EA, serverK(Na, Nb, M)) in

(* B combines both parts and sends the message to A *)
let ABCdata = (EA, HA) in out(s, ABCdata);

(** End of ABC logic **)

(** End of application data **)

(* Secrecy checks *)
(
  (* secrecy check on the secure channel *)
  (out(s, Sb)) |

  (* secrecy of ABC data *)
  (out(ABCdata, abcFlag)) |

  (* secrecy check on the Master secret *)
  (out(M, MSb)) |

  (* secrecy check on the Pre-Master Secret *)
  (out(PMS, PMSb)) |

  (* secrecy check on the finished **)
  (out(Finished, FinishedBFlag)) |

  (* dead code check *)
  (out(net, serverFinished))
).
```

A.4 Client process

```
let Client =
(** Start of initialization **)

(* A receives initial agent data over a trusted channel *)
in(clientInit, (A, B, clientKeypair, clientCert));

(* A extracts the secret key from the keypair *)
```

```
let SKc = sk(clientKeypair) in

(** End of initialization **)

(** Start of TLS handshake **)

(* A generates fresh nonce Na *)
new Na;

(* A lists supported cipher suites and compression methods *)
new SupportedOptions;

new Np; (* A generates fresh nonce for passport communication *)
new P;  (* A creates passport agent (required for the model) *)
new DG; (* A creates the DataGroups *)
let SOD = hash(DG) in (* Contains hashes of all DG values *)
let passportKeypair = keypair(P) in
let Passport = (DG, SOD, pk(passportKeypair)) in

(* A sends ClientHello to B *)
let CH = (ClientHello, A, Na, SupportedOptions) in out(net, CH);

(* A receives ServerHello from B *)
in(net, SH); let (=ServerHello, =B, Nb, SelectedOptions) = SH in

(* A receives ServerCertificate from B *)
in(net, SC); let (=ServerCertificate, serverCert) = SC in

(* A receives ServerKeyExchange from B and stores parameters *)
in(net, SKE); let (=ServerKeyExchange, p, a, b, =G, n, h, BG,
DHSignature) = SKE in

(* A retrieves the server's public key from its certificate *)
let unsignKey = verify(serverCert, B) in

(* A checks the signature on the parameters to ensure the message
   was really sent by B *)
let (=Na, =Nb, =p, =a, =b, =G, =n, =h, =BG) = unsign(DHSignature,
unsignKey) in

(* A receives ClientCertificateRequest from B *)
in(net, CCR); let (=ClientCertificateRequest, Acceptable_certifi-
cate_types, Acceptable_certificate_authorities) = CCR in
```

```
(* A receives ServerHelloDone from B *)
in(net, SHD); let = ServerHelloDone = SHD in

(* A sends ClientCertificate to B *)
let CC = (ClientCertificate, clientCert) in out(net, CC);

(* A generates a new secret multiplier *)
new alpha;

(* A sends ClientKeyExchange to B *)
let CKE = (ClientKeyExchange, sm(alpha, G)) in out(net, CKE);

(* A creates a hash of the past messages *)
let cvHash = hash((CH, SH, SC, SKE, CCR, SHD, CC, CKE)) in

(** Start of client authentication **)
event beginClientAuth(A, B, cvHash);

(* A sends CertificateVerify to B *)
let CV = sign((CertificateVerify, cvHash), SKc) in out(net, CV);

(* A computes the pre-master secret (X × YG) *)
let PMS = sm(alpha, BG) in

(* A computes the master secret *)
let M = PRF((PMS, Na, Nb)) in

(* A sends ClientChangeCipherSpec, indicating intention to switch
   to the encryption negotiated above *)
let CCCS = ClientChangeCipherSpec in out(net, CCCS);

(* A computes Finished using the hash function *)
let Finished = hash((CH, SH, SC, SKE, CCR, SHD, CC, CKE, CV,
CCCS, M)) in

(* A sends Finished to B *)
out(net, encrypt(Finished, clientK(Na, Nb, M)));

(* A receives ServerChangeCipherSpec from B, indicating a switch
   to the encryption negotiated above *)
in(net, SCCS); let = ServerChangeCipherSpec = SCCS in

(* A receives Finished from B *)
in(net, FB);
```

```
(* A verifies received finished *)
let = Finished = decrypt(FB, serverK(Na, Nb, M)) in

(** End of server authentication **)
event endServerAuth(A, B, Finished);

(** End of TLS handshake **)

(* A receives the secure channel created by the server *)
in(net, newChannel); let s = decrypt(newChannel, serverK(Na, Nb,
M)) in

(** Start of application data **)

(** Start of Passive Authentication **)
event beginPassiveAuth(A, B, Passport);

(* A encrypts the passport using the session key *)
let EP = encrypt((Passport, A, B, Np), clientK(Na, Nb, M)) in

(* A computes the HMAC of the encrypted passport,
   again reusing the session key *)
let HP = hmac(EP, clientK(Na, Nb, M)) in

(* A combines both parts and sends the combined message *)
let CP = (ClientPassport, EP, HP) in out(s, CP);

(** Start of Active Authentication **)

(* A receives a challenge from B *)
in(s, AAC); let ( = ActiveAuthenticationChallenge, Ch) = AAC in

(* A sends the response to the AA challenge *)
let AAR = (ActiveAuthenticationReponse, sign(Ch, sk(passport-
Keypair))) in out(s, AAR);

(** End of Active Authentication **)

(** Start of ABC logic **)

(* A receives the combined message from B *)
in(s, ABCs); let (EA, HA) = ABCs in
```

```
(* A checks the HMAC *)
let = HA = hmac(EA, serverK(Na, Nb, M)) in

(* A decrypts the Attribute Data (AD) *)
let AD = decrypt(EA, serverK(Na, Nb, M)) in

(* A verifies the participants and nonce and stores the ABC(s) *)
let (ABCdata,  = A,  = B,  = Np) = AD in

event endTransaction(A, B, ABCdata);

(** End of ABC logic **)

(** End of application data **)

(** Secrecy checks **)
(
  (* secrecy of secure channel *)
  (out(s, Sa)) |

  (* secrecy of passport data *)
  (out(CP, passportFlag)) |

  (* secrecy check on the Master secret *)
  (out(M, MSa)) |

  (* secrecy check on the Pre-Master Secret *)
  (out(PMS, PMSa)) |

  (* secrecy check on the Finished message *)
  (out(Finished, FinishedAFlag)) |

  (* dead code check *)
  (out(net, clientFinished))
).
```

A.5 Initializer process

```
let initializer =
(* Generate agent names (unique) *)
new C;
new S;
```

```
(* Generate keypairs *)
let clientKeypair = keypair(C) in
let serverKeypair = keypair(S) in
let issuerKeypair = keypair(S) in

(* Generate certificates *)
let clientCert = cert(C, pk(clientKeypair)) in
let serverCert = cert(S, pk(serverKeypair)) in
let issuerCert = cert(S, pk(issuerKeypair)) in
(
  (* Initialize agents *)
  out(clientInit, (C, S, clientKeypair, clientCert)) |
  out(serverInit, (S, serverKeypair, serverCert)) |
  out(initChannel, (S, issuerKeypair, issuerCert)) |

  (* Publish all non-secret information, otherwise we might miss
     attacks *)
  out(net, (C, S, clientCert, serverCert))
).
```

A.6 System

```
process !initializer | !Client | !Server
```

Bibliography

[1] David Adrian et al. *Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice*. Tech. rep. University of Michigan, May 2015. URL: https://weakdh.org/imperfect-forward-secrecy.pdf.

[2] Nadhem J. AlFardan and Kenneth G. Paterson. *Lucky Thirteen: Breaking the TLS and DTLS Record Protocols*. Tech. rep. Egham Hill, Egham, Surrey TW20 0EX United Kingdom: Royal Holloway University of London, Feb. 2013.

[3] Gergely Alpár and Bart Jacobs. 'Credential Design In Attribute-Based Identity Management'. In: *Bridging distances in technology and regulation, 3rd TILTing Perspectives Conference*. 2013, pp. 189–204.

[4] Tuomas Aura. *Network Security: TLS/SSL*. Lecture in Network Security course. Nov. 2010. URL: http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf.

[5] Vincent Bernat. *SSL/TLS & Perfect Forward Secrecy*. Nov. 2011. URL: http://vincent.bernat.im/en/blog/2011-ssl-perfect-forward-secrecy.html (visited on 21/08/2015).

[6] Simon Blake-Wilson et al. *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*. Informational. Internet Engineering Task Force (IETF) Network Working Group, May 2006. URL: https://tools.ietf.org/html/rfc4492.

[7] Bruno Blanchet, Ben Smyth and Vincent Cheval. *ProVerif 1.91: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*. 2015. URL: http://prosecco.gforge.inria.fr/personal/bblanche/proverif/manual.pdf.

[8] German Federal Office for Information Security (BSI). *Advanced Security Mechanisms for Machine Readable Travel Documents and eIDAS Token – Part 1 — eMRTDs with BAC/PACEv2 and EACv1 — version 2.20*. Tech. rep. TR-03110-1. Bonn, Germany: German Federal Office for Information Security (BSI), Feb. 2015.

[9] Jan Camenisch, Stephan Krenn and Victor Shoup. 'A Framework for Practical Universally Composable Zero-Knowledge Protocols'. In: *Advances in Cryptology – ASIACRYPT 2011*. Ed. by DongHoon Lee and Xiaoyun Wang. Vol. 7073. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 449–467. ISBN: 9783642253843. DOI: 10.1007/978-3-642-25385-0_24.

[10] Peter Cane and Joanne Conaghan. *The New Oxford Companion to Law*. Oxford University Press, 2008. ISBN: 9780199290543.

[11] Tom Chothia. *Secure Communication*. Lecture in Computer Security course. 2013. URL: http://www.cs.bham.ac.uk/internal/courses/comp-sec/2013/Lectures/8_SSL/8_RemoteAuth.pdf.

[12] John D. Day and Hubert Zimmermann. 'The OSI reference model'. In: *Proceedings of the IEEE* 71.12 (Dec. 1983), pp. 1334–1340. ISSN: 0018-9219. DOI: 10.1109/PROC.1983.12775.

[13] Tim Dierks and Eric Rescorla. *The Transport Layer Security (TLS) Protocol – Version 1.2*. Proposed Standard. Internet Engineering Task Force (IETF) Network Working Group, Aug. 2008. URL: https://tools.ietf.org/html/rfc5246.

[14] National Institute of Standards and Technology (NIST). *Digital Signature Standard (DSS)*. Tech. rep. FIPS PUB 186-4. Gaithersburg, MD 20899-8900: National Institute of Standards and Technology (NIST), July 2013. URL: http://dx.doi.org/10.6028/NIST.FIPS.186-4.

[15] Donald Eastlake 3^{rd}. *Transport Layer Security (TLS) Extensions: Extension Definitions*. Proposed Standard. Internet Engineering Task Force (IETF) Network Working Group, Sept. 2010. URL: https://tools.ietf.org/html/rfc6066.

[16] Gemalto. *Moving to the third generation of electronic passports – A new dimension in electronic passport security with Supplemental Access Control (SAC)*. Tech. rep. Gemalto, Oct. 2011. URL: http://www.securitydocumentworld.com/creo_files/upload/client_files/moving_to_the_third_generation_of_electronic_passports_october_20111.pdf (visited on 14/09/2015).

[17] Dan Harkins and Dave Carrel. *The Internet Key Exchange (IKE)*. Tech. rep. Internet Engineering Task Force (IETF) Network Working Group, 1998. URL: https://tools.ietf.org/html/rfc2409.

[18] Jaap-Henk Hoepman et al. 'Crossing Borders: Security and Privacy Issues of the European e-Passport'. In: *Advances in Information and Computer Security*. Ed. by Hiroshi Yoshiura et al. Vol. 4266. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 152–167. ISBN: 9783540476993. DOI: 10.1007/11908739_11.

[19] Information Sciences Institute. *Transmission Control Protocol – DARPA Internet Program – Protocol specification*. Tech. rep. 4676 Admiralty Way, Marina del Rey, California 90291: University of Southern California, Sept. 1981. URL: https://tools.ietf.org/html/rfc793.

[20] ISO/IEC JTC1 SC17 WG3/TF5. *Supplemental Access Control for Machine Readable Travel Documents – Part 1–3*. Tech. rep. Doc 9303. International Civil Aviation Organization (ICAO), Nov. 2010.

[21] Thomas Kinneging. 'Basic Access Control and Extended Access Control in ePassports'. In: *Technical Advisory Group on Machine Readable Travel Documents – Eighteenth meeting*. International Civil Aviation Organization (ICAO). May 2008. URL: http://www.icao.int/Meetings/TAG-MRTD/Documents/Tag-Mrtd-18/Kinneging.pdf.

[22] Kurt Kleiner. *Metal shields and encryption for US passports*. News article. Oct. 2005. URL: https://www.newscientist.com/article/dn8227-metal-shields-and-encryption-for-us-passports (visited on 18/06/2015).

[23] Paulan Korenhof et al. 'The ABC of ABC: An analysis of attribute-based credentials in the light of data protection, privacy and identity'. In: *Internet, Law and Politics — A decade of transformations*. Ed. by Joan Balcells Padullés et al. 1^{st} edition. Vol. 10. Barcelona: Huygens Editorial, June 2014, pp. 357–374. ISBN: 9788469708262.

[24] Hugo Krawczyk. 'Perfect Forward Secrecy'. In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg. Springer US, 2005, pp. 457–458. ISBN: 9780387234731. DOI: 10.1007/0-387-23483-7_298. URL: http://dx.doi.org/10.1007/0-387-23483-7_298.

[25] Nikos Mavrogiannopoulos. *The price to pay for perfect-forward secrecy*. Dec. 2011. URL: http://nmav.gnutls.org/2011/12/price-to-pay-for-perfect-forward.html (visited on 21/08/2015).

[26] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999. ISBN: 9780521658690.

[27] Wojciech Mostowski and Erik Poll. *Electronic Passports in a Nutshell*. Tech. rep. ICIS–R10004. Radboud University Nijmegen, June 2010. URL: https://pms.cs.ru.nl/iris-diglib/src/getContent.php?id=2010-Mostowski-ElectronicNutshell.

[28] *Mozilla Wiki — Server Side TLS*. Version 3.8. Oct. 2013. URL: https://wiki.mozilla.org/index.php?title=Security/Server_Side_TLS&oldid=1092713.

[29] National Security Agency (NSA). *Suite B - Cryptography Today*. Jan. 2009. URL: https://www.nsa.gov/ia/programs/suiteb_cryptography/index.shtml (visited on 22/08/2015).

[30] National Security Agency (NSA). *The Case for Elliptic Curve Cryptography*. 2009. URL: https://www.nsa.gov/business/programs/elliptic_curve.shtml (visited on 07/06/2015).

[31] OpenSSL Software Foundation. *OpenSSL manpage – ciphers*. 2015. URL: https://www.openssl.org/docs/manmaster/apps/ciphers.html (visited on 03/08/2015).

[32] Eric Rescorla. *Transport Layer Security (TLS) Parameters*. Last updated: 10-07-2015. 2005. URL: http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml (visited on 03/08/2015).

[33] Eric Rescorla et al. *Transport Layer Security (TLS) Renegotiation Indication Extension*. Proposed Standard. Internet Engineering Task Force (IETF) Network Working Group, Feb. 2010. URL: https://tools.ietf.org/html/rfc5746.

[34] Rijksoverheid. *Paspoort twee keer zo lang geldig, ID-kaart zonder vingerafdrukken*. Dutch. Jan. 2014. URL: http://www.rijksoverheid.nl/nieuws/2014/01/10/paspoort-twee-keer-zo-lang-geldig-id-kaart-zonder-vingerafdrukken.html.

[35] Ahmad Sabouri, Ioannis Krontiris and Kai Rannenberg. 'Attribute-Based Credentials for Trust (ABC4Trust)'. In: *Trust, Privacy and Security in Digital Business*. Ed. by Simone Fischer-Hübner, Sokratis Katsikas and Gerald Quirchmayr. Vol. 7449. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 218–219. ISBN: 9783642322860. DOI: 10.1007/978-3-642-32287-7_21.

[36] Carst Tankink and Pim Vullers. 'Verification of the TLS Handshake protocol'. May 2008.

[37] Tim Taubert. *The sad state of Server-Side TLS session resumption implementations*. Nov. 2014. URL: https://timtaubert.de/blog/2014/11/the-sad-state-of-server-side-tls-session-resumption-implementations (visited on 23/07/2015).