Radboud University Nijmegen

Master Thesis

Improving smart grid security using smart cards

Author:
Marlon Baeten
marlon@utf8.nl

Supervisors:
dr. ir. Erik Poll
dr. Bárbara Vieira

August 20, 2014

Abstract

The rapid changes in the electrical grid demand a smarter information infrastructure. The C-DAX project proposes a secure integrated communication and information infrastructure for future power distribution networks. Securing such an infrastructure is a difficult task since it has a flexible and distributed nature. The use of smart cards is an industry-proven solution for the difficult tasks of key distribution, key storage, and secure cryptographic algorithm execution. In this thesis we investigate to what extent the security functionalities in the C-DAX solution can be moved to a smart card, given its performance and throughput constraints. Moving parts of the security tasks to a smart card requires both a performance evaluation of smart cards and an analysis of the security implications. We implemented and analyzed different C-DAX security functionalities on a smart card. We propose a secure smart card installation method, including the design and evaluation of an activation protocol. In addition, we carried out an extensive benchmark of the smart card performance. The results show that our implementation can successfully be applied in within the C-DAX infrastructure.

Contents

1 Introduction **4**

1.1 The C-DAX project 4

1.2 Smart cards 6

1.3 Organization 8

I Security analysis of smart cards in the C-DAX project **9**

2 Architecture **10**

2.1 C-DAX infrastructure 10

2.2 Client architecture 11

2.3 Security requirements 11

2.4 Security protocols 12

2.5 Smart card communication 17

3 Activation protocol **19**

3.1 Actors 19

3.2 Threads and risks 20

3.3 Protocol security requirements 22

3.4 Personalization and upgrade procedures 23

3.5 Protocol specification 23

3.5.1 Assumptions 23

3.5.2 Requirement fulfillment 24
3.5.3 Protocol definition 25
3.5.4 Alternative protocol 27
3.6 Protocol verification 29
3.7 Logging and monitoring 29

II Smart card performance analysis 30

4 Implementation 31
4.1 Smart card and terminal hardware 31
4.2 Smart card applet implementation 34
4.2.1 HMAC implementation 35
4.2.2 AES padding 36
4.2.3 High level functions 36
4.3 Terminal implementation 37
4.4 Design decisions 38
4.4.1 Key length 38
4.4.2 Exception handling 38
4.4.3 Encoding sequence 38
4.4.4 Smart card memory limitations 39

5 Performance analysis 40
5.1 Materials 40
5.2 Method 41
5.3 Throughput performance analysis 42
5.4 Cryptographic performance analysis 44
5.4.1 Smart card benchmarks 44
5.4.2 Computer benchmarks 47

III Results **49**

6 Discussion **50**

6.1 Security analysis 50

6.2 Performance analysis 50

7 Conclusion **53**

7.1 Research summary 53

7.2 Applicability of smart cards in the C-DAX project 54

7.3 Future work 54

A Activation protocol verification in ProVerif **62**

Chapter 1

Introduction

With the modernization of the electrical grid, the information infrastructure is becoming more and more complex. Cyber-security is a growing concern for the security of our electrical grid; in the form of criminal threats, which can have a financial impact, or even terrorist threats, which can have disastrous effects. A robust network security is therefore crucial. Modern cryptographic algorithms such as RSA form the building blocks for a secure infrastructure. It is a difficult problem to protect the valuable assets in this infrastructure: cryptographic keys. An industry-proven way of distributing and protecting these keys are smart cards. This research investigates the application of smart cards in the C-DAX project. In order to test the applicability of these smart cards in the C-DAX project we have to assess the use of smart cards with respect to the C-DAX requirements. The communication throughput and execution time of cryptographic algorithms on these cards play an important role.

This chapter is divided into two parts. In the first part we describe the C-DAX project and related work, the second part gives a background on smart cards.

1.1 The C-DAX project

The rapid changes in the electrical power industry demand a smarter information infrastructure. Traditionally, the power flow over the grid was simple and predictable. Large power plants produced energy which industry and households consumed. With the increase in distributed renewable energy resources, the flow has become less predictable. The market share of energy sources such as windmills and solar panels is rapidly growing, and this makes the task of monitoring and controlling the power grid more complex. At the same time, the introduction of more sensors and actuators, as well as smart meters [VBN97] and electric vehicles [KT05], produces more data and the need for 'smarter' control of the grid. With the addition of new control mechanisms to the grid, the cyber-security of the infrastructure is a growing concern [Khu+10].

The electrical grid is of vital importance to the functioning of our modern society and should be sufficiently protected against all possible threats. With the increase in sensors placed on different parts of the grid, and the growing amount of measurement data on the consumer side, privacy concerns may also be involved.

C-DAX (Cyber-secure DAta and Control Cloud) is a EU FP7 project[1] that aims to provide a cyber-secure distributed information infrastructure to the energy distribution networks[2]. The C-DAX project investigates, among other things, in what way the integration of renewable energy resources can be accommodated and provides a platform for smart-grid applications. However, this is only one of the possible uses of the C-DAX communication infrastructure. There are many other tasks that the C-DAX communication infrastructure can accommodate within a future-proof electrical grid.

C-DAX uses an Information Centric Network (ICN) architecture that operates on top of the IP protocol. An ICN is well-suited to the challenges of the electrical grid since it is a distributed approach that aims to be highly scalable and more resilient to disruptions and failures [Ahl+12]. The flavor of ICN used in C-DAX is the Publish-Subscribe Internet Routing Paradigm [Ain+09]. C-DAX clients host smart grid applications and can play multiple roles as subscriber or publisher. For instance, a sensor device might be a publisher of sensor data, while a smart meter can publish measurements and subscribe to a data feed of energy prices. C-DAX provides the middle-ware to these publisher and subscriber clients for the interaction with the C-DAX cloud. The C-DAX cloud is responsible for routing and delivering the messages from publishers to subscribers in a safe, reliable and scalable manner. The cloud consists of a network of so-called C-DAX nodes, hosts located, for example, at distribution substations. Examples of clients are (smart) utility meters, Phasor Measurement Units (PMUs) and Intelligent Electronic Devices (IEDs).

Information on the C-DAX cloud is organized into topics. Publishers produce information for one or more topics. Topic information is stored in the C-DAX cloud, and replicated on one or possibly more C-DAX nodes.

The C-DAX requirements in [CDAX21] define the general requirements of the C-DAX platform. Besides these general system requirements, [CDAX21] considers 3 use cases that introduce additional and more specific system requirements:

- The first use case considers the control of normal network operation and stability in the medium voltage network. This involves communication between devices in distribution substations (such as remote terminal units and intelligent electronic devices) and master control systems in an utility distribution control center.
- The second use case considers the monitoring and control of the distribution grid, specifically the communication between PMUs placed in the medium voltage grid and the Phasor Data Concentrators (PDC) and Supervisory Control And Data Acquisition (SCADA) system. The PMUs measurements are used via these systems for fault detection, voltage control and other distribution system functions.
- The third use case tackles the retail energy market and the communications taking place in the future energy market, such as intelligently matching demand with supply.

These use cases give an overview of the practical use of the C-DAX infrastructure. They are relevant to the present research because they introduce a set of additional system requirements, for example, with respect to system performance. However, more important to the present research are the specific cyber-security properties that need to be ensured by the C-DAX cloud. A high-level requirement of C-DAX is to ensure availability of the C-DAX cloud and end-to-end security between C-DAX clients. In order to

[1] See `http://cordis.europa.eu/fp7`
[2] See `http://cdax.eu`

implement the requirement of availability, the protocols used in C-DAX have to involve authentication of all parties. End- to-end security between clients means that end-to-end confidentiality, and the end-to-end integrity of messages have to be guaranteed. Confidentiality has to be ensured on a topic basis and can typically be implemented by encrypting messages. Integrity can be implemented by signing messages, or adding a Message Authentication Code (MAC) at the publisher's side.

To implement the required security functionalities of C-DAX a various set of techniques, protocols and algorithms can be used. The challenge is to find the best-suited ones with respect to the non-functional requirements, such as the performance and the level of security. Several protocols are proposed for the particular application of the smart-grid, such as:

- [KKT12] based on deriving symmetric encryption keys from a long-term key that is distributed on a node bases;
- [Fou+11] based on Diffie-Hellman key exchange and uses a hash-based authentication code technique;
- [VP13] an efficient scheme based on pairings specifically designed to suit the C-DAX project needs.

However, the final decision for a C-DAX security protocol will be based on well-established and proven secure techniques. Using protocols and algorithms such as the ones recommended by the National Institute of Standards and Technology (NIST) is a good practice, since they are widely used in practice and thoroughly tested. Experimental techniques, such as the ones based on pairings, might have an application in the more distant future versions of the C-DAX security protocols.

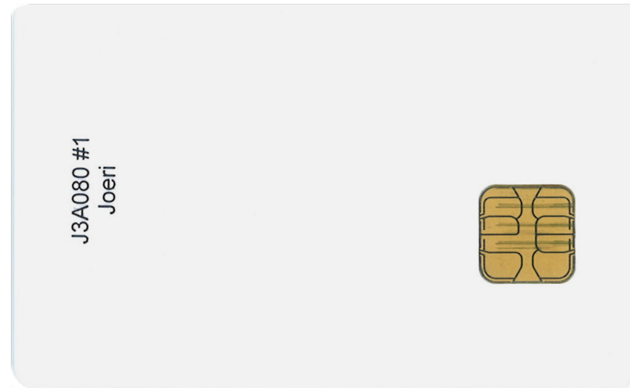
1.2 Smart cards

A smart card is a plastic card with an embedded chip. Smart cards are widely used for identification and authentication purposes, such as bank cards, or SIM cards in a mobile phones. The success of smart cards for these applications can be attributed to, inter alia, their excellent security properties.

The first serious use of smart cards was in 1984 by the French Telecom services, in the form of telephone cards [RE10]. Other payment cards, that used a machine readable magnetic strip, had been around for many years. This magnetic strip had a crucial weakness: an adversary with access to a card terminal device could read or write its contents. A smart card is much better secured against content manipulation. The first smart cards, such as the French Telecom card, were relatively simple memory cards with some added functionality to provide a digital wallet. The current microprocessor chips are more generic and a lot more complex. They offer modern cryptographic algorithms that make public key cryptography available on small chips. This generation of cards was first introduced in mobile telecommunication applications [RE10]. Today, cards with embedded microprocessor chips are the industry standard for inter alia banking, identification, and public transport. This last application in particular was made possible by the use of contactless smart cards, which do not need a physical contact between the smart card and a reader.

Smart cards are developed for security sensitive applications in, for instance, the financial sector, mobile phone industry, or even specifically for computer security applications. An important application for

Figure 1.1: J3A080 smart card



smarts cards is the use as a Secure Application Module (SAM) in payment terminals. Smart cards are used for this purpose because of the security measures taken on both the hardware and software side. Smart cards are engineered to be tamper resistant. It is really hard to extract information from the card, without permanently damaging it. See for instance [AK98; KK99] for various techniques and attacks on tamper resistance.

To support modern and strong security applications, such as authentication, modern smart cards are equipped with a cryptographic co-processor. This dedicated co-processor can efficiently execute cryptographic algorithms such as RSA, see [HP00]. A modern smart card typically provides several efficient implementations of both symmetrical and asymmetrical cryptographic algorithms, such as RSA, Hashing algorithms, and AES, see [BPR01].

In 1996, the smart card division of Schlumberger started the development of a new portable platform for smart cards[3]. They chose a subset of Java as the programming language for their new platform, which had to allow Java applications to run on a smart card. Java applications on a smart card are called applets. In that same year, Sun Microsystems completed the Java Card Platform specifications which was initiated by a group of companies, including Visa, Schlumberger, Gemplus, Philips and IBM. The Java Card subset has all language constructs that Java has, but many features such as dynamic class loading, two dimensional arrays, threading, cloning, etc. are left out. Particularly, the number of data types is reduced to fit the target system; the data types char, long, float, and double are not available. For a precise specification of the Java Card virtual machine, see [Mic06].

Java Card also adds specific security features, such as an *applet firewall* that isolates different applets running on the same card, or the support for atomic persistent updates and transactions. This last feature is crucial, since a smart card is externally powered and every operation could be interrupted at any moment.

At roughly the same time Java Card was introduced, another smart card operating system was introduced: MULTOS. MULTOS is an open standard, developed by a consortium of more than 20 companies, including MasterCard. In contrast with Java card, MULTOS applications are developed in the C language, and compiled to MULTOS byte code. MULTOS has several applications in for

[3]See `https://sites.google.com/site/personalhistoryofthejavacard/` for a personal history of the smart card by Bertrand du Castel

example banking and, more recently, in electronic passports[4].

In 1999 the GlobalPlatform association was founded. This association develops specifications to promote "secure and inter-operable deployment and management of multiple applications on secure chip technology"[5]. GlobalPlatform supports several developments related to smart cards, such as the development of tamper-resistant chips, but GlobalPlatform also manages responsibilities related to messaging. In summary, GlobalPlatform creates an international industry standard for building secure smart card applications.

A popular modern implementation of Java Card is the Java Card OpenPlatform (JCOP). The name originates from the Java Card specification and the former name of GlobalPlatform: OpenPlatform. JCOP was originally developed by IBM. In 2007 the support responsibilities have been passed on to NXP Semiconductors.

The findings in this research are based on modern variants of NXP JCOP smart cards, which provide both a contact and contactless interface. See Chapter 4 for a more precise specification of the cards used in this research.

1.3 Organization

This thesis is divided into two parts. The first part is concerned with the security analysis of smart cards within the C-DAX project. Chapter 2 gives an overview of the system architecture at several different levels. Chapter 3 specifies an activation protocol for smart cards within the C-DAX context. The second part of this thesis is concerned with the performance analysis of smart cards within various C-DAX security applications. Chapter 4 specifies the proof of concept implementation of some smart card security functionalities. Chapter 5 discusses the benchmark measurements of several C-DAX related algorithms performed on a smart card. The results, conclusion, and future work are discussed in chapters 6 and 7.

[4]See `http://www.multos.com/`
[5]See `http://globalplatform.org/aboutus.asp`

Part I

Security analysis of smart cards in the C-DAX project

Chapter 2

Architecture

This chapter discusses the architecture of the C-DAX communication infrastructure. In particular, the specified security requirements, protocols and cryptographic primitives are defined. Although there are several possibilities for cryptographic algorithms that can be used to secure the C-DAX information infrastructure, this research will concentrate on algorithms that are available on smart cards. Note that with the advancement of smart card technology, new techniques and algorithms will become available on smart cards. Therefore, this chapter does not, in any way, specify a *final* solution. See Section 7.3 for more details on possible future work.

2.1 C-DAX infrastructure

As already introduced in Section 1.1, the C-DAX communication model is based in the concept of Information Centric Networking (ICN). The idea of ICN is that the content plays the central role of the communication instead of the hosts. ICN relies on named-content as an alternative to having naming schemes for the source and destination hosts.

In C-DAX the content is organized into topics. Topics are a way of organizing different types of data for smart grid applications. This provides a lot of flexibility.

The C-DAX platform consists of two major components: the C-DAX middle-ware that provides publish-subscribe interfaces to C-DAX clients hosting smart grid applications, and the C-DAX *cloud* which consist of interconnected nodes responsible for the resolution and delivery of messages exchanged between publishers and subscribers.

C-DAX clients can play two different roles: the publisher role allows a client to publish information on a certain topic and the subscriber role allows the client to receive information on a certain topic.

The cloud consist of different types of nodes, which have different responsibilities within the network architecture. For the current research we do not consider a detailed distinction between types of nodes. In some of the examples presented in this thesis only a single node is considered to perform all cloud functionalities. This is just an abstraction for the sake of simplicity. We assume that all data is routed correctly from publishers to subscribers. A more precise specification of the C-DAX network

infrastructure can be found in [CDAX31].

Publishing and subscribing topic data is one of the two forms of communication considered in the C-DAX project. The other one is point-to-point communication between hosts. This type of communication is necessary for the management of keys, topics, nodes and clients in addition to monitoring. In [CDAX31] and [CDAX32] these two types of communication are called data-plane and control-plane communication.

2.2 Client architecture

A C-DAX client device can be any smart-grid device that runs the C-DAX client code and is connected to the C-DAX cloud. Any digital device can be a client and thereby play the role of publisher and/or subscriber on one of more topics. Access to the C-DAX infrastructure is provided through an API. On a client device, one or more smart grid applications can interact with this API to send or receive topic data or perform other C-DAX specific operations. We call these "client applications". The implementation of the C-DAX API is called the C-DAX Client. The C-DAX client directly communicates with the C-DAX cloud using the lower layer of the internet protocol suite such as TCP or UDP. When we refer to a "client", we also mean the part of the client device that is concerned with he C-DAX architecture, thus the code that implements the C-DAX API.

The C-DAX client can be split in two modules; the communication module and the security module. One of the responsibilities of the security module is to perform security functionalities such as authentication or encryption. When a smart card or other Secure Application Module (SAM) is part of the client device, the security module will handle all communication with this device. The communication module is responsible for all communication with the C-DAX cloud and handling interactions with client applications. For the client application the underlying C-DAX security implementation is abstracted through the API. This means the client applications never have to perform any action related to key management, encryption or authentication.

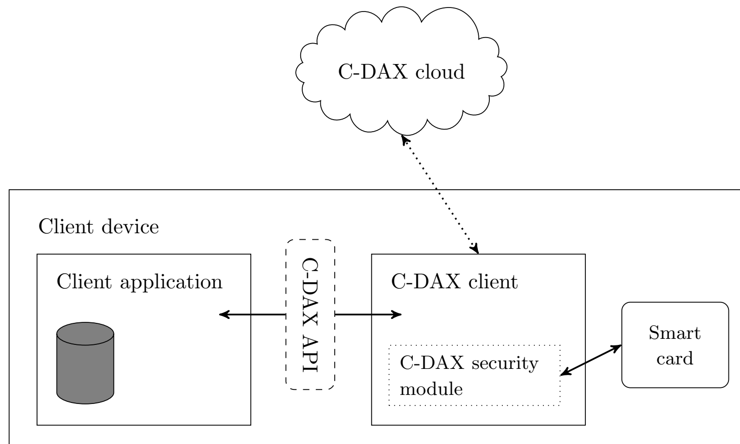
In this thesis we concentrate on the application of smart cards in a C-DAX client. In a practical use, these might not have the shape of, for instance, a banking card. A more practical shape for small devices is the form of a small SAM, such as a SIM card.

To communicate with an embedded SAM in a client device terminal hardware and drivers are needed. These operate between the C-DAX security module and the physical card. Figure 2.1 shows a schematic overview of a typical C-DAX client device and the software and hardware components relevant to this research.

2.3 Security requirements

The security requirements of the C-DAX project can be categorized in the three main security requirements: Confidentiality, Integrity, and Availability (CIA). Within these three, integrity and availability are most vital to the C-DAX project. It is important that the communication infrastructure is *available*, in order to avoid, for instance, power outages. *Integrity* is needed to protect equipment from incorrect measurements or other problems originating from modified, or fake data. If one can verify the integrity of a message and the source of that message, authentication is accomplished. Integrity

Figure 2.1: C-DAX client architecture



ensures the message is unchanged, while authenticity ensures the certainty of the message origin. If an authentication scheme ensures that a message originates from a certain publisher, a property called and non-repudiation is established; this means that a publisher cannot deny sending the message. *Confidentiality* means that the message contents remain confidential for all non-authorized entities. This is not a requirement in all use cases. PMU measurement data for instance, is not typically confidential, while individual meter readings of the consumer are privacy sensitive and thus confidential.

In order to meet these requirements, appropriate security mechanisms need to be in place. These mechanisms are required to be tailored to the application. In Section 2.2 we made a distinction between two types of communication: data-plane and control-plane. Given the architecture of the data- plane communication, which is data-centric and topic-based, the authentication mechanism should also be organized around data and topics.

2.4 Security protocols

The C-DAX project defines a set of elaborated networking and security protocols, see [CDAX31] and [CDAX32]. Most of the details described in these documents are not relevant to the current research, although a short background of these protocols helps to understand which security mechanism are relevant to the C-DAX client.

As stated before, C-DAX has two different communication mechanisms; data-plane and control-plane communication. Data-plane communication is topic-based and data-centric, while control-plane is based on point-to-point communication between hosts. For both of these types of communication, different security mechanisms need to be in place.

A solution to secure the control-plane communication, as defined in [CDAX31] and [CDAX32], is based on a Public Key Infrastructure (PKI). A PKI uses asymmetric cryptographic algorithms to encrypt, decrypt, sign, and verify data. The standard for these algorithms is RSA [RSA78]. Each host in a PKI has a pair of cryptographic keys, a public key and a private key. A public key is needed to encrypt a message or verify the signature of a message from a certain host. Two hosts that want to communicate

securely need each other's public key in advance, in order to be sure of each other's identity. Another method is storing only a public key of a trusted third party, called the Certification Authority (CA). The CA verifies, or makes sure in other ways, that a public key belongs to the identity of a certain host. Then the CA signs the public key of that host together with the identity, using the private key of the CA. This results in a certificate, which the host can subsequently be used to authenticate against other parties that trust the same CA.

In the C-DAX project, specifically the control-plane communication within the C-DAX architecture, both methods of authentication are considered. On the one hand, storing public keys in advance limits communication overhead, since no certificates have to be sent over for every communication. On the other hand, if there are a lot of hosts, it requires a lot of storage. To secure control- plane communication, every host, including clients, has a public/private key pair. These key pairs are distributed either in advance to all communicating parties, or are stored as certificates. Using these keys, hosts can set up a secure tunnel.

To secure data-plane communication another mechanism is used. Messages are authenticated, and checked for integrity based on a Message Authentication Code. These codes can only be generated or validated using a secret key that is shared between all parties that are allowed to access a certain topic. Confidentiality is obtained by encrypting messages with a topic-based secret key. The secret key for a certain topic, is only distributed to clients and nodes that are allowed to publish, subscribe or route messages belonging to that topic.

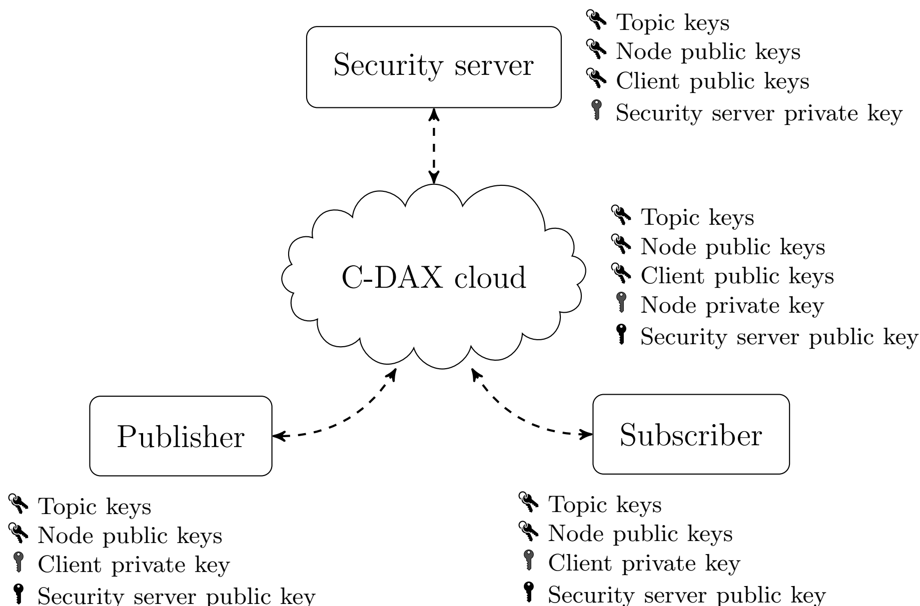
Note that the use of symmetric cryptography is not mandatory for topic communication, although it might be the most suitable choice for high bandwidth topic communication. An alternative is the use of the in-place public key infrastructure.

Thus data-plane communication is secured using symmetric cryptography. The standard for symmetric encryption is AES [DR02]. A standard for symmetric authentication is the use of Hash Message Authentication codes (HMAC) [KCB97]. HMAC uses a hash function and a secret key to generate an unique authentication code, based on the message contents. Any hash function can be used in HMAC, although the current industry standard is to use one of the SHA-2 or SHA-3 variants [FIP01].

One symmetric key per topic is enough to ensure confidentiality, authentication and integrity, by encrypting messages and adding an HMAC. To ensure end-to-end security between publishers and subscribers, independent from the intermediate nodes, a second key is needed. One key is used to perform topic-based end-to-end encryption and authentication and one key to allow intermediate nodes to verify the authenticity. We call the key that is used for end-to-end encryption and authentication $K_{\texttt{enc}}$. This key is distributed between all publisher and subscriber clients that are allowed to access the topic this key belongs to. Note that the encryption step is not required in all use-cases, since confidentiality is not always a requirement. We call the second key that is used for intermediate authentication $K_{\texttt{auth}}$. $K_{\texttt{auth}}$ allows the C-DAX cloud to provide authentication and strengthen availability, since it provides a method to filter all unauthorized data.

Figure 2.2 shows an abstract presentation of the C-DAX infrastructure. The dashed lines indicate control-plane communication, the solid lines indicate data-plane communication. The most important actors are the Security Server, the C-DAX cloud and publisher and subscriber clients. As discussed earlier, there are two ways to distribute public keys: by using certificates or by storing all public keys. This figure shows the second option. All parties store the security server public key. The security server stores the public keys of all nodes and clients and the topic keys of all topics. Nodes in the C-DAX cloud store the $K_{\texttt{auth}}$ of all topics they are allowed to handle, and the public keys of all clients

Figure 2.2: C-DAX key distribution



they communicate with. Clients store $K_{\texttt{enc}}$ and $K_{\texttt{auth}}$ of all topics they are allowed to subscribe to or publish on.

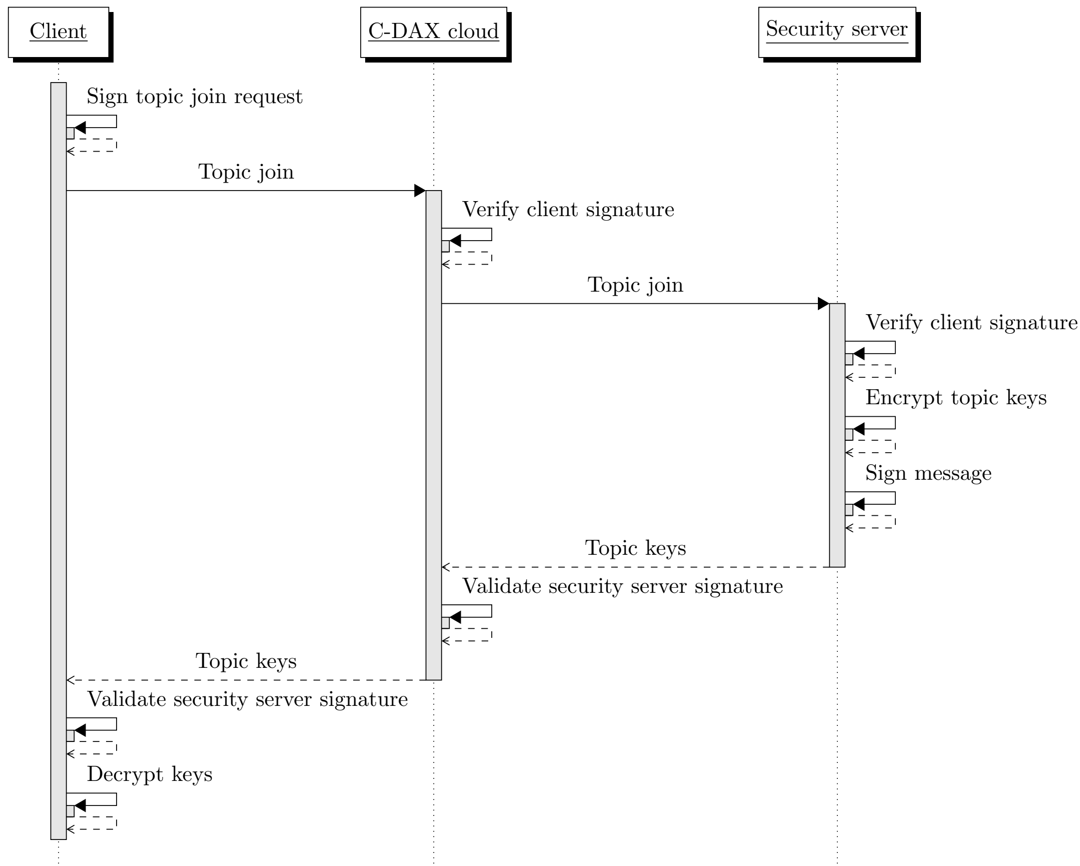
Note that Figure 2.2 only considers the keys needed for normal topic communication. To allow, inter alia, node-to-node communication, node public keys need to be stored as well.

There are several complex protocols described in [CDAX31] and [CDAX32], for instance, for key distribution, key revocation, topic communication, etcetera. Although these protocols are important for the C-DAX infrastructure, we will only discuss protocols that are relevant to the current research. To create a better understanding of the tasks a smart card can perform in these protocols we will highlight two important protocols.

Figure 2.3 shows a sequence diagram for a client topic join. This is the mechanism used by publishers and subscribers to obtain a certain topic key. The security server decides whether the client requesting the topic key is allowed to do so. First the client creates a 'topic join request' stating, inter alia, the identity of the client and the topic. The complete message is signed with the client private key. The message is sent to the security server via the C-DAX cloud. Each intermediate hub checks the authenticity of the message by verifying its signature. This check is also performed by the security server. The security server decides whether the client is allowed to access the specified topic. When the client is allowed access to the topic, the topic keys $K_{\texttt{enc}}$ and $K_{\texttt{auth}}$ are sent back to the client, encrypted with the client public key and signed with the security server private key. The nodes in the C-DAX cloud verify the message validity using the security server public key. Finally, the client verifies, decrypts and stores the topic keys. A node in the C-DAX cloud can perform a similar request to obtain $K_{\texttt{auth}}$ for a certain topic.

In Section 4 we only consider $K_{\texttt{enc}}$ for simplicity. Adding $K_{\texttt{auth}}$ to the smart card implementation is straight forward, but involves more processing on the smart card. A better solution is to leave $K_{\texttt{auth}}$

Figure 2.3: Topic join sequence diagram



on the C-DAX client device and adding an additional HMAC to the topic data on the device.

Figure 2.4 shows the content of a topic join request and a topic join response. In the request, the topic name, source identity and a timestamp are signed using RSA and the signature is appended to the message. The timestamp is needed to guarantee the freshness of the request. In the topic join response only the topic keys are RSA encrypted. The topic name, source identity and a timestamp together with the encrypted topic keys are signed using RSA and the signature is appended to the message.

After obtaining the topic keys, a client can encode and decode topic data. Encoding is performed by adding the HMACs to the message and optionally encrypting its data payload. Decoding is performed by the inverse procedure, verifying the HMACs and decrypting the payload content. Figure 2.5 shows the sequence diagram of publishing topic data. For the sake of simplicity the C-DAX cloud is represented as one node. There are two subscribers present in the diagram. The C-DAX cloud forwards the newly published topic data to all topic subscribers.

Figure 2.6 shows the content of a topic data message. It contains the topic name, source identity and a timestamp in plaintext. The message payload, which is the topic data AES encrypted with $K_{\texttt{enc}}$. Afterwards an HMAC over the complete message is appended, once using $K_{\texttt{enc}}$ and once using $K_{\texttt{auth}}$.

Figure 2.4: Topic join request and response

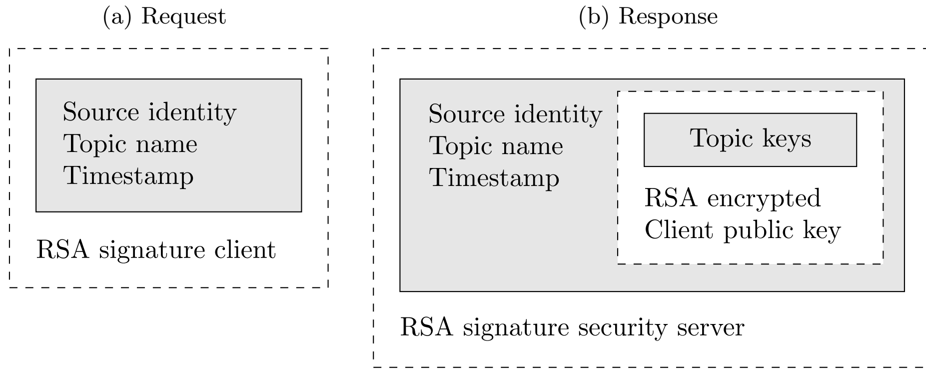


Figure 2.5: Topic communication sequence diagram

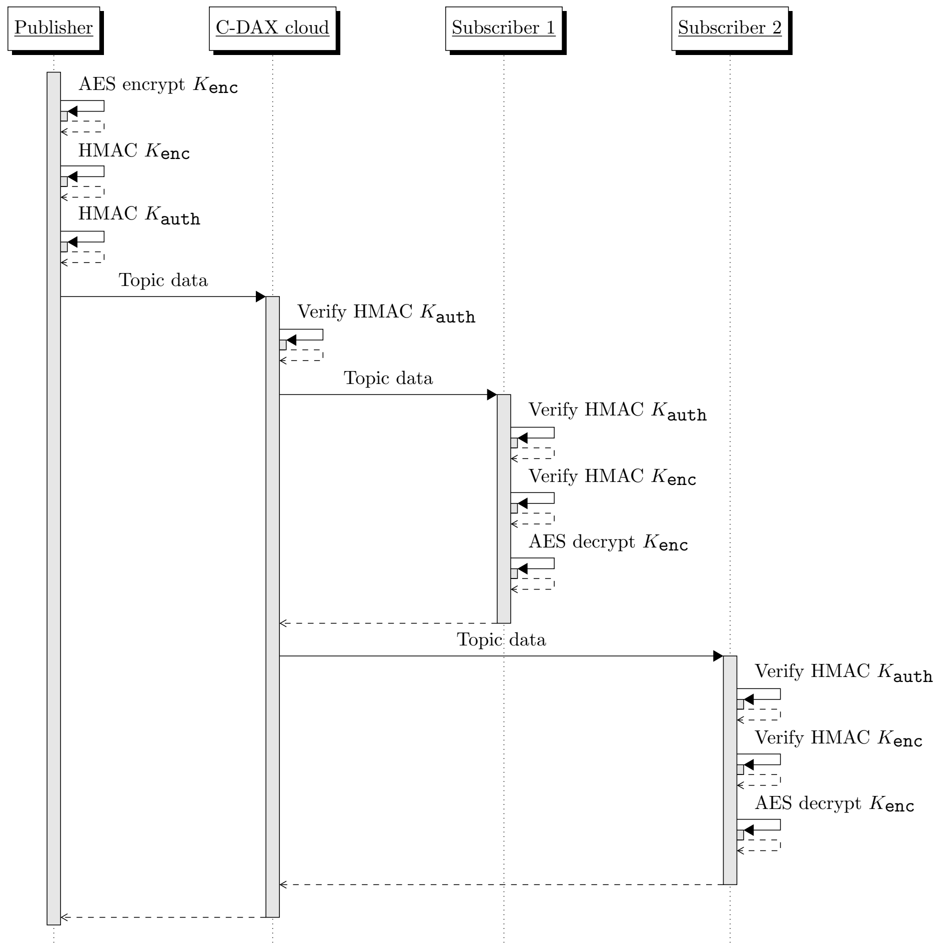
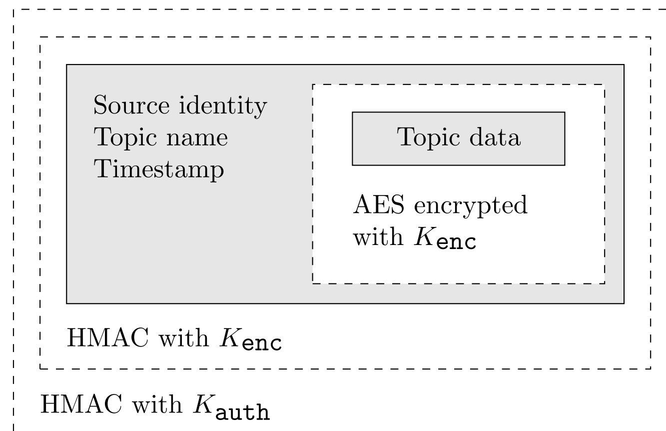


Figure 2.6: Topic data message



2.5 Smart card communication

The previous section describes some of the basic security functionalities and protocols. The control-plane communication is secured using asymmetric cryptographic primitives, while the data-plane communication is secured using symmetric primitives. The aim of this research is to push some of the functionalities to a smart card. Since there are multiple protocols and primitives, different divisions can be made between performing cryptographic algorithms on the smart card or on the C-DAX client.

The obvious role of a smart card is the distribution and storage of long term, asymmetric keys, which are the RSA key pairs. They are used in, inter alia, control-plane communication. Distributing and securing cryptographic keys in the field is a cumbersome task. Isolating the storage of the key to a SAM yields better security guarantees. Additionally, the practical distribution is easier. Carrying a SAM to a device in the field, such as a smart meter, is more practical than connecting to a client device and storing the key.

The obvious functionalities that can be performed on a smart card in a C-DAX client device are therefore as follows:

- generating and storing an RSA key pair;
- RSA signature generation and signature verification;
- RSA message encryption and decryption.

By performing these functionalities on the card, the private key never has to leave the secure environment of the smart card chip. It is generated on the card during a 'personalization phase'. The public key is then extracted and stored on the security server, the security server key is stored on the smart card. When the smart card is inserted in the client device, it has to be activated, see Section 3.5 of Chapter 3 for a detailed description of this protocol.

After activation the client can initiate a topic join. The join request is generated in the C-DAX client and then transmitted onto the smart card. The smart card then signs the request using its RSA

private key and transfers it back to the C-DAX client. When the C-DAX client receives the topic join response, the message is transferred to the smart card. The smart card verifies the signature using the security server public key. Subsequently, the message content is decrypted with the private key. The resulting plaintext, in this case the topic keys, is transferred back to the C-DAX client. Consequently, the C-DAX client can encode or decode topic messages using these topic keys.

Performing these functionalities on a smart card, can significantly slow down the system. However, a topic join is not a frequently performed functionality, since it is only performed once per topic.

In order to execute these functionalities on the smart card, the applicable cryptographic algorithms have to be available on the smart card platform. In this case RSA primitives need to be implemented. Section 4 provides the details of these cryptographic primitives and notes possible alternatives to classic RSA, such as Elliptic curve Cryptography (ECC).

Although the above mentioned partition of security functionalities properly secures the client key pair, the topic keys are not securely stored on the smart card. This problem yields the second partition option; storing topic keys on the smart card end performing topic data encoding and decoding on the smart card. This option results in the following additional functionalities:

- storing topic keys;
- AES encryption of topic data and generating an HMAC over topic data and message headers;
- AES decrypt and verify an HMAC over topic data and message headers.

In order to perform the above functionalities, the smart card has to be able to perform AES encryption and decryption and HMAC generation and verification. Section 4 provides the details of these algorithms on the smart card.

Chapter 3

Activation protocol

This chapter describes several measures to protect a smart card from unauthorized use within the context of the C-DAX project. These measures strongly depend on the capabilities of the device the smart card is installed into. Preventing unauthorized access implies the need for an authorization mechanism. This mechanism is implemented in the form of an activation protocol. This protocol involves communication between the device, the smart card, the security server and the installer in the field.

In this chapter the 'device' can be any C-DAX node or client that uses a smart card as part of the C-DAX security functionalities. We make a distinction between three types of devices for the activation protocol:

- A device with a data connection, such as a USB or Ethernet socket.
- A device with a display, such as a segment display or small video display.
- A device with an input device, such as a numeric keypad or small keyboard.

In Section 3.5 we will make a clear distinction between these different interfaces.

3.1 Actors

There are several actors involved during the installation of a smart card. The installation procedure will at least consist of the physical action of inserting a smart card in the C-DAX device, such as a smart meter. This could happen when the smart meter is first installed in a home or, for instance, when an old smart card is replaced with a newer one.

To prevent the unauthorized use of smart cards and other threats as described in Section 3.2, a short activation protocol needs to be executed after the insertion of the smart card. A protocol that tries to prevent possible attacks is described in Section 3.5.

The following actors play a role in this protocol:

1. **The security server**, a central server that performs security tasks within the C-DAX network. This server could, for instance, store public keys of all devices. This server should be reachable over a dedicated communication line by authorized devices or personnel. The security server also manages access control, such as the authorization of a certain smart card used in a specific device.
2. **Installer**, the person that inserts the smart card into the device. The installer could, for instance, be a skilled technician from the electrical company. The installer needs some kind of device to communicate with the security server.
3. **Installer Device**, the device used by the installer to communicate with the security server. The device should be certified, to ensure there are no security weaknesses or backdoors present. Optionally, it should be possible to set up a data connection from this device to the device the smart card is inserted into. Examples of installer devices are a PDA, smart phone or laptop with an internet connection.
4. **C-DAX device**, some device that needs to communicate with the C-DAX cloud. The device should contain a smart card terminal and either a display, some input method or a data connection interface, as listed above. An example of such a device is a C-DAX enabled smart meter.
5. **Smart Card**, the smart card that is installed into the device. The smart card should be personalized with a key pair and an identity number or string.
6. **C-DAX Cloud**, the intermediate communication layer, allowing the device to communicate with the security server.

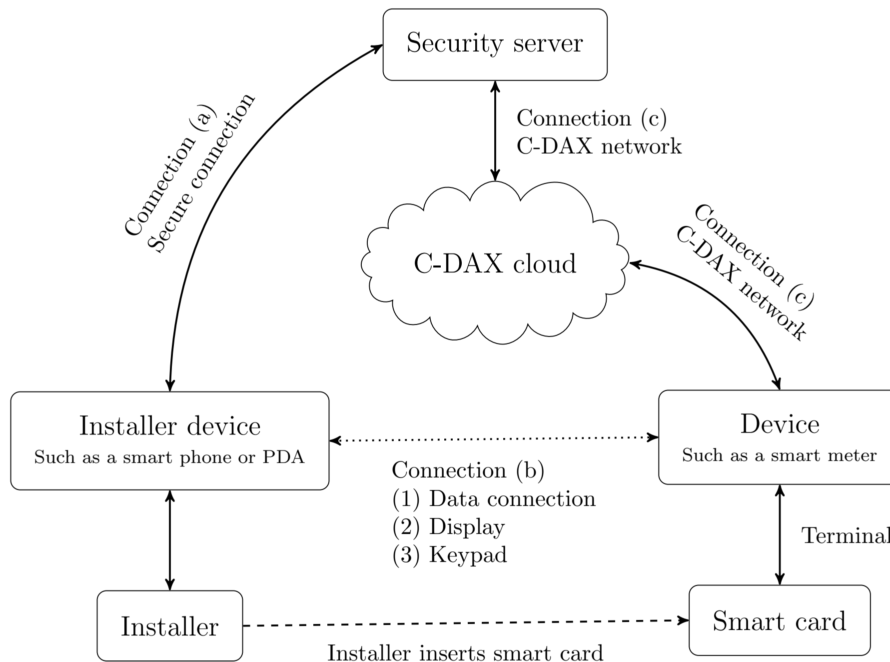
Figure 3.1 shows a brief overview of the parties involved in the activation protocol and their connections. The smart card is inserted into the device, the device can communicate with the security server via the C-DAX cloud (c). The Installer device can communicate with the security server (a) and there is a means of communication between the installer and the device (b).

3.2 Threads and risks

Before we can evaluate the security of an activation protocol, we first have to investigate the possible threats and their resulting risks, which need to be mitigated. A risk to the system under investigation exists when:

- there is a third party actor, such as a person or group, which has unauthorized access to the system in some way;
- this actor has a motivation to attack the system, such as financial gain or publicity;
- and there is some kind of vulnerability in the system, such as a flaw or weakness that the actor can exploit.

Figure 3.1: Activation protocol overview



According to [ISO13335] a risk is a combination of a threat, the probability the threat will exploit vulnerabilities, and the business impact this event has. In [ISO27002] a similar definition is given: an information security risk is the “potential that a given threat will exploit vulnerabilities of an asset or group of assets and thereby cause harm to the organization”.

In order to design a sound activation protocol, first possible vulnerabilities have to be identified. These vulnerabilities can be identified by listing several imaginable abuse cases of the system under investigation. An abuse case specifies a certain threat. We also consider abuse cases resulting from accidental action, for instance, a mistake by the installer.

Below, the abuse cases for a smart card within the C-DAX system are listed. Note that the following abuse cases are a carefully chosen sample out of all possible use cases. They all are a combination of an actor, motivation, vulnerability and a possible C-DAX smart grid component. Although the abuse cases are a subset of all possible abuse cases, they result in a complete set of security requirements. This is ensured by choosing the abuse cases in a way that all vulnerabilities are covered, as an alternative to all combinations of actors, motivation, vulnerabilities and systems. Most abuse cases focus on a smart meter, but other C-DAX enabled devices are considered as well to provide a varied subset of all possible abuse cases.

1. Incorrect billing:
   (a) by inserting the incorrect smart card in a device. If a smart card is coupled to a particular address, and accidentally the incorrect smart card is inserted in a smart meter by the installer, the billing of a particular customer could be incorrect.
2. Electricity theft:

(a) by stealing a blank smart card. If, for instance, a smart card could be used in an unauthorized and compromised smart meter, one could steal electricity by falsifying measurements.

(b) by stealing a smart card from another device. The same kind of electricity theft could occur when a smart card is pulled out of an authorized device and inserted and used by an unauthorized one.

(c) by tampering with an authorized device after the smart card is inserted.

3. Power cuts:

   (a) by sending incorrect measurement data using a stolen blank smart card. The correct operation of smart grid measurement units such as a PMU is vital to the availability of the grid. If measurement could be falsified using a stolen smart card this could result in power outages.

   (b) by sending incorrect measurement data using a stolen smart card from another device.

The actor in abuse case 1 is the installer. The actors in the other abuse cases could be any malicious individual, group or organization.

There is no actual motivation in abuse case 1, this threat is the result of inattention. The motivation in abuse case 2 could be financial gain buy electricity theft, or covering up some illegal drugs plantation. Abuse case 3 could be motivated by several aims such as targeting one or more individuals or companies or generating publicity.

A stolen smart card, such as in abuse cases 2a and 2b, could also be used to send incorrect data about the amount of energy being generated, for instance, with a private windmill or solar panels. Since these abuse cases are closely related to abuse cases 2a and 2b, we do not consider them.

3.3 Protocol security requirements

From the abuse cases above, we can deduce security requirements. We will number each requirement in order to identify them later.

1. **It should only be possible to use a card in the device for which it is configured.**
   This requirement is imposed by abuse case 1a and only applies when smart cards are actually coupled to a certain address/customer before it is inserted. The coupling could be performed during the smart cards personalizations phase, in which the device identity is configured in the card. From that point on, the smart card is only usable in a specific device.

2. **Blank smart cards can only be used by an authorized device.**
   Abuse cases 2a and 3a impose this requirement.

3. **Once a smart card is inserted and used by a certain device, one should not be able to use it in another device.**
   Abuse cases 2b and 3b impose this requirement.

4. **It should not be possible to tamper with an authorized device.**
   Abuse case 3c imposes this requirement.

Requirement 4 results in an assumption. We assume that an authorized device has not been tampered with. We can make this assumption because devices, such as smart meters, are designed to be more or less tamper resistant. Seals and other physical security measures could protect a device against tampering. A qualified installer should be able to detect possible tampering.

3.4 Personalization and upgrade procedures

Before a card can be activated and inserted into a C-DAX client device, the card needs to be personalized. This means that the software, or applet, is installed on the blank card and the initial credentials are created. When the applet is written to the card, the card can be locked. After locking the card, no applets can be installed, deleted or modified.

The most secure way of storing the initial card credentials is by letting the card generate them. When generating a (RSA) key pair on the card, the private key never has to leave the protected storage of the smart card. After generating the key pair, the public key has to be stored on the security server in order to authenticate the card later on and encrypt messages for the card. The personalization phase thus includes key pair generation and the transfer of the public key to the security server.

One of the benefits of using a smart card to execute security algorithms, is that they can easily be upgraded. Upgrading the signature algorithm, encryption algorithms and key lengths only involves replacing the smart card, instead of updating or even replacing the whole client device.

When an update of the smart card is executed the old card needs to be destroyed or securely recycled. Before the new card can be used, the activation protocol has to be executed. This allows the security server to register the new card and configure all future communications with the client device using the updated security algorithms.

3.5 Protocol specification

In order to implement the security requirements listed in Section 3.3, this section defines a security protocol performed during the activation of a smart card in a C-DAX device. The proposed protocol aims at implementing requirements 1, 2a, 3a and 3b.

3.5.1 Assumptions

The following assumptions are made about the participants of the activation protocol and the channels over which they communicate:

1. The participants in the protocol are genuine. This means, for instance, that it is assumed that the security server is not compromised, the installer is trustworthy, the smart card is unmodified since its personalization phase and the device is not compromised. Note that the security server and the smart card are able to validated each other's authenticity, since they have each other's public key beforehand.

2. There is a secure connection between the device and the smart card. This means that no malicious third party can access or change data that is communicated between the smart card and the device. We can assume this because the smart card is inserted in the device. If the previous assumption about the device holds, namely that it is not compromised, this assumption should also hold.

3. There is a secure channel between the installer and the security server. The installer is the person that installs the smart card into the device, for instance, when the device is initially configured. We assume that the installer can set up a secure connection with the C-DAX security server, for instance using a Secure Shell (SSH) or Transport Layer Security (TLS).

4. There is some trusted way of exchanging information between the installer and the device. For instance, the installer can input information using a keyboard or buttons or the device contains a small screen or display.

3.5.2 Requirement fulfillment

The protocol guarantees requirement 1 by involving an installer during the installation of a blank smart card. In other words, the installer has to prove to the security server that he or she is present during installation and can identify the device. Since the installer is a required party in this protocol a stolen blank smart card can not be used in any device.

Requirement 1 could be mitigated by checking the device identity after insertion and yielding an error when the device card combination does not match. This can also be left out of the activation protocol, since it is an optional and separate check, performed by the card.

Requirement 2 is guaranteed by letting the smart card bind to a specific device during installation. This binding is done by storing the identity string of the device on the smart card. This identity string is sent to the card once it is inserted into the device. We can assume the identity string is legitimate since we assume the device is authorized and untampered with. Before the smart card executes any security functionality, the device identity is compared to the stored identity string in the smart card. After every power outage of the card this check has to be performed again. The card can detect power outages by checking the contents of its volatile memory before any functionality is executed. The security mechanism of remembering the first device the card is inserted into is called the "Resurrecting duckling security policy", see [And08, Section 12.3.3.3]. The identity string of the device can be any number or string that is either:

- a fixed identifier in the device hardware or software such as a MAC address or serial number;
- a fixed identifier hard-coded in the C-DAX software layer;
- a unique random nonce generated by the smart card and stored on the device;
- a shared secret key agreed upon by the smart card and the device on first contact.

Note that if one could eavesdrop on the communication between the smart card and the device, the device identity could be recorded. If another device is modified to contain the same identity, the smart card could be interchanged. This is a serious weakness of the system. To overcome this problem one could consider a secure key exchange between the device and the smart card when they first

communicate. For instance, the *Diffie–Hellman key exchange* [DH76] could be used to established a shared key and subsequently set up a secure connection using a symmetric encryption scheme. Every time the device sends its identity to the smart card, it should be encrypted using the established key. This mechanism prevents the harmful consequences of eavesdropping on the connection between the device and the smart card. Using an initial key exchange honors the "Resurrecting duckling security policy", after the smart card is unlocked, no new key exchange has to be performed. While setting up a secure connection between smart card and device is a crucial point in the security of the system it is not included in the following definition of the activation protocol, for the sake of simplicity.

To perform the protocol there needs to be some communication between the installer and the device. The device needs either some kind of display, some kind of input source or a data connection with the installer to complete a successful activation. Since there are many devices in the field, we did not make any assumptions on the interface of the device. Instead there are two slightly different activation protocols to cope with these differences.

Section 3.5.3 defines the activation Protocol A, for devices with either:

- a data connection, such as a USB or Ethernet socket.
- a display, such as a segment display or small video display.

Section 3.5.4 defines the activation Protocol B, for devices with:

- an input device, such as a numeric keypad or small keyboard.

3.5.3 Protocol definition

The proposed protocol is stated below. The actors and message contents are defined as follows:

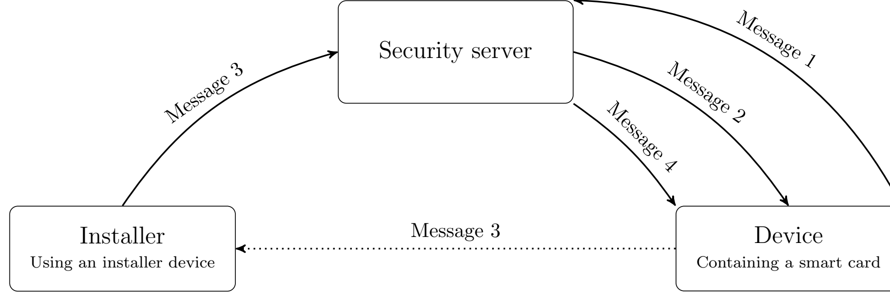
$Card$	Smart card
Dev	C-DAX Device, such as a smart meter
$Installer$	Person installing the smart card
$Server$	The C-DAX security server
ID_{Card}	Identity of the smart card
ID_{Dev}	Identity of the C-DAX device
N_{Card}	Fresh nonce generated by the smart card
N_{Server}	Fresh nonce generated by the security server
UNLOCK	The activation command for the smart card

With the notation $A \rightarrow B$ we mean "sends message to". The notation $A \rightarrow B \rightarrow C$ means that B forwards the message from A to C. The connection over which this message is sent can be either connection (a), (b) or (c) from Figure 3.1, depending on the parties involved. The type of connection (b) is dependent on the possibilities of the device, as stated previously.

When the device does not have a serial number or other identifiable number in the software, a random nonce is sent to the device in order to identify it. This nonce is sent over a secure channel.

$Card \rightarrow Dev : ID_{Dev}$

Figure 3.2: Activation protocol message diagram



The protocol is executed as follows: the identity of the device and the smart card are sent to the security server via the device. The security server sends back a fresh nonce, which is decrypted by the card and shown on the device. The installer sends the nonce and the device identity to the security server. Subsequently the security server completes the activation by sending an unlock certificate to the smart card, via the device. An unlock certificate is a special instruction, signed by the security server, which instructs the smart card to activate its security functionalities. Figure 3.2 provides a short overview of the actors involved and the different messages that are sent. A formal definition of the protocol is found below:

Protocol A

$Dev \rightarrow Card : ID_{Dev}$
The device initiates the activation protocol by sending its identity string to the smart card.

Message 1:
$Card \rightarrow Dev \rightarrow Server : (\{N_{Card}, ID_{Dev}, ID_{Card}\}_{PK_{Server}}, Sign_{Card})$
The smart card encrypts and signs a message for the security server containing the identity string of the smart card and the device and a fresh nonce. After transmitting the message from the smart card to the device, it is forwarded to the security server.

Message 2:
$Server \rightarrow Dev \rightarrow Card : (\{N_{Server}, N_{Card}, ID_{Dev}, ID_{Card}\}_{PK_{Card}}, Sign_{Server})$
The security server sends back the message with an added fresh nonce, this time encrypted and signed for the smart card. The device receives the message and transmits it to the smart card.

Message 3:
$Card \rightarrow Dev \rightarrow Installer \rightarrow Server : (\{N_{Server}, N_{Card}, ID_{Card}, ID_{Dev}\}_{PK_{Server}}, Sign_{Card})$
After verification and decryption, the smart card again encrypts and signs the same message content for the security server. This time the device transfers the message to the installer, using the display or a data connection. The installer sends the message to the security server, using a secure connection.

Message 4:
$Server \rightarrow Dev \rightarrow Card : (\{\text{UNLOCK}, N_{Card}, ID_{Dev}\}_{PK_{Card}}, Sign_{Server})$
After the security server receives the message with the two nonces and the identities of both smart card and device, it is proven the installer was present during the smart card installation. The security

server sends an unlock certificate back to the smart card. After that, the smart card is bound to one specific device identity, which is checked after every power loss. Consequently, unauthorized use in other devices is prevented, given that another device does not have the same identity.

After activation, the identity of the device is stored in the smart card. When the smart card is inserted into another device the identity will not match and the smart card is unusable. The security server has also stored the device identity together with the smart card identity. This can be used to manage future activations and for administrative purposes.

Figure 3.3 provides a schematic overview of the activation protocol A in a message sequence diagram. Implicit checks and other steps are modeled as function calls to the current thread. For instance, once the smart card receives the device identity, the identity is stored in order to use it later in the protocol. The security server checks access control policies to verify whether the smart card and device combination is authorized. Afterwards, the unlock command is signed and sent back to the smart card. For Protocol A Message 3 is displayed or sent over a data connection to the installer.

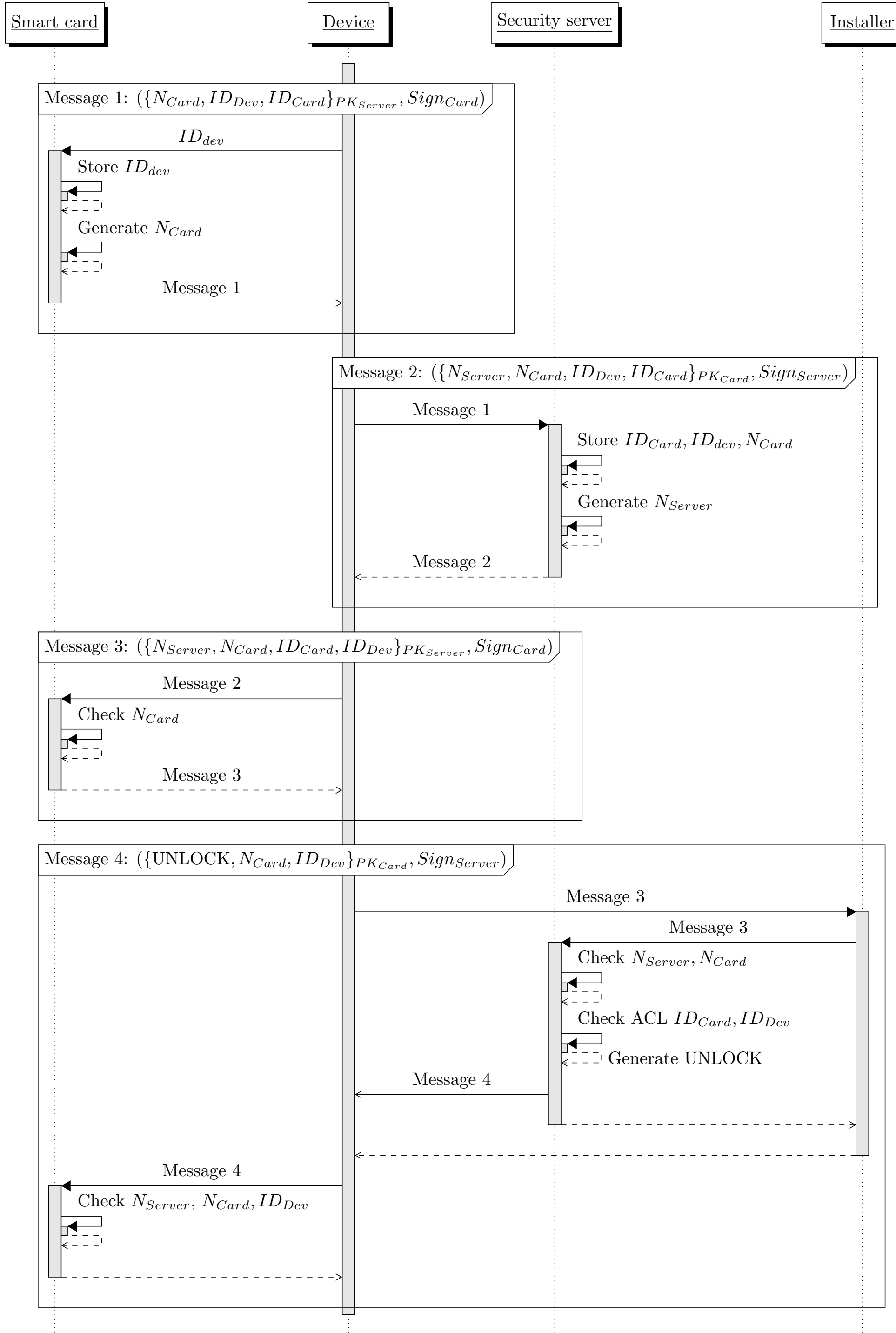
In case the message size is too large for the installer to practically read from a display, Message 3 can also be the output of a cryptographic hash function, with the input $N_{Server}, N_{Card}, ID_{Card}, ID_{Dev}$. The hash output is not encrypted or signed, but the security server can reproduce these values and thereby the output of the hash function. Depending on the size of the hash, this could lead to possible brute force attacks. If one could brute force the resulting hash value or guess the input value, this could result in a vulnerability in the protocol. The possibility of this attack depends on several factors:

- The size of the random nonces. When the nonce size is large enough, it becomes infeasible to brute force the message contents.
- The hash function used. When a strong hash function with a large output range is used, a brute force attack based on the hash output becomes infeasible. However, the size of the output range hash a direct influence on the effort needed by the installer. A four digit hash output is easy to retype by the installer, but also easy to brute force. A 40 digit number is hard to brute force, but also impractical to retype.
- The security mechanisms of the security server. When a solid anti brute force mechanism is in place on the security server, even small hash outputs become hard to brute force. For example, blocking each party that sends an incorrect hash value, would mitigate brute force attacks. However, these mechanism could also influence the availability of the system, when a small mistake results in an authentic party being blocked.

3.5.4 Alternative protocol

When the device contains an input source but no display or data connection on Connection (b) of figure 3.1, the protocol is slightly modified. The security server sends the message with the nonces and identities directly to the installer. After the installer inserts the message into the device, and the message is signed by the smart card, it is proven to the security server that the installer was present during the installation.

Figure 3.3: Activation protocol sequence diagram



Protocol B

$Dev \rightarrow Card : ID_{Dev}$
$Card \rightarrow Dev \rightarrow Server : (\{N_{Card}, ID_{Dev}, ID_{Card}\}_{PK_{Server}}, Sign_{Card})$
$Server \rightarrow Installer \rightarrow Dev \rightarrow Card : (\{N_{Server}, N_{Card}, ID_{Dev}, ID_{Card}\}_{PK_{Card}}, Sign_{Server})$
$Card \rightarrow Dev \rightarrow Server : (\{N_{Server}, N_{Card}, ID_{Dev}, ID_{Card}\}_{PK_{Server}}, Sign_{Card})$
$Server \rightarrow Dev \rightarrow Card : (\{\text{UNLOCK}, N_{Card}, ID_{Dev}\}_{PK_{Card}}, Sign_{Server})$

3.6 Protocol verification

In Appendix A a formal model of the activation protocol is provided in `ProVerif`. Using the `ProVerif` tool, two properties of the protocol related to authorization by the server, and the involvement of an installer during activation are proven. See Appendix A for a more elaborate description of the model and the protocol verification.

3.7 Logging and monitoring

After the activation protocol is executed, secure communication between the client device and the C-DAX cloud is possible. However, detection of errors remains an important task to improve the availability of the client device. When a smart card is unreachable by its host device or there is a defect in the smart card, this should be notified to the applicable authorities.

One way of monitoring the card is by the built-in C-DAX monitoring functionality, see [CDAX31]. Any errors on the smart card are detected in the terminal module and passed to the C-DAX client software. The C-DAX client software can log these errors and send them over to a central monitoring node.

Another way of evaluating the health of the smart card is by pinging. One could implement a ping protocol in which a C-DAX node sends over a random nonce to the C-DAX client device on a fixed time interval. The client device needs to answer with a signature over the nonce, which can only be generated by the smart card. This way a defect can actively be spotted by the C-DAX system.

Any defects or possible security incidents related to the use of smart cards should be thoroughly investigated to prevent future repeats. A common way of making these investigations possible is logging events. When each event is logged on the smart card, defects or even attempts of unauthorized use are traceable later on. Modern smart cards have a relatively large volatile memory. On a smart card with 80 KB EEPROM a rotating log of 1000 lines of 80 characters can be stored.

Part II

Smart card performance analysis

Chapter 4

Implementation

This thesis investigates the usability of a smart card for various security functionalities within the C-DAX project. Chapter 2 gives a explanation of the architecture and several of the protocols involved in the relevant part of the C-DAX project. In this chapter we take a closer look at the proof of concept implementation. We outline the possibilities of modern smart cards and how they can be applied to perform certain tasks within a larger system. During the implementation of the proof of concept, certain choices were made. These choices include assumptions about message size, number of topic keys, etcetera. These choices will be identified and explained.

This chapter will first detail the hardware used in the proof of concept. Subsequently, the terminal implementation is described, followed by the smart card applet. Finally we address a list of design choices.

4.1 Smart card and terminal hardware

The smart cards used in this proof of concept are the NXP J3A080 and J3A081 running JCOP v2.4.1 [NXP12]. JCOP is the operating system of the smart cards, and thereby implements the Java Card API. The choice for these two cards was based on two factors: the capabilities and the availability. With capabilities we mean the supported algorithms on the card, like the set of supported cryptographic block ciphers. With the availability we mean whether a certain type of card was obtainable by the author. Note that several other cards were tested, but found less applicable based on their supported cryptographic algorithms, supported RSA key size and general speed of operation. It is generally difficult to obtain unlocked and blank smart cards. Most smart cards, like the ones in your wallet, are locked – preventing the installation of new applets. Due to the extensive research with smart cards performed at the Radboud University Nijmegen, various Java Cards were available to the author. The two most modern ones, the J3A080 and the J3A081 were picked for this research, based on the speed of the chip, the available memory and the available cryptographic algorithm implementations. Table 4.1 depicts the technical specification of the cards, which can be found in [NXP12].

Table 4.1: Memory size and internal clock speed of the J3A080 and the J3A081 smart cards

Card type	J3A080	J3A081
EEPROM	80 KB	80 KB
ROM	200 KB	264 KB
RAM	6 KB	7.5 KB
Internal CPU clock speed	30 MHz	62 MHz

Table 4.1 shows that the J3A081 card has slightly more memory than the J3A080 card. The internal clock speed of the J3A081 is well over twice as fast as the J3A080. Note that the final clock speed also depends on the card reader and driver, since these smart cards have an external clock.

The card terminals used for this research are listed below.

- OMNIKEY AG CardMan 3121
- SCM Microsystems SCL011 contactless USB card reader
- SCM Microsystems SCR 3310 contact-based USB card reader

Initial tests were performed on the OMNIKEY 3121. However, this card reader was found unusable due to the lack of support for extended APDUs. An APDU is a smart card Application Protocol Data Unit, the message format between a smart card and a smart card terminal, as specified in [ISO7816-4]. In the original APDU specification, the message data length is restricted to 256 bytes, since the header field indicating the length of the message data consists of only a single byte. Extended APDU is a feature available since the Java Card Platform version 2.2.2, which allows a message size larger than 256 bytes. When extended APDUs are used, three bytes are available to indicate the data length. When the first byte is left empty, the following two field define the total length of the message data. This allows a theoretical message size of 65536 bytes. In practice the messages size is restricted by the card terminal, the terminal driver and the smart card, which often have a much smaller message buffer size.

Note that extended APDUs are not an absolute requirement for this research. However, it greatly improves the code complexity of both terminal and card software. Additionally, sending one extended APDU is faster than sending a number of small APDUs due to the overhead in message processing involved when chaining small messages.

The Omni Key 3021 throws an exception when sending large extended APDU messages. The other two smart card terminals support extended APDUs of greater length. For these readers we found that the bottleneck for the maximum message size was not present in the reader or the driver, but the smart cards used. Experimentally we found that the maximum transmittable APDU size is 1455 bytes with the addition of 7 header bytes, which makes a total of 1463 bytes as a maximum message size for the SCM card readers. Larger packets overflow the message buffer of the card. This imposes a practical limitation on the message size. However, for this research we assume that C-DAX messages, such as topic data, do not exceed 1024 bytes. The implementation of the smart card terminal software used in this research, PC/SC lite, lists[1] a number of smart card readers and their support for extended APDUs.

[1] See `http://pcsclite.alioth.debian.org/ccid_extended_apdu.html`

Figure 4.1: SCM SCR 3310 USB smart card reader

Figure 4.2: SCM SCL011 contactless USB smart card reader

Both the SCM card readers support extended APDUs. The SCL011, see Figure 4.1, is a contact-based reader; it makes physical contact to the *pinout* of the smart card. A smart card contains pins for high and low voltage power, ground, the reset signal, the clock signal and input/output. See [ISO7816-2; ISO7816-3] for a precise specification of the contacts, interface and transmission protocols of a standard smart card. The SCM CR 3310 is a contactless reader, see Figure 4.2. This reader communicates and provides power through radio frequency induction technology. It requires the smart card to be in the proximity of the reader to operate. Apart from the fact that the physical interface of the contactless reader differs from the contact reader, the communication protocols are also differ. See [ISO14443-1; ISO14443-2] for the precise specification of the contactless smart card interface.

There are several alternatives to the physical shape of the smart card used in this research. The J3A080 and the J3A081 have a classic credit card size. Security tokens, can provide the same functionalities in the shape of a small USB stick. Such token could provide a more efficient chip interface. However, a token that supports the Java Card Platform is difficult to obtain. The only Java Card USB tokens found by the author are not sold separately, or not sold to individuals, based on an internet search of several hours. Other security tokens, for instance, tokens that implement the PKCS #11 standard are much easier to obtain.

4.2 Smart card applet implementation

The proof of concept Java Card applet created for this research serves two different tasks. The first task is to provide a simple interface to perform benchmarks of, inter alia, the underlying cryptographic algorithms. The second task is to serve as a test implementation of some of the C-DAX security functionalities.

The applet implements 17 commands: 6 *high level* functionalities and 11 *low level* functionalities. The high level functionalities are the C-DAX security functionalities that require more than a single step, such as encrypting or signing data. The high level functionalities are:

- generating a key pair;
- storing the security server public key;
- signing a topic join request;
- handling a topic join response;
- encoding topic data;
- decoding topic data.

Note that the above functionalities are just a subset of all the C-DAX security functionalities that a typical C-DAX client supports. For instance, control-plane communication is not covered by these functionalities. This subset merely forms a proof of concept for some interesting functions, which can be moved onto the smart card.

The low level functionalities provide an interface to the smart cards cryptographic capabilities, such as AES encryption and decryption, RSA encryption and decryption, RSA signing and verification and HMAC. Two functions are used to measure the throughput of the data connection to the card. One command simply returns an empty response, the other commands sends an arbitrary response of which the length depends on a request parameter.

The handling flow of a command on the smart card is simple. The terminal sends an APDU to the card, containing information to select the correct applet, an instruction command, optionally some parameters and a message length and a message payload. The smart card applet implements a method that handles a single APDU message. This method has access to the message buffer in which the incoming APDU bytes are present and in which the outgoing APDU bytes can be written. Once the APDU processing method terminates the buffer is sent back to the terminal in a single APDU.

Most implementations of the simple cryptographic commands in the applet are straightforward. During the installation of the applet, the applet object is allocated. In the constructor of this object the key, cipher and signature objects are allocated, as well as buffer memory. Once an RSA key pair is generated and the security server public key is stored, the RSA keys are initialized. When, for instance, an RSA encrypt command is sent to the card, the `Cipher` object is initiated using the RSA key. Subsequently, the `Cipher` is performed on the message buffer by a library call. The resulting ciphertext is written to the message buffer, which in turn is returned to the card terminal.

There are two required functionalities that are not fully implemented on the available card; AES padding (PKCS #7) and HMAC. Although HMAC is present in the Java Card specification version

2.2.2, the operating system on the available card does not implements this function [2]. However, HMAC is not a complicated operation, and the underlying hash functions are available on the cards. The following sections describe the exact implementation of AES padding and HMAC in the applet.

4.2.1 HMAC implementation

A Hash-based Message Authentication Code is a simple specification of constructing a message authentication code using a hash function [KBC]. The inputs of an HMAC function are a message and a key. Constructing an HMAC is done according to the following specification:

$$HMAC(K,m) = H\left((K \oplus opad) | H((K \oplus ipad)|m)\right)$$

In which K is the key, padded with extra zeros on the right to the input block size (which in our case in 64 bytes), m is the plaintext message, | is concatenation, $\oplus$ is a *exclusive–or* operation, *opad* is 64 bytes all with the value value $0x5c$ and *ipad* is 64 bytes all with the value $0x36$.

The HMAC algorithm can easily be implemented using just a few memory copies. Nonetheless, performing these operation on a Java Card in an applet, is significantly more expensive than in a typical library implementation of HMAC. We therefore chose to precompute as much as possible, to allow the final HMAC computation to be as fast as possible. The two possible values that can be computed in advance are the key padded with the *opad* and the key padded with the *ipad*. These values are constructed once a topic key is stored on the card. See Section 4.4.4 for a discussion on the memory trade off for this precomputation. Computing an HMAC with the two padded keys involves two memory copies, loading the padded keys to the buffer and two calls to the hash function. In our case the hash function is SHA256, which also is the only hash function of the SHA-2 family available on the cards.

The first implementation of HMAC performed in the applet failed verifying a message of around 620 bytes in length. We can explain this by assuming that the maximum message buffer size is around 1450 bytes. The implementation performed a complete data copy to calculate the second HMAC, which yielded $620 * 2$ bytes message size and $128 * 2$ bytes for the padded key, thus 1496 of memory. This is solved by allocating an extra transient byte buffer during applet installation. This buffer of 1536 bytes allows a maximum message size of 1408 bytes. The size of 1536 bytes was chosen based on the available memory in the card, allocating more that 1536 transient bytes yielded a memory error, see Section 4.4.4.

[2] See `http://www.fi.muni.cz/~xsvenda/jcsupport.html` for an overview of supported algorithms in different Java Cards. The Java Cards in question (J3A080 and J3A081) do not support any HMAC algorithm

4.2.2 AES padding

Every block cipher requires the input value to be a multiple of the block length. For AES-128, the symmetric encryption standard used in the proof of concept implementation, the block length is 16 bytes. Therefore input data has to be padded. An often used padding scheme is PKCS #7, which is the default for many cryptography libraries, including Crypto++. PKCS #7 padding works by adding bytes with the value of the total number of bytes added. For instance, a 10 byte input block would be padded with 6 bytes of the value $0x06$. When an input value is a multiple of the block length, a complete padding block is added.

The AES implementation on a Java Card version 2.2.2, does not include a padding scheme. It is therefore either required to implement it on the card, or perform padding on the terminal side. However, given the simplicity of the padding scheme, performing the padding scheme on a smart card does not have a measurable performance impact. The benchmark measurement presented in Chapter 5 show the execution of AES encryption and decryption with padding performed on the smart card. When the same benchmark measurements were performed with the padding scheme executed on the terminal side, no significant performance difference could be detected.

4.2.3 High level functions

During the initialization or personalization phase of a smart card, initial credentials are created or loaded on the card. In the proof of concept applet this is done by generating an RSA keypair on the card and storing the security server public key. RSA key generation of a 2048 bit key takes about one minute on the card, but this has to be done only once. The security server public key is also stored only once. The public key is encoded in an exponent of 3 bytes (the value 65537) and a modulus of 245 bytes. Before the key is stored, it is checked whether the key is already initialized. The possibility to substitute the security server public key would impose a considerable security weakness. Trying to do so will therefore yield an error.

Creating a topic join request only requires to sign a message containing the topic identifier, a timestamp and the client identity. Handling a topic join response, involves a more complex processing. This is only needed if all topic keys are solely stored on the card and not on the device. After decrypting and verifying the topic join response a topic key remains. A topic key is a 128 bit randomly generated value used to encrypt with AES-128 in Cipher-Block Chaining mode (CBC) and HMAC with the SHA-256 hash function. A byte array is used to store all AES keys, which are concatenated in blocks of 16 bytes. There is a separate byte array to store precomputed HMAC keys. These are padded with the HMAC inner and outer pad value and sequentially stored in the byte array.

Using the topic key, topic data can be encoded and decoded on the card. Encoding involves encrypting the topic data using AES-128-CBC and adding an HMAC over the resulting ciphertext. The HMAC is appended to the ciphertext and sent back to the terminal. Decoding topic data is the inverse of the encoding process. First the HMAC over the same ciphertext is computed and compared to the concatenated HMAC value. A mismatch in the HMAC values yields an integrity or authentication breach and therefore produces an exception, see Section 4.4.2 for an explanation of exception handling. After the HMAC is verified, the topic data is decrypted and sent back to the terminal.

4.3 Terminal implementation

This section gives an overview of the terminal side implementation, written in C++, which offers an API to the smart card functionalities and can perform a simulation of some C-DAX functionalities.

The Java Card applet communicates with the rest of the C-DAX client software using the PC/SC lite library[3] in addition to the terminal driver. The PC/SC lite library is a Personal Computer/Smart Card library originally developed for Microsoft Windows. PC/SC lite is a free implementation of the Windows library for Linux and other Unix systems such as OSX. The library offers function and classes that enable the programmer to connect with a smart card reader and to send APDU messages.

The proof of concept terminal implementation encapsulates this library in an C-DAX specific API implemented in C++. This API gives an easy interface to the card for benchmark purposes and simulation of C-DAX functionalities.

The C-DAX simulation is performed by running several threads. There are threads that implement an abstraction of the security server, a C-DAX node and clients, which possibly use a smart card. Messages between these threads are sent over TCP in the form of a serialized version of the message class. The message class contains attributes for topic data, an HMAC or signature and other administrative fields such as the source identifier field, topic identifier and a timestamp. Data in the message class is stored using the `Bytestring` data type. This type is an extension of the `SecByteBlock` class of the Crypto++ library. The `SecByteBlock` class offers a secure data structure, that cleans its memory after deallocation, to prevent leakage of sensitive information. Messages sent to the smart card are also encoded in a `Bytestring`. The message class offers, besides access to the message attributes, a number of methods to encrypt, decrypt, sign or verify its contents using RSA, AES or HMAC. All cryptographic algorithms are implemented using the Crypto++ library. The simulation of the threads and transferring messages between threads over TCP is done using the C++ Boost library[4]. Boost offers high level functions to set up threads, TPC sockets and many other commonly used techniques.

The functionalities of the proof of concept simulation include:

- A complete topic join mechanism, initiated by a C-DAX client. The topic join request is verified and forwarded by a C-DAX node and a response is sent back by the security server.
- A publish and subscribe mechanism, in which one or more publishing clients generate topic data and one or more subscribing clients receive topic data. The topic data is distributed and verified by a single C-DAX node.

Optionally the simulation enables a client process to use a real life a smart card to perform the cryptographic functionalities.

[3]See `http://pcsclite.alioth.debian.org/`
[4]See `http://www.boost.org/`

4.4 Design decisions

4.4.1 Key length

The maximum RSA key size allowed by the smart card used in this research is 2048 bits, see [NXP09]. The NIST recommends keys of at least 2048 bytes in size until the year 2030 [Bar+12], after that, longer keys are advised [5]. Newer versions of the JCOP cards have support for RSA keys up to 4K in size. Another option is using Elliptic Curve Cryptography (ECC). A big advantage of ECC is the key size. Smaller ECC keys in length can provide the same level of security of RSA keys; for the same security of a 2048 bit RSA key only a 224 bit ECC key is required. Unfortunately in the Java Card specification (version 2.2.2) only a specific ECC signature scheme is present. In [MEÁ11] a method is described to implement ECC encryption on Java Cards. On page 6 of [MEÁ11] a table is presented which shows processing times with respect to the message length. The use of ECC is not present in the implementations created for the current research, it is however considered as possible future work, see Section 7.3.

4.4.2 Exception handling

When an encryption or authentication/verification error occurs, the C-DAX layer should notify the application layer. Exceptions that occur on the smart card will bubble up to the C-DAX security module and are eventually communicated to the application layer. Exception events should be logged or aggregated in some other way to detect, for instance, denial of service attacks. The C-DAX client code includes a monitoring module that performs these tasks, see Section 5.3 of [CDAX31].

4.4.3 Encoding sequence

During the encoding of topic data, the content is first encrypted using AES-128-CBC, subsequently an HMAC is calculated over the ciphertext. This combination of message encryption and authentication is called authenticated encryption. Generally there are three approaches to perform authenticated encryption:

- **Encrypt than MAC** First the plaintext is encrypted, then the MAC is appended to the ciphertext. The IPsec protocol uses this method.
- **MAC then Encrypt** First a MAC is appended to the plaintext, then the concatenation is encrypted. This method is adopted by the SSL protocol.
- **Encrypt and MAC** A MAC is calculated over the plaintext, then the plaintext is encrypted and afterwards the MAC is appended to the ciphertext. The SSH protocol uses this method of authenticated encryption.

The first method is an ISO standard [ISO19772], since it is the only method that natively provides integrity of the ciphertext. Some researchers, such as [BN00; Kra01], also advice the first method. The

[5] See http://www.keylength.com/ for an overview and comparison of recommendations.

other methods of authenticated encryption can also have secure implementations, such as encrypt and MAC in the SSH protocol, see [BKN02]. In [Kra01] it is stated that for a block cipher that uses CBC mode, MAC then encrypt is also secure, since the random initialization vector causes the ciphertext to be different each encryption. Given the fact that the underlying block cipher might change, encrypt than MAC is proven to be the most secure option for authenticated encryption.

For RSA the same choice occurs: sign then encrypt or encrypt then sign. In naive implementations both methods are insecure, since they are susceptible to surreptitious forwarding, see [Dav01]. The RSA implementations used on both the smart card and the terminal software implement the PKCS #1 standard. This standard specifies a carefully constructed padding scheme, RSA-OAEP, which prevents many attacks that are possible on naive RSA implementations.

Generally it is a bad practice to use the same key for both sign and encryption operations. In the proof of concept implementation only one key pair is used. Although there are no practical attacks on PKCS #1 using the same key for both encryption and signing, using different keys is still advised. One reason for this is that when a single key is used and lost, both operations are compromised. Also, the validity period of encryption and signing keys might differ. It is therefore advised to use two key pairs per client in any real world implementations.

4.4.4 Smart card memory limitations

The smart card has a finite storage capacity, for instance, there is a limit in the amount of stored topic keys. The proof of concept implementation has a capacity of 255 topic keys. This limit is due to the size of the parameter used to indicate the topic key index, which is one byte. Also, a larger number of topic keys would require a non native number type as an index for the HMAC key byte array, since a number of the type `short` would not suffice.

The 80 kilobyte EEPROM memory of the smart card can be used for several applications, for instance, storing logs and topic keys. A trade off between these can be made, with respect to the use case of the card application. Since such a trade of is specific to the use case, no concrete trade off is proposed in the current research.

Another option is to store some data on the device. To maintain the extra security a smart card provides, the data can first be encrypted using a randomly generated key on the smart card. The storage of the device is in that case used like an cryptographic vault by the smart card. In this way even a compromised device cannot get access to the sensitive, encrypted data stored in its memory.

Chapter 5

Performance analysis

This chapter presents benchmark results of the algorithms introduced in Section 4. Besides a performance analysis of some of the C-DAX security functionalities, separate benchmarks of the underlying cryptographic algorithms were performed on the card. These give better insight into the total execution time of the *higher level* functionalities. A large portion of the time it takes to execute certain functionalities on a smart card is the transmission time. This chapter also presents test results of the terminal to card communication speed, or throughput benchmarks, in Section 5.3.

To assess the impact on execution time when using a smart card in several C-DAX use cases in Section 5.4, we compare the execution time of cryptographic algorithms on the smart card to the execution time on other devices. These devices include a desktop computer, a laptop computer and a limited, single board computer.

In Chapter 6 we will discuss some of the findings in this chapter and compare the results to related literature.

5.1 Materials

As stated in Chapter 4, this research is based on two smart cards models manufactured by NXP. According to [NXP09] these smart cards differ in both size of the available types of memory and clock speed. Table 4.1 on page 32 lists these differences. The product numbers of the cards are P5CD080 and P5CD081. Throughout this thesis, these cards are identified by respectively J3A080 and J3A081, which are the combined names for the card model and operating system. The operating system is the same on both cards; JCOP v2.4.1 version J3A, which implements the Java Card API v2.2.2.

Both cards include an AES co-processor, and a 'FameXE co-processor' that supports fast RSA and ECC operations [NXP09]. These co-processors greatly improve the execution speed of the cryptographic algorithms used.

There are two different card readers used to perform the benchmarks; a contact- based card reader and contactless card reader, see Figures 4.1 and 4.2 on page 33. The version numbers of these readers are the following:

- **Contactless USB card reader**
  SCM Microsystems SCL011
- **Contact USB card reader**
  SCM Microsystems SCR 3310

The interface with the card is not the only difference between the two card readers, the drivers and the protocols differ as well. On a high level, all card readers send and receive APDUs. However, the lower level communication, using the card antenna or chip contact interface, relies on different protocols. The resulting throughput performance impact of these differences is presented in Section 5.3.

In Section 5.4.2 the same cryptographic primitives are benchmarked as in Section 5.4.1, but executed on several different devices. The following devices and terminology are used:

- **Desktop computer**
  MacMini with a 2.5 GHz Inter Core i5 and 8GB RAM running OSX 10.9.2
- **Laptop computer**
  MacBook with a 2.1 GHz Intel Core 2 Duo processor and 4GB RAM running OSX 10.9.2
- **Single-board computer**
  RaspberryPi with a 700 MHz ARMv6k processor and 512 MB RAM running Raspbian [1]

The benchmarks in Section 5.4.1 were both performed using the laptop and the desktop computers as controllers for the terminal. This means that the computers sent the APDU instructions to the smart card and measured the execution time. The computer used to control the benchmarks in Section 5.4.1 did not have any significant influence on the measured card performance. This gives us an indication of the reliability of the measurement.

5.2 Method

Conducting reliable and statistically sound benchmark is a difficult task. However, there are certain metrics that indicate the reliability of a benchmark. For all benchmark measurements presented in this chapter, the standard deviation is recorded besides the average execution time. The average standard deviation gives a good indication of the reliability of the complete benchmark and is mentioned for each table or chart in the following sections.

Another way of obtaining an accurate average execution time is the total number of measurement repetitions. All card measurements of the smart card were at least performed 10 times in sequence. Measurement on the computer are performed at least 100 times, since these execution times are much lower than those of the smart card.

All measurements are performed by a program running on the computer controlling the smart card. This program was written in C++, such as all other terminal side code. The function used to measure the time is `gettimeofday`. This function gives a microsecond precision on the system used. All results are stated in milliseconds, using decimals where appropriate.

[1] See `http://www.raspberrypi.org/` and `http://www.raspbian.org/`

In Section 5.4.1 we present benchmark results of the execution time of cryptographic algorithms on the card. These measurements do *not* include communication overhead, such as the time it takes to send and receive data from the card to the terminal. The communication overhead is stripped by performing two measurements. The first measurement includes communication time and the execution time of the cryptographic algorithm. For the second measurement a special flag is passed to the card that instructs the card to perform the algorithm two times on the input data. The final measurement is obtained by subtracting the total measurement of executing the algorithm once from the total measurement of executing the algorithm twice. Theoretically this leaves us with the time it takes to execute the algorithm on the card. The measurements resemble the measurements obtained by subtracting the time it takes to send and receive a random blob of data from the time it takes to send the same amount of data and executing the cryptographic algorithm. However, the method to measure the execution times of the algorithm when executed twice provided more accurate results with a smaller standard deviation of the measurements.

All cryptographic functions of the smart card are validated during the benchmarks. This means that, for instance, when the card encrypts a message, it is decrypted on the computer performing the benchmark and compared to the plaintext. This prevents exceptions on the smart card from being ignored during the benchmark. Likewise, HMACs and signatures are validated on the terminal side.

5.3 Throughput performance analysis

Figures 5.2 and 5.1 show the raw throughput speed of card communication, with a fine grained amount of bytes transmitted. Both the contact-based and the contactless card readers are evaluated. These measurements are obtained by immediately returning the APDU request data as a APDU response from the card. The APDU request contains a parameter that defines the length of the desired response data. This way we can differentiate between sending and receiving data. For measuring the time it takes to send data, this response length parameter is set to zero. For measuring the time it takes to receive data this parameter is varied, while the length of the data sent over remains zero.

The average standard deviation for Figure 5.2 is ~ 0.23 milliseconds, for Figure 5.1 this is ~ 0.32 milliseconds. In both Figure 5.2 and Figure 5.1 we see small hops in the measurements every time the bytes transmitted exceed a block of 255 bytes. This could be the result of allocating a new or larger buffer for every block of 256 bytes.

We observe that, for both readers, the time it takes to send and receive data (transceiving data) is almost the same as the sum of individually sending and receiving data with the same length.

The obvious difference between the two readers is that using the contactless reader results in an almost six times lower communication overhead. Tranceiving 512 bytes takes ~ 345.32 using the contact card reader, while the contactless card reader takes only ~ 53.26 milliseconds. This difference can be explained by different *baud rates* that are used for the reader and card combination. The NXP smart cards can be configured to a certain baud rate, using a special command. The possible NFC (contactless) baud rate for both cards are 106, 212, 424 or 848 kilobits per second. For contact communication this is configurable up to 1500 kilobits per second. However, further analysis of the specific baud rate used proved to be difficult, since it required specialized equipment to investigate the communication between the reader and the card. Changing the baud rate was not possible for the available cards, since the command to perform this task was blocked by the card vendor.

Figure 5.1: Throughput benchmark (J3A080-contactless)

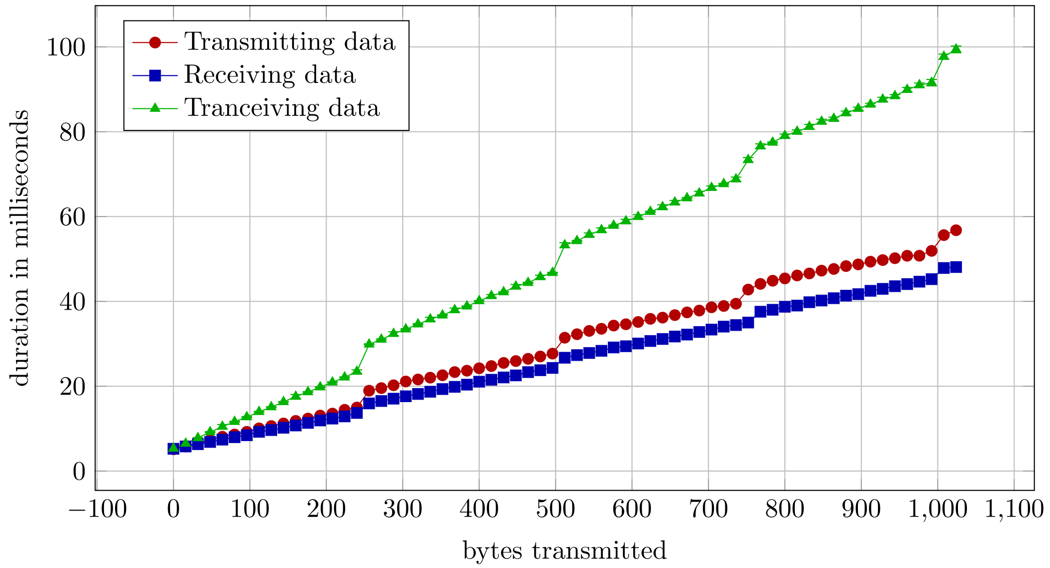
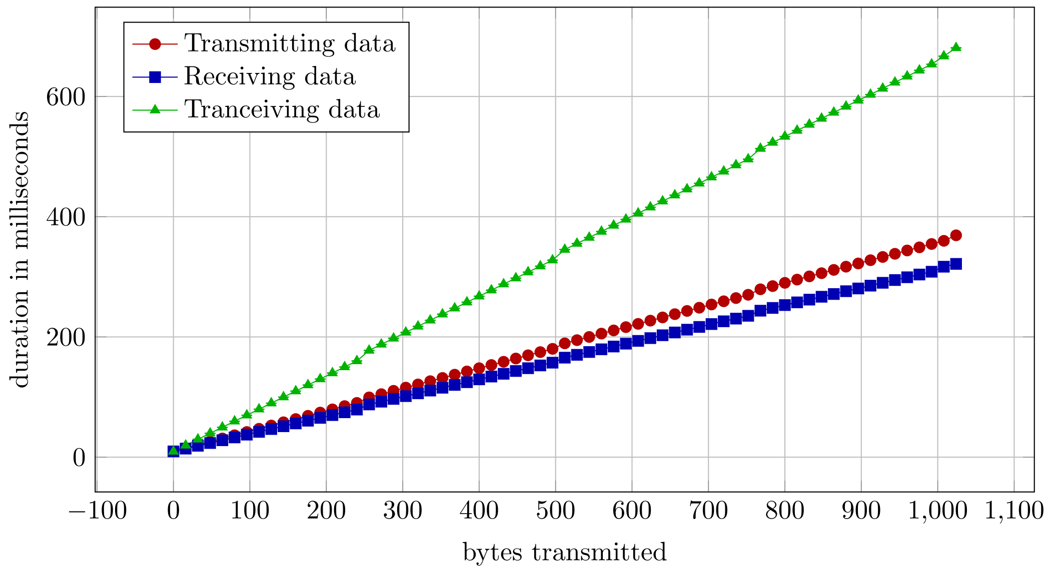


Figure 5.2: Throughput benchmark (J3A080-contact)



It is important to note that the acquired results in this section are based on a practical application of specific readers and cards. The authors are convinced that higher communication throughput is achievable when there is more control over the smart card, the reader and the terminal computer.

5.4 Cryptographic performance analysis

This section is divided in two parts. In the first part we present the measurements of the execution of various functions on the smart card. In Section 4.2 we divided the smart card applet functionalities into two groups; high and low level functionalities. Both the low level cryptographic algorithms and the high level C-DAX functionalities are benchmarked and the results are presented.

Section 5.4.1 presents the smart card benchmarks. In Section 5.4.2 we compare execution time of the low level cryptographic algorithms when executed on the smart card and on several types of computers.

5.4.1 Smart card benchmarks

Table 5.1 shows the average execution time of several cryptographic algorithms on the J3A080 smart card, set against the input message data length. The average standard deviation of all benchmarks in this table is ~ 0.49. The execution time is obtained by subtracting the time it takes to execute the algorithm twice with executing the algorithm once, as described in Section 5.2.

Table 5.1: Cryptography benchmark on a J3A080 smart card with contactless reader

Execution time in milliseconds against plain text length in bytes.

algorithm/bytes	8	16	32	64	128	256	512	1024
AES DECRYPT	38.84	39.36	40.48	42.48	45.94	135.18	226.89	411.94
AES ENCRYPT	38.65	39.3	39.63	40.9	45.91	51.68	63.89	88.89
HMAC	55.71	56.0	56.11	67.48	78.51	102.07	148.22	241.63
HMAC VERIFY	56.56	56.7	56.64	68.33	81.25	102.81	148.66	242.09
RSA DECRYPT	645.8	643.43	643.77	642.5	641.84			
RSA ENCRYPT	36.68	35.79	35.16	33.05	28.37			
RSA SIGN	655.49	652.42	652.74	658.49	666.06			
RSA VERIFY	99.23	99.02	99.34	105.51	110.66			

The results shown in Table 5.1 are as expected: symmetric algorithms are much faster than the asymmetric ones. In the same way, RSA decryption and signing algorithms are many times faster than encryption and verification ones.

If we compare the benchmark results presented in Table 5.1 to the results of same benchmark performed using the J3A081 smart card, there is no significant difference. All measurements are equal within a few milliseconds.

If we make a more fine-grained measurement of the execution time of AES CBC decryption on the card we see a unexpected occurrence. In Figure 5.3, the duration makes a unexpected jump around 220 bytes of input data size. We will come back to this result in Section 6.2.

Figure 5.3: AES CBC decryption benchmark (J3A080-contactless)

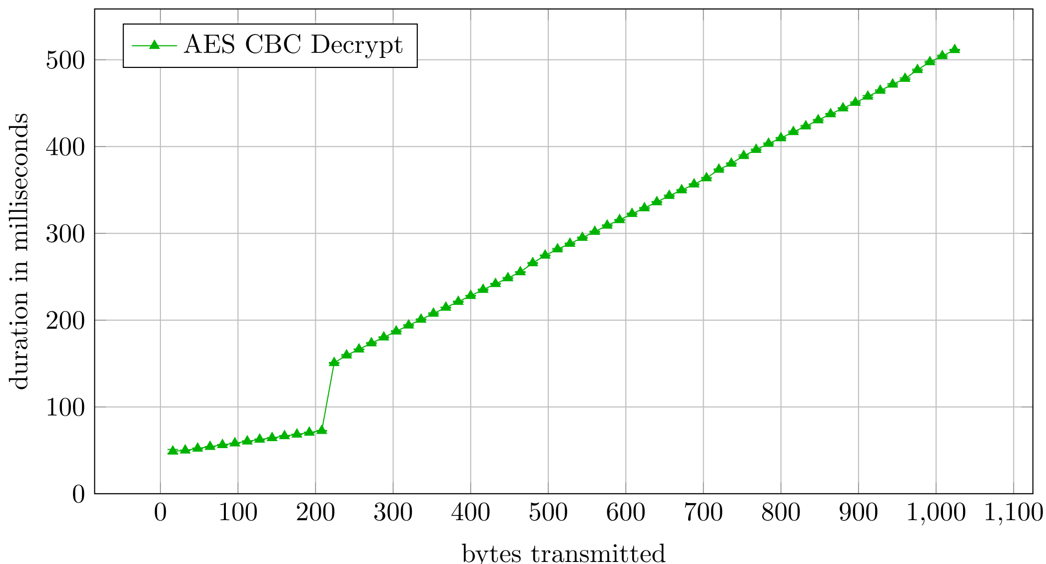


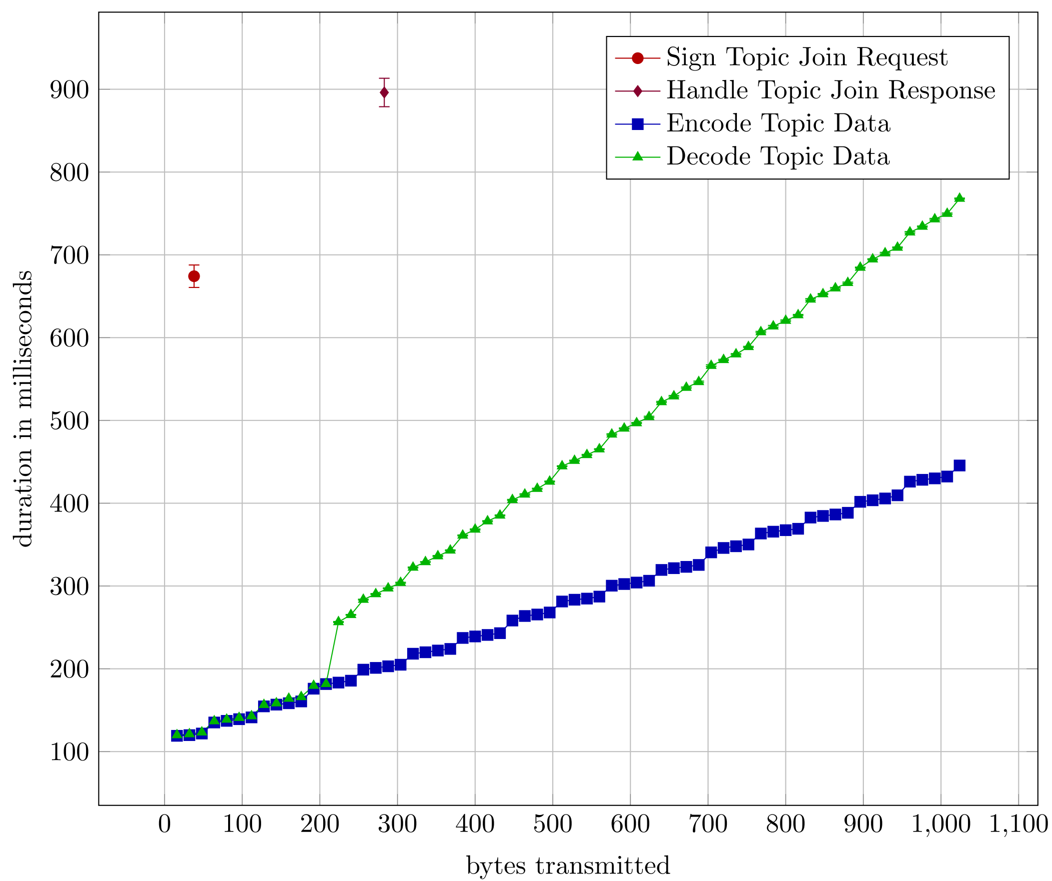
Figure 5.4 shows a benchmark of the high-level C-DAX client functionalities. These include signing a topic join request, verifying, decrypting and storing a topic join response and encoding and decoding topic data. Encoding means encrypting the message contents using AES-128 CBC and appending a single HMAC over the resulting ciphertext. Decoding means verifying the HMAC and decrypting the message contents. The results of the encoding and decoding operations are as expected when we compare these to the combined results of the throughput benchmark in Section 5.3 and the results of individual cryptographic functions in Table 5.1.

Signing a topic join request takes ~ 674 milliseconds with a standard deviation of ~ 6 milliseconds. The topic join request in our example is 38 bytes long, since it only contains the names of the topic, the source, and a timestamp. Storing topic keys on a card takes ~ 896 milliseconds with a standard deviation of ~ 5 milliseconds on average. Storing the topic key involves the verification of an RSA signature and the RSA decryption of the topic key. The topic join response message is 543 bytes long and contains message headers, RSA encrypted topic keys and RSA signature. We can explain these results by looking at the individual execution times of the RSA algorithms in Table 5.1 and in the throughput benchmarks.

From Figure 5.4 we can compute the possible data rates of the smart card for encoding or decoding topic data. Encoding is performed in ~ 0.8 to ~ 2.3 KiB/s for individual messages of respectively 128 and 1024 bytes in length. Topic data decoding is done in ~ 0.8 to ~ 1.3 KiB/s.

In Figure 5.4 the standard deviation per measurement is plotted in the chart. For encoding and decoding the topic data these *error bars* do not exceed the size of the icon on the measurement points.

Figure 5.4: High-level benchmark (J3A080-contactless)



5.4.2 Computer benchmarks

Tables 5.2, 5.3 and 5.4 show the execution duration of the same cryptographic algorithms as presented in Table 5.3. Tables 5.2 and 5.3 show benchmarks that are executed on a desktop and laptop computer respectively. Both tables have an average measurement standard deviation of $\sim$ 0.04 milliseconds. Table 5.4 shows the result of the same benchmarks on a single board computer. This single board computer might be more similar to some of the embedded devices we see in a practical C-DAX application. A smart meter, for instance, will not have the capabilities of a normal laptop or desktop computer. Even the single board computer used for the benchmark might be much more capable than a typical smart meter, since it contains 512 MB of RAM and a 700 MHz processor. However, our goal is just to give an approximation of the possible values.

Note that the measurements in Table 5.4 are less precise than the measurements performed on the other platforms. This can be attributed to the implementation of the time measurement function on this platform (ARM- Linux). The maximum standard deviation of these measurements does not exceed $\sim$ 4 and $\sim$ 9 milliseconds for respectively the symmetric and the asymmetric algorithms.

Table 5.2: Cryptography benchmark on a desktop computer

Execution time in milliseconds against plain text length in bytes.

algorithm/bytes	8	16	32	64	128	256	512	1024
AES DECRYPT	0.009	0.009	0.011	0.017	0.013	0.017	0.021	0.032
AES ENCRYPT	0.027	0.025	0.027	0.029	0.03	0.03	0.033	0.039
HMAC	0.009	0.008	0.013	0.009	0.01	0.012	0.02	0.021
HMAC VERIFY	0.011	0.01	0.01	0.012	0.013	0.013	0.023	0.024
RSA DECRYPT	2.329	2.761	2.499	2.333	2.371			
RSA ENCRYPT	0.235	0.219	0.221	0.202	0.177			
RSA SIGN	2.291	2.482	2.647	2.346	2.337			
RSA VERIFY	0.072	0.076	0.09	0.078	0.077			

Table 5.3: Cryptography benchmark on a laptop computer

Execution time in milliseconds against plain text length in bytes.

algorithm/bytes	8	16	32	64	128	256	512	1024
AES DECRYPT	0.015	0.017	0.018	0.019	0.021	0.034	0.034	0.049
AES ENCRYPT	0.041	0.045	0.043	0.053	0.052	0.052	0.053	0.062
HMAC	0.013	0.013	0.013	0.022	0.017	0.018	0.023	0.032
HMAC VERIFY	0.017	0.021	0.018	0.019	0.020	0.022	0.027	0.036
RSA DECRYPT	4.808	4.731	4.753	4.737	4.760			
RSA ENCRYPT	0.408	0.408	0.537	0.351	0.310			
RSA SIGN	5.843	4.776	4.802	4.737	4.85			
RSA VERIFY	0.142	0.143	0.137	0.139	0.138			

Table 5.4: Cryptography benchmark on a Raspberry Pi single board computer

Execution time in milliseconds against plain text length in bytes.

algorithm/bytes	8	16	32	64	128	256	512	1024
AES DECRYPT	0.999	1.131	0.844	0.769	1.325	1.252	1.4	1.908
AES ENCRYPT	1.334	1.26	1.251	1.394	1.342	1.551	1.75	1.993
HMAC	0.568	0.472	0.574	0.587	0.504	0.68	0.769	0.924
HMAC VERIFY	0.624	0.829	0.327	0.575	0.763	0.647	0.919	1.056
RSA DECRYPT	601.749	596.115	595.187	599.366	597.424			
RSA ENCRYPT	13.177	13.664	12.316	11.543	9.895			
RSA SIGN	594.802	595.373	596.786	596.418	597.194			
RSA VERIFY	8.202	11.095	8.525	9.56	10.172			

The benchmarks on the desktop and laptop computer run in just a fraction of the time that the benchmark takes on the smart card. Especially the symmetric algorithms perform much faster. On the single board computer, these benchmarks take a lot more time, but are still faster than the smart card. The asymmetric algorithms on the single board computer are the only exception; these take approximately the same time as on the smart card.

The benchmarks shown in tables 5.2, 5.3 and 5.4 are implemented in C++ using the Crypto++ library. The Crypto++ functions used implement the same standard algorithms as in the smart card. More benchmarks on different systems for the Crypto++ library can be found on [Cry]. Another resource of extensive cryptographic performance evaluations on different systems can be found at [Eba].

Part III

Results

Chapter 6

Discussion

The purpose of this study was to investigate how a smart card can be used to perform security tasks in C-DAX clients in order to improve device security. There were two parts that needed to be analyzed in order to draw a substantiated conclusion. The first part, in chapters 2 and 3, is the security implications for the C-DAX system as a whole when a smart card is used for certain functionalities. The second part, in chapters 4 and 5, is the applicability of a smart card for specific C-DAX use cases. For the latter part, both the performance of the cryptographic algorithms in question, and the throughput of the card are important.

6.1 Security analysis

The analysis of the first part was performed based on literature [Hen01; RE10] and further developed in a security analysis of the smart card integrated in a C-DAX client. In the literature we found that both hardware and software security features on a smart card can improve the security of a system significantly [Hen01; RE10]. For instance, smart cards have a better tamper resistance, which improves the protection of cryptographic keys. Additionally, the execution of cryptographic algorithms on a smart card is shielded from potentially compromised terminal applications, which also protects the keys. These properties are underlined by the many implementations of smart cards such as bank cards, SIM cards in mobile phones and personal identification cards.

The secure usage of a smart card involves a secure activation mechanism. The activation mechanism secures the deployment of smart cards in the field. In other words, it secures rolling out cards containing cryptographic keys to C-DAX client devices. In Chapter 3 we discussed the requirements of such an activation protocol and present two possible protocol variants that meet these requirements.

6.2 Performance analysis

In order to measure the performance of the implemented C-DAX security functionalities on the card a series of benchmarks were performed. We found that on the smart card available to us, a topic

join request could be signed on the smart card in approximately 674 milliseconds. Authenticating, decrypting and storing a topic key on the card took approximately 896 milliseconds. Encoding 256 bytes of topic data took around 199 milliseconds, decoding the same message size 284 milliseconds. The fastest results were obtained using a contactless card reader; the contact-based smart card reader was almost 6 times slower.

We found that performing the same cryptographic operation on a laptop or desktop computer only took a fraction of a millisecond. Heavy RSA operations take up to 6 milliseconds on these computers. For a low power computer, as one might find in an embedded device, these results differ significantly. Most symmetric cryptographic operations were performed in 1 to 2 milliseconds, while the heavy RSA operations were hardly faster than the same operation on the smart card. On the low power computer RSA decryption took around 600 milliseconds, on the smart card this operation took around 660 milliseconds.

In [MEÁ11] a method is described to implement ECC encryption and decryption on a Java Card. The benchmarks were preformed using the same Java Card as the one used in this thesis, the J3A080. The benchmarks results in [MEÁ11] show that using a 192 bit ECC key, encryption is possible in $\sim$ 614 milliseconds and decryption in $\sim$ 540 milliseconds. We found that the RSA implementation on the same card took $\sim$ 28 milliseconds for encryption and $\sim$ 642 milliseconds for decryption using a 2048 bit RSA key. While the ECC encryption and decryption might seem just a small improvement (or even a decline for encryption time) we have to take into account that the ECC keys are much smaller. ECC might also be more suitable for a resource restrained platform, such as a smart card, since it theoretically uses less computing power for the same level of security compared to RSA.

The throughput difference between the contact and the contactless card reader may be attributed to the baud rate the card is configured for. In the smart cards tested, the wireless protocol supported a higher baud rate than the wired protocol. Another possible influence could be the contact-based card readers we used, which might have support for low baud rates only.

The RSA benchmark results are mostly linear to the data size, with a small jump in execution speed for signing and verification of around 220 bytes. A similar, but more dramatic jump occurred when decrypting a ciphertext using AES-CBS 128. We do not have an explanation as to why this unexpected jump occurred. It might be related to the size of an internal buffer. In the throughput and high-level benchmarks we also observed little hops in execution time, at every 256 bytes. These might be due to new memory allocations.

One of the slowest algorithms is HMAC SHA-256. A possible explanation for this is the fact that this algorithm was not implemented on the card as a single low level library call. Instead it was implemented in relatively slow Java Card code. The custom implementation created for this research uses several buffer manipulations in addition to calling the underlying hash function twice, which could explain the slow execution time.

When we compared the J3A080 smart card to the J3A081, no significant differences were found. This could be explained by several causes. The difference between the cards in maximum internal clock speed might not result in different clock speeds, since the reader controls the clock. Both cards include the same cryptographic co-processors, we therefore expect that the cryptographic algorithm execution the be identical. The most significant difference between the cards is the amount of memory available. The available memory will typically not result in execution time differences.

Little scientific research has been published that explicitly states the speed of cryptographic algorithms of smart cards for RSA and AES cryptography. The most recent research that we could find was performed in 2006 [Mos+06]. Since 2006 the performance of cards has improved. The memory size of the card tested in [Mos+06] is almost half that of the cards used for this research. In 2006, the card that could perform AES-CBC the fastest, took around 77 milliseconds to encrypt and $\sim$ 89 milliseconds to decrypt a plaintext of 128 bytes. Our benchmark showed execution times of around 46 milliseconds for both operations. In [Mos+06] only RSA-CRT-PKCS#1 decryption was tested, executed with 245 bytes of input data. The fastest card performed this operation in approximately 509 milliseconds using a key of 2048 bits. The cards in this research can encrypt RSA-CRT-PKCS#1 in $\sim$ 642 milliseconds. This is not a performance increase compared to the research from 2006. More dramatic results were found for RSA signing and verification: in 2006, the fastest card took $\sim$ 645 milliseconds for verification, which is much slower than the $\sim$ 110 milliseconds found in the current research.

It should be stated that the results presented in this thesis are based on only two cards, from the same manufacturer. Moreover, a limited set of card readers was used to perform the benchmarks. Other cards or readers could show deviating performance results. In devices with limited performance, such as smart meters, the performance of the card will not be a bottleneck. The cryptographic co-processors embedded within these cards can even greatly improve the performance of the C-DAX security functionalities. With respect to the latency requirements as stated in [CDAX21], many use-cases could benefit from the added security improvements a smart card has to offer.

Chapter 7

Conclusion

In this chapter we conclude our research findings and reflect on the process that preceded these findings. In Section 7.3 we look ahead to possible future work and other topics that were not covered in this thesis.

7.1 Research summary

In order to gain insight in a typical C-DAX infrastructure, the author of this thesis started off with a proof of concept implementation of some of the C-DAX functionalities in Python. This high-level programming language allowed us to make a fast working example of a multi agent system in which several publishers and subscribers could communicate topic data using a single C-DAX node. The Python code can be found at [Bae14b].

After the Python implementation, a more serious C-DAX simulation was implemented in C++ using the Crypto++ library. The results can be found at [Bae14a]. the C++ implementation provided the basis for the Java Card applet implementation. The cryptographic algorithms used in the C++ implementation were adapted to the cryptographic possibilities of the card. The goal was to implement the cryptographic functions on the card so that they would exactly mimic the C++ implementation and the other way around. This involved sending over cryptographic keys to the card and comparing results of the output of cryptographic operations. Step by step all cryptographic functionalities were implemented on the card. This proved to be a difficult task in some situations, since the possibilities of debugging a smart card are cumbersome. If something went wrong on the card, it resulted to a single generic error code being returned on many occasions (`Error 0x67`). The resulting Java Card applet used in this research can be found at [Bae14a].

Once the smart card applet implementation was completed, benchmarks could be performed. When handling large messages, several shortcomings came to light. There appeared to be large differences between card readers. In total 6 different card readers were used, the only contactless card reader performed best. In addition implementation errors came to light using large messages. The message buffer was to small to append several intermediate results when verifying an HMAC. This resulted in another buffer allocation, during the installation of the applet.

The benchmark results gave us insight in the performance of the smart card for several C-DAX functionalities. When we compared the performance of RSA encryption and signature implementations of the smart card with a low power computer; a RaspberryPi, we found some interesting results. The performance of the smart card was comparable to the performance of the 700 MHz computer. This means that for small embedded devices, such as a smart meter, the smart card might actually improve the performance of the C-DAX client application with respect to the security functionalities. These results can be explained by the existence of a dedicated cryptographic co-processor in the smart card.

Apart from a possible performance improvement when using a smart card, the overall system security of a C-DAX client device in the field can also be improved. The tamper resistance of a smart card is proven by the numerous applications of smart cards in the field. There are countless applications of smart cards that provide security in a hostile environment. Applications range from banking to public transportation or personal identification. The protection of cryptographic keys is in particular a fitted task for a smart card. Rolling out new keys in the field would be greatly simplified when smart cards were used. If all cryptographic algorithms were performed on a smart card, upgrading the C-DAX security protocols would also be easier: replacing the smart card with a newer variant is enough to upgrade the cryptographic capabilities of a device. This prevents the need to replace the complete device and reduces the total costs of security upgrades.

In this thesis we also proposed a secure mechanism for the installation of a smart card in the field. This activation protocol provides the secure activation by the coupling of a smart card to a C-DAX device in the field. We show that the protocol fulfills the security requirements of such an activation protocol and thereby prevent several abuse cases of smart cards within the C-DAX system.

7.2 Applicability of smart cards in the C-DAX project

The aim of this research was to investigate the applicability of smart cards in the C-DAX project. We have found that smart cards are an excellent way of storing long term asymmetric keys. Performing asymmetric cryptographic functions on a smart card are possible for use cases that allow a delay of $\sim$ 689/136 milliseconds for signing/verification and $\sim$ 49/708 milliseconds for encryption/decryption with a message of 128 bytes in length. Soring topic keys on a smart card and encoding/decoding topic data is possible for date rates around 0.8 to 2.3 KiB/s for the encoding of messages. The data rate depends on the size of the individual messages; 0.8 KiB/s for 128 bytes messages and 2.3 KiB/s for 1024 byte messages. Message decoding is performed in data rates around 0.8 to 1.3 KiB/s, also depending on the individual message size. These data rates are based on the J3A080 smart card using AES-128 CBC and a single MAC using HMAC-SHA256.

7.3 Future work

This thesis takes the first steps in investigating the use of smart cards in the C-DAX infrastructure. However, there are many subjects that were not within the scope of this research, but relevant for the further incorporation of smart cards in the C-DAX project.

In the current research just a few smart cards were used. Newer smart cards might perform better and provide new cryptographic algorithms. There are several interesting cryptographic primitives to

research further, such as ECC. ECC might provide faster encryption and signature schemes with smaller key sizes. Another interesting subject is an alternative for the combination AES CBC and HMAC. Authenticated encryption such as AES GCM might simplify this operation and improve performance. Unfortunately, this mode of AES is not yet available on Java Cards. Other options for securing C-DAX communications include the use of pairing, see [VP13]. An implementation of pairing on a smart card has already been presented in [SCA06] and might be applicable to the C-DAX project.

Only a few high-level C-DAX functionalities were implemented in the proof of concept Java Card applet. In order to create a usable applet in practice, all C-DAX security functions should be implemented. In addition, possible key revocation or certificate schemes have to be implemented between card and terminal software.

A clear distinction between the C-DAX client responsibilities and the smart card should be established before any practical application is possible. This means a cut-off between the security functionalities has to be made. In this research just two possible cut-offs are proposed: storing asymmetric keys and only performing control-plane functionalities, or additionally storing topic keys and performing both control-plane and data-plane communication. There are many other cut-offs possible, certainly for the complete set of C-DAX security functionalities. In a practical implementation one of these possibilities has to be chosen.

A clear distinction of the responsibilities, between the card and the client, also includes a clear exception handling strategy. Such a strategy concerns the overall security of the system, since an exception might be the result of tampering or other malicious actions. A decent exception handling strategy defines what should happen when an exception occurs and which parties are responsible for the further handling of the exception. Therefore an adequate monitoring and logging functionalities should be present on both the client device and the smart card.

List of Figures

1.1 J3A080 smart card 7

2.1 C-DAX client architecture 12

2.2 C-DAX key distribution 14

2.3 Topic join sequence diagram 15

2.4 Topic join request and response 16

2.5 Topic communication sequence diagram 16

2.6 Topic data message 17

3.1 Activation protocol overview 21

3.2 Activation protocol message diagram 26

3.3 Activation protocol sequence diagram 28

4.1 SCM SCR 3310 USB smart card reader 33

4.2 SCM SCL011 contactless USB smart card reader 33

5.1 Throughput benchmark (J3A080-contactless) 43

5.2 Throughput benchmark (J3A080-contact) 43

5.3 AES CBC decryption benchmark (J3A080-contactless) 45

5.4 High-level benchmark (J3A080-contactless) 46

List of Tables

4.1 Memory size and internal clock speed of the J3A080 and the J3A081 smart cards 32

5.1 Cryptography benchmark on a J3A080 smart card with contactless reader 44

5.2 Cryptography benchmark on a desktop computer 47

5.3 Cryptography benchmark on a laptop computer 47

5.4 Cryptography benchmark on a Raspberry Pi single board computer 48

Bibliography

[Ahl+12] Bengt Ahlgren et al. "A survey of information-centric networking". In: *Communications Magazine, IEEE* 50.7 (2012), pp. 26–36.

[Ain+09] M Ain et al. "D2.3 Architecture Definition, Component Descriptions, and Requirements". In: *Deliverable, PSIRP 7th FP EU-funded project* (2009).

[AK98] Ross Anderson and Markus Kuhn. "Low cost attacks on tamper resistant devices". In: *Security Protocols*. Springer. 1998, pp. 125–136.

[And08] Ross Anderson. *Security engineering*. John Wiley & Sons, 2008.

[Bae14a] Marlon Baeten. *C-DAX security functionalities simulation and smart card API in C++ and the proof of concept Java Card applet*. `https://github.com/marlonbaeten/cdax-crypto-python`. 2014.

[Bae14b] Marlon Baeten. *C-DAX security functionalities simulation model in Python*. `https://github.com/marlonbaeten/cdax-crypto-python`. 2014.

[Bar+12] Elaine Barker et al. "Recommendation for Key Management–Part 1: General". In: *NIST Special Publication 800-57* (2012).

[BKN02] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. "Authenticated encryption in SSH: provably fixing the SSH binary packet protocol". In: *Proceedings of the 9th ACM conference on Computer and communications security*. ACM. 2002, pp. 1–11.

[Bla+10] Bruno Blanchet et al. *Proverif: Cryptographic protocol verifier in the formal model*. 2010.

[BN00] Mihir Bellare and Chanathip Namprempre. "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm". In: *Advances in CryptologyASIACRYPT 2000*. Springer, 2000, pp. 531–545.

[BPR01] Johan Borst, Bart Preneel, and Vincent Rijmen. "Cryptography on smart cards". In: *Computer Networks* 36.4 (2001), pp. 423–435.

[CDAX21] C-DAX Consortium. "Deliverable 2.1: C-DAX Requirements - Use Case Descriptions for Domains 1, 2 and 3 and Derived C-DAX Requirements". In: *Deliverable, C-DAX 7th FP EU-funded project* (Apr. 2013). `http://cdax.eu/sites/default/files/C-DAX%20Requirements%20D2.1.pdf`.

[CDAX31] C-DAX Consortium. "Deliverable 3.1: Specification of the initial C-DAX Architecture and Basic Mechanisms, Protocols and Algorithms". In: *Deliverable, C-DAX 7th FP EU-funded project* (Sept. 2013). `http://cdax.eu/sites/default/files/C-DAX-D3.1.pdf`.

[CDAX32] C-DAX Consortium. “Deliverable 3.2: Specification of the Security and Privacy Techniques for the C-DAX Infrastructure”. In: *Deliverable, C-DAX 7th FP EU-funded project* (Mar. 2013).

[Cry] *Crypto++ 5.6.0 Benchmarks*. http://www.cryptopp.com/benchmarks.html. Accessed: 2014-06-01.

[Dav01] Don Davis. “Defective Sign & Encrypt in S/MIME, PKCS# 7, MOSS, PEM, PGP, and XML.” In: *USENIX Annual Technical Conference, General Track*. 2001, pp. 65–78.

[DH76] Whitfield Diffie and Martin E Hellman. “New directions in cryptography”. In: *Information Theory, IEEE Transactions on* 22.6 (1976), pp. 644–654.

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer, 2002.

[Eba] *eBACS: ECRYPT Benchmarking of Cryptographic Systems*. http://bench.cr.yp.to/. Accessed: 2014-06-01.

[FIP01] NIST FIPS. *180-2: Secure hash standard (SHS)*. Tech. rep. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf. Technical report, National Institute of Standards and Technology (NIST), 2001, 2001.

[Fou+11] Mostafa M Fouda et al. “A lightweight message authentication scheme for smart grid communications”. In: *Smart Grid, IEEE Transactions on* 2.4 (2011), pp. 675–685.

[Hen01] Mike Hendry. *Smart card security and applications*. Artech House, Inc., 2001.

[HP00] Helena Handschuh and Pascal Paillier. “Smart card crypto-coprocessors for public-key cryptography”. In: *Smart card research and applications*. Springer. 2000, pp. 372–379.

[ISO13335] ISO. *ISO/IEC TR 13335-1 - Management of information and communications technology security – Part 1: Concepts and models for information and communications technology security management*. 2004. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=39066.

[ISO14443-1] ISO/IEC. *ISO/IEC 14443-1:2008, Identification cards - Contactless integrated circuit cards - Proximity cards - Part 1: Physical characteristics*. 2008. URL: http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=39693.

[ISO14443-2] ISO/IEC. *ISO/IEC 14443-2:2010, Identification cards - Contactless integrated circuit cards - Proximity cards - Part 2: Radio frequency power and signal interface*. 2010. URL: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=50941.

[ISO19772] ISO/IEC. *ISO/IEC 19772:2009 Information technology - Security techniques - Authenticated encryption*. 2009. URL: http://www.iso.org/iso/catalogue_detail?csnumber=46345.

[ISO27002] ISO. *ISO/IEC 27002:2005 - Information technology – Security techniques – Code of practice for information security management*. 2005. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50297.

[ISO7816-2] ISO/IEC. *ISO/IEC 7816-2:2007, Identification cards - Integrated circuit cards - Part 2: Cards with contacts - Dimensions and location of the contacts*. 2007. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=45989.

[ISO7816-3] ISO/IEC. *ISO/IEC 7816-3:2006, Identification cards - Integrated circuit cards - Part 3: Cards with contacts - Electrical interface and transmission protocols.* 2006. URL: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=38770.

[ISO7816-4] ISO/IEC. *ISO/IEC 7816-4:2013, Identification cards - Integrated circuit cards - Part 4: Organization, security and commands for interchange.* 2013. URL: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=54550.

[KBC] H. Krawczyk, M. Bellare, and R. Canetti. *RFC2104 - HMAC:Keyed-Hashing for Message Authentication.* http://www.ietf.org/rfc/rfc2104.txt.

[KCB97] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. "HMAC: Keyed-hashing for message authentication". In: *RFC 2104, Network Working Group* (1997). http://tools.ietf.org/html/rfc2104.

[Khu+10] Himanshu Khurana et al. "Smart-grid security issues". In: *Security & Privacy, IEEE* 8.1 (2010), pp. 81–85.

[KK99] Oliver Kömmerling and Markus G Kuhn. "Design principles for tamper-resistant smartcard processors". In: *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology.* USENIX Association. 1999, pp. 2–2.

[KKT12] Young-Jin Kim, Vladimir Kolesnikov, and Marina Thottan. "Resilient end-to-end message protection for large-scale cyber-physical system communications". In: *Smart Grid Communications (SmartGridComm), 2012 IEEE Third International Conference on.* IEEE. 2012, pp. 193–198.

[Kra01] Hugo Krawczyk. "The order of encryption and authentication for protecting communications (or: How secure is SSL?)" In: *Advances in CryptologyCRYPTO 2001.* Springer. 2001, pp. 310–331.

[KT05] Willett Kempton and Jasna Tomić. "Vehicle-to-grid power implementation: From stabilizing the grid to supporting large-scale renewable energy". In: *Journal of Power Sources* 144.1 (2005), pp. 280–294.

[MEÁ11] V Gayoso Martínez, L Hernández Encinas, and C Sánchez Ávila. "Java Card implementation of the Elliptic Curve Integrated Encryption Scheme using prime and binary finite fields". In: *Computational Intelligence in Security for Information Systems.* Springer, 2011, pp. 160–167.

[Mic06] Sun Microsystems. *Java Card Specifications Version 2.2.2.* Available at http://download.oracle.com/otndocs/jcp/java_card_kit-2.2.2-fr-oth-JSpec/, version 2. 2006.

[Mos+06] Wojciech Mostowski et al. *A Comparison of Java Cards: State-of-Affairs.* 2006.

[NXP09] NXP. *Identification product range.* Available at http://www.smartcardsource.com/pdf/NXP-JCOP-JA-cards.pdf, Document order number: 9397 750 16728. July 2009. DOI: Documentordernumber:939775016728.

[NXP12] NXP. *SmartMX for programmable, high-security, multi-application smart cards.* Available at http://www.nxp.com/products/identification_and_security/smart_card_ics/smartmx_dual_interface_controllers/. 2012.

[RE10] Wolfgang Rankl and Wolfgang Effing. *Smart card handbook.* John Wiley & Sons, 2010.

[RSA78] Ronald L. Rivest, Adi Shamir, and Len Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[SCA06] Michael Scott, Neil Costigan, and Wesam Abdulwahab. "Implementing cryptographic pairings on smartcards". In: *Cryptographic Hardware and Embedded Systems-CHES 2006*. Springer, 2006, pp. 134–147.

[VBN97] Khoi Vu, MM Begouic, and Damir Novosel. "Grids get smart protection and control". In: *Computer Applications in Power, IEEE* 10.4 (1997), pp. 40–44.

[VP13] Bárbara Vieira and Erik Poll. "A security protocol for Information-Centric Networking in smart grids". In: *Proceedings of the first ACM workshop on Smart energy grid security*. ACM. 2013, pp. 1–10.

Appendix A

Activation protocol verification in ProVerif

In Chapter 3 we introduced an activation protocol, which facilitates the secure installment of a smart card in a C-DAX device. In this chapter we provide a formal verification of some of the properties of this protocol.

The verification is performed using `ProVerif` [Bla+10], which is a cryptographic protocol verifier. The model, as listed below, consists of four different processes or threads. One for the C-DAX client device, one for the smart card, one for the installer and one for the security server. Each process is replicated, to simulate a system containing multiple cards, installers and devices. The modeled protocol is roughly identical to the protocol as defined in Section 3.5.3. There is one small deviation: the smart card sends its identity along with the first message to the security server. This allows the security server to lookup the correct public key of the smart card for signature verification.

Note that this formal verification only proves some specific security properties of the activation protocol and is in no way a complete proof of correctness.

The model below contains two main queries. The tool will attempt to construct a proof of the defined statements in these queries. Together, these queries verify security requirement 2 of Section 3.3, namely: "Blank smart cards can only be used by an authorized device". The verification model only considers requirement 2 because it is applicable to the security *protocol*. Requirements 1 is optional, and is like requirement 4 fulfilled by behavior of the smart card software. Requirement 3 also results in certain behavior of the smart card software: once the card is activated, it can not be activated again in another device. This behavior is mimicked in the verification model, since each smart card performs the activation protocol only once.

The first query is defined as follows:

```
query evinj:unlock(idc, idd) ==> evinj:startUnlock(idc, idd).
```

This says that a card can only be unlocked if and only if the security server has sent an unlock certificate, called *Message 4*. This statement is implemented with two events. One is fired before the security server sends the unlock certificate and one is fired when the smart card has processed the

unlock certificate. Both events contain the identity of the smart card and the identity of the device as parameters. For a correctness proof, these parameters have to be unifiable between the two events. The output of this query is shown below:

```
RESULT evinj:unlock(idc_2532,idd_2533) ==> evinj:startUnlock(idc_2532,idd_2533) is true.
```

This means the query was proven; the `unlock` event only occurs if and only if the `startUnlock` event has occurred.

The second query states that the installer has to take part in every protocol execution in which the security server generates an unlock certificate. The first event `startUnlock` is fired right before the installer forwards *Message 3*, and contains the encrypted and signed message as event parameter. The `installer` event is fired before the security server sends the unlock certificate.

```
query evinj:installer(msg) ==> evinj:startInstaller(msg).
```

The resulting output of this query is:

```
RESULT evinj:installer(msg_78) ==> evinj:startInstaller(msg_78) is true.
```

Which means that every successfully generated unlock certificate has to be preceded by the installer forwarding the second last protocol message, and the other way around: if the installer forwards the second last message, the security server always generate an unlock certificate.

The complete protocol model and queries is defined below:

```
1  (* public key lookup function *)
2  fun pk/1.
3
4  (* asymmetric encryption function *)
5  fun enc/2.
6
7  (* asymmetric encryption and decryption *)
8  reduc dec(enc(x,pk(y)),y) = x.
9
10 (* signature generation function *)
11 fun sign/2.
12
13 (* asymmetric signature scheme *)
14 reduc check(sign(x,y),pk(y)) = x.
15
16 (* public channel C–DAX cloud or internet *)
17 free net.
18
19 (* unlock command *)
20 free unlock.
21
22 (* private channels for card−device, device−installer and
       installer−server communication *)
23 private free CardDev, DevInstaller, InstallerServer.
24
```

```
25 (* private key of the server *)
26 private free KeyServer.
27
28 (* start unlock event occurs *)
29 query ev:startUnlock(idc, idd).
30
31 (* smart card only unlocks when the security server sends the unlock
     command *)
32 query evinj:unlock(idc, idd) ==> evinj:startUnlock(idc, idd).
33
34 (* start installer event occurs *)
35 query ev:startInstaller(msg).
36
37 (* security server only sends unlock if installer forwards message 4 *)
38 query evinj:installer(msg) ==> evinj:startInstaller(msg).
39
40 let device =
41   new IdDev;
42   (* message 1 *)
43   out(CardDev, IdDev);
44   in(CardDev, enc_msg1);
45   out(net, enc_msg1);
46   (* message 2 *)
47   in(net, enc_msg2);
48   out(CardDev, enc_msg2);
49   (* message 3 *)
50   in(CardDev, enc_msg3);
51   out(DevInstaller, enc_msg3);
52   (* message 4 *)
53   in(net, enc_msg4);
54   out(CardDev, enc_msg4).
55
56 let securityserver =
57   (* message 1 *)
58   in(net, (enc_msg1, IdCard));
59   let (NonceCard, IdDev, =IdCard) = dec(check(enc_msg1, pk(KeyCard)),
       KeyServer) in
60     (* message 2 *)
61     new NonceSS;
62     let msg2 = (NonceSS, NonceCard, IdDev, IdCard) in
63       let enc_msg2 = sign(enc(msg2, pk(KeyCard)), KeyServer) in
64         out(net, enc_msg2);
65         (* message 3 *)
66         in(InstallerServer, enc_msg3);
67         let (=NonceSS, =NonceCard, =IdDev, =IdCard) =
             dec(check(enc_msg3, pk(KeyCard)), KeyServer) in
68           (* message 4 *)
69           let msg4 = (unlock, NonceCard, IdDev) in
70             let enc_msg4 = sign(enc(msg4, pk(KeyCard)), KeyServer) in
```

```
71                  event installer(enc_msg3);
72                  event startUnlock(IdCard, IdDev);
73                  out(net, enc_msg4).
74
75  let smartcard =
76    new IdCard;
77    (* message 1 *)
78    in(CardDev, IdDev);
79    new NonceCard;
80    let msg1 = (NonceCard, IdDev, IdCard) in
81      let enc_msg1 = sign(enc(msg1, pk(KeyServer)), KeyCard) in
82        event initiate(IdCard, IdCard);
83        out(CardDev, (enc_msg1, IdCard));
84        (* message 2 *)
85        in(CardDev, enc_msg2);
86        let (NonceSS, =NonceCard, =IdDev, =IdCard) = dec(check(enc_msg2,
            pk(KeyServer)), KeyCard) in
87          (* message 3 *)
88          let msg3 = (NonceSS, NonceCard, IdDev, IdCard) in
89            let enc_msg3 = sign(enc(msg3, pk(KeyServer)), KeyCard) in
90              out(CardDev, enc_msg3);
91              (* message 4 *)
92              in(CardDev, enc_msg4);
93                let (=unlock, =NonceCard, =IdDev) = dec(check(enc_msg4,
                    pk(KeyServer)), KeyCard) in
94                  event unlock(IdCard, IdDev).
95
96  let installer =
97    (* message 3 *)
98    in(DevInstaller, enc_msg3);
99    event startInstaller(enc_msg3);
100   out(InstallerServer, enc_msg3).
101
102 (* the system *)
103 process
104   (!device)
105 | (!securityserver)
106 | (!smartcard)
107 | (!installer)
```