Thomas Wolfgang Pieber, BSc

Design and Implementation of a Secure User Authentication Protocol for Smart Sensors

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Assoc. Prof. Dipl.-Ing. Dr. Christian Steger

Institute for Technical Informatics
Graz University of Technology, Austria

Advisors
Assoc. Prof. Dipl.-Ing. Dr. Christian Steger
Dipl.-Ing. Holger Bock (Infineon Technologies Austria AG)

Graz, September 2016

Abstract

In the near future almost every device will be connected via the Internet. This vision is called “Internet of Everything” or “Internet of Things” or, in an industrial environment, “Industrie 4.0”. The process of connecting everything with each other yields huge benefits, but has also the potential of being misused - unknowingly or conscious. To counter that, security measures must be taken in order to restrict the access to authorized users. Furthermore, the communication between users and machines or within machines must be encrypted to weaken the threat caused by any adversary.
This thesis is written within the context of the EU-project IoSense, which motivates with the vision of a production facility of the near future. It then describes the development process of a method to perform an authenticated key exchange on microcontrollers. The developed design was implemented with the support of Infineon Technologies. The evaluation of the protocol is done by measuring the computation time of the whole authentication and key exchange process. It is compared to the time that is needed to exchange messages on its own. With these statistics the overhead of other protocols, that also perform authentication, can be calculated. The thesis is concluded with a prospect for future research.

Kurzfassung

In naher Zukunft wird fast jedes Gerät mit dem Internet verbunden sein. Diese Vision wird "Internet von Allem" oder "Internet der Dinge" oder, in einem industriellen Umfeld, "Industrie 4.0" genannt. Dieser Prozess in dem Alles mit Allem verbunden wird bringt riesige Vorteile mit sich, hat aber auch das Potential missbraucht zu werden - unwissentlich oder absichtlich. Um dem entgegenzuwirken müssen Sicherheitsmaßnahmen ergriffen werden um den Zugriff auf autorisierte Personen zu beschränken. Weiters muss die Kommunikation zwischen den Benutzern und Maschinen, oder unter den Maschinen verschlüsselt werden, um die Gefahr, die von Angreifern ausgeht, zu minimieren.
Diese Arbeit, welche im Rahmen des EU-Projektes IoSense angefertigt wurde, ist motiviert durch eine Vision von einer Produktionsanlage der nahen Zukunft. Es wird danach der Entwicklungsprozess einer Methode zur Durchführung von einem authentifizierten Schlüsselaustausch, welcher auf einem Mikrocontroller durchgeführt werden kann, beschrieben. Das entwickelte Design ist dann, mit der Hilfe von Infineon Technologies, umgesetzt worden. Die Bewertung dieses Protokolls geschieht mit der Messung der Zeit, die für den gesamten authentifizierten Schlüsselaustausch benötigt wird. Diese Messung wird dann mit der Zeit, die für einen Nachrichtenaustausch benötigt wird, verglichen. Mit diesen Werten kann der benötigte Mehraufwand verglichen mit anderen Protokollen, die ebenfalls eine Authentifizierung durchführen, berechnet werden. Diese Arbeit wird mit einem Ausblick für zukünftige Forschung beendet.

AFFIDAVIT

Note of Thanks

Here I want to thank all those who have guided me in the course of this master's thesis.

I want to express my thankfulness to Assoc. Prof. Dipl.-Ing. Dr. Christian Steger who has tutored me while conducting this work. His professional and personal support guided me throughout this time. The skillful assistance of my supervisors Dipl.-Ing Holger Bock and Dipl.-Ing Thomas Ruprechter and the competent help of Andreas Wallner improved the experience of this work much. Likewise, I want to thank my colleagues at Infineon for their support. Foremost, I want to show my appreciation for Dipl.-Ing. Dr. Christian Lesjak who helped me take foot at Infineon.

Beyond that I want to thank my parents Josef and Edith and my brothers Daniel and Roman who enabled me to participate in this study and encouraged me throughout it. Their support has led me through many tough times.

A special thank goes to my fellow students Jakob, Marco, Sarah, and Thomas who helped on countless occasions throughout my time in university. The discussions with them have inspired and have brightened the time. I found true friends with them.

Last but not least I want to thank all my friends who supported my moral patiently.

Contents

List of Figures

List Of Abbreviations

AES	Advanced Encryption Standard
APDU	Application Protocol Data Unit
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CDH	Computational Diffie-Helman
CPU	Central Processing Unit
DDH	Decisional Diffie-Helman
DES	Data Encryption Standard
DH	Diffie-Helman
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Helman
ECDHE	Elliptic Curve Diffie-Helman Ephemeral
EKE	Encrypted Key Exchange
FPGA	Field Programmable Gate Array
FTP	File Transfer Protocol
HF	High Frequency
IIoT	Industrial Internet of Things
IoT	Internet of Things
ISO	International Organization for Standardization
KDF	Key Derivation Function

MAC	Media Access Control
MAC	Message Authentication Code
MitM	Man-in-the-Middle
MMU	Memory Management Unit
NFC	Near Field Communication
NIST	National Institute of Standards and Technology
NVM	Non Volatile Memory
OSI	Open Systems Interconnection
PIN	Personal Identification Number
PLC	Programmable Logic Controller
RAM	Random Access Memory
ROM	Read Only Memory
RSA	Rivest, Shamir, Adleman
SHA	Secure Hash Algorithm
SIM	Subscriber Identity Module
SMS	Short Message Service
SPAKE	Simple Password-based Authenticated Key Exchange
SSL	Secure Sockets Layer
TLS-SRP	Transport Layer Security - Secure Remote Password
UI	User Interface
US-CERT	United States Computer Emergency Readiness Team
VLC	Visible Light Communication
WSN	Wireless Sensor Network
XML	Extensible Markup Language
XOR	Exclusive OR

1 Introduction

The following chapter gives an introduction to the "*IoSense*" project and to this thesis. It explains the vision we have and introduces to the topics of NFC-based sensor systems, password-based authentication, and key exchange. After that the contribution of this thesis to the IoSense project is explained. This chapter concludes with the outline of the conducted thesis.

1.1 Introduction to the Project

This thesis was carried out within the EU-funded project "*IoSense - Flexible Front End / Back End Sensor Pilot Line for the Internet of Everything*" which aims to add intelligence into sensors used in the "Internet of Things" (IoT). That is done in order to empower semiconductor manufacturers to be the "*key drivers for innovation and employment and creator for answers to the challenges of the modern society*"[1].

The project was initiated by Infineon Technologies to show that Europe still has a leading role in development, and that the fourth industrial (r)evolution - called "Industrie 4.0" - remains in Europe, as did the other three. To achieve this goal the project aims at increasing the pilot production capacity and improving the Time-to-Market for innovative microelectronics by establishing fully connected semiconductor pilot-lines in Europe [1].

[1] `http://www.iosense.eu` - accessed on 24.05.2016

Within this project a demonstrator called "*TrustWorSys*" will be implemented. This name is a compound of Trustworthy and System. From this name one can say that the aim is to give sensors logic to become smart sensors that we can trust. To do so we need to connect and configure them. On the microcontrollers a lightweight protocol should run that allows to secure the communication with the sensor and the configuration stored on it.

1.2 Motivation

This section describes the application possibilities of the theis' contribution in three parts. These segments are dedicated to industry, home, and science.

1.2.1 In Industry

We have envisioned a futuristic production facility that works with the concept of "Industrie 4.0". This term describes a part of the "Internet of Things" (IoT) and is also often referred to as "Industrial Internet of Things" (IIoT). The idea of this concept is that all devices in a factory become smart and can communicate with each other. This facility enables the company to produce low volume, high quality, fully customizable products. To enable the production of those products the configuration of the machines and maybe also the path, goods need to take between the machines, have to be changed frequently. To comply with this vision robots are used to transport the items between the machines. These robots need to know which item they currently transport and where it should go next. Furthermore the machine needs to know which item it just received and what to do with it. Therefore the machines need to communicate with each other as well as with the robots and the robots need to be capable of configuring the machines. As sometimes a human engineer or maintenance worker needs to interact with the machines and robots, they should provide a communication interface that is easily accessible and commonly available. As all the communication between the operating entities can be stored, and the machines are capable of frequent reconfiguration, we can furthermore envision that a central server is able to find optimal item flows and configurations with the help of computational intelligence. These configuration files can then be deployed during manufacturing and increase the efficiency of the facility.

In this link the human operator is capable of reconfiguring big parts of the production. Therefore he must be authenticated against the machines. A simple and widely used mechanism to proof that a dedicated user wants to use a device, is the usage of passwords. With the knowledge of the password - the secret key - the user can log into an account and perform operations on the machine. These accounts have different privileges and must therefore be secured from access of unauthorized persons. If the user wants to log into a machine via a remote terminal the key has to be securely transferred to the other end. An authenticated user can use the elevated rights to access secure information about the device (like measurement values or settings) and configure the device. Configuration of a device is a key component of a devices' lifecycle management. It needs to be configured in the manufacturing process to be tested, reconfigured if it is sold or resold, and configured and reconfigured by the enduser during the operational phase.

As the communication vector for configuration should not interfere with other links a short ranged wireless technique should be used. The best solution would be Near Field Communication (NFC - Section 2.3.3) as it is short ranged and already widely deployed on almost every new smartphone.

1.2.2 At Home

This technique to configure smart devices can also be used at home. When thinking of a smart home, many settings need to be changed to fit the users' needs. To be able to configure some devices, one must even connect to it with a cable. This can be done easier when utilizing the communication capabilities of a smartphone. The user just opens the app for his home, configures the parameters to better suit his needs, and taps the device with the phone. The phone uses the provided NFC interface to authenticate the user and configure the device. Another possibility to configure any appliances is to use a web-interface provided by the vendor, configure the parameters there, download the configuration file to the smartphone, and transmit it wirelessly to the apparatus.

1.2.3 In Science

A technology for researchers are wireless sensor networks consisting of sensor nodes. These are small, usually battery driven, devices. The nodes have sensing capabilities, a microprocessor, and some form of wireless communication technology. These nodes can be deployed almost everywhere. A requirement for them is to perform their operations for months. Sometimes they have to be reconfigured as the goals of the scientists change or better algorithms are developed. This software update is currently transmitted via the same channel as the sensor data is sent. Furthermore, these sensor nodes need to use their very limited resources to send the update to all nodes and they need to verify the correctness of the received data. A microcontroller performing cryptographic functions can also use a lot of precious energy. In order to resolve this issue, the update can be sent via NFC. With this technology the node can perform all energy consuming tasks while being powered with the NFC field, conserving the energy provided by the battery. These updates also need to be authenticated as not everyone should be allowed to change operational data on scientific instruments. Therefore a lightweight but secure algorithm needs to be executed to authenticate the user and provide additional security.

1.3 Configuring Devices

Many devices can perform more operations than those, which are noticeable by users. These are operations for testing the device, changing the firmware, or operations used during the development process. It is useful to restrict the customer from using these functionalities as they can disclose internal operations and company secrets. Other functions can be enabled if a customer purchases them separately. This is done to have one product capable of everything and selling it to different companies. For example a company might want to use only parts of a product. They do not want additional capabilities and therefore ask for a cheaper version. They can get this by restricting functionalities. Of course these settings must be secured as the customer could re-enable the full functionalities without having paid for them.

Configuration of a product to the users environment must be accessible by the user. These settings can be custom policies for communication (with the Internet), changing of cryptographic keys to be able to secure communication, setting a sensor to different modes in order to get better performance, changing the operational mode of actuators, setting up user accounts with elevated privileges, to name a few. These configurations must certainly be secured as well. Therefore the privileged users need to authenticate themselves to the device they want to configure - usually this happens by showing knowledge of a certain secret (a password). As the devices may be connected to a terminal over an insecure channel, the communication must be secured against intruders as they could get hold of the password as it is transmitted. This in turn requires the use of an encrypted channel.

1.3.1 Why Using Passwords

In order to use computers or change the operational mode of machinery, the operators need to authenticate themselves. This authentication gives them permission to operate the device in a certain manner. As different users have different needs why they want to use a device, different users get different levels of access. For example a normal user can see some operational data and an administrator is also allowed to change the mode in which the machine operates. As the users need to log into an account to get the needed privileges to operate a device, they need to prove their identity. The simplest way to give such a proof is to show that the user has knowledge about a shared secret. As this method of authentication uses passwords it is called password-based authentication. This pass-phrase should be kept a secret to ensure that only the authorized persons get access to critical operations. Therefore it should not be written down somewhere. That in turn means, that the passwords needs to be memorable.

1.3.2 What To Consider Using Passwords

As humans are not particularly good at remembering a random sequence of characters, numbers and other special characters, passwords are normally constrained to be words. With a dictionary attack these passwords can be obtained easily [BPR00]. To be more secure the first letters of a sentence can be taken as a pass-phrase. To get additional entropy in this password some letters can be replaced by similar-looking numbers or other characters (leetspeak). This gives a little amount of entropy for the password, but not as much as a complete random sequence of characters. It is still very probable that the password is a sequence of lower-case letters of the alphabet which are in the American Standard Code for Information Interchange (ASCII)-table at positions 97 to 122. Even with a random sequence of characters one only uses letters 20 to 126 of the 256 possible 8-bit characters.

1.3.3 Why Exchange Keys

For others to read encrypted data they first need to decode it. The decoding is done using a publicly available algorithm. This means that the information that secures the data has to come from somewhere else: the key. Possessing this key enables the holder to decrypt the message. If the key is sent with no protection, anyone who captures both, the key and the message, can read it. Therefore special key-exchange or key-agreement procedures are necessary to securely exchange key-material over the insecure medium ”Internet”.

1.3.4 What To Consider Exchanging Keys

When exchanging secret material like cypher-keys, one should always use methods that assure that the transmitted secret stays secret and secured. One of the best ways to securely give secret material to another person is to hand over the material in a personal meeting. As this cannot be done in the Internet, other methods need to be used. These methods must consider that, in the Internet, the communication is transmitted over many nodes where some of them may be compromised and also send the data to unauthorized persons. This means that an algorithm, which transmits secret data, must secure the secret even if the whole communication is stored and analyzed by an unauthorized user.

The key-exchange should also provide measurements to authenticate the communication parties. This is done to detect if an adversary reroutes the communication and places himself in the middle of the communication. In this scenario the adversary can not only listen to the whole communication but can also change every data-packet at will. This is called a man-in-the-middle attack.

1.4 Password-based Authentication

As described previously, passwords are used as a simple way to prove that a person is the one who he claims to be by showing knowledge of a shared secret. As passwords are used to authenticate a human operating a machine, this human has to hold the knowledge. When using passwords to authenticate oneself against someone else on a remote place, the secret has to be transmitted to the other party. This implies that the knowledge has to leave the communication device in some form - either in plain text or encrypted. A better way to authenticate oneself is to use the password as a seed to generate a key which is then transmitted. The remote party uses the same password to generate the second side of the key exchange protocol. When the protocol is finished successfully both parties have a session key that is generated by the password of the user. If the password is wrong (the user is not authenticated) the key will not match and the messages cannot be decoded. This method for remote authentication is a simple approach to use a password to generate an authenticated communication with another party whilst the password does not need to leave the communication-devices.

1.5 Exchanging Keys

Cryptographic keys are the fundamental building blocks that enable the secure data transfer over an insecure medium - the Internet. They provide not only access to this data, they can also enable any person to verify the origin and integrity of the data. These keys can also represent the owner of every secret transferred over the Internet. Therefore keys are also used to authenticate the communicating parties against each other.

If encrypted data is transferred to another party, this recipient should be able to decipher the message. Therefore a key is needed. For bigger amounts of data a symmetric cryptography is used. That means that the same key is used to en- and decipher the message. But this requires the sender to also transmit the used key to the recipient. As the information about the key needs to be transmitted over many nodes to the recipient the key also needs to be encrypted. This may seem like a chicken-egg-problem: the data is encrypted and the key is also encrypted - so we also need a key to decrypt the key. But also this key needs to be secured. This can be solved by using a different cryptographic style - asymmetric cryptography.

This method uses two keys. One of them is open to the public, the other one is secret. A message encrypted with one of these keys can only be decrypted by the other. This method is is more complex regarding computational effort and is therefore only used for small amounts of data - like a key.

1.6 Contribution of the Thesis

Within the IoSense project this thesis is located within the TrustWorSys demonstrator. An overview of this demonstrator is depicted in Figure 1.1. The marked region in this picture is where the thesis gives its contribution. Its main focus is on the "*Security and Trust*" aspect of the communication between the elements. To establish trust between the electronic device (smart sensor) and the user a password-based authentication scheme is used. With the use of the implemented protocol mutual authentication can be achieved. The security of the communication between the smart sensor and the user is achieved with a Diffie-Helman key exchange. The aim of the thesis is to perform the key exchange and the authentication in one step.
This thesis gives a detailed view on password-based authentication techniques and how the authentication is done. It furthermore builds a concept on how the authentication step can be used while exchanging cryptographic keys. Furthermore, a library for password-based authentication using the asymmetric Elliptic Curve Cryptography (ECC) is built. This library should be easily portable to an Infineon-type cryptocontroller. The prototypical implementation uses the OpenSSL cryptolibrary. This implementation in C++ features many aspects that are expected of authentication processes - e.g.: supporting multiple connections, multiple curves, and the change of passwords, and - of course - securely using the passwords.

To get the implemented code running on the cryptoprocessor and using the hardware-accelerations that are featured in the processor, the ECC-specific commands need to be changed to those that use the processor-specific cryptolibrary. An interface between the processor-specific commands and the OpenSSL library is implemented to be able to test the code. This version of the protocol contains all features that are available in the previous version.

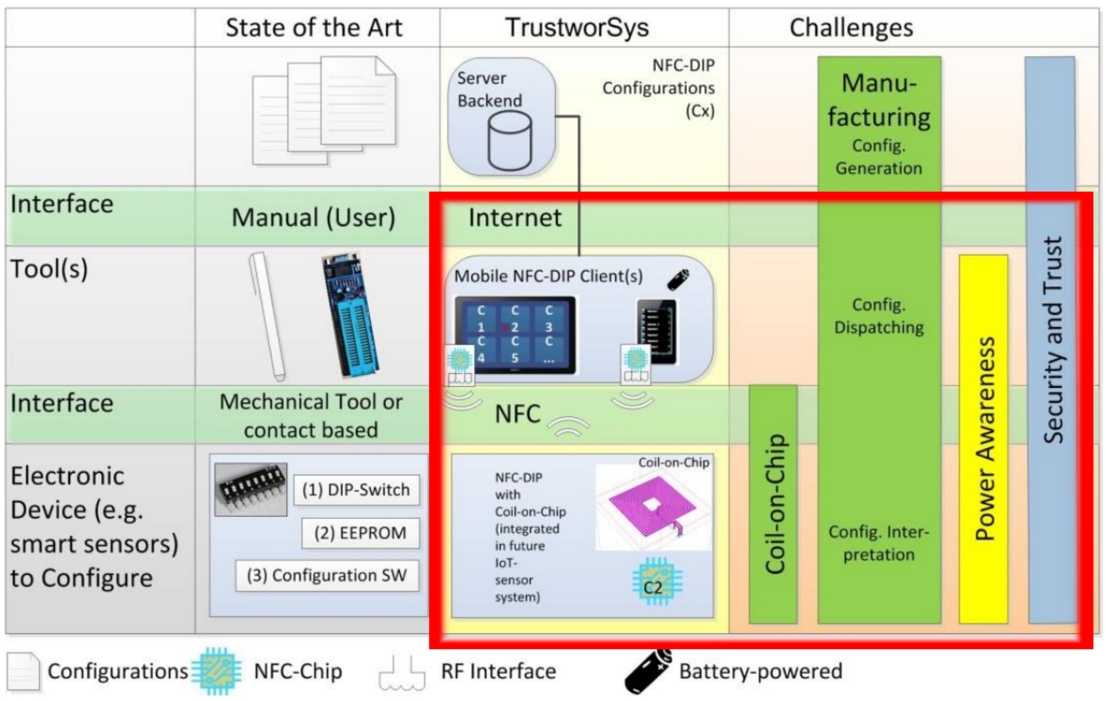


Figure 28: Overview on TrustWorSys

Figure 1.1: The TrustWorSys Demonstrator overview. The marked region is where this thesis is located.

The generated code is then transferred to the tool that creates the code that can be flashed to the cryptoprocessor. To be able to test the implementation a simulation software for the cryptoprocessor is used. In the version that is used on the simulation, only the functions for the machine-part are used, as the use case defines that the cryptoprocessor is on the machine-side. To be able to communicate with the protocol-implementation, Application Protocol Data Units (APDUs) are defined and the communication is implemented.

The demonstrator consists of a simulation for the cryptoprocessor running the generated code that is communicating with the computer that is running a user-side interface. The demonstrator is then able to communicate with the cryptoprocessor via the defined APDUs. It is able to perform the authenticated key exchange, change the name of the machine, configure the users of the machine, and perform an authenticated echo-request as a dummy operation.

1.7 Outline

In Chapter two related work on the topic of password-based authentication is analyzed. This chapter starts by motivating an authenticated key exchange and addresses security challenges, linked to authentication, that arise in the “Internet of Things”. This chapter further gives an introduction to the commonly used cryptographic primitives and algorithms. At the end of this chapter the necessary prerequisites are described.

Chapter three is dedicated to describe the design of the demonstrator as a whole, the authentication-process, the key-exchange algorithm, and the evaluation of the performance of the algorithm. Furthermore, the developed designs of the implementation iterations are described. It begins with an analysis of different scenarios where the thesis’ contribution can be used.

The fourth Chapter focuses on the details of the implementation. This chapter also describes the used implementation environments and the different implementations of the algorithm in each development iteration. Key points in this chapter are the authentication and the key exchange. It furthermore describes the used algorithm in more detail and gives better insight into the reference implementation.

The results of the performance evaluation process are described in Chapter five. In this chapter the evaluation process is described and the results are depicted. The chapter concludes with a Interpretation of the found results.

In the next chapter, Chapter six, a prospect into future work is given. The main focus is the combination of this thesis’ results with the results of other theses within the same or related projects. Furthermore, the goal of the development within the project is described.

This thesis ends with Chapter seven. In this chapter a conclusion of the whole work is given.

2 Related Work and Background

This chapter addresses works that are done on related topics to password-based authentication and key exchange. It begins by describing the process of key agreement and the basics of password-based authentication. With those sections the next one, which describes the authenticated key exchange, is motivated. After that the topic of "Internet of Things" and the security issues that arise with that are described. Furthermore, the mathematical background, for the used primitives in cryptography and key exchange, is described. At the end of this chapter the necessary prerequisites, to understand the remainder of this work, are described.

2.1 Related Work

In this section the problems of authentication and key agreement in the "Internet of Things" are described. To have a better understanding of the topic key agreement and password-based authentication are explained beforehand. With that the authenticated key exchange, which is used in many IoT-systems, is motivated. Finally the security issues, linked to authentication, in the "Internet of Things" are presented.

2.1.1 Key Agreement

Since ancient times, messages have been encrypted. The method of encrypting the data has changed often since then. But the problem to exchange the secret material for deciphering the message has stayed the same. In the beginning the key was attached to the message and it was hoped that no one would guess what to do with that. Or it was also a seemingly random string of characters that blend in with the encrypted material. Only when you know how to interpret the received data, one could decrypt the received text. This symmetric cryptography brings the capabilities of encrypting large amounts of data with comparably small computational effort. As the methods of encrypting data get standardized one cannot rely on the obscurity of the key-text-concatenation to keep messages secure. This implies that the key has to be sent to the receiver with a different method. This can be a personal meeting where the secret, used to encrypt data, is handed over to the receiver, or using a different communication channel. To get better security the key should be changed after every conversation. To do this one could send the next key within the last conversation. That, on the other hand, implies that, if the encryption is broken once, all the following conversations can be decrypted as the key is then known by the adversary. To solve this problem a key agreement must be performed. A well-known key agreement protocol invented is by W. Diffie and M. Hellman in [DH76]. This method is based on the assumption that it is infeasible to calculate the discrete logarithm of big numbers. In their protocol the two communicating parties agree publicly on a basis g and a modulus p. Additionally they choose a secret number x or y. Both parties calculate $X = g^x \; mod \; p$ or $Y = g^y \; mod \; p$ respectively. These numbers can then be transmitted. As it is currently not possible to calculate $g^{x \cdot y} \; mod \; p$ in a reasonable amount of time if x and y are big numbers and only X and Y are known. This problem is called the Computational Diffie-Hellman problem (Section 2.2.6).

A modern implementation of the Diffie-Hellman key agreement protocol is the Elliptic Curve Diffie-Hellman (ECDH). In this method every user generates a Elliptic Curve key pair. The public key is distributed. The shared secret between two parties is the multiplication of the own private key with the public key of the other entity. Because the public key can be linked to a certain person authentication is given. This implies that in the ephemeral version of ECDH (ECDHE) the authentication must be done in another way.

The generated shared secret can then be used directly as a key to use cryptographic functions. But in [LMQ$^+$03] it is recommended to use a key derivation function on this secret to remove weak bits that appear during the Diffie-Hellman key agreement.

2.1.2 Password-based Authenticaion

W. Diffie and M. E. Hellman stated in [DH76] that:

> The problem of authentication is perhaps an even more serious barrier to

> the universal adoption of telecommunications for business transactions than the problem of key distribution. Authentication is at the heart of any system involving contracts and billing. Without it, business cannot function.

This is not only true for businesses, but for almost every interaction that use digital parts. We need to authenticate ourselves against our cell phone, the mailserver, the router at home, the computer(s) and machinery at work, to name a few. Most of these systems rely on the use of passwords (or PIN codes). These secrets must be secure as they allow anyone to impersonate the owner. These passwords must therefore not be written down but remembered. That makes them an easy target for a dictionary attack. To prevent this from happening and to empower people to generate secure passwords the *United States Computer Emergency Readiness Team (US-CERT)* provides a guideline to create a secure passphrase on their website[2]. In this work the authors mention the most common mistakes and offer remedies. They further claim that an attacker can gain information on passwords with every other one. That implies that there is no such thing as a "less important" password and that all have to be made as secure as possible. One can think now, that the secure password would do a good job as a key for a cryptographic function. After all, passwords are the most common way of authenticating oneself and, with a strong one, performing encryption of the message should not be easier. But as S. Halevi and H. Krawczyk state in [HK99]:

> [. . .] using a low-entropy password as a key to a cryptographic function, can transform an otherwise strong function into a weak one. Namely, when using passwords as cryptographic keys, one makes the assumption that these functions remain secure even when the keys are chosen from a very small set. These assumptions are so unusual that [. . .] no one has been able to formally define the requirements from these cryptographic functions under which existing protocols can be proved secure.

The term "low-entropy" means that even a random string of characters uses only the letters 20 to 126 of the 256 ($= 2^8$) possible 8-bit characters of the ASCII-table. That means that passwords should not be used for encrypting data, only for authentication. Halevi and Krawczyk show some basic approaches for password-based authentication mechanisms:

(1) **Transmission in clear:** This mechanism is the classic one used in Unix systems and in remote authentication with **FTP** and **telnet**.

(2) **Challenge-response system:** In this scheme the password is used to compute

[2] `https://www.us-cert.gov/sites/default/files/publications/PasswordMgmt2012.pdf` - accessed on 27.06.2016

a secret function on the challenge of the server (the challenge gets encrypted with the password as key). The server, knowing the password, can decrypt the challenge and check if it is fulfilled. This mechanism is prone to offline attacks as the adversary knows the challenge and can try to guess the password.

(3) **One-time passwords:** Using these with a challenge-response system avoids the password-guessing attack by replacing the key after every use. This however, leaves the user with the inconvenience of carrying a long list of one-time passwords.

Later on, the authors propose mechanisms to provide additional functionalities such as mutual authentication, authenticated key exchange, and user-identity protection.

There are many algorithms that use a password for authentication. One of them is proposed by M. Peyravian and N. Zunic in [PZ00]. This protocol is especially well suited for microcontrollers as it does not use any encryption at all. The only cryptographic method used is a collision-resistant hash function H. In this protocol the user generates a random number (ru) and sends it to the server with his username (un). The server replies with another random number (rs). The user calculates the hash $H(H(un, pw), ru, rs)$ and sends the result to the server. The server, knowing the used hash with the username and password (pw) ($H(un, pw)$), can verify the message and grant access. This very simple method, depicted in Figure 2.1, also complies with a secure storage of the password as proposed by the National Institute of Standards and Technology (NIST) in their *Guide to Enterprise Password Management (Draft)*[3]. This document also describes the "salting" of hashes, as this decreases the chance of a birthday- or dictionary attack with precomputed hashes. This method of storing the passwords must be done because hardware-security modules are not feasible for storing many passwords.

[3] `http://www.tier3md.com/media/800-118.pdf` - accessed on 27.6.2016

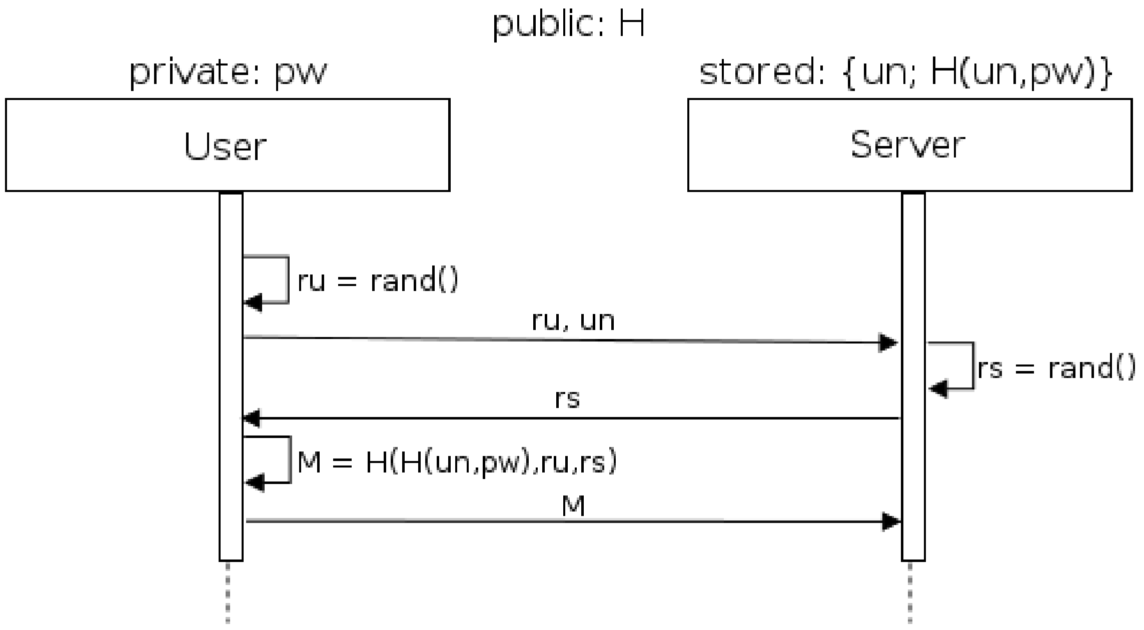


Figure 2.1: Lightweight authentication protocol. [PZ00]

2.1.3 Authenticated Key Exchange

With the knowledge from the preceding sections it comes easy to see that in the future of the "Internet of Things", and especially within the industrial context, we need encryption of the data and therefore key exchange mechanisms. Furthermore, authentication of users, or in general communication parties, is of high priority. As we have seen, many of these futuristic applications use microcontrollers as their computing element. They have limited calculation power. Therefore an efficient way of combining authentication and the secure exchange of keys must be used. A method that fulfills the criterion was proposed by S. M. Bellowin and M. Merrit in [BM92] in the year 1992. This protocol is called the "Encrypted Key Exchange"-protocol (EKE). They show the use of this protocol with various asymmetrical cryptography algorithms like RSA, ElGamal, and exponential key exchange. At this stage the proposed protocol uses six messages between the parties to perform a secured and authenticated link. It starts by symmetrically encrypting the public key and session key with the password as a key. The authentication is therefore done as only the two parties know the password. Many other variants of that protocol are proposed in the literature such as [BR00], [KI03], and [Kra03].

In "*The AuthA Protocol for Password-Based Authenticated Key Exchange*" by M. Bellare and P. Rogaway [BR00] the public RSA or ECC key is encrypted with a key, derived from the passwords and parties names, and sent to the other party. With that a master key and session keys can be calculated. They claim that this protocol is secure against dictionary attacks and the "Denning-Sacco attack" [DS81], provides perfect forward secrecy, and more. The latter protocol proposed by K. Kobara and H. Imai in [KI03] calculates another key with the password and another generator (a randomly chosen number or point on the ECC-curve). With this key the public key is masked. On the other side the inverse mask can be calculated and the Diffie-Hellman key exchange can be finished.

In more recent years the TLS-SRP (Transport Layer Security - Secure Remote Password) ciphersuite was created. This is an augmented password-authenticated key agreement (PAKE) protocol that works with values derived from, in advance shared, passwords. It was proposed in [TWMP07] by Taylor et. al. This specific protocol is designed to be cryptographically secure with weak passwords as an eavesdropper or Man-in-the-Middle cannot obtain enough information to guess a password without further interaction with the users.

In [DOW92] W. Diffie, P. C. Oorschot and M. J. Winter define that:

> A *secure protocol* is a protocol such that the following conditions hold in all cases where one party, say Alice, executes the protocol faithfully and accepts the identity of another party:
>
> - At the time that Alice accepts the other party's identity (before she sends or receives a subsequent message), the other party's record of the partial or full run matches Alice's record.
> - It is computationally infeasible for the exchanged key accepted by Alice to be recovered by anyone other than Alice and possibly the party whose identity Alice accepted. (This condition does not apply to authentication without key exchange.)

In addition to this definition of a secure protocol other desirable characteristics for a cryptographic protocol are: perfect forward secrecy, direct authentication, and no timestamps. That means that the exchanged messages should stay secret even if the long-term secret key is disclosed, the authentication should be established at the end of the protocol run - and not be done with a subsequent protocol, and it is not necessary for the communicating parties to have (synchronized) clocks.

These traits are found in the SPAKE (Simple Password-Authenticated Key Exchange)-protocol proposed by M. Abdalla and D. Pointcheval in [AP05]. This protocol performs a Diffie-Hellman key exchange with the messages masked using the authentication information. This protocol is not only secure by the definition above, but also efficient as it only needs two messages to be exchanged to perform an authenticated key-exchange. M. Abdalla states in a later published paper [Abd14] that:

> [. . .] the simple password-authenticated key exchange protocol [. . .] to which we refer as SPAKE [. . .] is among the most efficient PAKE schemes based on the EKE protocol.

This protocol relies on the random oracle protocol as a key derivation function is used to generate the session key out of the exchanged credentials. Therefore, it cannot be secure in the standard model of cryptography. A better, but computationally more expensive) protocol is the *Gennalo-Lindell PAKE protocol* [GL06]. This uses additional public key cryptography to get the added security in the standard model.

2.1.4 Internet of Things

The term "Internet of Things" (IoT) describes a vision where most of the everyday objects are connected with each other over the Internet. This was proposed by Mark Weiser in the 1991 Article "*The Computer for the 21st Century*" [Wei91]. He and his colleges proposed hundreds of computers in a single room, all connected with each other, performing tasks to ease everyday life. It is estimated that by 2020 the number of connected devices is about 50 billion. In 2008 the amount of connected devices surpassed the human population. Within this world humans interact with objects and these objects report their state to other objects. On a small scale that can be that the flower pot senses that the soil is too dry. The pot communicates that information to the phone of the owner, which should refill the water. Or the pot sends the information to an automated watering system which acts accordingly. This vision has many vulnerabilities and security challenges.

Cisco Systems proposed a framework for securing the IoT in [4]. In this scheme the authentication of IoT machinery (embedded sensors or endpoints) against the infrastructure. The machines can authenticate themselves with X.509 certificates, the MAC address or some sort of hardware trust anchor. The endpoint, typically a human user, can be identified by human credentials like a fingerprint or username and password. On top of the authentication authorization must be accomplished to form a trust relationship between the machine and the endpoint. As a third layer of the secure IoT framework network enforced policies come to use. This layer is established to transport the data securely to the endpoint. To gain visibility of the IoT environment (and maybe controlling the ecosystem) a fourth layer is added. This layer can perform statistical analysis to detect outliers and potential threats. The major challenges are that the computing platforms have limited resources and that encryption uses high processing power. Furthermore, the devices may need to be reconfigured such that security features can be changed if they are not secure

[4] `http://www.cisco.com/c/en/us/about/security-center/secure-iot-proposed-framework.html` - accessed on 27.8.2016

anymore. The lack of interest in security of IoT devices has put the IoT in a position that does not provide reliability, safety, and security - in short dependability.

The LEAD-project “Dependable Things” [5] should devise methods to increase the level of dependability of IoT devices. The four ongoing subprojects use WSNs as IoT devices and perform the research with them. These and even more challenges are to be addressed in an industrial environment (IIoT - Industrial IoT), where the data from the connected machines can reveal company secrets. The topic of industrial security becomes even more critical when looking at the “Stuxnet”-incident [Kar11].

This computer-worm infected a host computer and searched for peripheral devices that are used to program PLCs (Programmable Logic Controllers). On the PLC the worm manipulates the actuators to a point of failure whilst presenting a normal state at the superordinate control instance. This incident was a wakeup call for many industries that had seen security as an afterthought and add-on to their processes.

This problem is tackled in the doctoral thesis of C. Lesjak in [Les16] and the related publications [LRH$^+$14], [LRB$^+$14], [LHH$^+$15], [LHW15] and, [LB16] from the *ARROWHEAD*[6] project. This work describes the use of a broker-system to get maintenance data from the factory to the device vendor. In this work a mediator was attached to the industrial machine. This mediator encrypts the required data and sends it to the broker. There the snapshots from all customers of a vendor get collected. To ensure that the device sends only the maintenance data, the factory owner can request the own data. The concept further ensures that any other company cannot read data from potential competitors.

As the literature shows Near Field Communication (NFC) is a promising technology for the IoT. Due to its short range it does not interfere with other communication in the vicinity. Furthermore, the data rate is sufficient to transmit the required messages in a reasonable time. NFC is also used as a tool to exchange keys for another connection with higher bandwidth like WiFi. This short-range out-of-band configuration ensures the authenticity of the other device whilst the key exchange is inaudible to eavesdroppers on the high-bandwidth channel. The reconfiguration of the NFC tag yields weak spots if the user or the tag are not authenticated. To counter that problem J. Baek and H. Y. Youm present a lightweight protocol for authentication against NFC tags in [BY15]. Their scenario describes the malicious reconfiguration of a NFC tag to infect smartphones, steal data, and perform phishing and smishing (phishing over SMS). Other protocols try to secure the data transmission of NFC. The technology proposed by R. Jin and K. Zeng in [JZ15] secures the communication at the physical layer of the OSI-stack by modifying the

[5] `https://www.tugraz.at/projekte/dependablethings/home/` - accessed on 29.8.2016
[6] `http://www.arrowhead.eu/`

field strength randomly during the transmission. This protocol seems to be well suited to counter most attacks. One violation this protocol cannot handle is in the case that the reader itself is compromised. This is especially dangerous in case of a payment system based on NFC. The protocol by O. Jensen et. al. in [JGQ16] is claimed to be able to counter that form of attack by performing a challenge-response protocol between the NFC tag and the backend-server (at the bank).

2.2 Background

This section describes the mathematical background for different cryptographic methods and the used cryptographic primitives. Furthermore, the most notable parts of the described mechanisms are pointed out. A good reference to start with cryptography is [PP09]. Another very useful source for Elliptic Curve Cryptography is [CFA$^+$05].

2.2.1 Cryptography

A cryptographic scheme should have the following security features:

- **Authentication:** An efficient way to authenticate the creator of a message is by signing it. This is done by calculating a hash of the message and encrypting it with the private key. As everyone can have the public key it is easy to decrypt the hash. If it is consistent with the digest calculated with the message the author is fixed.
- **Data integrity:** This term explains that the received data should be the same as the sender intended it to be. This can be achieved by calculating a message digest. If something on the message changed, either through an error in the communication or by an attacker, the digest would show this.
- **Data security:** With the use of (symmetric) cryptography the data can be made secure against eavesdroppers. Even if the eavesdropper gets hold of the message he cannot decipher it.
- **Anti-tampering:** A temper-resistant message can be created by encryption, and authentication. In this way the message is secured against an attacker that can manipulate the message. But he cannot encrypt the message digest in the way the intended "author" could.
- **Anti-counterfeit:** This trait is gained with asymmetric cryptography. Only the author can encrypt the message digest with his private key.

In addition to those features the “Open Group” proposed in the “OISM3” standard an operational definition of security objectives. These are *confidentiality*, *integrity*, *availability*, and *non-repudiation*. These terms can be linked with the features above. Confidentiality and integrity can be gained with data security and -integrity. Availability describes the fact that the message should be decipherable at any time and non-repudiation can be achieved with the use of a certificate and public-key cryptography. The certificate is signed by a trusted third party and links the keys to a dedicated entity.

2.2.2 Symmetric Cryptography

Symmetric Cryptography is a method to en- and decipher messages using the same key. An early example for this method was the Caesar-cipher. In this method the secret key was a number by which the letters were shifted [Kip06].
A common and widely used method is the Data Encryption Standard (DES). As the length of the key is only 56 bit it is considered as not secure enough. Methods to increase the keylength are for example the Triple-DES (TDES or 3DES). In October 2000 the successor for DES - the Advanced Encryption Standard (AES) - was announced by the National Institute of Standards and Technology (NIST).

The Data Encryption Standard

The DES-scheme takes the input, splits it in two equally sized blocks and performs 16 Feistel-rounds. Each of these rounds looks as follows: The input on one side gets expanded to 48 bit. Then it gets mixed (XORed) with the subkey of this round. After that the 48 bit get split into eight 6-bit blocks. These blocks are then transformed to four bits each. The result gets rearranged by a fixed permutation. The next step is the XOR-ing of the output with the data from the other side. As a last step the two sides get swapped.

The Advanced Encryption Standard

The AES-scheme is based on a substitution-permutation network. The network gets repeated 10 to 14 times, based on the key-length. Each repetition follows the order:

1. Form a n-by-n block.
2. Replace each byte of the block by another one based on a non-linear function.
3. Shift the rows of the block to avoid that columns are independent.
4. Multiply each column by a fixed polynomial.
5. XOR the round key to the data.

In the first round only the last step is performed, in the last round all but the third (multiplication with the polynomial).

2.2.3 Asymmetric Cryptography

Asymmetric Cryptography or public-key-cryptography describes methods that use different keys for encryption and decryption. That means that a message that was encrypted with key A can only be deciphered with an inverse key B that is not equal to A. This form of encryption is expensive regarding computational effort and is therefore mostly used for small amounts of data. For example to distribute keys (Diffie-Hellman key exchange - DH) for symmetric cryptography methods or to digitally sign messages (Digital Signature Algorithm - DSA). Some other methods provide both functions. These are for example the "Rivest, Shamir and Adleman" (RSA) and "Elliptic Curve Cryptography" (ECC).

Rivest, Shamir and Adleman

The security of RSA is based on the difficulty of factoring the product of two large prime numbers - the factoring problem. When using RSA the user generates two keys (one public and one private) and an auxiliary number. The auxiliary number is the product of the used primes and is the divisor in the modulo operation. One of the keys is a number that is coprime (only integer that divides them is 1 [7]) with the totient (Euler's totient function [8]) of the divisor. The other key is the modular multiplicative inverse to the first key.
To encrypt a message one has to take the message to the power of the key modulo the divisor. The decryption is the same operation with the private key as the exponent.

Elliptic Curve Cryptography

ECC is based on the algebraic structure of elliptic curves over finite fields. The problem of discrete logarithms on elliptic curves is more difficult than normal, therefore the key-length can be smaller to reach the same amount of security. Thereby also the calculation-time is reduced. The curve is represented by the formula $y^2 = x^3 + a \cdot x + b \ mod \ p$ where $4 \cdot a^3 + 27 \cdot b^2 \neq 0$. There are 6 parameters for an ECC-system:

1. p: Prime, Defines the field in which the curve operates. All operations are made modulo p.
2. a, b: Integers, Define the curve.

[7] `https://en.wikipedia.org/wiki/Coprime_integers` - accessed on 27.05.2016
[8] `https://en.wikipedia.org/wiki/Euler's_totient_function` - accessed on 27.05.2016

3. G: A point on the curve. The start (generator) on the curve. G is maybe given as g_x and g_y, both integers.
4. n: Integer, Order of the curve generator point. This number specifies the number of reachable points on the curve based on multiplying with a scalar. This is only used for ECDSA (Elliptic Curve Digital Signature Algorithm).
5. h: Cofactor of the curve. This is the number of curve points.

The private key is an Integer in [1, n-1] mod n. The according public key is the point that one gets when multiplying the generator G with the private key. Generating a symmetric key is done by choosing a random number, multiplying this number with the generator point and transmitting the resulting point. Also the random number is multiplied by the public key. From the resulting point a symmetric key can be derived.
Not all curves that fit in this scheme are safe to use. A summary of the most used curves and their security status is shown in [9].

2.2.4 Cryptographic Hash

A cryptographic hash is a function that maps any input of different lengths to an output (hash value, hash, message digest or digest) of fixed length. This mathematical function is designed to be a one-way-function, that means that it is infeasible to invert the function. An ideal cryptographic hash functions has four main properties:

- The digest is quick to compute for any input.
- The digest is of uniform length regardless of the input.
- It is infeasible to generate a message from it's hash except by trying all possible inputs.
- It is infeasible to find two different messages with the same digest.
- Even a small change in the input changes the output so drastically that is appears uncorrelated with the other hash value.

Hash functions are mainly used for digital signatures, message authentication code generation, key derivation, checksums, indexing and finding duplicates and different kinds of authentication.
Commonly used cryptographic hash functions are the MD5 and the SHA versions.

[9] `https://safecurves.cr.yp.to/index.html` - accessed on 01.06.2016

Principle of Hash Functions

There are two main methods of generating hash-values. The MD5 is based on a custom designed structure and works with a compression-function at its heart. A block from the input is taken and together with the previously calculated hash placed in the compression function that works with bitwise boolean functions (AND, OR, NOT, XOR). This is done until the last block and the last calculated value is given as the message digest or hash. This model for calculating the hash is called a *Merkle-Damgård hash function* [Mer79].

Another method for obtaining hash values is by utilizing a block cypher. This is done in SHA-1. In this method the last output is used as a seed for the key generation for the encryption of a block. This is done using a m-to-n (bit) mapping. After the encryption of a block the output is XORed to the original block. This is the output for one message-block. The last output is then returned as the message digest. This construction is called *Matyas-Meyer-Oseas hash function*. There exist many other variants of this method. One uses the message-block as key and the hash is encrypted (*Davies-Meyer*) another one also uses the old hash value to be XORed to the result (*Miyaguchi-Preneel*).

Key derivation

Key derivation is done using a Key Derivation Function (KDF) which generates new key material from old ones or to generate session keys (or ephemeral keys) from the shared secret derived in the beginning of the communication. Deriving new keys is done because if an attacker gets hold of a key he can only decipher a small portion of the communication [PP09].
A simple way of deriving a new session key is to send a random nonce (value only used once) to the remote party and use this as the seed for the next key. The real key is then calculated by encrypting the nonce with the shared key (the key agreed on in the beginning of the communication) and uses this value as the new key. Another possibility is the use of hash functions on the random nonce. If a counter is used as nonce the new key can be calculated by both parties independent and therefore the no additional value needs to be transmitted.
A KDF needs to be a one-way-function. This is done because if the attacker can get hold of a session-key he should not be able to recalculate the shared secret as this would allow the attacker to calculate all other session keys.

2.2.5 Message Authentication Code

A Message Authentication Code (MAC) is also called keyed hash function. From this name one can see that this is calculated using the hash of the message. A MAC is sometimes confused with a digital signature, but a MAC is calculated with a symmetric cryptography whilst a signature is calculated using asymmetric cryptography. This means that MACs do not provide nonrepudiation (after the communication no one can blame that a certain message is from him).
MACs are used as a simple way to ensure that a message was not manipulated. This is done by sending the message and the encrypted hash of the message. As the adversary is not able to fake an encrypted hash he cannot manipulate the message. With this mechanism in place message integrity is provided. In respect of this also the authentication is provided as the other communication party has to have generated the MAC.

Another method of creating MACs is by utilizing Block Ciphers. As ciphers can also be used to generate hash values they can also be used to generate MACs. This is done in the same way as to create a hash. That means that the message is split into blocks, the blocks are XORed with the previous output, this input is encrypted using the shared secret. The output after the last block is then used as MAC. As the key is used to generate the MAC, message authenticity is provided.

2.2.6 Diffie-Hellman

The Diffie-Hellman (DH) key exchange protocol works with two asymmetrical keypairs. It is a method to generate a shared secret based on the public key of one and the private key of the other party. This key-exchange protocol is based on the assumption that it is infeasible to calculate a discrete logarithm for big numbers as the discrete exponentiation is a one-way-function. That means that the function $y = g^x \ mod \ p$ can be calculated efficiently but there is no fast algorithm to calculate x if y, g and p are given.

Computational Diffie-Hellman

The Computational Diffie-Hellman (CDH) problem describes the complexity to calculate the discrete logarithm given the result, basis and modulus. The full problem is to get the shared secret $g^{a \cdot b} \ mod \ p$ given p, g, $A = g^a$ and $B = g^b$. This means that the CDH problem is to derive the shared DH secret. Therefore it is used in modern key agreement schemes (Section 2.1.1).
If an adversary can break the CDH problem he can also break the Decisional Diffie-Hellman problem. The inversion of that argument does not hold.

Decisional Diffie-Hellman

The Decisional Diffie-Hellman (DDH) problem is the described as the problem that, given $A = g^a \; mod \; p$, $B = g^b \; mod \; p$ and $C = g^c \; mod \; p$ and a, b and c are randomly distributed in $\{0, \ldots, p-2\}$ or $c = a \cdot b \; mod \; (p-1)$, one has to decide if $g^c = g^{a \cdot b}$. The DDH problem is to recognize the DH shared secret.

2.3 Prerequisites

This section describes the prerequisites needed for the development, simulation, and emulation of an NFC-based algorithm on a secure element.

2.3.1 Secure Element

A secure element is a type of processor that has many hardware features to protect the system from unauthorized access. These features can be sensors that check for incoming light signaling that the casing is breached or if an attack including lasers is attempted. Another sensor can be sensory lines in a higher layer of the controller hardware that can signal if a probe tries to reach data lines running from the memory to the executing unit. Other sensors can detect if a probe has reached the data-, clock- or other lines. Furthermore, active parts can be that the bus that transports the data is encrypted or that execution units get their parameters in more steps. Another active element to protect the data on the bus is the injection of random bits on the bus. Furthermore, it might be the case that a parameter is transmitted and in another step a mask is transmitted that must be XOR-ed to the parameter before execution. Some secure elements also perform the execution twice or in parallel to detect if an execution unit was faulty or attacked. Upon detection of an attack measures to prevent the extraction of data are taken. These actions can range from simply shutting down, over erasing the whole memory, to setting up little explosive charges to destroy the memory for good.
Examples for secure elements are the SIM-card (Subscriber Identity Module[10]) of mobile phones or satellite receivers. Many of such secure elements also feature special execution units that are designed to perform cryptographic functions such as signing a value, performing a key exchange, or confirming a signature.

[10] `https://en.wikipedia.org/wiki/Subscriber_identity_module`

2.3.2 Field Programmable Gate Array

A Field Programmable Gate Array (FPGA) is a kind of processor that allows for very accurate measurements of algorithm performance. It is devised that it can emulate almost any kind of hardware. To do so hardware-components can be configured and connected freely. This programming is done by providing a so called "net-list" to the FPGA. With this file the logic blocks of the FPGA get reconnected and the blocks itself get configured. Most of the FPGAs have additionally one or more embedded microcontrollers and the related peripherals to be able to emulate a "system on chip". Some even have analog components integrated to emulate a mixed-signal system. Modern FPGAs also have the ability to be reprogrammed at runtime. This means that at the start of a emulation one part emulates a certain part of another processor and later in the same emulation the same part "impersonates" a different part of the emulated processor. This can be done in order to emulate different coprocessors. For example at first a processor for data analysis (a Digital Signal Processor) can be emulated and, if that part is not needed anymore, the same hardware can be used to emulate a special circuit that is needed for the application (Application Specific Integrated Circuit - ASIC).
FPGAs are not only used in development but also in consumer products with low market volume such as early products and niche products. This is because the per-unit cost of an ASIC is comparatively high for small production values. An example for such products are digital oscilloscopes.

2.3.3 NFC Technology

The Near Field Communication (NFC) technique gives a way to send data over short distances (up to 10 cm). NFC is based on a set of standards and specifications known as Radio Frequency IDentification (RFID) or ISO/IEC 14443 A and B. Like the RFID standard NFC operates in the industrial, scientific and medical (ISM) radio band of 13.56 MHz. This communication uses direct links without any networking mechanisms such as routing. The standards specify three different data rates of 106 kBit/s, 212 kBit/s and 424 kBit/s. Most of the devices deployed only support 106 kBit/s. A data exchange format for NFC (called NDEF - NFC Data Exchange Format) is used. These NDEF-messages can be composed of more NDEF-records which contain the data.
As NFC is based on RFID there are passive tags which are powered through the RF-field emitted by the reader.

The fact that an element of the communication can be powered with the EM-field is especially beneficial as, in our case, the security controller can be run with that energy. This is helpful as in some use cases the controller is mounted on a battery-driven mobile device. If then the device begins to move a voltage spike could disrupt the functionality of the controller. If the controller is powered via the EM-field a voltage spike is unlikely to happen.

2.3.4 Commonly Used Terms

Birthday attack: This kind of attack exploits mathematics from the birthday problem. This states that it is likely for a relatively small group of persons that two of them have their birthday on the same date. In other words it says that for splitting a group of inputs to a larger set of outputs it is likely that at least two of them fall into the same output category. This problem is related to the pigeonhole principle.

Dictionary attack: This attack is used to retrieve weak passwords. These are passwords that come from a relatively small set of possibilities - like a dictionary. There are also dictionaries of weak passwords available on the internet[11].

Eavesdropping: When eavesdropping one tries to get to secret information shared by others. During that the communication of the other parties is not disturbed.

Man-in-the-Middle: The Man-in-the-Middle (MitM) attack is a form of attack in which the adversary can read or alter the content of the messages. This is a entity that is in the middle of the communication between two parties. This entity is able to do anything he likes to the packets. One simple form is to alter or not deliver it.

Phishing: This term describes the attempt to get to personal data with the help of faked websites, email, or short messages.

Pigeonhole principle: This principle states that, if one wants to classify at least $n + 1$ items into n states, at least one state contains more than one item.

PLC: A Programmable Logic Controller is a device that is intended to control automated electromechanical processes. That means that it is connected to sensors and actuators to perform a controlling mechanism on industrial machinery.

Relay attack: A relay attack is a kind of Man-in-the-Middle attack. In this scenario the MitM does not necessarily alter or read the messages. This kind of attack can be used to respond like another person's identity card. The attacker initiates the communication, relays the messages to the other person's card and relays the response back to the system.

[11] `http://weakpass.com/` - accessed on 27.8.2016

Skimming: Skimming is another form of a MitM attack. Here the attacker listens to the targets communication, records enough data to pretend to be this person. This can also be seen as illegal copy of an electronic document.

Worm: A worm in the computer is a malware program that stands alone, unlike a virus. This malware can do significant harm to all digital systems.

3 Design

Within this chapter an analysis of the project yields us a rough design. The further analysis of this design gives specific requirements for the used protocol. Building on top of these requirements a demonstrator is designed that contains the fundamental functionalities to get to the vision of the motivation (Section 1.2). This chapter furthermore describes the design of the underlying protocol for authenticated key exchange. After the detailed description of the protocol the prototypical designs of the implementation are explained. This chapter concludes with the design of the evaluation phase.

3.1 Application Scenarios

This section is dedicated to describe the scenarios that are mentioned in the Motivation (Section 1.2). The configuration and authentication of users is a vital part in these schemes.

3.1.1 Future Industry

The "*Industrial Internet Consortium*"[12] states that:

> The Industrial Internet will dramatically improve productivity and efficiencies in the production process and throughout the supply chain.

[12] `http://www.iiconsortium.org/` - accessed on 13.9.2016

> Processes will govern themselves, with intelligent machines and devices that can take corrective action to avoid unscheduled breakdowns of machinery. Individual parts will be automatically replenished based on real time data.

They say that new instruments will interlink millions of things and that "*customization [of products and production processes] will be automatic*". In state of the art industries the production flow is basically static as illustrated in Figure 3.1. The raw material is fetched from a warehouse or some other form of storage. It is then placed in the input of the production street. There, highly specialized machinery treats the incoming items and forms new components, which in turn should be the input to another process. To connect the apparatuses conveyor belts are used that are capable of transporting a huge amount of products very efficiently. At the end of this course the produced goods are packaged and transported to another storage place, ready for shipment.

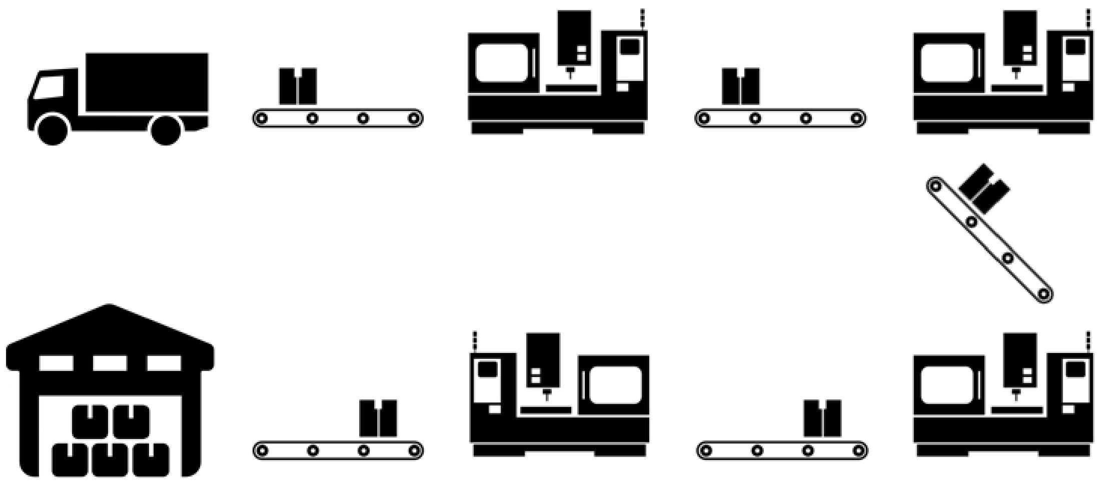


Figure 3.1: A static production line. Goods are transported on a certain path between machines.

With the increasing power of digital information processing and the development of better communication possibilities, it is only natural that also the production machinery gets digitalized. More sensors on the appliances can give better insight on the process, thereby provide information for analysts that can work up even better ways of producing the goods. In recent years the consumer demand for more individualized products rose to a new level. This trend requires the company to get more flexible in producing new components. That implies on the other hand that the efficiency of the machinery decreases as lower production volume generates less data for analysis. Even more noticeable is the fact that for reconfiguration some machines need to be stopped, the flow of items need to be rerouted or postponed, and the machine itself needs to be reconfigured by a trusted engineer. This reconfiguration does not even imply that, for some products it may be necessary to skip some machines, or to change their position in the street. This is illustrated in Figure 3.2

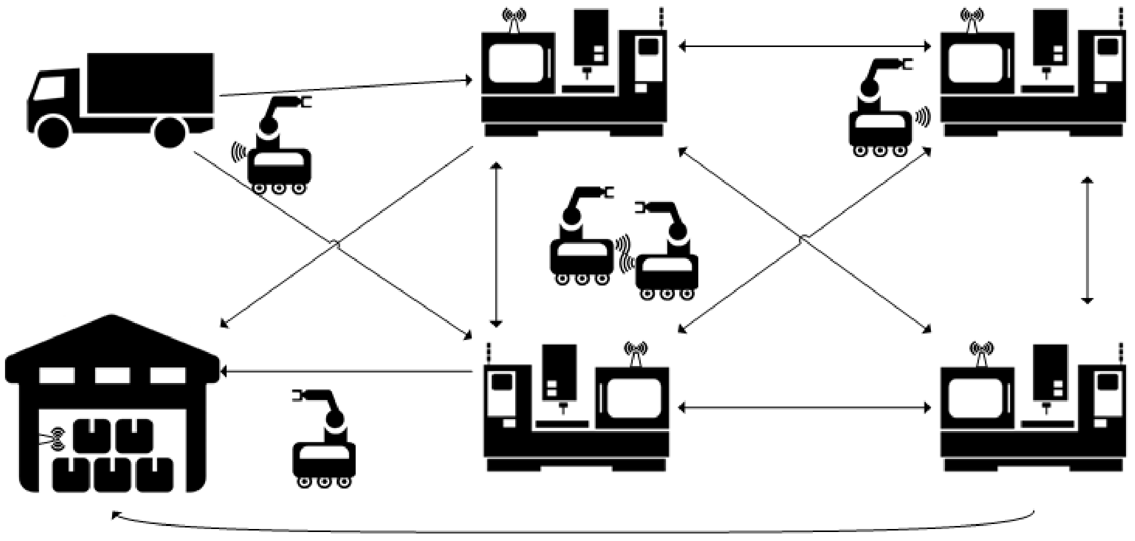


Figure 3.2: New production process with "Industrie 4.0" in mind. Everything can communicate with everyone and goods can be transported in any way between the machines.

With the usage of new technology this problem of reconfiguring machines and directing the flow of goods between the appliances can be solved. Therefore the apparatuses need to be equipped with means to communicate with each other. "*Industrie 4.0*" is the key under which these changes are described. In the near future the customers want to have even more possibilities to individualize their products. To comply with these wishes, a better way to transport goods between machines and to configure those machines needs to be developed. We envisioned a facility that uses a completely automatized production floor. Robots carry any item between the machines to provide a maximum of flexibility for the workflow. The robots can furthermore configure the machines that process the items. Every machine senses the operating parameters and can send this data to a central server that calculates even better configurations for the next time this setting is chosen. The communication between the machines and robots should be easily usable, stable, secure, and not interfere with the communication of other machine-robot pairs. That implies a short-ranged, wireless transmission technology. One of the best solutions to this is the usage of Near Field Communication (NFC).

An advantage that comes with the use of NFC is that also human engineers and service technicians can use this technology to operate the service interface of the appliances. This means that the worker can use his NFC-enabled smartphone to configure any machine in the facility. This furthermore improves the situation in the way that, for machines that need an engineer in order to be allowed to start, the technician needs to be present to give the command to start up.

On the down side of this arrangement is that any person owning a NFC-enabled phone could read configuration data and, in the worst case, also sabotage the whole facility. To counter that encryption and authentication processes need to be established. With that only authorized personnel is able to decrypt the received data.

3.1.2 Smart Home

Nowadays many new homes are becoming so called “Smart Homes”. That idiom describes a house where almost anything can be controlled by a smartphone or tablet computer. The “*COYERO*” project [13] of *CISC Semiconductor* operates in this field and aims to connecting people with local services, the car-charging mechanism and their homes. The owner of such a “smart home” can for example switch on the lights in every room, operate the door locks, change the state of the security measures, set the heating or air conditioning, open the garage door, and much more. It is easy to see what benefits can come with that if everything works properly. In the case that something is misconfigured it can also make life more difficult, uncomfortable, or even dangerous. With all appliances in the home connected to the Internet, it is easy to imagine someone hacking into the system and generating havoc. This attacker can set the heating and air conditioning to full load, switch the lights randomly, lock all doors, signal the police that an intruder has broken into the home, and blocking the real owner from getting access to all functions. Thinking of such an attack it is just clear that all connected items must be secured and that only authenticated persons get access to the settings. Another possibility to ease the threat of an attacker is the use of NFC as a second way to configure the appliances. With NFC being a short ranged communication medium it is assured that the person attempting to change settings is directly in front of the device. Therefore the settings made with this method are higher in priority than other settings. The only thing needed to get everything back to order is a NFC-enabled phone. If the configuration is already stored on the phone (for example the devices were initially configured with the phone) the user just needs to touch the devices and insert the password to authenticate himself. After that the connection between the device and the phone is secured and the configuration can be transmitted. To get an even better protection a security controller can be used as a trusted anchor to secure all data. This controller can also be directly powered with the field of the NFC-communication. That ensures that the high priority configuration data can

[13] `http://www.coyero.net/` - accessed on 13.9.2016

always be received.

This method of configuring devices is also favorable for “normal” homes. As an example one can think of configuring a WiFi-router to allow a new device in the wireless network. For some models configuring can only be done via cable. This method is highly unpractical as it is likely that guests may want to use the network and therefore it needs to be reconfigured every so often. With the configuration done over NFC this can be done by adjusting the saved configuration on the phone, authenticating the user with the administrator password, and touching the router with the cell phone.

3.1.3 Next Generation Research

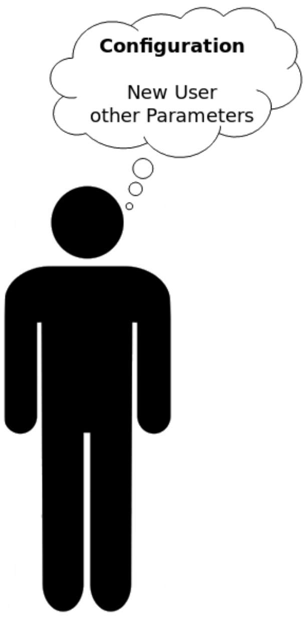
Wireless sensor networks are widely used in research. These networks consist of sensor nodes that are connected with each other. These devices usually consist of a microcontroller, sensors, an antenna, and a battery or a mechanism for energy harvesting. They furthermore have the requirement to last for months without maintenance. In research the sensor nodes get reprogrammed very often to support different studies. Therefore, they need to be collected, connected to the computer, the software replaced and distributed again. If the nodes support the wireless transmission of a new program only one node needs to be collected. This results in a serious vulnerability. If a node gets reprogrammed by an attacker and can transmit this “update” to the other nodes the adversary can take the whole system. To counter that we have envisioned a possibility that the nodes can be equipped with a security controller. This controller is usually powered down in order to not use any energy. If a software update has to be made, it can be transmitted with NFC technology. Thereby the secure controller can be powered with the HF-field and perform the energy consuming calculations without drawing any of the needed energy from the other controller’s resource. As the user gets authenticated in the update-process, a malicious attacker cannot overwrite the program. When the reprogramming is done the security controller can again be powered down to save energy.
After that the update can be transmitted within the network to update all nodes. To secure also this part of the update process, the secure controller only needs to verify the received data. This can be done by calculating the hash of the received program and comparing it with a signature generated by the first controller that got the update from the user.

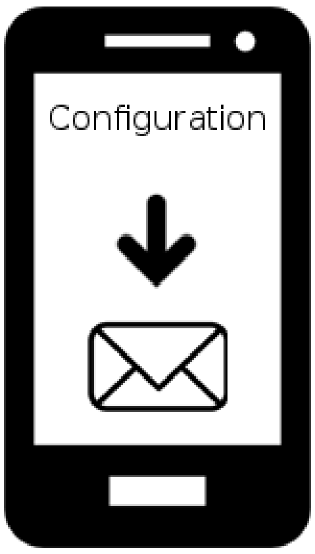
3.2 Design Analysis

Within the IoSense project smart sensors are used as a central topic. This is a very vague definition as basically everything can be seen as a sensor. Some are just that, sensors, other can be as complex as smartphones or robots. This thesis focuses on the configuration of such sensors.

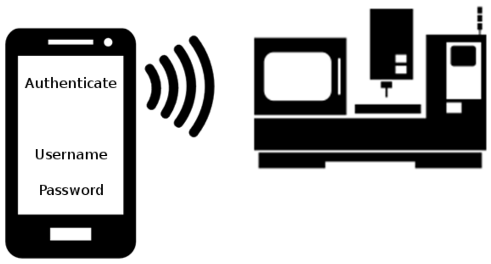
To see the wider picture we thought of a newly designed smart factory as already described in Section 1.2. The machines are able to communicate with each other, the transport of items between machines is done with robots, everything can be controlled by configuring the machines and robots. Furthermore, we envisioned that with all the gathered data, optimization techniques coming from computational intelligence can improve the efficiency quickly. In such a factory different end products can be designed by reconfiguring some machines and rerouting the item-flow.
Then we thought of how the configuration of robots and machines can be done. Analyzing that situation and adding knowledge we gathered from the RoboCup Logistics League we came to a conclusion that reliable configuration of machines and robots should be done with the Near Field Communication (NFC) technology. This is because the communication with moving machines is better with a contactless channel, but WiFi has a very big radius and will be used by the machines to communicate with the central computer. When the configuration data is also transferred via this channel more latency is introduced, furthermore many machines are affected if only one machine is to be configured. This gets even worse if the machines are configured by the robots delivering and gathering items. The best solution is therefore a stable, wireless communication technique with a short range or that is very directed. With that in mind NFC is the logical conclusion.
Having that, it is just reasonable that a human operator or service technician will configure the machines or robots. Therefore he only needs a NFC-enabled smartphone. With that a user can set up the configuration data on the phone (e.g. when a machine fails and it must not be switched on until a technician inspected it) or get the configuration from a central place - maybe the server running the optimization algorithms. This leaves us with an (potentially untrusted) human in charge to configure a production facility.

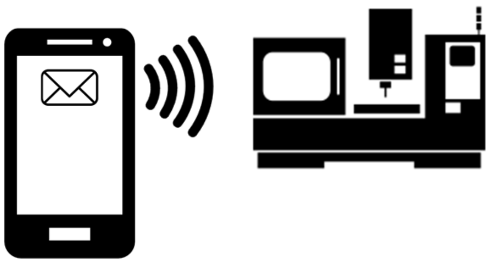
To face this problem we decided that a secured and authenticated channel must be used if the technician configures the machine. As described earlier (Section 2.1.2) passwords are a good start. With the use of the password an authentication is done, and with a key exchange the communication is secured. As the trust on the machines comes from a security controller - usually they have very limited resources in terms of memory and processing speed - we wanted an efficient protocol. This protocol should handle both, authentication and security, in one step with one exchange of messages. A possible process of configuring a machine is depicted in Figure 3.3.





(a) The user thinks of a new configuration (b) The configuration gets packaged





(c) The user has to authenticate himself against the machine (d) The configuration gets sent to the machine

Figure 3.3: Information flow from the idea of a configuration to the machine

3.3 Demonstrator

The demonstrator application will be a form of authenticated Echo-Service. This will be demonstrated by simulating an Infineon-type security controller running a SPAKE-implementation as a machine, and by connecting to this microcontroller with the computer. Figure 3.4 shows how the commands and requests are sent from the user to the machine and how the data is sent back. This process is invoked for every communication and is omitted in the rest of the diagrams.

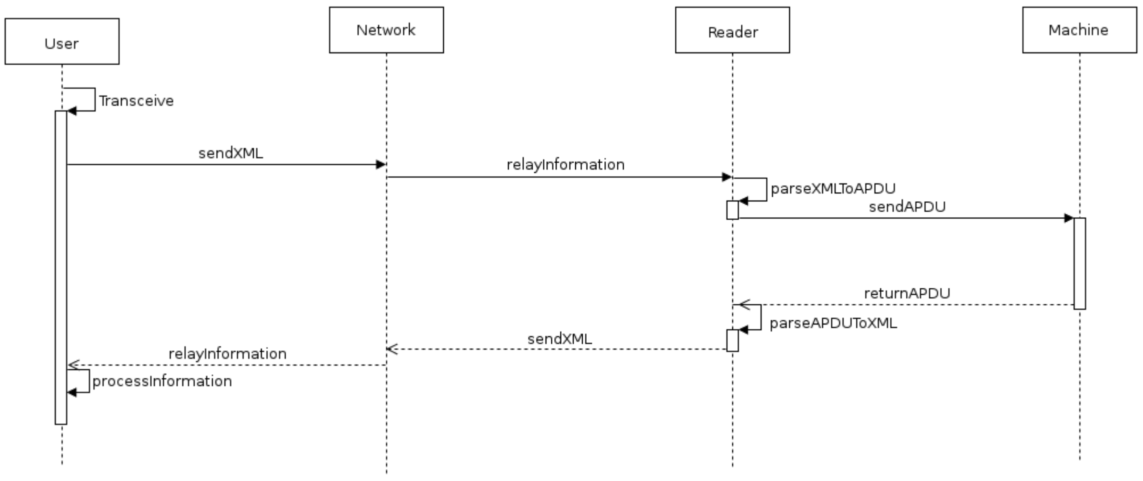


Figure 3.4: Design of the communication between user and machine.

The use cases depicted in Figure 3.5 show the critical tasks to be performed by the user and the machine. With these a communication between the two parties can be established and the machine can be configured. In this scenario the user wants to configure the machine to be able to communicate with another user. In Figure 3.6 the calculation of the masked public keys is depicted and in Figure 3.7 the mask is subtracted and the second part of the Diffie-Hellman key exchange is performed.
After the calculation of the masked public keys these are exchanged. This is done by the user sending his masked key X_s which invokes the storage of this key in the machine. After that, the user requests the remote masked public key which is answered by the machine with Y_s as payload.
The combination of the described use-cases enables the password-based authentication and key exchange. This is performed to connect to the machine and establish a trusted connection. This process is shown in Figure 3.8.
After this the secured connection is established and critical operations can be performed. Within the context of SPAKE a critical operation is the configuration of users on the machine. This must only be done by trusted users.
For the demonstrator another ”critical” operation is introduced. This is a command that takes user-payload and also returns payload. For simplicity reasons this

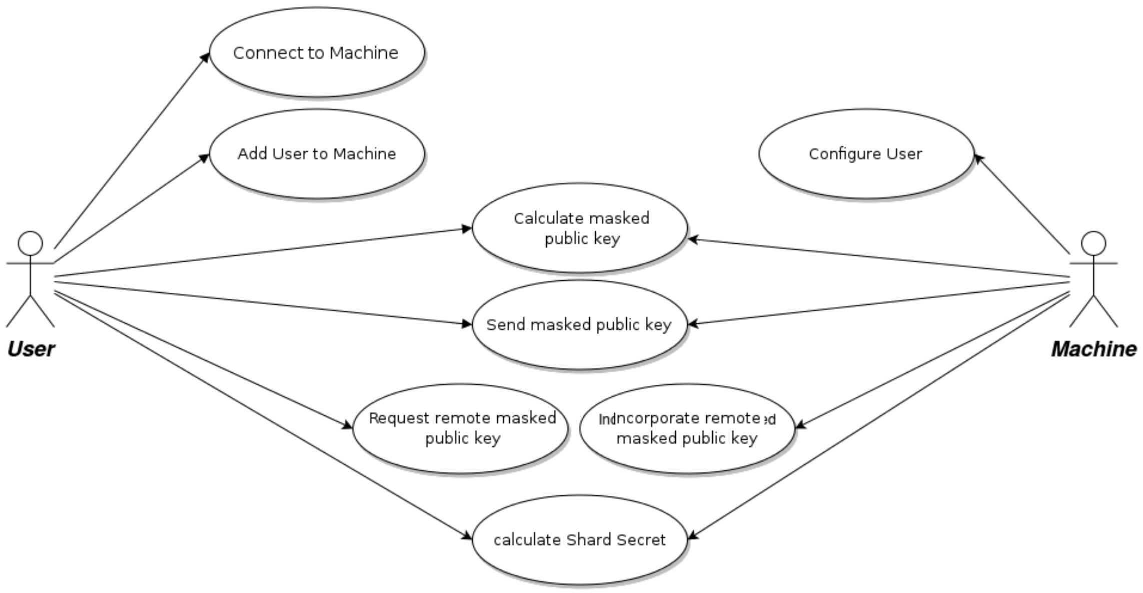


Figure 3.5: Use Cases of a communication between user and machine.

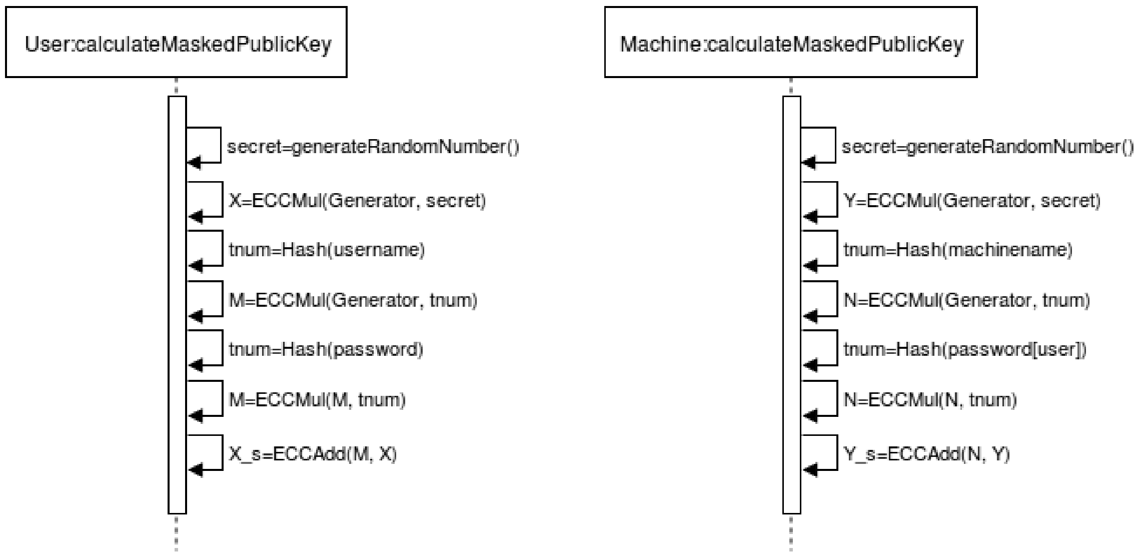

User:calculateMaskedPublicKey
secret=generateRandomNumber()
X=ECCMul(Generator, secret)
tnum=Hash(username)
M=ECCMul(Generator, tnum)
tnum=Hash(password)
M=ECCMul(M, tnum)
X_s=ECCAdd(M, X)
Machine:calculateMaskedPublicKey
secret=generateRandomNumber()
Y=ECCMul(Generator, secret)
tnum=Hash(machinename)
N=ECCMul(Generator, tnum)
tnum=Hash(password[user])
N=ECCMul(N, tnum)
Y_s=ECCAdd(N, Y)

Figure 3.6: Calculation of the masked public key of user and machine.

is an echo-service that can only be called if the user is authenticated otherwise the echo-request will fail. After the user has performed the needed operations he should disconnect from the controller.

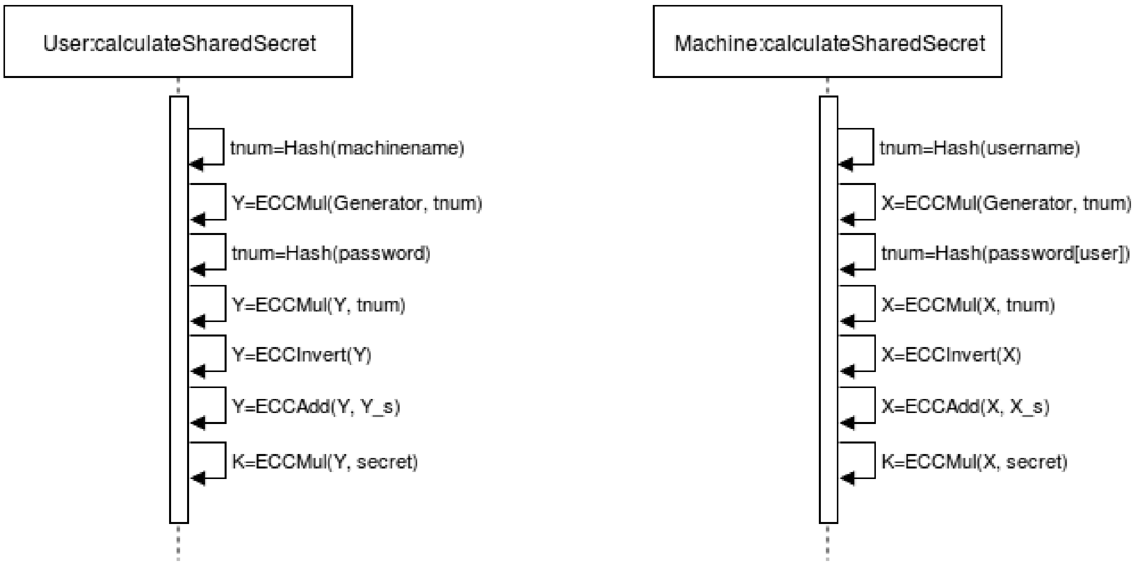


Figure 3.7: Calculation of the shared secret between user and machine.

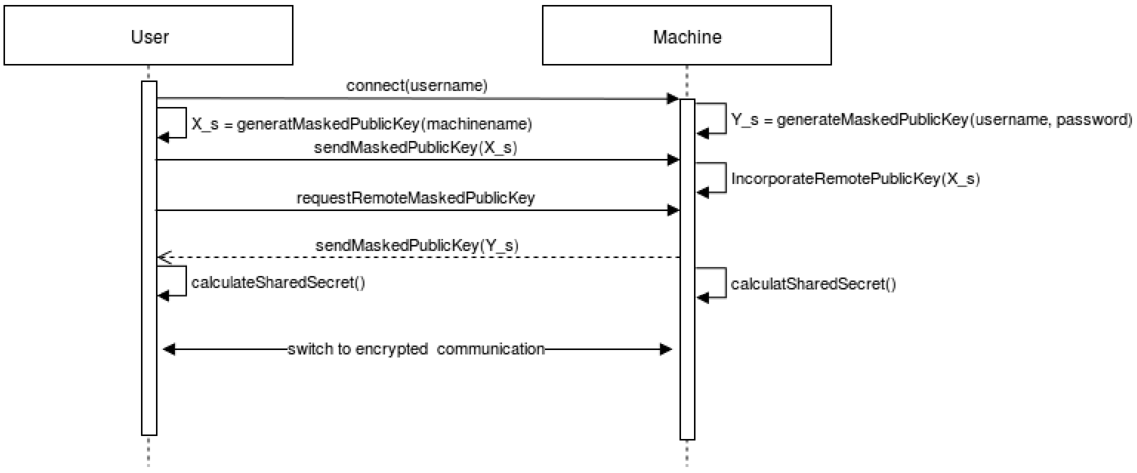

User
Machine
connect(username)
X_s = generatMaskedPublicKey(machinename)
Y_s = generateMaskedPublicKey(username, password)
sendMaskedPublicKey(X_s)
IncorporateRemotePublicKey(X_s)
requestRemoteMaskedPublicKey
sendMaskedPublicKey(Y_s)
calculateSharedSecret()
calculatSharedSecret()
switch to encrypted communication

Figure 3.8: Establishment of a secured connection between user and machine.

Initial Configuration

To be able to connect to the secure element at startup, a default configuration is needed. With this information a first key establishment can be performed. After that the name of the machine can be changed and another user can be defined. With this the authentication at the first login is disabled as this information is publicly available. This however does not disable the normal key exchange. For this an algorithm such as ECDH (Elliptic Curve variant of the Diffie-Hellman key exchange Section 2.2.6) can be chosen.

3.4 Authentication

The selection of the communicating parties is done by using the names of the user and the remote party - in the used case a machine. To generate a key a shared secret (a password) must be known to both parties. This can be hardcoded in the machine or can be transmitted to it via a secured channel or another transport vector like VLC (Visible Light Communication). On the other side, the user-side, the password must be entered by the engineer. Using this technique only users which have access to the password can connect to the machine. The authentication process does use the password in the key exchange process to calculate a mask for the ECDH (Elliptic Curve Diffie-Hellman)-algorithm. The final key generation uses the usernames of both parties, the transmitted masked points, the password, and a generated pre-shared secret to generate the secure session key. If the password of the user and the password stored in the machine does not match, the key will be different and no communication is possible. The authentication of the user against the machine is done by providing the right password in the process. If the password stored in the machine and the entered do not match the resulting key is different and the encrypted communication cannot proceed. The authentication of the machine against the user can be obtained if for every machine a new password is used. Another method of mutual authentication is the use of a second shared secret used for one mask.

3.5 Key Transfer

The used algorithm to transfer the key material is the SPAKE2-algorithm proposed in [AP05]. This method allows for exchanging a strong shared key based on a weak shared secret e.g.: a password. This is done with a form of ECDH (Elliptic Curve Diffie-Hellman) algorithm as introduced in Section 2.1.1. In this version the transmitted key-point is masked with another point derived from the username of the transmitting party and the shared secret. The algorithm works as follows:

- Given are an elliptic curve and the according parameters (a, b, p, G) and an hash algorithm H, as well as the names of the connecting parties.
- User A selects a random number x and calculates the point $X = G \cdot x$.
  User B generates another Point Y in the same manner with the random number y.
- User A calculates the mask with an given point M that is associated with user A and the password. $X_{mask} = M \cdot password$.

User B calculates the mask Y_{mask} with the given point N that is associated with user B and the password in the same manner.

- User A generates the masked point $X^* = X + X_{mask}$.
  User B generates the masked point $Y^* = Y + Y_{mask}$.
- User A sends the masked point X^* and receives the masked point Y^* generated by user B.
- User A calculates the pre-shared secret K_A by subtracting the mask of user B and multiplying the result with x. $K_A = (Y^* - N \cdot password) \cdot x$. The subtraction is achieved by calculating the inverse element modulo p and adding it.
  User B generates K_B in the same way. In an honest execution K_A equals K_B.
- To generate the shared keys SK_A and SK_B a hash is calculated with the names of A and B, the masked points X^* and Y^*, the password, and the calculated pre-shared secrets K_A or K_B respectively.

The operations $+$, $\cdot$ and $Inv(\dots)$ are used according the rules for Elliptic curves. For better visualization of the algorithm see Table 3.1.

public Information: a, b, G, M, N, H
private shared Information: $password$

User A		User B
$x = rand()$		$y = rand()$
$X = G \cdot x$		$Y = G \cdot y$
$X^* = X + M \cdot password$		$Y^* = Y + N \cdot password$
	$\overset{X^*}{\rightarrow}$	
	$\overset{Y^*}{\leftarrow}$	
$K_A = (Y^* + Inv(N \cdot password)) \cdot x$		$K_B = (X^* + Inv(M \cdot password)) \cdot y$
$SK_A = H(A, B, X^*, Y^*, password, K_A)$		$SK_B = H(A, B, X^*, Y^*, password, K_B)$

Table 3.1: Visualization of the SPAKE2 Algorithm using Elliptic curves.

As described in [AP05] the calculated keys SK_A and SK_B are the same, as, in an honest execution of the protocol, the points K_A and K_B are equal. The points K_A and K_B are equal to $G \cdot x \cdot y$.
The public information in this algorithm is the information about the used curve (the parameters and the generator point), the used key-derivation-function H (in this implementation a hash-function), and the Points associated with the user (M) and the machine (N).

When looking at the algorithm one can see that the basic Diffie-Helman key exchange is extended with the addition of the mask before sending the public key ($X^* = X + M \cdot password$), and the removal of the mask as first step after receiving the remote public key ($\cdots + Inv(N \cdot password) \ldots$).

Reference Implementation

When looking at the implementation found at [14] one can see that the points M and N are chosen statically, the curve cannot be selected, the password cannot be changed, and the user and the machine only support one connection. In a real-world environment a single user may want to connect to multiple devices which in turn gets used by multiple users. Furthermore, a user may be required to change the password and the used cryptographic curve as sometimes a curve is shown not to be secure for certain applications. Furthermore, this library uses the generated key as an ephemeral key which means that the process of connecting needs to be done again when the session expires. One could use the generated key as a shared secret which is used to generate the ephemeral session keys. As a final point the authors of the code claim to have implemented SPAKE2 from [AP05]. This is not the case as the authors of the code do not even use the hash-function to generate the key (SPAKE1). SPAKE2 also uses the password in the hash function to generate a concurrent version of SPAKE.
In the implementation these features will also be found. Because of this we cannot test the implemented algorithm against the reference.

3.6 Different Designs

In the many cycles of the Softwaredevelopment many different designs of the SPAKE-algorithm have been developed. The class-diagram for the conceptual evaluation for the SPAKE protocol is depicted in Figure 3.10. This implementation enables the user to test the key exchange rigorously. It consists of users and machines which can have an arbitrary number of connections to the other class. With the use of the OpenSSL-EC, -SHA, and -Rand libraries the crucial parts of the SPAKE protocol are built.
The second development-iteration uses the class-diagram shown in Figure 3.11. In this iteration the developed software should be easily portable to the security controller of Infineon. As this controller has a hardware-acceleration for Elliptic Curve Cryptography (ECC) special data-types, defined in Figure 3.9, must be used. Furthermore, classes are not supported, therefore the SPAKE-implementation has all necessary functions in one file. The helper-functions and the implementation of the

[14] `https://weave.googlesource.com/weave/libuweave/+/HEAD` - accessed on 17.06.2016

interface, which should behave like the real crypto-processor, use the functionalities of the OpenSSL. As the secure memory is limited in size memory-optimizations have to be made. Therefore only two connections per user or machine are possible.

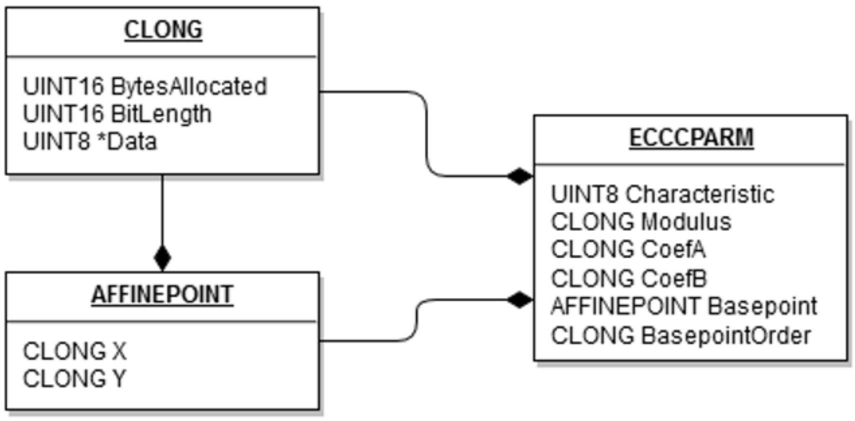


Figure 3.9: Datastructure for storing long integers and the ECC related data.

The next development step is the implementation with the tool that yields the micro-controller code. In this implementation the interface functionalities are performed by the processor-specific asymmetrical crypto-lib. Furthermore the hashing and random-number-generation is done by the processor. As this implementation-phase yields code that can be executed on the hardware-controller (or the emulator), many more optimizations are needed. This includes the minimization of memory usage as well as the reduction of additional features. As the implementation should reflect the communication on the machine-side, the userside-related functions are omitted. The resulting diagram is shown in Figure 3.12. Aside from this SPAKE-implementation additional functionalities need to be implemented to be able to communicate with the user. To do this Application Protocol Data Units (APDUs) are defined. The instructions for the communications are:

setMachine: With this command the machinename can be changed - if the user is already authenticated and the connection is encrypted.

setUser: This command can configure the currently not used user. So if user 1 calls this command user 0 is configured. The configuration-data is in the payload of the message. Also this command can only be called if the calling user is already authenticated and the connection is encrypted.

setUserMessage: This command is called to start an authenticated session. Within the payload of the command the masked public key of the user is transmitted.

getMachineMessage: This is the request command for the masked public key of the machine. In the answer of the transceive-operation the requested data is transmitted.

After all steps are taken the communication can be switched to be encrypted. The cryptographic key can be derived from the exchanged credentials, the connecting user and the password of the user. If any of these parameters is wrong the key is different and the communication cannot be done encrypted. If this happens the authenticated key-exchange needs to be performed again.

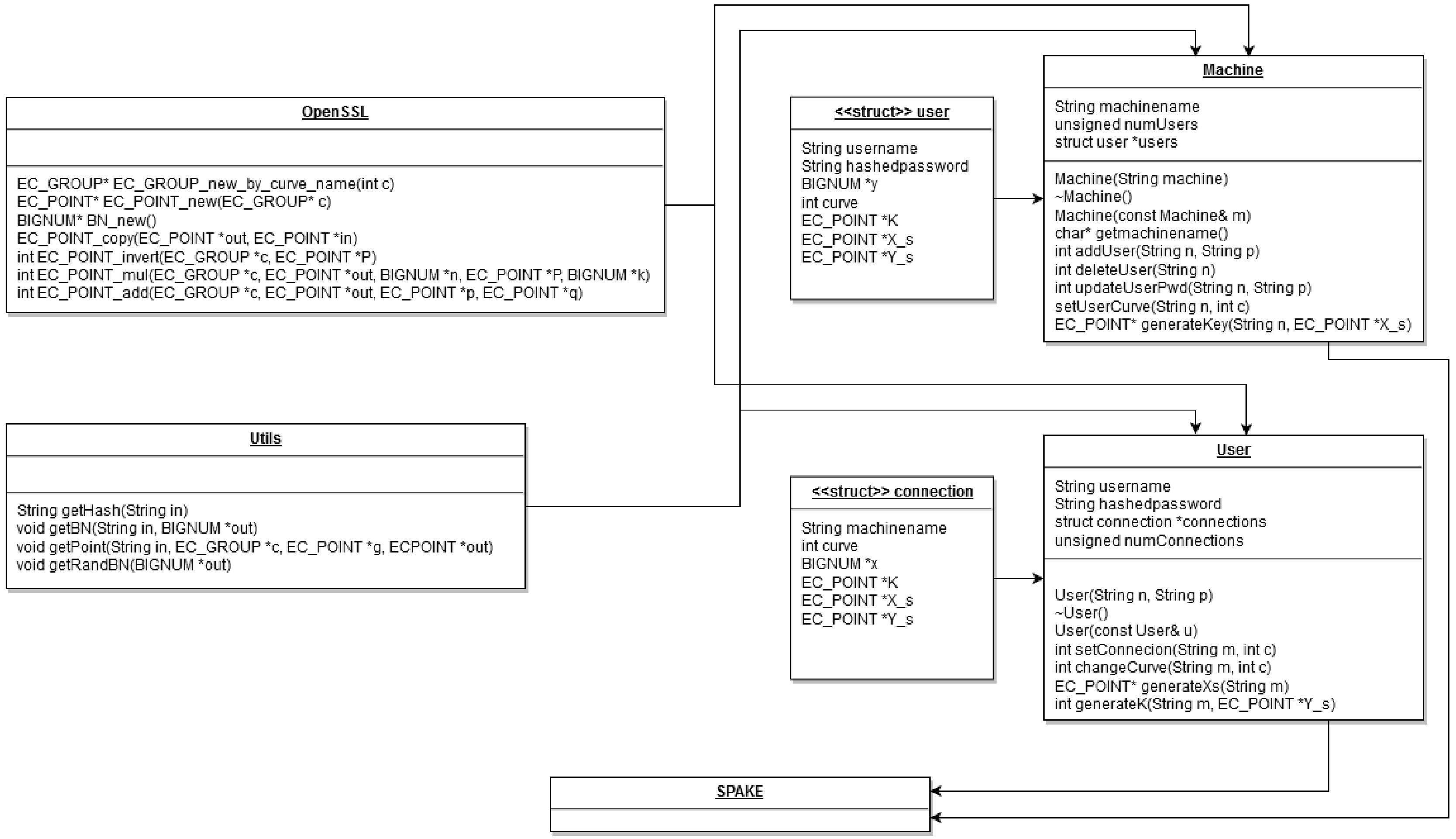

OpenSSL
EC_GROUP* EC_GROUP_new_by_curve_name(int c)
EC_POINT* EC_POINT_new(EC_GROUP* c)
BIGNUM* BN_new()
EC_POINT_copy(EC_POINT *out, EC_POINT *in)
int EC_POINT_invert(EC_GROUP *c, EC_POINT *P)
int EC_POINT_mul(EC_GROUP *c, EC_POINT *out, BIGNUM *n, EC_POINT *P, BIGNUM *k)
int EC_POINT_add(EC_GROUP *c, EC_POINT *out, EC_POINT *p, EC_POINT *q)
Utils
String getHash(String in)
void getBN(String in, BIGNUM *out)
void getPoint(String in, EC_GROUP *c, EC_POINT *g, ECPOINT *out)
void getRandBN(BIGNUM *out)
<<struct>> user
String username
String hashedpassword
BIGNUM *y
int curve
EC_POINT *K
EC_POINT *X_s
EC_POINT *Y_s
Machine
String machinename
unsigned numUsers
struct user *users
Machine(String machine)
~Machine()
Machine(const Machine& m)
char* getmachinename()
int addUser(String n, String p)
int deleteUser(String n)
int updateUserPwd(String n, String p)
setUserCurve(String n, int c)
EC_POINT* generateKey(String n, EC_POINT *X_s)
<<struct>> connection
String machinename
int curve
BIGNUM *x
EC_POINT *K
EC_POINT *X_s
EC_POINT *Y_s
User
String username
String hashedpassword
struct connection *connections
unsigned numConnections
User(String n, String p)
~User()
User(const User& u)
int setConnecion(String m, int c)
int changeCurve(String m, int c)
EC_POINT* generateXs(String m)
int generateK(String m, EC_POINT *Y_s)
SPAKE

Figure 3.10: Software design for the conceptual evaluation of the SPAKE protocol.

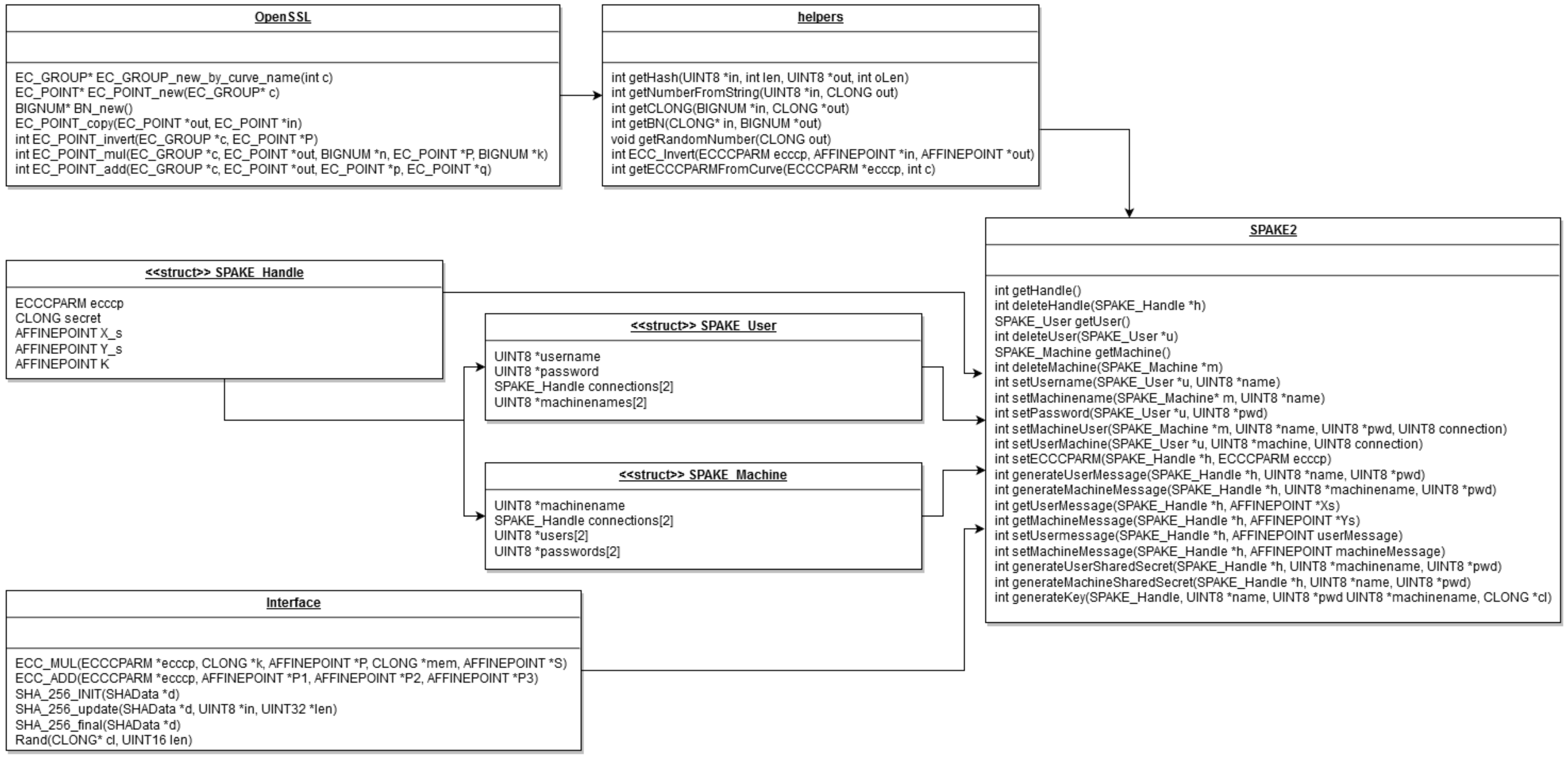

OpenSSL
EC_GROUP* EC_GROUP_new_by_curve_name(int c)
EC_POINT* EC_POINT_new(EC_GROUP* c)
BIGNUM* BN_new()
EC_POINT_copy(EC_POINT *out, EC_POINT *in)
int EC_POINT_invert(EC_GROUP *c, EC_POINT *P)
int EC_POINT_mul(EC_GROUP *c, EC_POINT *out, BIGNUM *n, EC_POINT *P, BIGNUM *k)
int EC_POINT_add(EC_GROUP *c, EC_POINT *out, EC_POINT *p, EC_POINT *q)
helpers
int getHash(UINT8 *in, int len, UINT8 *out, int oLen)
int getNumberFromString(UINT8 *in, CLONG out)
int getCLONG(BIGNUM *in, CLONG *out)
int getBN(CLONG* in, BIGNUM *out)
void getRandomNumber(CLONG out)
int ECC_Invert(ECCCPARM ecccp, AFFINEPOINT *in, AFFINEPOINT *out)
int getECCCPARMFromCurve(ECCCPARM *ecccp, int c)
SPAKE2
int getHandle()
int deleteHandle(SPAKE_Handle *h)
SPAKE_User getUser()
int deleteUser(SPAKE_User *u)
SPAKE_Machine getMachine()
int deleteMachine(SPAKE_Machine *m)
int setUsername(SPAKE_User *u, UINT8 *name)
int setMachinename(SPAKE_Machine* m, UINT8 *name)
int setPassword(SPAKE_User *u, UINT8 *pwd)
int setMachineUser(SPAKE_Machine *m, UINT8 *name, UINT8 *pwd, UINT8 connection)
int setUserMachine(SPAKE_User *u, UINT8 *machine, UINT8 connection)
int setECCCPARM(SPAKE_Handle *h, ECCCPARM ecccp)
int generateUserMessage(SPAKE_Handle *h, UINT8 *name, UINT8 *pwd)
int generateMachineMessage(SPAKE_Handle *h, UINT8 *machinename, UINT8 *pwd)
int getUserMessage(SPAKE_Handle *h, AFFINEPOINT *Xs)
int getMachineMessage(SPAKE_Handle *h, AFFINEPOINT *Ys)
int setUsermessage(SPAKE_Handle *h, AFFINEPOINT userMessage)
int setMachineMessage(SPAKE_Handle *h, AFFINEPOINT machineMessage)
int generateUserSharedSecret(SPAKE_Handle *h, UINT8 *machinename, UINT8 *pwd)
int generateMachineSharedSecret(SPAKE_Handle *h, UINT8 *name, UINT8 *pwd)
int generateKey(SPAKE_Handle, UINT8 *name, UINT8 *pwd UINT8 *machinename, CLONG *cl)
<<struct>> SPAKE_Handle
ECCCPARM ecccp
CLONG secret
AFFINEPOINT X_s
AFFINEPOINT Y_s
AFFINEPOINT K
<<struct>> SPAKE_User
UINT8 *username
UINT8 *password
SPAKE_Handle connections[2]
UINT8 *machinenames[2]
<<struct>> SPAKE_Machine
UINT8 *machinename
SPAKE_Handle connections[2]
UINT8 *users[2]
UINT8 *passwords[2]
Interface
ECC_MUL(ECCCPARM *ecccp, CLONG *k, AFFINEPOINT *P, CLONG *mem, AFFINEPOINT *S)
ECC_ADD(ECCCPARM *ecccp, AFFINEPOINT *P1, AFFINEPOINT *P2, AFFINEPOINT *P3)
SHA_256_INIT(SHAData *d)
SHA_256_update(SHAData *d, UINT8 *in, UINT32 *len)
SHA_256_final(SHAData *d)
Rand(CLONG* cl, UINT16 len)

Figure 3.11: Software design when implementing with the processor interface in mind.

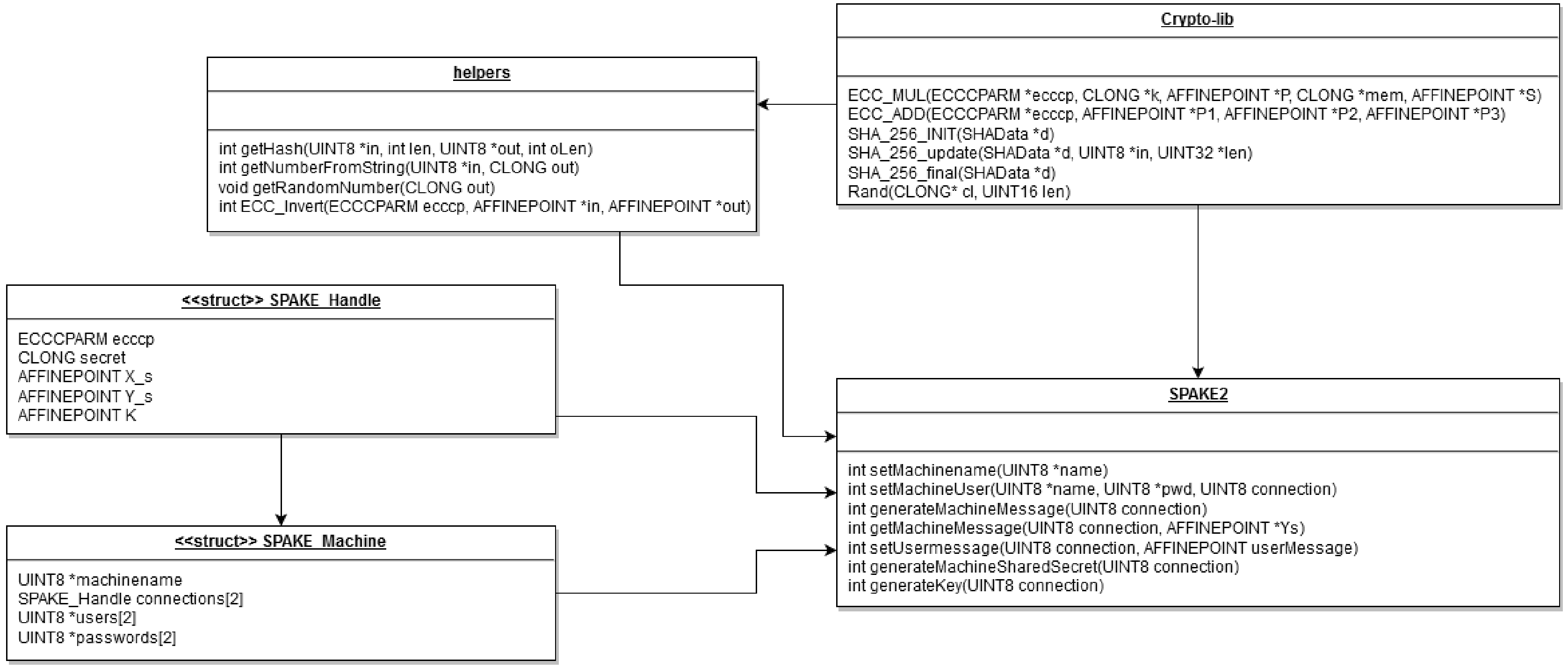

Crypto-lib
ECC_MUL(ECCCPARM *ecccp, CLONG *k, AFFINEPOINT *P, CLONG *mem, AFFINEPOINT *S)
ECC_ADD(ECCCPARM *ecccp, AFFINEPOINT *P1, AFFINEPOINT *P2, AFFINEPOINT *P3)
SHA_256_INIT(SHAData *d)
SHA_256_update(SHAData *d, UINT8 *in, UINT32 *len)
SHA_256_final(SHAData *d)
Rand(CLONG* cl, UINT16 len)
helpers
int getHash(UINT8 *in, int len, UINT8 *out, int oLen)
int getNumberFromString(UINT8 *in, CLONG out)
void getRandomNumber(CLONG out)
int ECC_Invert(ECCCPARM ecccp, AFFINEPOINT *in, AFFINEPOINT *out)
<<struct>> SPAKE_Handle
ECCCPARM ecccp
CLONG secret
AFFINEPOINT X_s
AFFINEPOINT Y_s
AFFINEPOINT K
SPAKE2
int setMachinename(UINT8 *name)
int setMachineUser(UINT8 *name, UINT8 *pwd, UINT8 connection)
int generateMachineMessage(UINT8 connection)
int getMachineMessage(UINT8 connection, AFFINEPOINT *Ys)
int setUsermessage(UINT8 connection, AFFINEPOINT userMessage)
int generateMachineSharedSecret(UINT8 connection)
int generateKey(UINT8 connection)
<<struct>> SPAKE_Machine
UINT8 *machinename
SPAKE_Handle connections[2]
UINT8 *users[2]
UINT8 *passwords[2]

Figure 3.12: Part of the design of the SPAKE algorithm when implementing for the microcontroller. Here only the part for the machine is implemented.

3.7 Evaluation

The evaluation of the implementation on the secure element is performed by testing it against the prototypical implementations. The test against the reference implementation of Google cannot be performed as described in Section 3.5. As a next step the correct functionality can be tested with defined test cases that include the correct usage of a user and the transmission of corrupted or wrong messages. This tests how the cryptocontroller handles corrupted data.

As another metric of the implementation, the time of the calculation is measured. This will be done in different ways. Firstly, the time from sending the data to receiving the result is measured. Subtracting the time used for sending and receiving an echo command gives the time the simulator spends in the calculation of the “useful” part of the application.
Multiple measurements have to be taken to get accurate values. This is because the laptop-computer, on which the simulation is performed on, will likely do other tasks in the background and does not react as soon as the message arrives. To keep the background-load of the computer low only the necessary programs should be started.

A more accurate measurement can be done by tracking the CPU cycles the simulated processor takes. With this information and the knowledge about the CPU-speed a better timing profile for the heavyweight functions can be calculated.

To get an even better result of the CPU-cycles an emulator with a FPGA (Field Programmable Gate Array) can be used. Figure 3.13 shows the setup for the cycle-accurate emulation. The processor-netlist gives the hardware-configuration of the security controller to the FPGA. After the hardware is set up the code for the machine application is loaded. The FPGA has an attached analog front end for the interfaces (HF and ISO). The HF interface gets the direct input from the EM field of the card reader. The ISO interface has the pins of the card connected to it. As the chosen Use Case uses the NFC technology the HF interface is connected. As a card-interface the REFPICC module connects converts the HF signal to an EM field. Over this field the data is exchanged with the ISO14443 protocol. The other communication partner is a card reader with a special firmware that allows for sending any data. With the use of a proprietary protocol the reader communicates with an application interface (API). This API converts between a XML stream and the proprietary format. The user-application, which is formed from the second prototype, communicates with the reader API.

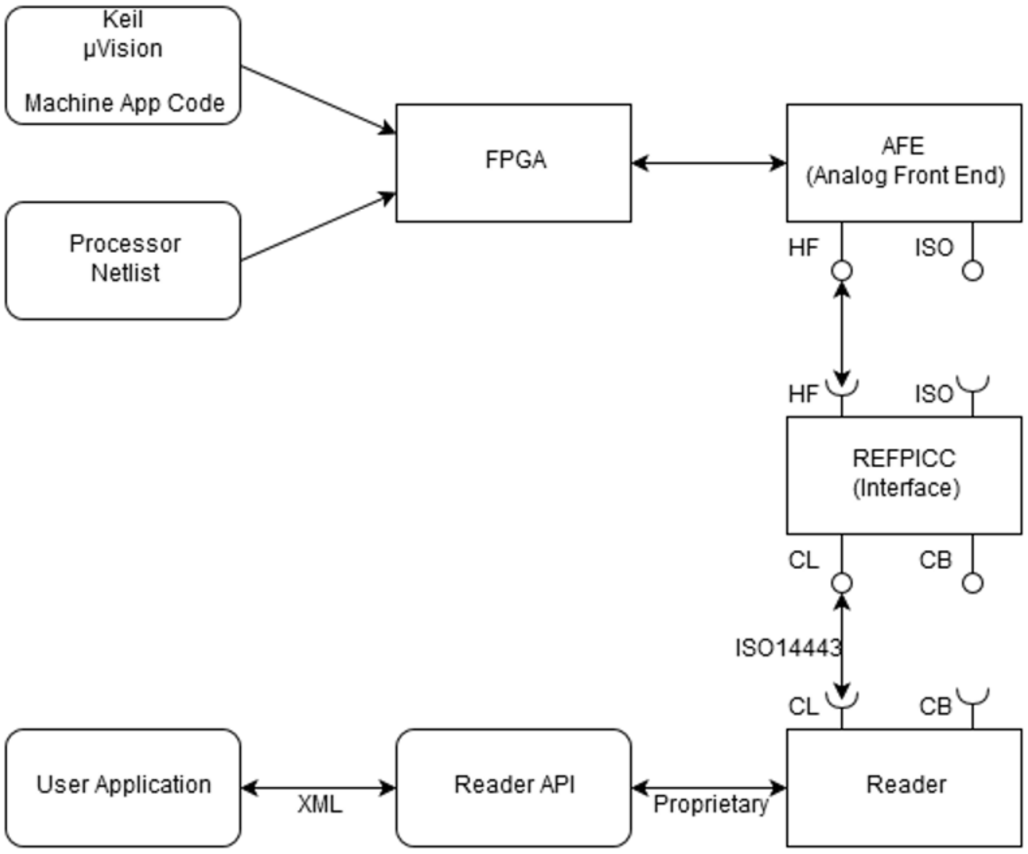


Figure 3.13: Design of the cycle-accurate evaluation.

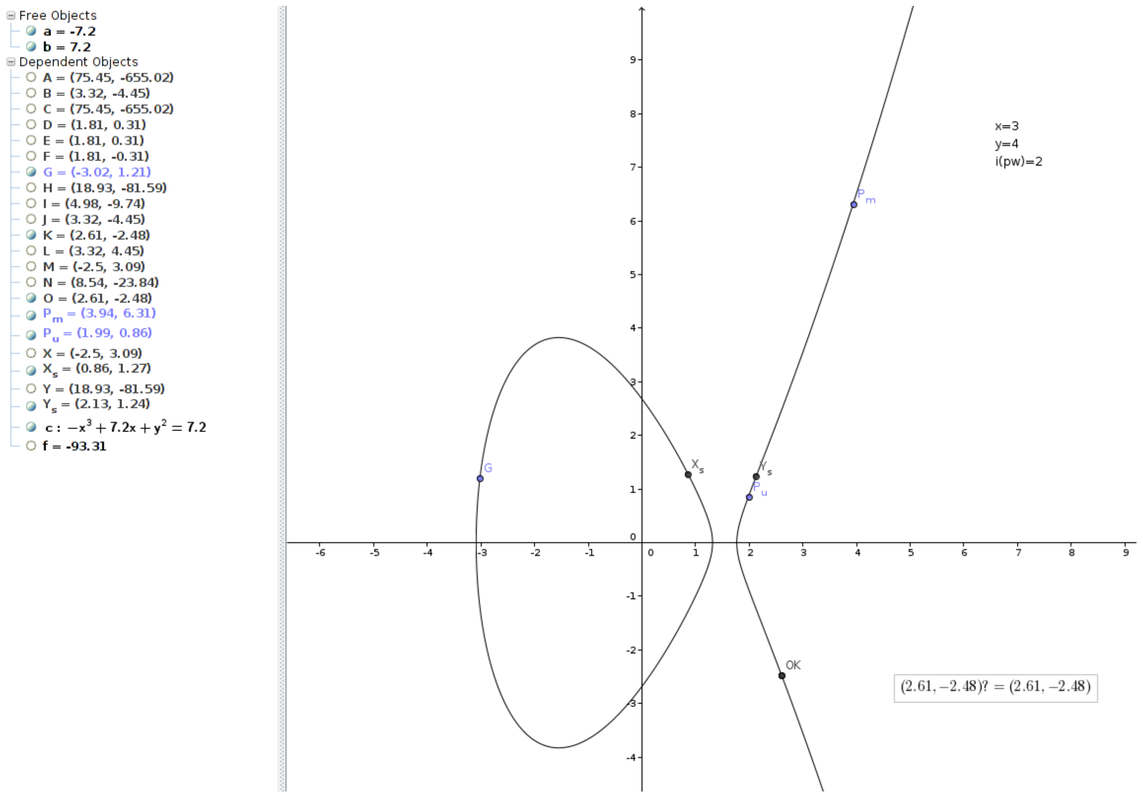
In the simulation the code is executed in Keil's μVision. The reader API connects to the simulator via a network interface transmitting the APDUs. The user-application communicates with the reader API with the same XML stream as in Figure 3.13.

4 Implementation

A core part of the conducted work is the implementation of the designed protocol. This chapter focuses on that. It shows the mathematical concept behind the SPAKE2 protocol. Furthermore, the specific part how the authentication and key exchange is done gets explained. After that the implementation is at the focus. The different implementations during the development process are described. Finally the implementation of the demo-application is analyzed.

4.1 Mathematical Concept

To evaluate the concept of SPAKE2 as described in Table 3.1, a small proof of concept is developed. For usability reasons the used numbers are set to small values. In Figure 4.1 the output of the mathematical concept is shown. In this figure the curve can be parametrized on will. Furthermore, the points G (generator-point) and M and N (points associated with the users) can be set freely. The random variables x and y are set to 3 and 4 respectively, as well as the number generated by the password is set to 2. This can be done as this is only a proof of concept. During the execution of the algorithm the points X_S and Y_S are exchanged. These points are displayed in the plot. When calculating the rest of the algorithm the points O and K are derived. These points need to be equal (the same key is derived). As this simple implementation shows, the used concepts can be developed with more advanced techniques. The next step is the prototypical implementation with OpenSSL. This mathematical showcase is implemented using GeoGebra.


Free Objects
- $a = -7.2$
- $b = 7.2$

Dependent Objects
- $A = (75.45, -655.02)$
- $B = (3.32, -4.45)$
- $C = (75.45, -655.02)$
- $D = (1.81, 0.31)$
- $E = (1.81, 0.31)$
- $F = (1.81, -0.31)$
- $G = (-3.02, 1.21)$
- $H = (18.93, -81.59)$
- $I = (4.98, -9.74)$
- $J = (3.32, -4.45)$
- $K = (2.61, -2.48)$
- $L = (3.32, 4.45)$
- $M = (-2.5, 3.09)$
- $N = (8.54, -23.84)$
- $O = (2.61, -2.48)$
- $P_m = (3.94, 6.31)$
- $P_u = (1.99, 0.86)$
- $X = (-2.5, 3.09)$
- $X_s = (0.86, 1.27)$
- $Y = (18.93, -81.59)$
- $Y_s = (2.13, 1.24)$
- $c: -x^3 + 7.2x + y^2 = 7.2$
- $f = -93.31$

x=3
y=4
i(pw)=2

$(2.61, -2.48)? = (2.61, -2.48)$

Figure 4.1: Mathematical proof of the design of the SPAKE-algorithm.

4.2 Authenticate the User

For the authentication of the user a point associated with the user needs to be generated or stored. The generation of such a point can be achieved by calculating a hash-function on the username and using the digest as a number. By multiplying the generator-point G of the curve with that number a point on the curve, defined by the username, can be generated. The same method (hashing and using the digest as number) can be used when multiplying the point associated with the user (or the one associated with the machine) with the password. To not store the password on the machine in clear text the hash-value can be stored or, even better, a secure element with secure memory can be used to hold the data.

4.3 Key Exchange

The key exchange follows the SPAKE-algorithm shown in Table 3.1. The parties generate a random number which is used as the private key in an ECC (Elliptic Curve Cryptography) system. With this the user and the machine generate the public Keys

X and Y. The points M and N are calculated by multiplying the generator-point G of the curve with the hash of the own name. The resulting mask is then calculated by multiplying M respectively N with the hash of the password. In this case the hash-function is used for key expansion (generating a big number out of a small value). The masked public key is then generated by adding the mask to the public key. After the masked keys are sent the mask of the other party is calculated (with the hash of the remote name and the hash of the password), inverted, and added to the received key. The resulting point is then multiplied with the own secret key. This is done according to the Diffie-Hellman key exchange. After this the hash of the concatenation of the names, the masked keys, the password, and the generated point is calculated and used as shared key.

4.4 Design Flow

In Figure 4.2 the path from the source-code to the executable file on the hardware is shown. The C89-code consists of multiple files that implement an operating system and the applications for the secure element. This is developed in Keil's μVision. With the *Keil C251 Compiler* the C-code is converted to the optimized Assembly-code. The next step towards the final code is performed by the *Keil A251 Assembler*. This process results in an hex-file suited for the processor. The secure element features a MMU (Memory Management Unit). Therefore, a configuration file for the memory must be provided to a *Postlocator*-application. This application defines where the used variables are placed in the memory. It also defines which variables are placed in the RAM, ROM, or in the NVM (Non Volatile Memory - Flash). The hex-file and the file containing the memory information are then used by one of the three following applications:

Keil Simulator: The simulator takes the files and executes the final application on the computer. This tool is used in the early development process as it can be used quickly and one does not need additional hardware. The simulation is very slow compared to the other two scenarios but features excellent debugging possibilities.

Keil X51 Emulator: The emulation of the final application is done as an intermediate step between the simulation and the execution on the intended hardware. The emulator connects to an FPGA board that emulates the intended hardware. This process is faster and can unveil more errors. It has only limited debugging possibilities.

SolidFlash: The SolidFlash application is used to get the hex-file to the microcontroller. The execution on this hardware is the fastest of the three possibilities but has no default debug functionalities. If one wants to debug on this hardware special commands are needed to get the wanted data.

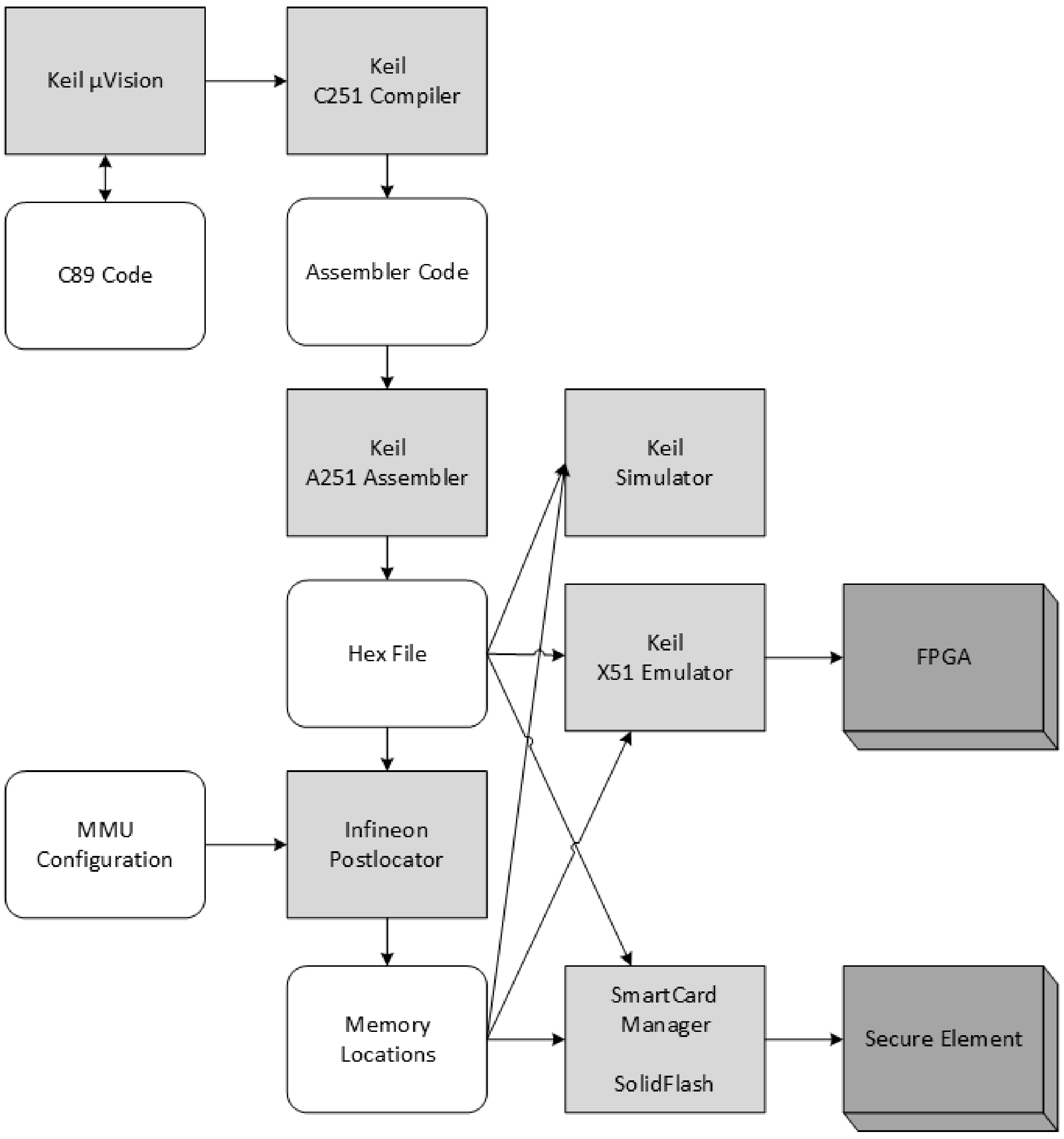


Figure 4.2: Toolchain for the software development process.

4.5 Implementation Environments

This section describes briefly what libraries and implementation environments are used within the implementation of the SPAKE algorithm.

4.5.1 Used Libraries

The library used in most modern crypto software on a computer is the OpenSSL-library. This is a free library to perform Transport Layer Security (TLS) formerly Secure Sockets Layer (SSL). This library contains the code to get, create, and save certificates and to perform basic cryptographic functions.
Within this library many cryptographic schemes are provided. This library features modern crypto-schemes like RSA, ECC, AES, DES, DH, SHA, X509, and many more. Furthermore, some older schemes like MD5, Camellia, ChaCha (Salsa20), and others are featured to be able to communicate with older systems.
As the OpenSSL library is open source, many bugs have been reported and fixed, making it to a safe to use library. Nevertheless, some bugs can be found. The last one was the "*Heartbleed*"-bug ([DKA+14]). Persons exploiting this bug could read parts of the working memory of other computers, getting access to private data such as cryptographic keys of certificates, usernames, and passwords. This bug was fixed in April 2014, showing that even this well understood and widely used library is not perfectly secure.

4.5.2 Used Applications

Eclipse C++ is used to implement the first prototypical version of SPAKE. To perform the cryptographic challenges an OpenSSL library is needed.
The second prototype (the one that uses the microcontroller's data types) is also being programmed in Eclipse C++ and further developed in Microsoft Visual Studio Express 2012. The definition of the interface and datatypes of the microcontroller is given by Infineon. In the background the OpenSSL calculates the cryptographic operations which are provided by the interface.
The implementation for the security controller is done using Keil's μVision5 and Infineon-type operating systems and emulators for the security controller. With this simulation the cryptographic operations are calculated with a Infineon-type crypto-library, that is fitted to the operations on the security controller.
The demonstrator is developed in Microsoft Visual Studio Express 2012 on the basis of the second prototype. This version uses the Windows sockets from the *Ws2_32.lib* library. The developed application connects to a Infineon reader with

a simulator for the contactless interface of a smart card. This reader connects to the simulation of the crypto-processor's contactless interface. This simulation is done within Keil's μVision.

4.6 Prototypical Implementation

The first implementation in C++ is done by generating a class for users and one for machines. The needed data for a connection is stored in a struct. The used data for a connection is the secret number x or y, the transmitted masked Points X^* and Y^*, the used curve, the username or machine-name, the hashed password (only for the machine), and for convenience the generated K_A or K_B. Every user and machine has an array of these structs for storage of the different connections. The machine stores additionally its own name, and the user stores his username and, for convenience, the hashed password.

In Listing 4.1 the class of the user is defined. It contains an array of "Connections" to different machines. Furthermore, the possibilities of the user are defined with the function headers. The machinename is then used as a key to find the connection to the correct machine. Therefore these names must be unique.

```
class User
{
  typedef struct Connection{
    char* machinename;
    int curve;
    EC_POINT* X_s;
    EC_POINT* Y_s;
    EC_POINT* K;
    BIGNUM* x;
  } Connection;
private:
  char* username;
  char* hashedpassword;

  Connection* connections;
  unsigned numConnections;
public:
  User();
  User(char* user, char* pwd);
  User(const User& u);
  virtual ~User();
  void changePassword(char* pwd);
  void changeUsername(char* user);
  char* getHashedPassword();
  char* getUsername();

  int setConnection(char* machinename, int curve);
  int changeCurve(char* machinename, int curve);
  EC_POINT* generateXs(char* machinename);
  int generateK(char* machinename, EC_POINT* Y_s);
  int deleteConnection(char* machinename);
  int initConnection(char* machinename);

  char* debugGetKey(char* machinename);
  void debugPrintConnections();
  BIGNUM* getX(char* machineneame);

  int getCurve(char* machinename);
};
```

Listing 4.1: The definition of the "User"-class

To support multiple connections to machines but not overwrite old data due to wrong usage the “setConnection”-function checks the current connections for matches. This is shown in Listing 4.2.

```
int User::setConnection(char* machinename, int curve){
  if(curve==0){
    printf("User:setConnection Error: curve is not set\n");
    return 0;
  }
  Connection* oldconnections = new Connection[numConnections];
  for(unsigned i=0; i<numConnections;i++){
   oldconnections[i]=connections[i];
  }
  delete[] connections;
  connections = new Connection[numConnections+1];
  for(unsigned i=0;i<numConnections;i++){
    connections[i]=oldconnections[i];
    if(connections[i].machinename==machinename){
      printf("User:setConnection Error: Connection already exists\n");
      delete[] connections;
      connections=new Connection[numConnections];
      for(unsigned j=0;j<numConnections;j++)
      {
        connections[j]=oldconnections[j];
      }
      delete[] oldconnections;
      return 0;
    }
  }
  delete[] oldconnections;
  connections[numConnections].X_s=NULL;
  connections[numConnections].Y_s=NULL;
  connections[numConnections].K=NULL;
  connections[numConnections].curve=curve;
  connections[numConnections].machinename=machinename;
  BIGNUM* r = BN_new();
  getRandBN(r);
  connections[numConnections].x=r;
  numConnections++;
  return 1;
}
```

Listing 4.2: A new connection gets initiated. If the machinename is already found the operation is terminated and the old state is reset.

The deletion of a connection between user and machine is also an operation that is necessary for complete testing of an industrial environment. The code in Listing 4.3 shows this operation. One can also see that the memory management is done within the function. This is necessary as C/C++ does not take care of that.

```
int User::deleteConnection(char* machinename){
  if(numConnections==0){
    printf("User:deleteConnection Error: No Connections\n");
    return 0;
  }
  bool found=false;
  Connection* oldconnections = new Connection[numConnections];
    for(unsigned i=0; i<numConnections;i++){
     oldconnections[i]=connections[i];
     if(connections[i].machinename==machinename){
       found=true;
     }
    }
    if(!found){
      printf("User:deleteConnection Error: No Connection to %s found\n",machinename);
      delete[] oldconnections;
      return 0;
    }
    delete[] connections;
    bool deleted=false;
    connections = new Connection[numConnections-1];
    unsigned added=0;
    for(unsigned i=0;i<numConnections;i++){
      if(oldconnections[i].machinename != machinename){
        connections[added]=oldconnections[i];
        added++;
      }else{
        if(oldconnections[i].X_s){EC_POINT_free(oldconnections[i].X_s);}
        if(oldconnections[i].Y_s){EC_POINT_free(oldconnections[i].Y_s);}
        if(oldconnections[i].K)  {EC_POINT_free(oldconnections[i].K);}
        BN_free(oldconnections[i].x);
        deleted=true;
      }
    }
    if(!deleted){
      //printf("User:deleteConnection Error: No connection to %s found\n",machinename);
      delete[] connections;
      connections = new Connection[numConnections];
      for(unsigned i=0;i<numConnections;i++){
        connections[i]=oldconnections[i];
      }
      delete[] oldconnections;
      return 0;
    }
    delete[] oldconnections;
    numConnections--;
    return 1;
}
```

Listing 4.3: An old connection is deleted. Therefore the remaining connections must be copied to a smaller memory.

In Listing 4.4 the generation of the masked public key by the user is shown. This function starts by generating a EC_GROUP from a curve identifier. From the generator point (G) and the secret of the user (x) the public key (X) is calculated. The function getPoint(. . .) generates a number based on the first parameter. This function calculates a new point using this number, the curve parameters, and a point on the curve as starting point. In the implementation the point on the curve is the ordinary generator point, but as some other schemes use two generator points on a curve, this parameter is added.

```
EC_POINT* User::generateXs(char* machinename){
  if(numConnections>0){
    for(unsigned i=0;i<numConnections;i++){
      if(connections[i].machinename==machinename){
        if(connections[i].curve==0){
          printf("User:generateXs Error: curve not specified\n");
          return NULL;
        }
        if(connections[i].X_s==NULL){
          printf("Connection not initialized properly\n");
          return NULL;
        }
        if(connections[i].Y_s==NULL){
          printf("Connection not initialized properly\n");
          return NULL;
        }
        if(connections[i].K==NULL){
          printf("Connection not initialized properly\n");
          return NULL;
        }
        EC_GROUP* c;
        if((c=EC_GROUP_new_by_curve_name(connections[i].curve))==NULL){
          printf("User:generateXs Error: group generating not possible\n");
          EC_GROUP_free(c);
          return NULL;
        }
        EC_POINT* G = EC_POINT_new(c);
        if(0==EC_POINT_copy(G,EC_GROUP_get0_generator(c))){
          printf("User:generateXs Error: copy returned with an error\n");
          EC_POINT_free(G);
          EC_GROUP_free(c);
          return NULL;
        }
        EC_POINT* X = EC_POINT_new(c);
        if (0==EC_POINT_mul(c,X,NULL,G,connections[i].x,NULL)){
          printf("User:gemerateXs Error: Mul1 returned with an error\n");
          EC_POINT_free(G);
          EC_POINT_free(X);
          EC_GROUP_free(c);
          return NULL;
        }
        EC_POINT* M = EC_POINT_new(c);
        getPoint(username, c,G,M);
        BIGNUM* pw = BN_new();
        getBN(hashedpassword,pw);
        EC_POINT* X_i = EC_POINT_new(c);
        if (0==EC_POINT_mul(c,X_i,NULL,M,pw,NULL)){
          printf("User:generateXs Error: Mul2 returned with an error\n");
          EC_POINT_free(G);
          EC_POINT_free(X);
          EC_POINT_free(X_i);
          EC_POINT_free(M);
          BN_free(pw);
          EC_GROUP_free(c);
          return NULL;
        }
        EC_POINT* X_s = EC_POINT_new(c);
        if(0==EC_POINT_add(c,X_s,X_i,X,NULL)){
          printf("User:generateXs Error: Add1 returned with an error\n");
          EC_POINT_free(G);
          EC_POINT_free(X);
          EC_POINT_free(M);
          EC_POINT_free(X_i);
          BN_free(pw);
          EC_GROUP_free(c);
          return NULL;
        }

        if(0==EC_POINT_copy(connections[i].X_s,X_s)){
          printf("User:generateXs Error: Copy of X_s returned with an error\n");
          EC_POINT_free(G);
          EC_POINT_free(X);
          EC_POINT_free(M);
          EC_POINT_free(X_i);
          EC_POINT_free(connections[i].X_s); connections[i].X_s=NULL;
          BN_free(pw);
          EC_GROUP_free(c);
          return NULL;
        }

        EC_POINT_free(G);
        EC_POINT_free(X);
        EC_POINT_free(M);
        EC_POINT_free(X_i);
        BN_free(pw);
        EC_GROUP_free(c);
        return X_s;
      }
    }
    printf("User:generateXs Error: No connection to machine %s found\n",machinename);
    return NULL;
  }
  printf("User:generateXs Error: No connections found\n");
  return NULL;
}
```

Listing 4.4: generation of the masked public key of the user.

When the generation of the masked public key (X_S) is finished it is sent to the machine and its masked key is requested. After that key is received the shared secret (K) is calculated. The function performing that is shown in Listing 4.5.

```
int User::generateK(char* machinename, EC_POINT* Y_s){
  if(numConnections>0){
    for(unsigned i=0; i<numConnections;i++){
      if(connections[i].machinename==machinename){
        if(connections[i].X_s==NULL){
          printf("User:generateK Error: Connection not initialized properly\n");
          return NULL;
        }
        if(connections[i].Y_s==NULL){
          printf("User:generateK Error: Connection not initialized properly\n");
          return NULL;
        }
        if(connections[i].K==NULL){
          printf("User:generateK Error: Connection not initialized properly\n");
          return NULL;
        }
        EC_GROUP* c;
        if((c=EC_GROUP_new_by_curve_name(connections[i].curve))==NULL){
          printf("User:generateK Error: group generating not possible\n");
          EC_GROUP_free(c);
          return 0;
        }
        if(0==EC_POINT_copy(connections[i].Y_s,Y_s)){
          printf("User:generateK Error: copy of Y_s returned with an error\n");
          EC_POINT_free(connections[i].Y_s);
          connections[i].Y_s=EC_POINT_new(c);
          EC_GROUP_free(c);
          return 0;
        }
        EC_POINT* G = EC_POINT_new(c);
        if(0==EC_POINT_copy(G,EC_GROUP_get0_generator(c))){
          printf("User:generateK Error: copy returned with an error\n");
          EC_POINT_free(G);
          EC_POINT_free(connections[i].Y_s);
          connections[i].Y_s=EC_POINT_new(c);
          EC_GROUP_free(c);
          return 0;
        }
        EC_POINT* N = EC_POINT_new(c);
        getPoint(machinename, c,G,N);
        BIGNUM* pw = BN_new();
        getBN(hashedpassword,pw);

        EC_POINT* Ki = EC_POINT_new(c);
        if (0==EC_POINT_mul(c,Ki,NULL,N,pw,NULL)){
          printf("User:generateK Error: Mul1 returned with an error\n");
          EC_POINT_free(G);
          EC_POINT_free(connections[i].Y_s);
          connections[i].Y_s=EC_POINT_new(c);
          EC_POINT_free(N);
          EC_POINT_free(Ki);
          BN_free(pw);
          EC_GROUP_free(c);
          return 0;
        }
    if(0==EC_POINT_invert(c,Ki,NULL)){
          printf("User:generateK Error: Invert returned with an error\n");
          EC_POINT_free(G);
          EC_POINT_free(connections[i].Y_s);
          connections[i].Y_s=EC_POINT_new(c);
          EC_POINT_free(N);
          EC_POINT_free(Ki);
          BN_free(pw);
          EC_GROUP_free(c);
          return 0;
        }
    EC_POINT* Ki2 = EC_POINT_new(c);
        if(0==EC_POINT_add(c,Ki2,Ki,Y_s,NULL)){
          printf("User:generateK Error: Add1 returned with an error\n");
          EC_POINT_free(G);
          EC_POINT_free(connections[i].Y_s);
          connections[i].Y_s=EC_POINT_new(c);
          EC_POINT_free(N);
          EC_POINT_free(Ki);
          EC_POINT_free(Ki2);
          BN_free(pw);
          EC_GROUP_free(c);
          return 0;
        }
        if (0==EC_POINT_mul(c,connections[i].K,NULL,Ki2,connections[i].x,NULL)){
          printf("User:generateK Error: Mul2 returned with an error\n");
          EC_POINT_free(G);
          EC_POINT_free(connections[i].Y_s);
          connections[i].Y_s=EC_POINT_new(c);
          EC_POINT_free(N);
          EC_POINT_free(Ki);
          EC_POINT_free(Ki2);
          BN_free(pw);
          EC_GROUP_free(c);
          return 0;
        }

        getRandBN(connections[i].x);
        EC_POINT_free(G);
        EC_POINT_free(N);
        EC_POINT_free(Ki);
        EC_POINT_free(Ki2);
        BN_free(pw);
        EC_GROUP_free(c);
        return 1;
      }
    }
    printf("User:generateK Error: No connection to machine %s found\n",machinename);
    return 0;
  }
  printf("User:generateK Error: No connections found\n");
  return 0;
}
```

Listing 4.5: With the parameters of the connection and the received key the shared pre-secret (K) is calculated.

The Listing 4.6 shows the definition of the “Machine”-class. For connecting to the user the machine needs to store other information (not only the username but also the (hashed) password). The other function definitions are almost identical to the ones from the user.

```
class Machine
{
  typedef struct users{
    char* username;
    char* hashedpassword;
    BIGNUM* y;
    int curve;
    EC_POINT* K;
    EC_POINT* X_s;
    EC_POINT* Y_s;
  } StrUsers;
private:
  char* machinename;
  StrUsers* users;
  unsigned int numUsers;
public:
  Machine(char* name);
  virtual ~Machine();
  Machine(const Machine& m);

  char* getMachinename();
  int addUser(char* name, char* hpwd);
  int deleteUser(char* name);
  int updateUserPwd(char* name, char* hpwd);
  int setUserCurve(char* username, int curve);
  EC_POINT* generateKey(char* username, EC_POINT* X_s);

  char* debugGetKey(char* username);
  void debugPrintUsers();
  BIGNUM* getY(char* username);
};
```

Listing 4.6: The definition of the “Machine”-class

The establishment of new connections and the deletion of old ones, and the calculation of the own public key and of the preshared secret is similar to the ones of the “User” (Listings 4.2, 4.3, 4.4, and 4.5). The code for the calculation of the preshared secret and the session key can be found in the Appendix Section A.1, Listing A.2.
The calculation of the session key is done the same way by the user and the machine. The code for that is shown in Listing 4.7.

```
char* Machine::debugGetKey(char* username){
  if(numUsers>0){
    for(unsigned i=0;i<numUsers;i++){
      if(users[i].username==username){
        if(users[i].X_s!=NULL && users[i].Y_s !=NULL && users[i].K != NULL && users[i].curve!=0){
          char* o = new char[SHA224_DIGEST_LENGTH*12+6];
          char* md = getHash(users[i].username);
          strcpy(o,md);
          strcat(o,"-");
          delete[]md;
          md=getHash(machinename);
          strcat(o,md);
          strcat(o,"-");
          EC_GROUP* c;
          if((c=EC_GROUP_new_by_curve_name(users[i].curve))==NULL){
            //printf("Machine:debugGetKey Error: generating group not possible\n");
            EC_GROUP_free(c);
            delete[] o;
            return NULL;
          }
          delete[]md;
          md = getHash(streamPoint(c,users[i].X_s),NULL,true);
          strcat(o,md);
          strcat(o,"-");
          delete[]md;
          md = getHash(streamPoint(c,users[i].Y_s),NULL,true);
          strcat(o,md);
          strcat(o,"-");
          delete[]md;
          md = getHash(users[i].hashedpassword,NULL,false);
          strcat(o,md);
          strcat(o,"-");
          delete[]md;
          md = getHash(streamPoint(c,users[i].K),NULL,true);
          strcat(o,md);
          o=getHash(o,NULL,true);
          EC_GROUP_free(c);
          delete[]md;
          return o;
        }
        printf("Machine:debugGetKey Error: Not all steps to generate key taken\n");
        return NULL;
      }
    }
    printf("Machine:debugGetKey Error: No connection to user %s found\n",username);
    return NULL;
  }
  printf("Machine:debugGetKey Error: No connections found\n");
  return NULL;
}
```

Listing 4.7: Calculation of the session key as done by the machine. It is calculated the same way by the user.

To establish a connection between an user and a machine the user needs to get the memory for the saved variables and initialize the values. The machine also needs to acquire the used memory, initialize it and set the used cryptographic curve. To perform the key exchange the user generates the X^* (in the code X_S). This is then given to the machine which calculates the key. As a by-product Y^* (Y_S in the code) is calculated. This is given to the user, which can complete the computation of the key. The final key can then be calculated by concatenating the names of the parties, the transmitted keys, the password, and the generated key. This concatenation can then be hashed and the digest is used as key.
In our implementation the items of the concatenation are hashed at first to get an input of fixed length.

As input to the mathematical functions on the EC-points, the algorithm uses the names of the user and the machine. To expand these inputs the hash of the values is calculated and interpreted as number (this is a practical way for key-expansion). This method yields a big enough number to be useful for ECC.

This algorithm can then be tested with many scenarios. These include the normal use with only one user and one machine, the use of multiple users with multiple machines, the case if a user wants to connect multiple times to the same machine, etc. To test many different scenarios a randomized test was generated which generates randomly valid commands and tries to use this command. One very interesting fact is that even if a user can guess the password of another user he cannot connect to the machine if the second user has already initialized the connection to the machine.

One of the tests simulating normal use is shown in Listing 4.8. In this test two users connect to two machines. They use different curves for encryption. This test prints the session-keys from the four connections at the end. Not shown is the test if these are equal.

```
void SPAKE_test_2(){
  printf("\n\nTest 2\nnormal use\n");
  User* u1 = new User("User1","password1");
  User* u2 = new User("User2","@home");
  Machine* m1 = new Machine("Machine1");
  Machine* m2 = new Machine("Machine2");

  connectUserAndMachine(u1,m1,NID_secp224k1);
  connectUserAndMachine(u1,m2,NID_secp224k1);
  connectUserAndMachine(u2,m1,NID_secp224r1);
  connectUserAndMachine(u2,m2,NID_secp224r1);

  performExchange(u1,m1);
  performExchange(u1,m2);
  performExchange(u2,m1);
  performExchange(u2,m2);

  printKeys(u1,m1,"11");
  printKeys(u1,m2,"12");
  printKeys(u2,m1,"21");
  printKeys(u2,m2,"22");

  delete(u1);
  delete(u2);
  delete(m1);
  delete(m2);
}
```

Listing 4.8: A sample test containing normal usage.

A more rigorous test is the random generation of valid commands and sending that to the system. This test is shown in Listing A.1.
The code for generating the machine-side of the key exchange (calculating the public key and calculating the shared key) is basically the same as on the user-side. It can be found in Listing A.2.

As this rigorous tests performed as expected the next step is the implementation of the interface to of the crypto-processor. This interface should perform as the processor but in the background the OpenSSL-library is working. With this interface the SPAKE-implementation can be changed to work on the crypto-processor. This gives the advantage that the implementation stays debug-able while simulating the crypto-processor.

4.7 Transitional Implementation

As the final implementation should be deployed on an Infineon-type cryptoprocessor, a specification of the usable functions and used data-types had been provided. These functions do not contain a way to invert EC-points, therefore this function had to be developed. The prototypical implementation had to be changed to use the provided interface and used data-types. As the software is still executed on a Laptop-computer, the functions, which should be performed by the crypto-hardware, had to be implemented using the OpenSSL library. Therefore a conversion from the OpenSSL data-types to the ones used by the hardware and vice versa had to be developed. The structure of the used datatypes is depicted in Figure 3.9.
The new data-flow looks as follows:

- The simulation calls functions provided by the SPAKE implementation (and some helper functions to emulate the communication).
- The SPAKE implementation uses the interface and the helper functions (e.g.: point inversion, hashing data, allocating memory, ...).
- The interface calls the helpers for converting the provided data-types to the OpenSSL-types, uses the OpenSSL, and converts back the results.

The processor's crypto-library supports many different ECC functions. Only the ones needed were implemented in the interface to the OpenSSL. The code in Listing 4.9 shows the conversion from processor-specific data types to the OpenSSL ones, the use of the OpenSSL EC-Point multiplication, and the conversion back to the specific data structure.

```
CLIB_STATUS ECC_DHMask(           ECCCPARM    huge *ecccp     // [in]  ECC parameters, defining the
       ECC curve
                                , CLONG       huge *k1        // [in]  Secret scalar value of party 1
                                , CLONG       huge *k2        // [in]  Secret scalar value of party 1
                                , AFFINEPOINT huge *P         // [in]  Public point provided by 2nd
                                         party
                                , CLONG       huge *mem       // [in]  Temporary working memory
                                , AFFINEPOINT huge *S         // [out] SharedSecret->X will be the
                                         shared secret integer.
                                )
{
  BIGNUM *k_1,*k_2,*k,*a,*b,*p,*x,*y;
  EC_GROUP *c;
  EC_POINT* p1;
  k_1 = BN_new();
  getBN(*k1,k_1);
  k_2 = BN_new();
  getBN(*k2,k_2);
  k = BN_new(); //k1+k2
  BN_add(k,k_1,k_2);
  BN_free(k_1);BN_free(k_2);
  a = BN_new(); //ecccp->CoefA
```

```c
  getBN(ecccp->CoefA,a);
  b = BN_new(); //ecccp->CoefB
  getBN(ecccp->CoefB,b);
  p = BN_new(); //ecccp->Modulus
  getBN(ecccp->Modulus,p);
  x = BN_new(); //P->X
  getBN(P->X,x);
  y = BN_new(); //P->Y
  getBN(P->Y,y);

  c = EC_GROUP_new_curve_GFp(p,a,b,NULL);
  BN_free(a);BN_free(b);BN_free(p);

  p1 = EC_POINT_new(c);
  EC_POINT_set_affine_coordinates_GFp(c,p1,x,y,NULL);
  if(0==EC_POINT_mul(c,p1,NULL,p1,k,NULL))
  {
    printf("Error");
  }
  BN_free(k);
  EC_POINT_get_affine_coordinates_GFp(c,p1,x,y,NULL);
  getCLONG(x,&S->X);
  getCLONG(y,&S->Y);
  BN_free(x);BN_free(y);
  EC_POINT_free(p1);
  EC_GROUP_free(c);
  return CLIB_STATUS_SUCCESS_UNSPECIFIC;
}
```

Listing 4.9: Implementation of the Multiplication of ECC points with a scalar with OpenSSL. The scalar is inserted as a sum to get additional security.

The crypto-library is intended to perform operations common in smart cards. Therefore not all operations (like the point inversion) are available to programmers. As the implemented protocol needs an inversion step to unmask the public key, this has to be implemented. The code performing the inversion is shown in Listing 4.10.

```c
CLIB_STATUS ECC_Invert(         ECCCPARM    huge *ecccp     // [in]  ECC parameters, defining the ECC curve
                 , AFFINEPOINT huge *P         // [in]  Affine point P to be inverted
                 , AFFINEPOINT huge *Po    // [out] Resulting affine point Po=P^(-1)
                 )
{
  INT16 l;
  UINT16 t;
  if(ecccp->Modulus.BytesAllocated != P->Y.BytesAllocated){
    return CLIB_STATUS_ERROR_PRECOND_FAIL;
  }
  l = ecccp->Modulus.BytesAllocated;
  copyClong(P->X,&Po->X);
  copyClong(P->Y,&Po->Y);
  t=0;
  for (l--;l>=0;l--){
    if(t==0)
      Po->Y.Data[l] = ecccp->Modulus.Data[l] - P->Y.Data[l];
    else
      Po->Y.Data[l] = ecccp->Modulus.Data[l] - P->Y.Data[l] - 1;
    t=(ecccp->Modulus.Data[l] - P->Y.Data[l])>>8;
  }
  return CLIB_STATUS_SUCCESS_UNSPECIFIC;
}
```

Listing 4.10: Implementation of the point inversion for Points on a GF(p) field.

A common part in the protocol is the generation of a number from a string. To do this a hash-algorithm that returns a digest of the optimal length (the length of the curve modulus) is used. This operation is shown in Listing 4.11. One can see that the "*getHash*" function is called twice. In the first run this function returns the length of the hash digest.

```
CLIB_STATUS getNumberFromString(UINT8 *str, CLONG *cl)
{
  UINT16 len;
  UINT16 inlen=0;
  UINT8  *md;

  inlen = strlen(str);
  if(CLIB_STATUS_SUCCESS_UNSPECIFIC != getHash(NULL,0,NULL,&len)){
    return CLIB_STATUS_ERROR_UNSPECIFIC;
  }
  md = (UINT8*)malloc(sizeof(UINT8)*len);
  if(CLIB_STATUS_SUCCESS_UNSPECIFIC != getHash(str,inlen,md,&len)){
    free(md);
    return CLIB_STATUS_ERROR_UNSPECIFIC;
  }
  memcpy(cl->Data,md,len);
  free(md);
  cl->BitLength = len*8;
  return CLIB_STATUS_SUCCESS_UNSPECIFIC;
}
```

Listing 4.11: Generation of a big number from a string.

The machine-part of the communication consists of three functions: the "generateMachineMessage"-, "generateMachineSharedSecret"-, and the "generateKey"-function.
As the crypto-processor has very limited resources in terms of memory, the amount of users per machine (as well as the amount of machines per user) has been reduced to two. Furthermore, as the targeted processor has hardware security-features in place, the password can be saved in clear on the memory. The definition of these stripped down "classes" can be seen in Listing 4.12.

```
typedef struct _SPAKE_HANDLE{
  ECCCPARM ecccp;
  CLONG secret;
  AFFINEPOINT X_s;
  AFFINEPOINT Y_s;
  AFFINEPOINT K;
} SPAKE_HANDLE;

typedef struct _SPAKE_User{
  UINT8 *username;
  UINT8 *password;
  SPAKE_HANDLE *connections[2];
  UINT8 *machinenames[2];
} SPAKE_User;

typedef struct _SPAKE_Machine{
  UINT8 *machinename; // string
  SPAKE_HANDLE *connections[2];
  UINT8 *users[2];
  UINT8 *passwords[2];
} SPAKE_Machine;
```

Listing 4.12: Definition of the "Machine" and "User" in the transitional implementation.

The functionalities for converting between the internal data types and the OpenSSL ones, and the obtaining of the curve parameters is shown in Section A.2. Because this implementation should be executable by the crypto-processor, a further implementation is made. This implementation is written with an emulator of the processor. This development should yield a binary file that can be uploaded directly to the processor.

4.8 Changes for the Security Controller

In the implementation for the security controller the functions for the user-side of the communication have been omitted as the controller should secure the machine-side of a communication. Furthermore, the memory usage has been reduced as better improvements are possible due to the fact that the used memory needs to be allocated at compile time and therefore the addresses are fixed. To be able to communicate the simulation-layer has been changed to an intermediate layer that converts between the SPAKE-functions and the sent and received APDUs (Application Protocol Data Unit). The communication interface has six functions that are used to perform the exchange and configure the algorithm. These commands are:

Set Machine: This command sets the machinename. It can be called if the user is authenticated.

Set User: This command sets the username and password of the one of the users of the machine.

Set UserMessage: This command is to set the masked public key of the user.

Get MachineMessage: This command is a request for the masked public key of the machine.

Authenticate: With this command the session key is calculated. After that the communication can be performed encrypted and the critical commands can be used. If the user is not authenticated the sending of such a command results in an "error"-return message.

Disconnect: This command terminates the connection and deletes the session key, the public keys and the private key.

Echo: The payload is replied if the user is authenticated.

One more command is added to this list for the demonstration application. This debug-command allows the authenticated user to read out the derived key that can be used for the encryption. This command needs to be removed in a deployed version. This can be done by inserting a compile-switch.
In Figure 4.3 the sequence to configure the machinename, the other user of the machine, or both is shown. This command-sequence performs the key exchange with authentication, encrypts the messages, sends the configuration data - here also other commands such as the authenticated echo-service can be executed, and disconnects from the machine.

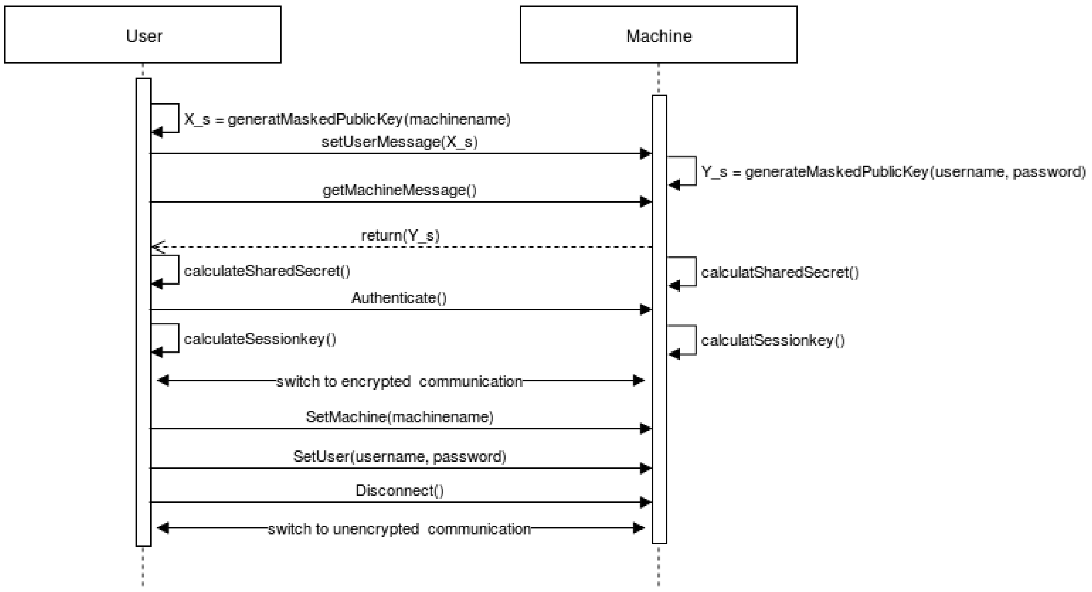


Figure 4.3: Steps to configure the machineame and users on a machine.

4.9 Implementation of the Demonstrator

The demonstrator consists of two applications. The first one is the readily implemented algorithm that is executed on the simulator. This program performs the machine-side of the key exchange. The second application is made of the transitional implementation that is made beforehand (Section 4.7). The secure element calculates the machine-side of the communication. It also cannot initiate a communication to the other side. The functions for the user-side of the communication is performed by the other program on a computer. On top of this a user interface (UI) is implemented that can take commands from the operator and initiate the operations and messages necessary for the protocol. These messages are then converted to an XML-string that is sent to the interface-simulator which converts this string to APDUs (Application Protocol Data Units). The simulator of the secure element executes the code that would normally be run on the secure element. This execution results in another message that is sent to the user. The interface-simulator takes this message and converts it to an XML-string and sends this to the user where it is parsed and used by the program.

The user can activate the secure element (getting the device into operating distance), perform a key agreement with his password, configure the machine (users and machinename) if the authentication is good, execute the dummy echo-command, and disconnect again. Furthermore, the debug command can be used to get to - otherwise secret - material and the current state of the users can be displayed.

The implementation of the demonstrator uses the functions described in Section 4.7.

The biggest difference is that the commands for the machine are not sent to the functions implemented in that section, but are sent to the “reader API”. Additionally the calling of those functions is not done in a fixed sequence but the operator of the program can initiate the functions. This can be seen in Listing A.5. The “authdis”-macro performs an authentication and disconnects again (this command is to test the time it takes to disconnect an authenticated user).

All functions that communicate with the secure element need to take the following steps:

- Ask the user for data that specifies which of the possible users should be taken.
- (Optionally) Ask the user for the password.
- Generate the XML-string that is sent to the reader API.
- Transmit the generated string and wait for an answer from the reader (transceive).
- Extract the result of the answer. Determine if this command was handled as expected.

As an example the “echo”-command is shown in Listing 4.13. The string in Line 7 specifies the APDU header. This is manipulated with the “setUser”- and “setLength”-functions. The message that is sent to the reader is composed in lines 19 to 24. After that the transceiving is performed. Depending on the “debug”-variable this part is done in various cycles (to get more accurate measurements). In line 49 the result of the (last) transceive command is extracted and handled.

```
 1  void echocommand(char* userbuffer, char* tempbuffer, char* inbuffer, int debug)
 2  {
 3    char* message;
 4    unsigned long duration;
 5    char stringfront[]="<?xml version=\"1.0\" encoding=\"UTF-8\"?><call
          method=\"TransceiveData\"><data>";
 6    char stringback[]="</data><calcCRC>false</calcCRC><disableCheckCRC>true</disableCheckCRC>
          <typeOfFrame>standard</typeOfFrame></call>";
 7    char echo[] = "00CF0000000000";
 8    unsigned int messagesize = strlen(stringfront) + strlen(stringback) + 14;
 9    int i;
10
11    getMoreData("Echo request from which User?",1,userbuffer,USERSIZE);
12    if(userbuffer[0]=='0'){setUser(echo,0);}
13    else if(userbuffer[0]=='1'){setUser(echo,1);}
14    else {printf("User can be '1' or '2'\n");return;}
15    getUserData("What message to send?",1,userbuffer,USERSIZE);
16    for(i=0;i<strlen(userbuffer);i++)
17      sprintf(tempbuffer+(i)*2,"%02X",userbuffer[i]);
18    setLength(echo,strlen(userbuffer));
19    message = (char*) malloc(sizeof(char) * (messagesize + 2*strlen(userbuffer) + 1));
20    memcpy(message,stringfront,strlen(stringfront));
21    memcpy(message+strlen(stringfront),echo,strlen(echo));
22    memcpy(message+strlen(stringfront)+strlen(echo),tempbuffer,2*strlen(userbuffer));
23    memcpy(message+strlen(stringfront)+strlen(echo) +
          2*strlen(userbuffer),stringback,strlen(stringback));
24    message[strlen(stringfront)+strlen(echo) + 2*strlen(userbuffer) + strlen(stringback)]='\0';
25    if(debug){printf("Message:\n");
26    printf(message);
27    printf("\n");}
```

```
  clearBuffer(inbuffer,MESSAGESIZE);
  if(debug>1)
  {
    int t=debug;
    unsigned long avg=0;
    unsigned long min=ULONG_MAX;
    unsigned long max=0;

    for(;t>0;t--)
    {
      transceive(message,strlen(message),inbuffer,&duration);
      avg+=duration;
      if(duration<min){min=duration;}
      if(duration>max){max=duration;}
    }
    printf("Min: %d\nAverage: %d\nMax=%d\n",min,avg/debug,max);
  }else{
    transceive(message,strlen(message),inbuffer,&duration);
  }
  free(message);
  message = 0;
  extractResult(inbuffer);
  if(debug){printf("Result:\n");
  printf(inbuffer);
  printf("\n");}
  printAsciiString(inbuffer);
  if(debug){printf("Duration of command: %d\n",duration);}
  clearBuffer(inbuffer,MESSAGESIZE);
}
```

Listing 4.13: Code that is executed to perform the “Echo”-command.

The APDU-header-strings of all the commands are shown in Listing 4.14. In the header the first two hex-characters (one byte) determine the class of command. The second byte is the command itself. After this the two parameters and the length of the payload are placed. The last two bytes determine the requesting user. After this command-header the payload is added. Normally the payload consists of only one field or multiple fields of fixed length - like a key consisting of X- and Y-coordinates with 256 bits each. The “set user”-command has two fields of variable length. Therefore the length of the first field must be added as parameter. To raise an error if this is forgotten the characters in the string are set to invalid values.

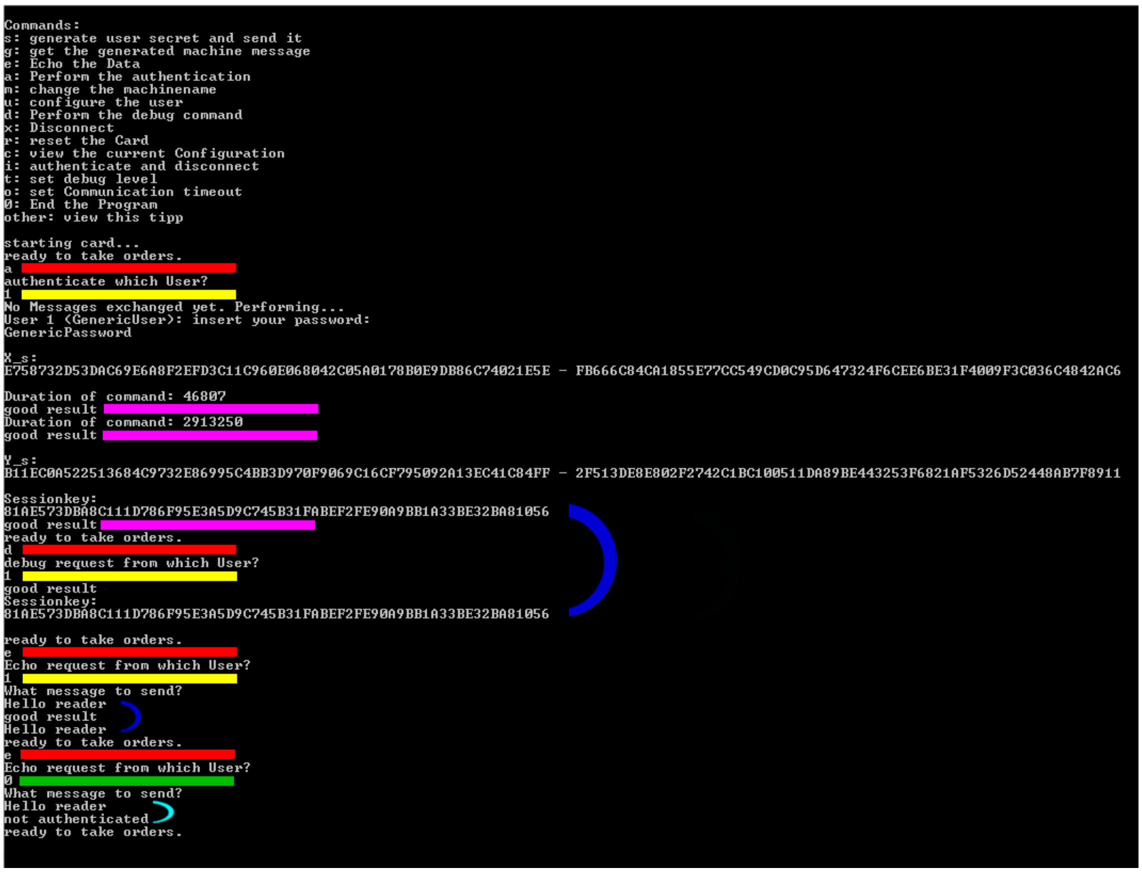
```
/*Echo command - data must be requested by the user*/
char echo[] = "00CF0000000000";
/*Set user message command - length is always 0x42 bytes (2 for additional data, 0x40 for the key)*/
char setuserpubkey[] = "00C30000420000";
/*Get machine message command - length always 2*/
char getmachinepubkey[] = "00C20000020000";
/*Perform Authentication - length always 2*/
char authenticate[] = "00C40000020000";
/*Perform the debug command - length always 2*/
char debug[] = "00CE0000020000";
/*Set machine name command - the machinename must be requested from the user*/
char setmachine[] = "00C00000020000";
/*Set user command - username and password must be requested; replace xx by length of username*/
char setuser[] = "00C1xx00070000";
/*perform the disconnection - length always 2*/
char disconnect[] = "00C50000020000";
```

Listing 4.14: APDU headers for the different commands.

The user interface of the demonstrator is a classical console window. The commands can be entered by the user via the keyboard. Figure 4.4 shows the console after performing an authentication and two “Echo”-commands.

The first red marker indicates the issuing of an authentication step. This is followed by the announcement that user “1” should perform the selected action (first yellow

marker). As the exchange of the public keys has not happened yet the necessary commands are executed beforehand (results at the first two purple markers). The third purple marker shows the result of the authentication request. For debug purposes the exchanged keys and the calculated session key are displayed. The second red marker points at the issuing of the “Debug”-command. Also this one is called by user “1” (second yellow marker). This command returns the session key on the secure element. The big blue arc connects the two corresponding session keys. These must be equal as the subsequent communication will be encrypted with this key.
After that the third red marker indicates the call of the “Echo”-command from user “1”, who is already authenticated. The small blue arc connects the outgoing message and the incoming one. These are also equal. At last the fourth red marker issues another “Echo”-command. This time it is called by user “0” (green marker). As this user is not authenticated the command returns with “not authenticated” (light blue arc).



```
Commands:
s: generate user secret and send it
g: get the generated machine message
e: Echo the Data
a: Perform the authentication
m: change the machinename
u: configure the user
d: Perform the debug command
x: Disconnect
r: reset the Card
c: view the current Configuration
i: authenticate and disconnect
t: set debug level
o: set Communication timeout
0: End the Program
other: view this tipp

starting card...
ready to take orders.
a
authenticate which User?
1
No Messages exchanged yet. Performing...
User 1 (GenericUser): insert your password:
GenericPassword

X_s:
E758732D53DAC69E6A8F2EFD3C11C960E068042C05A0178B0E9DB86C74021E5E - FB666C84CA1855E77CC549CD0C95D647324F6CEE6BE31F4009F3C036C4842AC6

Duration of command: 46807
good result
Duration of command: 2913250
good result

Y_s:
B11EC0A522513684C9732E86995C4BB3D970F9069C16CF795092A13EC41C84FF - 2F513DE8E802F2742C1BC100511DA89BE443253F6821AF5326D52448AB7F8911

Sessionkey:
81AE573DBA8C111D786F95E3A5D9C745B31FABEF2FE90A9BB1A33BE32BA81056
good result
ready to take orders.
d
debug request from which User?
1
good result
Sessionkey:
81AE573DBA8C111D786F95E3A5D9C745B31FABEF2FE90A9BB1A33BE32BA81056

ready to take orders.
e
Echo request from which User?
1
What message to send?
Hello reader
good result
Hello reader
ready to take orders.
e
Echo request from which User?
0
What message to send?
Hello reader
not authenticated
ready to take orders.
```

Figure 4.4: Console window of the demonstrator. The interesting parts, which are described in the text, are marked.

5 Performance Evaluation

This chapter describes the performance evaluation process and shows the results of the measurements. It ends with the interpretation of the results.

5.1 Process

The evaluation starts with the measurement of the duration of the different commands. As the secure element is simulated the execution is slower than it would be with the real hardware. Furthermore, the computer also executes the user-side of the protocol and the task may be interrupted at any time by the operating system. Accounting for this many measurements are taken. The amount of measurements depends on the variation in execution-time.

The unauthenticated “Echo”-command does use much computation time on the secure element and can therefore be used as a reference for how long the communication and other overhead takes. With this data the calculation time of the secure element can be approximated.

As a second step the simulation can be set to show the CPU-cycles of the secure element. With the knowledge of the CPU-frequency a more accurate estimation is possible. This approximation must also be taken with care as the simulation does not need to be accurate as some commands need to be executed on the hardware but not in the simulation. These can only be found if the implementation is ported to a hardware element or emulation of the element on an FPGA.

5.2 Results

The first measurements are taken from the demonstrator. There the time is measured from the beginning of the command-transmission to the reader API until the end of the reception of the answer message. These measurements are affected by different processes on the computer. Furthermore, the simulation of the processor is usually slower than the execution would be at a hardware element.

Table 5.2 shows the results of the measurement of the timing of the commands that do not have to calculate any cryptographic operations. The results of the other commands are shown in Table 5.3. As expected these commands take one to two orders of magnitudes longer to compute. The commands with the "-a" added are executed while the user was authenticated. The tables show that this does not change the execution time for the "simple" commands. The only difference is in the "debug" command as only for this operation long execution times are expected which get omitted if the user is not authenticated.

In Table 5.1 the durations of the "echo"-command with some selected payload lengths are shown. These lengths are chosen as it is expected that the time consumption will rise linear with the payload length and these measurements span the possible range of lengths to show if the assumption is broken. Furthermore, measurements with 32 and 64 bytes are taken as these are common payload-lengths within the protocol.

characters/time [μs]	echo	echo-a
1	4148	4368
4	4548	5234
32	8240	13266
40	9329	15634
64	12472	22368
100	17173	32704
140	22648	44215
180	27840	55176
200	30371	61087
240	35773	72422

Table 5.1: Mean durations of the Echo command with different message lengths.

Every command has been measured 500 times to get reasonably good results. Figures B.1, B.2, B.3, and B.4 show the results of those measurements. In the first figure all commands are depicted. One can see that the debug command takes about ten times longer than the "simple" commands and that the computationally expensive functions are still one order of magnitude slower. The other three figures show the commands in a higher resolution.

The functions using cryptographic elements (generating the machine-side message, calculating the shared point, calculating the shared key) are also measured in CPU-cycles. The results are shown in Table 5.4. As these functions work with randomized values and the generation of random values can vary in time the exact amount of cycles can vary, but the approximated timing will still stay within that range.

time [μs]	echo	echo-a	send	send-a	user	user-a	machine	machine-a	disc	disc-a
min	4133	9009	20843	19976	10360	10423	9748	9826	9338	2024
average	4645	17354	21844	20740	11888	12121	11430	11490	10883	7801
max	8021	57838	24075	23981	14949	14875	15425	14216	14736	17046

Table 5.2: Time measurements of different commands that do not calculate much.

time [μs]	get	get-a	auth	auth-a	debug	debug-a
min	2840533	2835791	2840194	2835209	3831	196075
average	2880437	2879542	2879424	2876860	4288	207929
max	2925237	2937215	2891951	3948819	8145	236325

Table 5.3: Time measurements of different commands that perform cryptography.

Command	CPU cycles	Time [sec]
generateMachineMessage	$1.75 \cdot 10^6$	0.035
generateMachineSharedSecret	$1.76 \cdot 10^6$	0.035
generateKey	$1.3 \cdot 10^6$	0.026

Table 5.4: CPU-cycles and timing of cryptographic heavy functions.

5.3 Interpretation

The results of Figure 5.1, which is created from Table 5.1, can be understood in the way that the basic transmission of data takes in average eleven microseconds. One can also see that the transmission duration is linear proportional to the sent data. If the user is authenticated it takes approximately twice as long, because the same data needs to be returned as well. When looking at the trend line this becomes obvious. The gradient of the authenticated slope is $\frac{284\ \mu s}{character}$ whereas the unauthenticated one is $\frac{132\ \mu s}{character}$. The slight more than doubled slope-pitch of the authenticated command also expresses the time needed to copy the characters from the input to the output. This speed of transmission can be converted into a data rate of about $\frac{60\ kbit}{s}$. Also the constant time of the authenticated one is 135 μs longer than the unauthenticated one. This can be explained because of the additional operations like copying the data from input to output and resetting the transmission length.

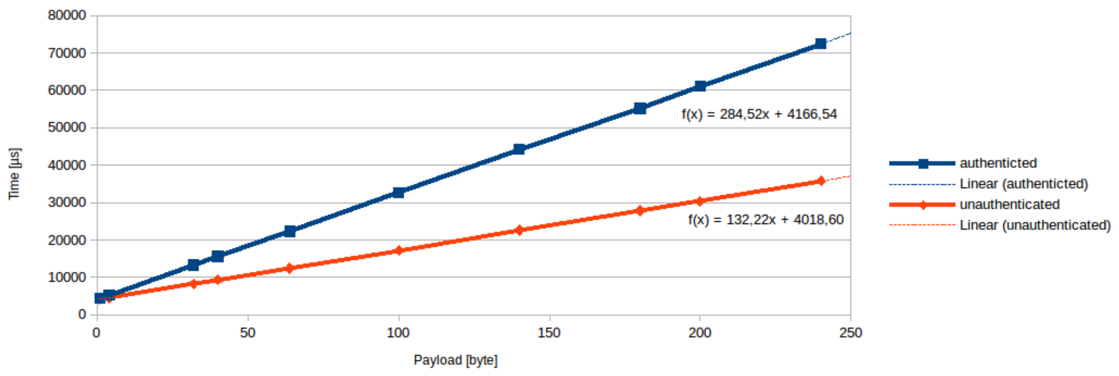


Figure 5.1: Comparison between different payloads of the authenticated and not-authenticated Echo-command.

This data can be used as baseline for measurement of the other commands as the echo-command does only do the minimum of required operations.

To compare the echo- against the send-command (sending the user-key to the machine and saving it) the unauthenticated echo with a payload of 64 bytes can be taken as baseline. This is because the unauthenticated echo sends the packet with the payload and receives an empty packet with “not authenticated” as return and the send-command sends a packet header with the point as payload and gets “good” as return.
The echo command uses about 10 ms of time for data transmission. The send command about 18 ms. This means that the execution of the command on the secure element uses about 8 ms.

A more complex command is the “get-command”. This sends an empty packet and receives 64 bytes of the machine-key. That means that the communication time is again comparable to the echo-command with 64 bytes of data. Furthermore, some data has to be saved to the NVM taking about 16 ms. The rest of the time is the calculation of the cryptographic functions. This means that these functions take up about 2.8 seconds of time in the simulation.

Another interesting function is the “debug”-command. This command can be compared to an unauthenticated echo-command with a payload of 32 bytes. In this function the session key of the user is calculated out of the previously received and calculated data. When assuming the transmission takes 7 ms and the complete process uses about 208 ms the resulting time to calculate the message digests is 201 ms in the simulation.

As these timings are very long for these kinds of operations the amount of processor-cycles was measured. With the knowledge of the processor speed the timing can be calculated. As the simulation may behave in a different way than the real hardware also these results need to be taken with a grain of salt. But they should be more accurate than the results of the first measurement.
When looking at the “get”-command it performs the “generateMachineMessage” function at it’s heart. This function would only take 0.035 seconds to compute on the hardware but uses approximately 2.8 seconds in the simulation. That tells us that the simulation speed is 80 times slower than real time for ECC-functions.

The “debug”-command uses the “calculateKey” function which takes 0.026 seconds in CPU-time and 0.201 seconds in the simulation making the simulation only 7.7 times slower for the hash-function. The interesting part of the “debug”-command is the difference of time between the authenticated and not authenticated user. This can be explained by the fact that only a authenticated user is allowed to see the debug output. This is because the session key, which is sent as reply to the command, cannot be calculated for an unauthenticated user. Thus the command returns with an error code.

As the time used to communicate is not likely to reduce when switching to a real hardware, but the calculation time is reduced drastically, as shown with the CPU-cycle measurements (Table 5.4), the strength of this algorithm gets visible. The implemented protocol only needs two messages to be sent. The protocol overhead can be reduced even more if the calculation of the machine-key is started after the user-key is received and the key is replied instead of the standard-“OK”-message.
Other protocols that also perform authentication establish a secured channel at first and perform the authentication step separately, requiring many more messages to be sent. With the use of SPAKE2 that time can be used to transmit the configuration data instead.

6 Conclusion and Future Work

In the following sections a conclusion of this thesis and the limitations of the implemented protocol are described. This chapter furthermore describes the possibilities if this work is combined with the works of my colleagues in the project and what further work can be done to improve the implemented protocol.

6.1 Conclusion

This thesis motivates with a future production facility built on the vision of "Industrie 4.0". That enables producers to manufacture high quality products that are fully customizable at low cost. In this scenario the production goods are transported by robots between the machines. These in turn are configurable by the robots or a main computer system that keeps track of all items. Robots are used to keep the item flow variable at all times. In such an environment a good way of configuring devices (machines or robots) is via the NFC technology. As sometimes a human engineer needs to configure a device this engineer must be authenticated against it and vice versa. Furthermore, the communication must be secured with encryption to counter a possible adversary.
Also the other visions from the smart home and the fields of the sciences are inseparable connected to the security features brought from an authenticated key exchange to weaken the threat caused by the connection of everyday appliances to the Internet.

This work's main focus is on the authentication and key exchange for the encryption in such a system. It gives a design that is based on the SPAKE2 algorithm from [AP05]. This design is then implemented in multiple steps and evaluated. This protocol's strength is in its little communication cost as it only needs two messages to perform an authenticated key exchange. After that the secured communication can be performed.

6.1.1 Limitations

The security of the SPAKE protocol is only given under the random oracle model. This proof of security is weaker than one that does not need a random oracle (standard model of cryptography).

Because of the limited memory of the security controller the amount of users is reduced to two. This can be improved by using strong cryptographic methods and storing the encrypted material on an external memory. That would add longer latency for the system as it may need to search through encrypted memory.

6.1.2 Combination With Other Work

When looking at the motivation (Section 1.2) and the works of my colleagues from the "IoSense" and "Semi40" projects one can see how this vision can become reality in the near future.

The other members of the "IoSense" team from TU Graz and Infineon Technologies Graz have been working on other methods for configuring smart sensors with NFC-enabled devices, implementing a method for securely storing the configuration data in a backend-server, transmitting it to a portable device with NFC-technology, and transmitting the secured data to the smart sensor. As a robot can be seen as a smart, portable sensor with attached actuators this method can be used as a second way of configuring the devices.

The colleagues from the "Semi40" project are also working on a way of securely connecting machines with each other in an "Industrie 4.0" fashion. This research can be used to build a prototypical smart production floor.

6.2 Future Work

The next development steps should be that the algorithm is tested on an FPGA development board. After those tests are concluded, and possible bugs are fixed, measures to get the algorithm running on the secure element can be taken. Further testing is then to be carried out.

Additionally more improvements and developments can be made. These include:

1. Replacement of the “debug” and “echo” commands with more useful commands.

2. Functionalities to receive and verify configuration data for a host machine can be implemented.

3. Combination of the developed algorithm with other works from the “IoSense” project to get closer to the “TrustWorSys” demonstrator.

4. The users can be redefined to have different access rights. Furthermore, a way to support more users can be implemented.

5. Evaluation of the implementation against other mechanisms that perform an authenticated key exchange.

6. With minor changes this algorithm can be used to authenticate machines against each other. It can be evaluated if this algorithm is usable for this kind of authentication.

Bibliography

[Abd14] Michel Abdalla. Password-based authenticated key exchange: an overview. In *International Conference on Provable Security*, pages 1–9. Springer, 2014.

[AP05] Michel Abdalla and David Pointchevall. Simple Password-Based Encrypted Key Exchange Protocols. In *Lecture Notes in Computer Science*, pages 191–208. Springer Science + Business Media, 2005.

[BM92] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: password-based protocols secure against dictionary attacks. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*. Institute of Electrical & Electronics Engineers (IEEE), 1992.

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated Key Exchange Secure against Dictionary Attacks. In *Advances in Cryptology — EUROCRYPT 2000*, pages 139–155. Springer Science + Business Media, 2000.

[BR00] Mihir Bellare and Phillip Rogaway. The AuthA Protocol for Password-Based Authenticated Key Exchange. In *IEEE P1363*, pages 136–3, 2000.

[BY15] Jonghyun Baek and Heung Youl Youm. Secure and Lightweight Authentication Protocol for NFC Tag Based Services. In *2015 10th Asia Joint Conference on Information Security*. Institute of Electrical & Electronics Engineers (IEEE), May 2015.

[CFA$^+$05] Henry Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography (Discrete Mathematics and Its Applications)*. Chapman and Hall/CRC, 2005.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, 22(6):644–654, nov 1976.

[DKA$^+$14] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

[DOW92] Whitfield Diffie, Paul C. Van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and cryptography*, 2(2):107–125, 1992.

[DS81] Dorothy E. Denning and Giovanni Maria Sacco. Timestamps in Key Distribution Protocols. *Commun. ACM*, 24(8):533–536, August 1981.

[GL06] Rosario Gennaro and Yehuda Lindell. A Framework for Password-based Authenticated Key Exchange1. *ACM Trans. Inf. Syst. Secur.*, 9(2):181–234, May 2006.

[GMS15] Jorge Granjal, Edmundo Monteiro, and Jorge Sa Silva. Security for the Internet of Things: A Survey of Existing Protocols and Open Research Issues. *IEEE Communications Surveys & Tutorials*, 17(3):1294–1312, 2015.

[HK99] Shai Halevi and Hugo Krawczyk. Public-key Cryptography and Password Protocols. *ACM Trans. Inf. Syst. Secur.*, 2(3):230–268, August 1999.

[HVM04] Darrel Hankerson, Scott A. Vanstone, and Alfred J. Menezes. *Guide to Elliptic Curve Cryptography.* Springer, 2004.

[JGQ16] Oliver Jensen, Mohamed Gouda, and Lili Qiu. A secure credit card protocol over NFC. In *Proceedings of the 17th International Conference on Distributed Computing and Networking - ICDCN '16.* Association for Computing Machinery (ACM), 2016.

[JZ15] Rong Jin and Kai Zeng. SecNFC: Securing inductively-coupled near field communication at physical layer. In *2015 IEEE Conference on Communications and Network Security (CNS).* Institute of Electrical & Electronics Engineers (IEEE), Sep 2015.

[Kar11] Stamatis Karnouskos. Stuxnet worm impact on industrial cyber-physical system security. In *IECON 2011 - 37th Annual Conference of the IEEE Industrial Electronics Society.* Institute of Electrical & Electronics Engineers (IEEE), nov 2011.

[KI03] Kazukuni Kobara and Hideki Imai. Pretty-Simple Password-Authenticated Key-Exchange Under Standard Assumptions. Cryptology ePrint Archive, Report 2003/038, 2003. `http://eprint.iacr.org/2003/038`.

[Kip06] Rudolf Kippenhahn. *Verschlüsselte Botschaften: Die Geheimschrift Des Julius Caesar, Geheimschriften Im I. Und II. Weltkrieg, Das Codebuch Des Papstes, Enigma.* Nikol, Hamburg, 2006.

[Kra03] Hugo Krawczyk. SIGMA: The 'SIGn-and-MAc'approach to authenticated Diffie-Hellman and its use in the IKE protocols. In *Annual International Cryptology Conference*, pages 400–425. Springer, 2003.

[LB16] Christian Lesjak and Eugen Brenner. Securing Smart Service Connectivity for Industrial Equipment Maintenance - A Case Study. 2016.

[Les16] Christian Lesjak. *Secure Smart Service Connectivity for Industrial Equipment Maintenance.* PhD thesis, Graz University of Technology, 2016.

[LHH+15] Christian Lesjak, Daniel Hein, Michael Hofmann, Martin Maritsch, Andreas Aldrian, Peter Priller, Thomas Ebner, Thomas Ruprechter, and Gunther Pregartner. Securing smart maintenance services: Hardware-security and TLS for MQTT. In *2015 IEEE 13th International Conference on Industrial Informatics (INDIN).* Institute of Electrical & Electronics Engineers (IEEE), Jul 2015.

[LHW15] Christian Lesjak, Daniel Hein, and Johannes Winter. Hardware-security technologies for industrial IoT: TrustZone and security controller. In *IECON 2015 - 41st Annual Conference of the IEEE Industrial Electronics Society.* Institute of Electrical & Electronics Engineers (IEEE), Nov 2015.

[LMQ+03] Laurie Law, Alfred Menezes, Minghua Qu, Jerry Solinas, and Scott Vanstone. An Efficient Protocol for Authenticated Key Agreement. *Designs, Codes and Cryptography*, 28(2):119–134, 2003.

[LRB+14] Christian Lesjak, Thomas Ruprechter, Holger Bock, Josef Haid, and Eugen Brenner. ESTADO - Enabling Smart Services for Industrial Equipment through a Secured, Transparent and Ad-hoc Data Transmission Online. *The 9th International Conference for Internet Technology and Secured Transactions (ICITST-2014)*, 2014.

[LRH^{+}14] Christian Lesjak, Thomas Ruprechter, Josef Haid, Holger Bock, and Eugen Brenner. A secure hardware module and system concept for local and remote industrial embedded system identification. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. Institute of Electrical & Electronics Engineers (IEEE), Sep 2014.

[Mer79] Ralph Charles Merkle. *Secrecy, Authentication, And Public Key Systems*. PhD thesis, Stanford University, Stanford, CA, 1979.

[PP09] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer, 2009.

[PZ00] Mohammad Peyravian and Nevenko Zunic. Methods for Protecting Password Transmission. *Computers & Security*, 19(5):466 – 469, 2000.

[TWMP07] David Taylor, Tom Wu, Nikos Mavrogiannopoulos, and Trevor Perrin. Using the Secure Remote Password (SRP) Protocol for TLS Authentication. Technical report, nov 2007.

[Wei91] Mark Weiser. The computer for the 21st century. *Scientific american*, 265(3):94–104, 1991.

A Listings

A.1 Prototype

```
void SPAKE_test_random(unsigned runs, bool runTillFound=false){
  User** users = new User*[0];
  Machine** machines = new Machine*[0];
  unsigned numUsers = 0;
  unsigned numMachines = 0;

  srand(time(NULL));
  unsigned char percent = 0;
  bool check=(!runTillFound);
  do{
    if(runTillFound==false){
    printf("|");
    for(int i=0;i<98;i++){printf("_");}
    printf("|\n");
    }
  for(unsigned z=0;z<runs;z++){
    if(runTillFound==false){
      if((int)(((float)z/(float)runs)*100)>percent){
      percent=(unsigned char)(((float)z/(float)runs)*100);
      printf("*");
      if(percent==99){printf("\n");}
      fflush(stdout);
      }
    }
    unsigned r = rand()%14;
    switch(r){
    case 0: // add User
    {
      User** tu = new User*[numUsers];
      for(unsigned i=0;i<numUsers;i++){
        tu[i]=users[i];
      }
      delete[] users;
      users = new User*[numUsers+1];
      for(unsigned i=0;i<numUsers;i++){
        users[i]=tu[i];
      }
      delete[] tu;
      char* u=new char[21];
      gen_random(u,20);
      char* p=new char[11];
      gen_random(p,10);
      User* nu = new User(u,p);
      users[numUsers] = nu;
      numUsers++;
      delete[] p;
      break;
    }
    case 1: //remove User
    {
      if(numUsers>0){
      unsigned j=rand()%numUsers;
      //printf("  %d\n",j);
      User** tu = new User*[numUsers];
      for(unsigned i=0;i<numUsers;i++){
        tu[i]=users[i];
      }
      delete[] users;
      users = new User*[numUsers-1];
      for(unsigned i=0;i<numUsers;i++){
        if(i<j){users[i]=tu[i];}
        if(i==j){
          delete[]tu[i]->getUsername();
          delete(tu[i]);}
        if(i>j){users[i-1]=tu[i];}
      }
      delete[] tu;
      numUsers--;
      }
      break;
    }
    case 2: // add Machine
    {
      Machine** tm = new Machine*[numMachines];
      for(unsigned i=0;i<numMachines;i++){
        tm[i]=machines[i];
      }
      delete[] machines;
      machines = new Machine*[numMachines+1];
      for(unsigned i=0;i<numMachines;i++){
        machines[i]=tm[i];
      }
      delete[] tm;
      char* m=new char[21];
      gen_random(m,20);
      Machine* nm = new Machine(m);
      machines[numMachines] = nm;
      numMachines++;
      break;
    }
    case 3: //remove Machine
    {
      if(numMachines>0){
      unsigned j=rand()%numMachines;
      Machine** tm = new Machine*[numMachines];
      for(unsigned i=0;i<numMachines;i++){
        tm[i]=machines[i];
      }
      delete[] machines;
      machines = new Machine*[numMachines-1];
      for(unsigned i=0;i<numMachines;i++){
        if(i<j){machines[i]=tm[i];}
        if(i==j){delete[]tm[i]->getMachinename();delete(tm[i]);}
        if(i>j){machines[i-1]=tm[i];}
      }
      delete[] tm;
      numMachines--;
      //printf("  %d\n",numMachines);
      }
      break;
    }
    case 4: // change Userpassword
    {
      if(numUsers>0){
      unsigned j=rand()%numUsers;
      char* p=new char[21];
      gen_random(p,20);
      users[j]->changePassword(p);
      delete[] p;
      }
      break;
    }
    case 5: // set Connection
    {
      if(numUsers>0 && numMachines>0){
        unsigned i=rand()%numUsers;
        unsigned j=rand()%numMachines;
        unsigned k=rand()%2;
        users[i]->setConnection(machines[j]->getMachinename(),k==0?NID_secp224r1:NID_secp224k1);
      }
      break;
    }
    case 6: //initConnection
    {
      if(numUsers>0 && numMachines>0){
        unsigned i=rand()%numUsers;
        unsigned j=rand()%numMachines;
        users[i]->initConnection(machines[j]->getMachinename());
      }
      break;
    }
    case 7: // addUser
    {
      if(numUsers>0 && numMachines>0){
        unsigned i=rand()%numUsers;
        unsigned j=rand()%numMachines;
        machines[j]->addUser(users[i]->getUsername(), users[i]->getHashedPassword());
      }
      break;
    }
    case 8: // setUserCurve
    {
      if(numUsers>0 && numMachines>0){
        unsigned i=rand()%numUsers;
        unsigned j=rand()%numMachines;
        machines[j]->setUserCurve(users[i]->getUsername(),
               users[i]->getCurve(machines[j]->getMachinename()));
      }
      break;
    }
    case 9: // performExchange
    {
      if(numUsers>0 && numMachines>0){
        unsigned i=rand()%numUsers;
        unsigned j=rand()%numMachines;
        int work=performExchange(users[i],machines[j]);
        if(work){
          char* a = new char[10];
          gen_random(a,9);
          check = printKeys(users[i],machines[j],a);
          delete[] a;
        }
      }
      break;
    }
    case 10: // deleteConnection
    {
      if(numUsers>0 && numMachines>0){
        unsigned i=rand()%numUsers;
        unsigned j=rand()%numMachines;
        users[i]->deleteConnection(machines[j]->getMachinename());
      }
      break;
    }
    case 11: // updateUserPwd
    {
      if(numUsers>0 && numMachines>0){
        unsigned i=rand()%numUsers;
        unsigned j=rand()%numMachines;
        machines[j]->updateUserPwd(users[i]->getUsername(),users[i]->getHashedPassword());
      }
      break;
    }
    case 12: // deleteUser
    {
      if(numUsers>0 && numMachines>0){
        unsigned i=rand()%numUsers;
        unsigned j=rand()%numMachines;
        machines[j]->deleteUser(users[i]->getUsername());
      }
      break;
    }
    case 13:
    {
      if(numUsers>0 && numMachines>0){
        unsigned i=rand()%numUsers;
        unsigned j=rand()%numMachines;
        unsigned k=rand()%2;
        users[i]->changeCurve(machines[j]->getMachinename(),k==0?NID_secp224r1:NID_secp224k1);
      }
      break;
    }
    default:
    {
      printf("%d\n",z);
      break;
    }
    }
  }
  }while(!check);

  for(unsigned i=0;i<numUsers;i++){
    delete[] users[i]->getUsername();
    delete users[i];
    users[i]=NULL;
  }
  delete[] users;
  for(unsigned i=0;i<numMachines;i++){
    delete[] machines[i]->getMachinename();
    delete machines[i];
    machines[i]=NULL;
  }
  delete[] machines;
}
```

Listing A.1: Random valid commands are generated and sent to the machines and users.

```
EC_POINT* Machine::generateKey(char* username, EC_POINT* X_s){
  if(numUsers>0){
    for(unsigned int i=0; i<(numUsers);i++){
      if(users[i].username==username){
        if(users[i].curve==0){
          printf("Machine:generateKey Error: No Curve defined\n");
          return NULL;
        }
        EC_GROUP* c=NULL;
        if((c=EC_GROUP_new_by_curve_name(users[i].curve))==NULL){
          printf("Machine:generateKey Error: error generating group\n");
          EC_GROUP_free(c);
          return NULL;
        }
        if(0==EC_POINT_copy(users[i].X_s,X_s)){
          printf("Machine:generateKey Error: copy of X_s returned with an error\n");
          EC_POINT_free(users[i].X_s);
          users[i].X_s=EC_POINT_new(c);
          EC_GROUP_free(c);
          return NULL;
        }

        EC_POINT* G = EC_POINT_new(c);
        if(0==EC_POINT_copy(G,EC_GROUP_get0_generator(c))){
          printf("Machine:generateKey Error: copy of G returned with an error\n");
          EC_POINT_free(users[i].X_s);
          users[i].X_s=EC_POINT_new(c);
          EC_POINT_free(G);
          EC_GROUP_free(c);
          return NULL;
        }

        EC_POINT* Y = EC_POINT_new(c);
        if (0==EC_POINT_mul(c,Y,NULL,G,users[i].y,NULL)){
          printf("Machine:generateKey Error: Mul1 returned with an error\n");
          EC_POINT_free(users[i].X_s);
          users[i].X_s=EC_POINT_new(c);
          EC_POINT_free(G);
          EC_POINT_free(Y);
          EC_GROUP_free(c);
          return NULL;
        }

        EC_POINT* N = EC_POINT_new(c);
        getPoint(machinename, c,G,N);
        BIGNUM* pw = BN_new();
        getBN(users[i].hashedpassword,pw);
        EC_POINT* Yi = EC_POINT_new(c);
        if (0==EC_POINT_mul(c,Yi,NULL,N,pw,NULL)){
          printf("Machine:generateKey Error: Mul2 returned with an error\n");
          EC_POINT_free(users[i].X_s);
          users[i].X_s=EC_POINT_new(c);
          EC_POINT_free(G);
          EC_POINT_free(Y);
          EC_POINT_free(N);
          EC_POINT_free(Yi);
          BN_free(pw);
          EC_GROUP_free(c);
          return NULL;
        }

        EC_POINT* Y_s = EC_POINT_new(c);
        if(0==EC_POINT_add(c,Y_s,Yi,Y,NULL)){
          printf("Machine:generateKey Error: Add1 returned with an error\n");
          EC_POINT_free(users[i].X_s);
          users[i].X_s=EC_POINT_new(c);
          EC_POINT_free(G);
          EC_POINT_free(Y);
          EC_POINT_free(N);
          EC_POINT_free(Yi);
          BN_free(pw);
          EC_GROUP_free(c);
          return NULL;
        }
        if(0==EC_POINT_copy(users[i].Y_s,Y_s)){
          printf("Machine:generateKey Error: Copy of Y_s returned with an error\n");
          EC_POINT_free(users[i].X_s);
          users[i].X_s=EC_POINT_new(c);
          EC_POINT_free(users[i].Y_s);
          users[i].Y_s=EC_POINT_new(c);
          EC_POINT_free(G);
          EC_POINT_free(Y);
          EC_POINT_free(N);
          EC_POINT_free(Yi);
          BN_free(pw);
          EC_GROUP_free(c);
          return NULL;
        }

        EC_POINT_free(N);
        EC_POINT_free(Y);
        EC_POINT_free(Yi);

        // Second part
        EC_POINT* M = EC_POINT_new(c);
        getPoint(users[i].username, c,G,M);
        EC_POINT* Ki = EC_POINT_new(c);
        if (0==EC_POINT_mul(c,Ki,NULL,M,pw,NULL)){
          printf("Machine:generateKey Error: Mul3 returned with an error\n");
          EC_POINT_free(G);
          EC_POINT_free(M);
          EC_POINT_free(Ki);
          BN_free(pw);
          EC_GROUP_free(c);
          return NULL;
        }
        if(0==EC_POINT_invert(c,Ki,NULL)){
          printf("Machine:generateKey Error: Invert returned with an error\n");
          EC_POINT_free(G);
          EC_POINT_free(M);
          EC_POINT_free(Ki);
          BN_free(pw);
          EC_GROUP_free(c);
          return NULL;
        }
        EC_POINT* Ki2 = EC_POINT_new(c);
        if(0==EC_POINT_add(c,Ki2,Ki,users[i].X_s,NULL)){
          printf("Machine:generateKey Error: Add2 returned with an error\n");
          EC_POINT_free(G);
          EC_POINT_free(M);
          EC_POINT_free(Ki);
          EC_POINT_free(Ki2);
          BN_free(pw);
          EC_GROUP_free(c);
          return NULL;
        }

        if (0==EC_POINT_mul(c,users[i].K,NULL,Ki2,users[i].y,NULL)){
          printf("Machine:generateKey Error: Mul4 returned with an error\n");
          EC_POINT_free(G);
          EC_POINT_free(M);
          EC_POINT_free(Ki);
          EC_POINT_free(Ki2);
          BN_free(pw);
          EC_GROUP_free(c);
          return NULL;
        }
```

```
        getRandBN(users[i].y);
        EC_POINT_free(Ki2);
        EC_POINT_free(Ki);
        EC_POINT_free(M);
        EC_POINT_free(G);
        BN_free(pw);
        EC_GROUP_free(c);
        return Y_s;
      }
    }
    printf("Machine:generateKey Error: Username %s not found\n",username);
    return NULL;
  }
  printf("Machine:generateKey Error: No users connected\n");
  return NULL;
}
```

Listing A.2: Machine part of the key exchange. The own public key is generated in the beginning, then the shared secret is calculated.

A.2 Transition

```
UINT8 getBN(                     CLONG cl                      // [in]
                         , BIGNUM* bn                    // [out]
                 )
{
  unsigned char data[64+1];
  UINT16 i=0;
  UINT8 t[3];
  UINT8 te;
  for(i=0;i<65;i++){data[i]=0;}
  t[0]=t[1]=t[2]=0;
  for(i=0;i<cl.BitLength/8;i+=2)
  {
    te=cl.Data[i];
    sprintf(t,"%02X",(unsigned char)te);
    data[i]=t[0];
    data[i+1]=t[1];
  }
  BN_bin2bn(cl.Data,cl.BitLength/8,bn);
  return 0;
}

UINT8 getCLONG(                  BIGNUM* bn          // [in]
                 , PCLONG cl           // [out] // empty on call
                 )
{
  UINT16 bnlen = getDataLength(bn);
  if(cl->BytesAllocated < getDataLength(bn))
  {
    printf("error\n");
    return 1;
  }
  BN_bn2bin(bn,cl->Data);
  if (bnlen == 0) {bnlen = 1;}
  cl->BitLength = bnlen*8;
  return 0;
}
```

Listing A.3: Conversion between CLONG and BIGNUM.

The AFFINEPOINT consists of two CLONGs. This conversion is not performed in a function.

```
int getECCCPARAMFromCurve(PECCCPARM ecccp, int curve){
  BIGNUM *x,*y,*a,*b,*p,*order;
  EC_GROUP *c;
  EC_POINT *G;
  c = EC_GROUP_new_by_curve_name(curve);
  G = EC_POINT_new(c);
  if(!ecccp || !ecccp->Basepoint.X.Data)
  {return 0;}
  EC_POINT_copy(G,EC_GROUP_get0_generator(c));
  x = BN_new();
  y = BN_new();
  EC_POINT_get_affine_coordinates_GFp(c,G,x,y,NULL);
  getCLONG(x,&ecccp->Basepoint.X);
  getCLONG(y,&ecccp->Basepoint.Y);
  BN_free(x);BN_free(y);

  ecccp->Characteristic = 0;

  a = BN_new();
  b = BN_new();
  p = BN_new();
  EC_GROUP_get_curve_GFp(c,p,a,b,NULL);
  getCLONG(a,&ecccp->CoefA);
  getCLONG(b,&ecccp->CoefB);
  getCLONG(p,&ecccp->Modulus);
  BN_free(a);BN_free(b);BN_free(p);

  order = BN_new();
  EC_GROUP_get_order(c,order,NULL);
  getCLONG(order,&ecccp->BasepointOrder);
  BN_free(order);
  return 1;
}
```

Listing A.4: Obtaining the ECCCPARM data from the OpenSSL curve identifier.

A.3 Demonstrator

```
void demo()
{
  SPAKE_User* u = getUser();
  SPAKE_User* u2 = getUser();
  AFFINEPOINT data;
  CLONG sessionkey;

  /*primary names and passwords*/
  char machinename[61] = "GenericMachine";
  char name[61] = "GenericUser";
  char password[61] = "GenericPassword";
  /*communication buffer*/
  char timeout[]="<call method=\"SetProperty\"><CommTimeout>3500</CommTimeout></call>";
  char inbuffer[MESSAGESIZE]={0x00};
  /*variables for running*/
  int run = 1;
  char userbuffer[USERSIZE] = {0x00};
  char tempbuffer[USERSIZE] = {0x00};
  int state0 = 0,state1=0;

  int debug=0;

  viewTooltip();

  setUsername(u,(UINT8*)"");
  setPassword(u,(UINT8*)"");
  setUsername(u2,(UINT8*)name);
  setPassword(u2,(UINT8*)password);
  allocAffinepoint(&data,32); //helper
  allocClong(&sessionkey,32);
  clearBuffer(inbuffer,MESSAGESIZE);
  if(!connectToCard())
  {
    printf("Could not connect to Reader!\n");
  }

  /*reset the card*/

  printf("starting card...\n");
  resetcommand(inbuffer,&state0,&state1,debug);
  clearBuffer(inbuffer,MESSAGESIZE);
  transceive(timeout,strlen(timeout),inbuffer,NULL);

  while(run)
  {
    clearBuffer(inbuffer,MESSAGESIZE);
    clearBuffer(userbuffer,USERSIZE);
    getMoreData("ready to take orders.",1,userbuffer,USERSIZE);
    switch (userbuffer[0])
    {
    case '0': // exit program
      getMoreData("Exit program? y/n",1,userbuffer,USERSIZE);
      if(userbuffer[0]=='y')
        run=0;
      else if(userbuffer[0]=='n')
        break;
      else
        printf("no clear answer - assumed no\n");
      break;
    case 'r': // reset card
      resetcommand(inbuffer,&state0,&state1,debug);
      break;
    case 'e': // echo command
      echocommand(userbuffer,tempbuffer,inbuffer,debug);
      break;
    case 's': // calculate X_s and send it
      sendcommand(userbuffer,tempbuffer,inbuffer,&state0,&state1,u,u2,debug);
      break;
    case 'g': // request Y_s - key gets calculated
      getcommand(userbuffer,tempbuffer,inbuffer,machinename,&state0,&state1,u,u2,debug);
      break;
    case 'a':
      authenticatecommand(userbuffer,tempbuffer,inbuffer,machinename,&state0,&state1,u,u2,debug);
      break;
    case 'd':
      debugcommand(userbuffer,inbuffer,debug);
      break;
    case 'm':
      setmachinecommand(userbuffer,tempbuffer,inbuffer,machinename,&state0,&state1,debug);
      break;
    case 'u':
      setusercommand(userbuffer,tempbuffer,inbuffer,&state0,&state1,u,u2,debug);
      break;
    case 'x':
      disconnectcommand(userbuffer,inbuffer,&state0,&state1,debug);
      break;
    case 'c':
      printf("Machinename:\n%s\n",machinename);
      printf("User 0:\n");
      printf("name:\n%s\npwd:\n%s\n",u->username,u->password);
      printf("secret:\n");printClong(u->connections[0]->secret);printf("\n");
      printf("X_s:\n");printAffinepoint(u->connections[0]->X_s);
      printf("Y_s:\n");printAffinepoint(u->connections[0]->Y_s);
      printf("K  :\n");printAffinepoint(u->connections[0]->K);
      printf("\nUser 1:\n");
      printf("name:\n%s\npwd:\n%s\n",u2->username,u2->password);
      printf("secret:\n");printClong(u2->connections[0]->secret);printf("\n");
      printf("X_s:\n");printAffinepoint(u2->connections[0]->X_s);
      printf("Y_s:\n");printAffinepoint(u2->connections[0]->Y_s);
      printf("K  :\n");printAffinepoint(u2->connections[0]->K);

      getMoreData("change? y/n",1,userbuffer,USERSIZE);
      if(userbuffer[0]=='y'){
        getUserData("What is the name of the Machine?",1,machinename,61);
        getUserData("What is the name of User 0?",1,name,61);
        getUserData("What is the password of User 0?",1,password,61);
        setUsername(u,(UINT8*)name);
        setPassword(u,(UINT8*)password);
        getUserData("What is the name of User 1?",1,name,61);
        getUserData("What is the password of User 1?",1,password,61);
        setUsername(u2,(UINT8*)name);
        setPassword(u2,(UINT8*)password);
      }
      else if(userbuffer[0]=='n')
        break;
      else
        printf("no clear answer - assumed no\n");
      break;
    case 'i':
      authdis(userbuffer, inbuffer, debug);
      break;
    case 't':
      printf("Debug level\n0: no debug\n1: debug output enabled\nn: perform command n times\n");
      getUserData("Set the new value for the debug variable:",1,userbuffer,61);
      debug=atoi(userbuffer);
      break;
    case 'o':
      setCommunicationtimeout(userbuffer,inbuffer);
```

```
            break;
        default:
            viewTooltip();
            break;
        }

    }

    CloseConnection();
    freeClong(&sessionkey);
    freeAffinepoint(&data);
    deleteUser(u);
}
```

Listing A.5: Main function of the demonstrator application. The "classes" of the users are instantiated here and all commands from the user are handled in this function.

B
Figures

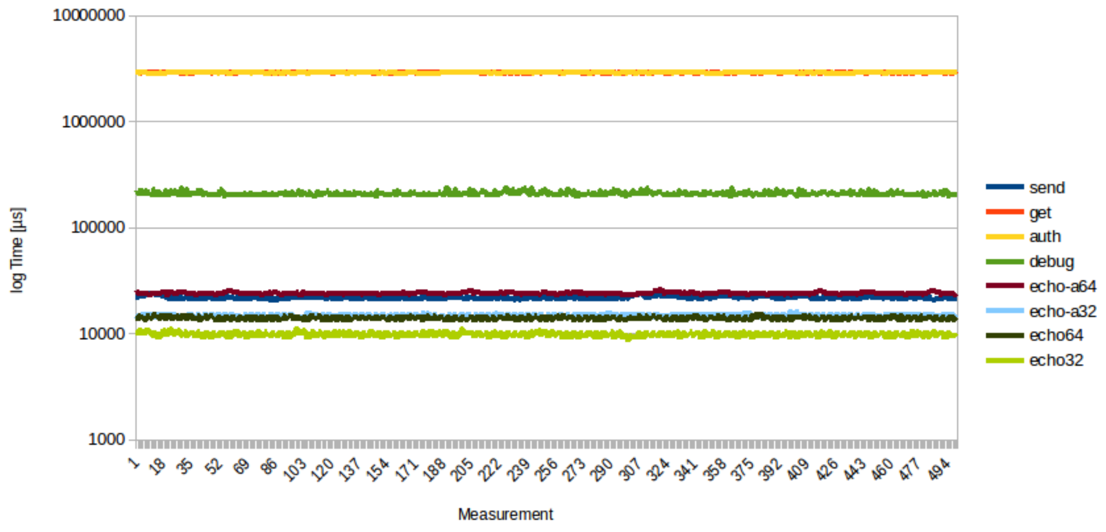


Figure B.1: Timings of 500 measurements of different commands.

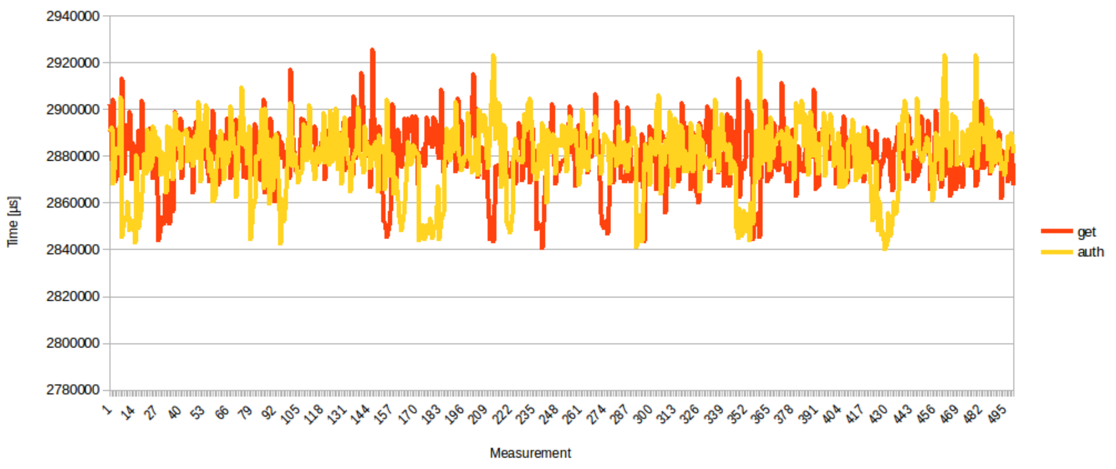


Figure B.2: Timings of 500 measurements of the challenging commands.

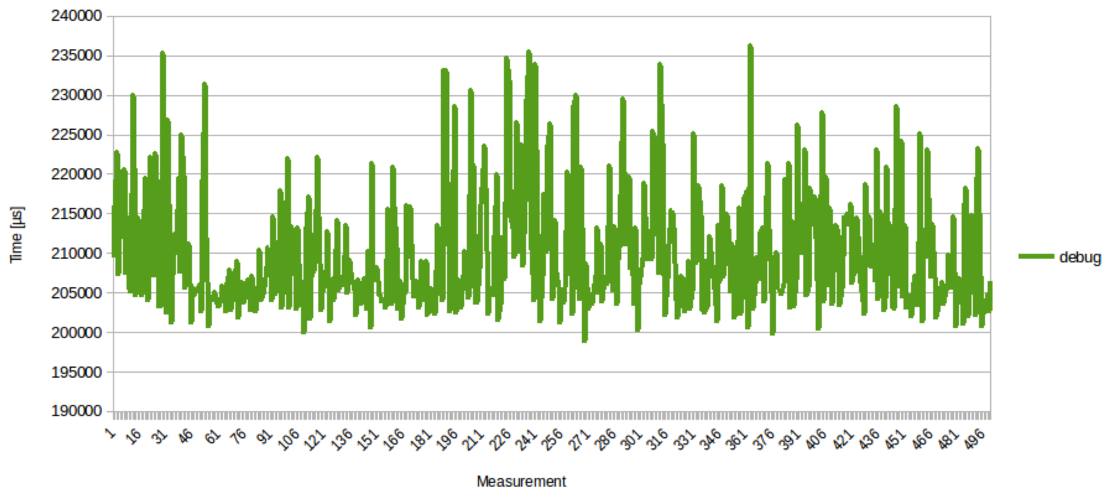


Figure B.3: Timings of 500 measurements of the debug-command.

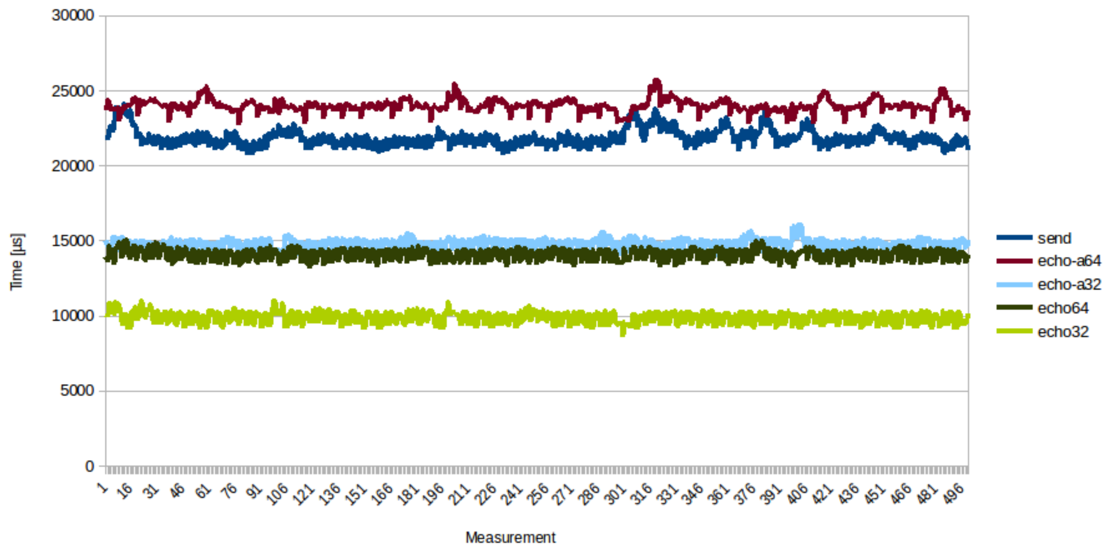


Figure B.4: Timings of 500 measurements of the simple commands.