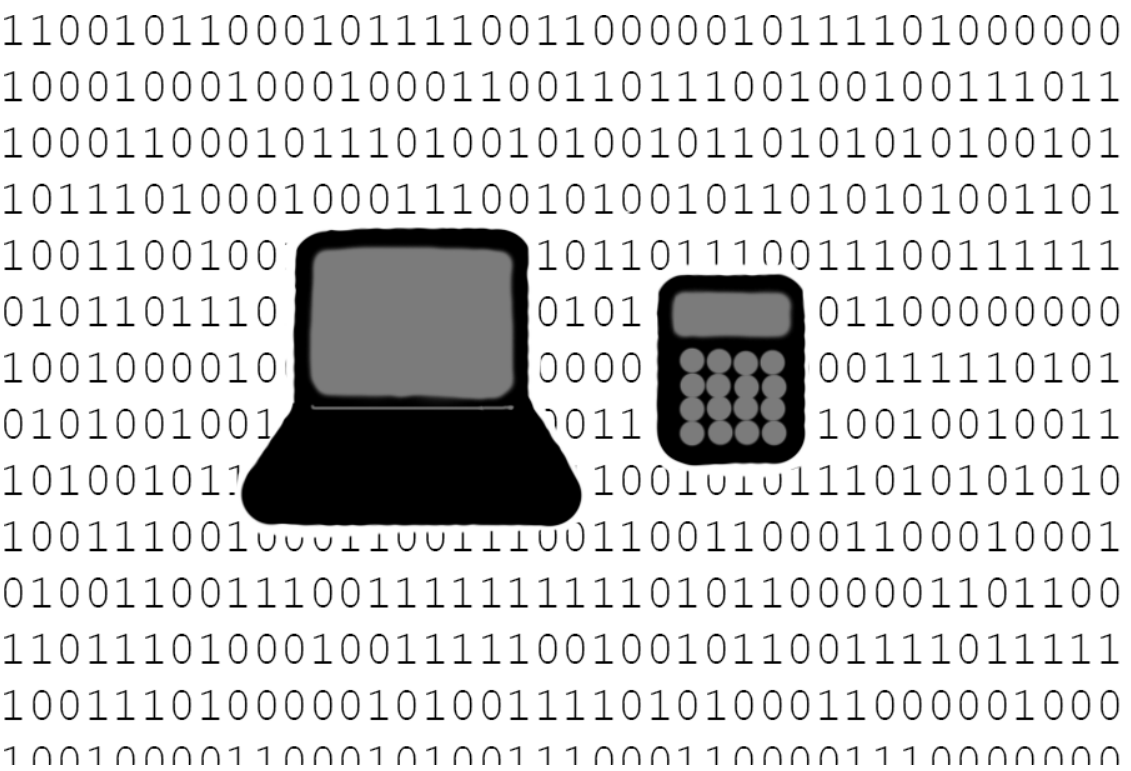
CHALMERS | GÖTEBORGS UNIVERSITET



# Open Source Security Token for Linux

## A more secure login authentication model

Bachelor's thesis in Computer Science and Engineering

JOHAN BEN MOHAMMAD
ADAM FREDRIKSSON
CHRISTOFFER MATHIESEN
ELIOT ROXBERGH
GUSTAV ÖRTENBERG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Bachelor's thesis 2017

# Open Source Security Token for Linux

A more secure login authentication model

JOHAN BEN MOHAMMAD
ADAM FREDRIKSSON
CHRISTOFFER MATHIESEN
ELIOT ROXBERGH
GUSTAV ÖRTENBERG


CHALMERS
UNIVERSITY OF TECHNOLOGY


UNIVERSITY OF
GOTHENBURG

Department of Computer Science and Engineering
*Division of Computer Engineering*
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

**Open Source Security Token for Linux**
A more secure login authentication model

Johan Ben Mohammad
Adam Fredriksson
Eliot Roxbergh
Christoffer Mathiesen
Gustav Örtenberg



Supervisor: Lars Svensson
Examiner: Arne Linde

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden 2017

Open Source Security Token for Linux
A more secure login authentication model
JOHAN BEN MOHAMMAD
ADAM FREDRIKSSON
CHRISTOFFER MATHIESEN
ELIOT ROXBERGH
GUSTAV ÖRTENBERG
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

# Abstract

The project investigates and implements a two-factor authentication system utilizing the RSA cryptography scheme. The system consists of an FPGA security token and a PAM module for Linux. Two similar solutions were made, one air-gapped with a shorter key (version A), whereas the other communicated over USB (version B). The cryptography module in the FPGA supports no more than 512-bit RSA and is the greatest area of improvement - since a longer key would provide more security and still be supported by the rest of the system. Additionally, interesting follow-up projects could be to explore quantum safe cryptography schemes - especially if to be used for decades to come. Altogether, the prototype created is a basic, yet fully functional, two-factor system with no obvious security flaws if deployed correctly. The project is released as open source under the BSD license.

# Sammanfattning

Detta projekt undersöker och implementerar ett tvåfaktorsautentiseringssystem som använder sig av RSA kryptografi. Systemet består av en koddosa och en Linux PAM-modul. Två liknande lösningar skapades, där en lösning har en kort nyckel och ej kopplas direkt till datorn (version A), medan den andra lösningen använder USB-kommunikation (version B). Den befintliga krypteringskärnan kan maximalt stödja 512 bitars RSA-nycklar, vilket är systemets största förbättringspotential. Ty längre nycklar skulle kunna hanteras av systemet i övrigt, samt ge en förhöjd säkerhet av tvåfaktorslösningen. Vidare kan intressanta uppföljningsprojekt vara att undersöka kvantsäker kryptografi, speciellt om projektet skall användas decennier framöver. Sammanfattningsvis är prototypen ett grundläggande, men fullt fungerande, tvåfaktorssystem utan självklara säkerhetsbrister givet att systemet är konfigurerat korrekt. Projektet i sin helhet släpps som öppen källkod, licensierad under BSD.

**Keywords:** FPGA, Linux, Open source, OpenSSL, PAM, RSA, Security, Two-factor, VHDL.

# Glossary

**AFL** - American Fuzzy Looper, a program that tests different inputs to find bugs
**API** - Application Programming Interface
**APT** - A very dedicated attacker with a specific goal
**ASCII** - American Standard Code for Information Interchange
**Baud** - Bits per second a serial port can transfer
**Binary (file)** - Compiled code run by an executable device
**Bitstream file** - A file of a binary sequence, can be used to program an FPGA
**BRAM** - Block Random Access Memory
**BSD-license** - Open source software license
**DMZ** - Demilitarized Zone, isolated section of a network. Used to separate public from private features of a network
**FIFO** - First-In First-Out (memory)
**FPGA** - Field-Programmable Gate Array, reprogrammable hardware
**Fuzzing** - Finding Bugs by testing a lot of random inputs
**GCC** - GNU Compiler Collection. A compiler included in GNU utilities
**GNU utilities** - Basic software included in most Linux distributions
**Hash** - In this report's context: result of a one-way function that makes passwords safer to store
**IPS** - Intrusion Prevention System used on computer networks
**ISE** - Development tool for Xilinx FPGA:s
**JTAG** - Joint Test Action Group, a debugging interface
**Key pair** - A pair of cryptography keys (a private and a public)
**LUT** - Look Up Table
**on-chip general purpose FPGA logic** - Premade circuit on an FPGA that can be used by the design for different functionallities
**PAM** - Pluggable Authentication Module
**Pseudo-random** - Statistically random
**Quantum computer** - A computer using quantum bits for which factorization is trivial
**RSA** - An asymmetric cryptography scheme
**Scrum** - An agile software development framework
**Sign** - Encrypt a message with the private key
**Slice LUT** - Primary programmable component in an FPGA
**SOC** - System On a Chip
**SSH** - Secure Shell used for remote login
**Top module** - The VHDL file that integrates all needed submodules, implements logic to control them and defines I/O
**Verify** - Decrypt a message with the public key (encrypted by the private key)
**Version A** - Our first version. Uses the user as communication link
**Version B** - Our second version. Uses USB for communication, more secure
**VHDL** - Very high speed integrated circuit Hardware Description Language

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Authentication in computer security is when a user is prompted to verify herself to a system in order to access information or perform some type of action. This is necessary in order to keep information private, to restrict permissions and to make users accountable for their actions when using a system. Authentication in most computer systems today is done by password, which is meant to be unique and known to one user only. Unfortunately, passwords are frequently discovered by unauthorized individuals, primarily since humans are poor in choosing and storing passwords [42]. A leaked password may lead to devastating effects for an individual or a company, not limited to economic consequences. A solution for a more secure login is a second authentication step for business critical systems - so-called two-factor authentication. Two-factor authentication is growing in popularity and is often provided as a smartphone application or as a biometric reader. However, since smartphones often are attacked [21] and biometrics can be faked [29, 43], the second login-factor for this project will consist of a custom physical security token.

This project's software will be released under the BSD-3-Clause license. Thus interested individuals and companies can use the code in order to produce their own tokens or use it as a building block for other projects. Furthermore, sharing the code enables external audits to confirm and improve the security of the project.

## 1.1 Aim

The project aims to find out how secure a challenge-response system based on RSA cryptography can be. A challenge-response system on an FPGA will be designed that implements RSA cryptography. Additionally, the system will be used to extend the login authentication on a Linux system. Since human interaction is required in the system, user-friendliness also needs to be an important factor during the analysis.

## 1.2 Scope

The project will focus on developing a non-portable prototype of a security token, with the software needed to extend the login authentication functionality in Linux via PAM. It is outside the scope of this project to investigate security issues that could arise in parts of the system (when integrated with a solution) that is not developed by the project group, e.g. servers and operating systems.

The project code will be released as an open source software, allowing its users to audit and change the code themselves. It is of utmost importance to release the code of any software using cryptography openly - giving transparency to its users, and proving that no backdoor, calling-home functionality or other unwanted functionality is included.

# 2 Theory and Technical Background

This chapter describes central concepts and technologies. Briefly introducing important topics for the project and the developed product.

## 2.1 Two-Factor Authentication

Two-factor authentication is a method to make computer authentication more secure by introducing an extra step to the login procedure. There are three different categories of authentication: “something you know", “something you have", and “something you are" (e.g. password, credit card, and fingerprint respectively) [5].

When designing a two-factor authentication system, two authentication methods are chosen. It is common to use a password as well as a device that the person possess [44].

## 2.2 Security

Correctly implemented two-factor authentication helps to protect the user against, amongst others: bad passwords, powerful brute-force attacks, and passwords observed by a third-party. The prevalence of bad passwords in organizations has prompted a plethora of two-factor authentication solutions. However, not all two-factor authentication systems are equal, sadly many solutions are proprietary or connected to the Internet - possibly weakening the security [15, 25]. Connecting a device to the Internet increases the attack surface greatly. Furthermore, proprietary solutions require utmost trust in its creators since they are the only ones able to analyze the source code.

A security token only protects against password attacks, e.g. brute-force attacks and leaked passwords. More specifically, if the host computer has been compromised, i.e. somehow infected with malicious software, it is assumed the data is at risk regardless of any authentication solution.

## 2.3 Asymmetric Cryptography

Asymmetric cryptography is a form of a cryptographic system that creates two paired keys, one may be shared (public) and one is secret (private). A benefit to asymmetric cryptography, when compared to symmetric cryptography, is that a communication can be set up without the need of two parties meeting up in person and physically sharing keys. Key exchange can instead be done by sending public keys to each other via unencrypted communication, without concern if the key is seen by others. However, one must be careful if the key can be changed by attackers before arriving at the destination.

In an asymmetric cryptosystem, a message encrypted with a public key must only be decipherable with its related private key. Furthermore, inversely a message encrypted with a private key must only be decipherable with its public key. A message is said to be *signed* if it is encrypted with the private key, thus anyone with the public key can *verify* the signer's identity. On the other hand, a message *encrypted* with the public key can only be *decrypted* by the private key - thus providing confidentiality and integrity [1].

## 2.4 RSA

RSA (Rivest-Shamir-Adleman) [45] is an asymmetric cryptosystem and thus utilizes public and private keys. In order to generate the keys, as well as to perform the algorithm itself, RSA uses properties of prime numbers to ensure high security. The security is derived from the fact that large numbers are difficult to prime factorize for today's computer. If either key is used to encrypt data, the other key can always decrypt it. However, encrypting data with the private key, to sign it, is primarily done in digital significates.

RSA utilizes the modular arithmetic properties of prime numbers, and the equations below briefly show how to generate keys, encrypt messages as well as decrypt them.

$$Public\ key\ [e, N],\ Secret\ key\ [d, N]$$
$$Choose\ p, q \in prime,\ N = p * q,\ \gamma = (p-1)(q-1)$$
$$0 \leq e \leq \gamma,\ gcd(e, \gamma) = 1$$
$$d * e \equiv 1\ (mod\ \gamma)$$
$$ciphertext \equiv plaintext^e\ (mod\ N)$$
$$plaintext \equiv ciphertext^d \equiv plaintext^{e*d}\ (mod\ N)$$

According to the NIST foundation, RSA-keys shorter than 2048 bits are not deemed secure for applications where the public key is disclosed in plaintext [3].

RSA is much used and recognized by many authorities. However, RSA is not considered quantum-safe. Specifically it is assumed that - decades in the future - quantum computers will be able to prime factorize arbitrarily and easily. Thus, a quantum computer can, in theory, break the security of the RSA cryptography system [48]. There are cryptosystems which are quantum-safe, e.g. Lattice-based algorithms [32]. Nevertheless, many applications still use RSA since it is well known, well understood and well supported.

## 2.5 Multiprecision Operations

When operating on big numbers, the bus width of the processor could be exceeded. However, with a multi-precision algorithm, this issue can be avoided and any normal operation can be performed. An analog to multi-precision multiplication is how many school kids perform manual multiplication. More precisely, to multiply two two-digit numbers, a well-used algorithm is to perform multiple separate multiplications on single digits and to use carry and summing to complete the multiplication.

A computer works in the same way, but instead of digits there are bits, and the processor's bus width represents the number of bits that can be calculated at once. If two numbers with 32 bits individually were to be multiplied on a 16-bit wide bus, it would require multiple steps of multiplications, carrying and adding to perform the whole calculation [22].

Multiprecision operations become very relevant even for 64-bit processors when handling numbers with sizes relevant for RSA encryption, which often are in the lower thousands (of bits).

## 2.6 Modular Exponentiation

Among other things, the RSA algorithm calculates $m^e \ mod \ N$, which can be done in the two separate steps - $m^e = i$ and $i \ mod \ N$. The calculation is neither memory nor calculation efficient, because the product of a number with amount of digits $x$ to the power $e$, is a number with $x \times e$ digits. The only exception is when the base number is one - then the amount of digits remains the same. Although for small $m$ and $e$, this is usually not a problem for today's computers. However, when performing RSA, the need for larger $m$ and $e$ forces developers to find more efficient algorithms to use. Modular exponentiation can be done in a way to make such calculations more efficient by dividing the whole calculation process into smaller steps [52] .

Utilizing the fact that:

$$x \ mod \ m \equiv (a * b) \ mod \ m$$
$$x \ mod \ m \equiv [(a \ mod \ m) * (b \ mod \ m)] \ mod \ m$$

An algorithm can be realized where many small steps are being performed where multiplication and modulus are alternating. Thus reducing the number of computations, as well as reads and writes to memory. Because in each step, the numbers operated on can never exceed more than twice the size of the modulus. The algorithm below shows such a program [52].

```
int modPower(int base, int exp, int modulus)
    int tmp=base
    while(exp not 1) do
        if (exp.even = True){
            exp = exp/2
            tmp = tmp*tmp
        }
        else{
            exp = exp-1
            tmp = tmp*base
        }
        tmp = tmp mod modulus
    return tmp
```

[46]

## 2.7 Random-Number-Generator (RNG)

Random-number-generators are extensively used in applications that utilize cryptography and thus require random data, e.g. in key-generation. Software-based-RNG uses either mathematical algorithms (pseudo-random) or readings of physical entropy (true randomness) from for example atmosphere noise (lightning discharges) or radioactive decay [56].

## 2.8 PAM - Pluggable Authentication Modules

PAM is an API used to create and configure different methods for authenticating users and handling user sessions. On most computer systems, the default authentication requires you to enter a password. The password is then checked against the hash stored on the computer, paired to the chosen username.

It is possible to use multiple layers of authentication in addition to the default password, such as an RFID-chip or fingerprint. Any PAM-aware application can be configured to require the user to perform multiple tasks of authentication, before given access. Furthermore, each PAM module can enable one such later of authentication.

A PAM-configuration file can be unique for each application. The configuration file decides what modules are needed - e.g. *pam_unix.so* for default password authentication - and of how high priority they are. For example, the configuration for user login can state that it is sufficient for the user to scan her fingerprint. However, if the same user declines to login via fingerprint, she could be prompted to use the standard password authentication in addition to any other methods - if declared in the configuration file [18].

PAM is licensed under the *BSD 3-clause*, which in short enables anyone to use the code according to their wish, however, no liability is assumed by its creators and the copyright notice must be kept intact [23].

## 2.9 OpenSSL

OpenSSL is a software library that is commonly used in applications that require secure communication. OpenSSL implements a variety of cryptographic functions, as well as their utility functions. Additionally, OpenSSL offers command line utilities for most of its functions. This is useful if one, for example, does not want to generate keys inside a program or to perform simple testing [40].

## 2.10 CentOS/Red Hat

Red Hat Enterprise Linux is a Linux distribution directed towards businesses, and CentOS is the community-based free version of Red Hat. Since these distributions use the Linux kernel and the GNU utilities, they are largely compatible with other GNU/Linux distributions.

## 2.11 FPGA

An FPGA (*Field-Programmable Gate Array*) is an integrated circuit that can be (re-)configured according to specifications given by a programmer. Such specifications come in the terms of code and constraints (e.g. power consumption). An FPGA frequently comes mounted on a development board, with many different peripherals such as memory, connectors, displays, power supplies and LEDs. Furthermore, FPGA development boards often have different buttons and switches for debugging and prototyping purposes [57].

## 2.12 VHDL

A hardware description language is used to program an FPGA - one such language is VHDL. VHDL describes how physical signals interact, are stored and manipulate the program flow. Contrary to software descriptive languages where the code specifies in which order low-level instructions are executed. To avoid enormous files which contain a whole program, code can be subdivided into modules which perform separate tasks on their own. This not only helps with making the code easier to understand but also makes it easier to debug since each module can be tested separately.

## 2.13 USB

USB is a serial communications interface which is frequently used in modern devices. USB connected devices uses baud rate, to decide how many bits of information to communicate per second. When setting up a USB connection, an agreed upon baud rate is selected so that both units know at which rate they will be transmitting bits. More specifically, a high signal for one second in a 9600 baud rate setup, is 9600 ones being transmitted - since a high signal is interpreted as one and a low signal as a zero. [6]

## 2.14 Open Source

Open source is a license type for intellectual property and appears in many different forms. Individuals interested in a particular open source software can use, modify and distribute the source code quite freely. However, different licenses give different sets of permissions to the user. So called copyleft licenses forces *"any program derived from it"* to also be released under the same license, as in the case of GPL [19]. However, most open source licenses are permissive - i.e. there are no limits to its use as long as the license is included.

An open source license can be beneficial to use especially for software dealing with security since it enables transparency. More specifically, open source makes it possible for anyone to analyze software for security issues, backdoors or privacy violations.

# 3 Method

In this chapter, the tools and methods used throughout the project are described. Thus giving transparency and enabling reproducing the project.

## 3.1 Planning

The project was Scrum based [28] and utilized two-week sprints, that resulted in five product iterations labeled Mark: I, II, III, IV, and V. The project had two groups, with the focuses: hardware and software. Due to the different challenges encountered by each group, the sprint goals differed at times.

On the hardware side Mark I focused on getting basic I/O functionality working and Mark II implemented the RSA cryptography. Mark III was a collaboration with the software group to make the two systems work together. In Mark IV we changed direction and changed how I/O worked, namely, the USB communication was implemented. Like Mark III, Mark V collaborated with the software group to make the two systems work together correctly.

In the software group, Mark I included the configuration of PAM to meet our needs. Mark II explored how to use the built-in functions in OpenSSL for use in the RSA cryptography. Mark III collaborated with the hardware group to make the two systems work together as intended. Mark IV implemented USB connectivity instead of the manual system, and finally, Mark V again was a collaboration with the hardware group to make the two systems work together.

The project used git as its version control system and Google Docs for time logging. Three special assignments were dealt out to the project members: meeting convener, git responsible, and logbook writer. Other responsibilities were taken collectively and assigned at the weekly meetings. Furthermore, to simplify communication the project group spent most days working in the same location.

## 3.2 Tools

To fulfill the security token system and its functions, this project has been dependent on a couple of tools to compile and execute different programming code. The programming tools have been selected based on what PAM and the FPGA require, but also based on what was available and familiar to the project group.

### 3.2.1 Linux

CentOS was used to ensure compatibility with the Chalmers environment. More specifically *CentOS 6.8 (Final)*, with Linux kernel *2.6.32-642.15.1.el6.x86_64 SMP*, and *Gnome Display Manager 2.30.4*.

### 3.2.2 OpenSSL

OpenSSL was used on the host machine, for random number generation as well as all RSA functionality.

### 3.2.3 PAM

PAM was used to handle authentication on the computer. To begin with, other already existing PAM modules were investigated [37, 51, 60]. The PAM modules we designed are written in C and compiled with GCC.

### 3.2.4 Mobilefish.com

Many calculators are cumbersome to use when calculating the very large numbers used in RSA encryption. On the website www.mobilefish.com, there are tools for conversion between different number bases for large numbers, as well as calculators for them. In this project, the site's tools - `Big number converter` [30] and `Big number equation calculation` [31] - were used. With these tools it was possible to calculate what signed message (ciphertext) should be returned from the token, when testing.

### 3.2.5 QuestaSim

To avoid testing the VHDL code on hardware directly, and to simplify finding faults and bugs, the simulation tool QuestaSim was used. This program provides a code editor, and it also makes it possible to step through a design and change parameters for debugging. While it is no complete guarantee, a module that works correctly in QuestaSim is also very likely to function as intended once synthesized into an FPGA.

### 3.2.6 Xilinx ISE

Xilinx ISE is, like QuestaSim, a development tool for VHDL and Verilog, which is able to compile and simulate implementations. Unlike QuestaSim, this tool can be used to make the VHDL code into a bitstream file which in turn is used to program an FPGA. The conversion is done by using the built-in tools which map and configure specific pins on the FPGA to our design. As well as optimizing the: implementation, placement, and timings of the design itself onto the FPGA. Finally, it outputs the result as bitstream files.

Besides providing functionality for realizing code in hardware, Xilinx ISE can also be used for analyzing the implementations size requirements, power usage, and max speed. The tool was used for all mentioned functionalities besides simulation.

### 3.2.7 Development Board

A development board is a circuit board containing an FPGA, as well as many peripherals and connection points. In this project the primary board was the Digilent Nexys-3 housing the FPGA Xilinx Spartan-6 XC6SLX16-CSG324C [13]. In the case that the Nexys-3 was insufficient the Digilent Atlys housing the FPGA Spartan-6 XC6SLX45CSG324C [12] was used instead.

### 3.2.8 Adept

Adept from Digilent is a tool for easy loading of bitstream files onto a development board. While programming the development board with a bitstream file could be done in Xilinx ISE, Adept provides a much more streamlined process. Additionally, Adept provides easy to use self-tests for a development board, which can be used to ensure correct functionality of I/O.

### 3.2.9 Docklight

To test the functionality of the USB communication, data has to be able to be read and written to a port. On Linux systems this can easily be done with included GNU utilities, but on Windows systems it is not as easy. A useful tool that can provide these actions is Docklight, which was used when testing the USB on the Windows system.

# 4 Implementation

This chapter describes the process of designing and implementing the two versions of the security token system.

## 4.1 Design of the Security Token System

The design of the complete system is based on the security derived from asymmetrical cryptography. RSA is the cryptosystem used since it is recommended by multiple standardization bodies, such as ISO [9]. Furthermore, RSA is quite simple from a mathematical standpoint. The system implemented in this project acts as a challenge-response system, where either a user or a USB connection is the communication channel between the computer and the security token. The security is affected by the behavior of each user, e.g. if the user does not log off when she leaves her PC unattended the token does not make a difference. Hence the system should depend as little as possible on the user and still enhance security. The computer uses Linux (CentOS) and extending the PAM-login module is sufficient to add the security token as a second factor of authentication. A system overview is shown in Figure 4.1.

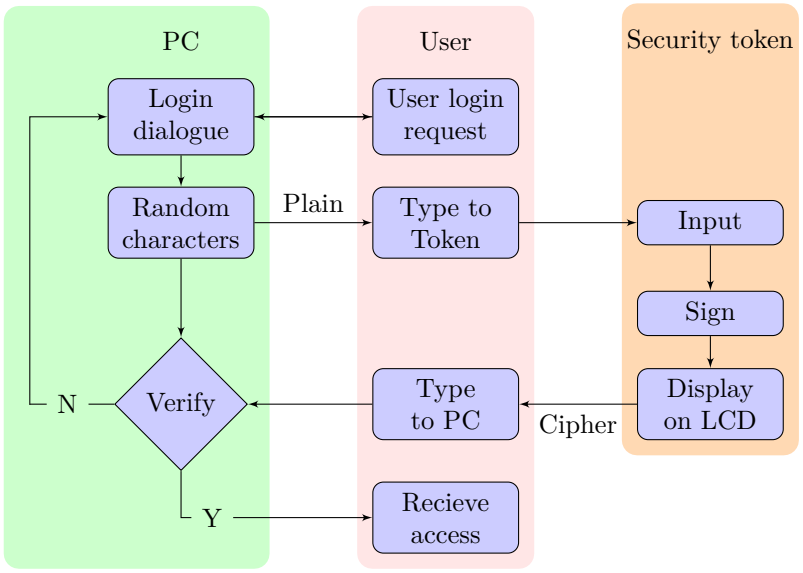



**Figure 4.1:** System Overview

Version A uses a human as the communication link, which impose the constraint that information transferred from the token to the computer needed to be easily read and written by humans. Thus a character-set of 64 ($2^6$) characters was created. The character set includes the characters: zero to nine, a to z, A to Z, ! and ", each translated to a unique 6-bit value. This allows for data to be transferred densely, and more precisely it enables a signed message (72 bits) to be communicated in only twelve characters ($12 * 6 = 72$).

In version B the human link was exchanged for serial communication between the security token and the computer, unlocking the possibility for vastly longer key lengths.

## 4.2 Implementation of the Security Token

The security token's top module is responsible for housing all submodules, designating in/out pins, the PIN-code, controlling program flow and configuring constants.

To make the configuration of the token easier, the top module includes a number of configurable constants. These constants are propagated down into the modules which need them and/or configures the modules' implementation (e.g. the RAM size, vector width, RSA exponent value).

The top module implements the flowchart to the right (Figure 4.2) as a series of states. It first waits for the LCD to be initialized, as it needs some time to do this. Once the LCD signals that it is ready for commands, the normal program flow follows:

A message prompting the user to input the PIN is written to the LCD, read from a predefined message placed in ROM. If the entered PIN is correct, the program continues to print out a message that prompts the user to enter a randomly generated message from the computer. Is the PIN incorrect, however, the program goes back to the PIN state unless the maximum amount of tries has been reached. In the project's default config, the user is given three tries. If the number of tries has been exhausted the program locks itself permanently, and the FPGA needs to be reprogrammed. To reset the FPGA is not enough since the try-counter will not reset unless the FPGA is reprogrammed.

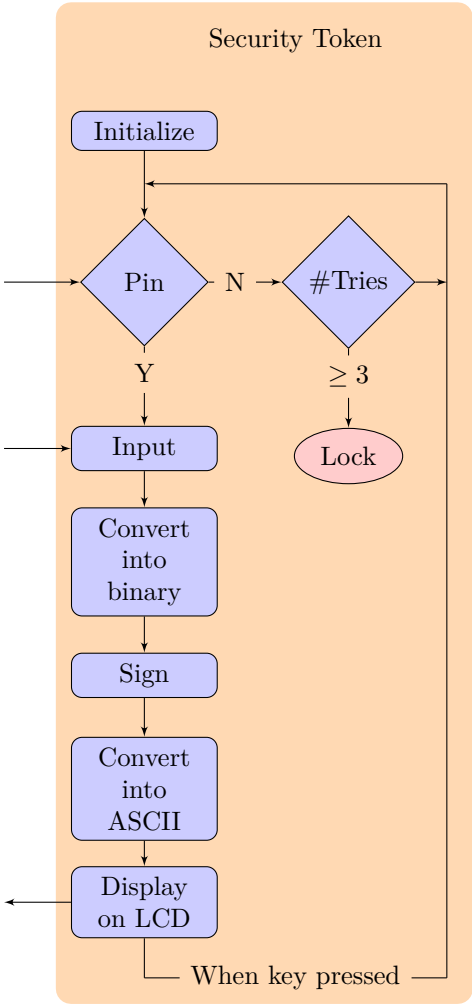



**Figure 4.2:** Flowchart of Security Token

After a successful PIN, the user is prompted to enter the message shown on the computer. This input is transferred into the FPGA's RAM, in the form of hexadec-

imal values. To clarify, the input 8, 6 becomes the hexadecimal value 86 in RAM. Once enough values are entered, as defined in the constants mentioned above, the program proceeds to the signing phase.

Once the RSA signing is done, the resulting value is present in RAM and has to be returned to the user. However, first the values have to be converted into Alphanumerical++ (see Section 4.2.4) to avoid invalid characters. To do this however it first needs to be rearranged from eight bit chunks to six bit cunks. Those two processes are done by the *byte splitter*(Section 4.2.3) and then the message is printed to the LCD.

After the return message has been printed, the program does nothing until a button on the keypad is pressed. Once a button is pressed, the RAM is cleared by writing zeroes to all cells and then the program returns to the PIN state - ready for another message to sign.

### 4.2.1 Hexadecimal Keypad

In the project, a hexadecimal keypad that consists of sixteen buttons was used. This keypad was chosen because it simplifies the translation of input messages to bit values since each button conveys a 4-bit value. Furthermore, a larger keypad would not be needed since the cleartext entered is only a couple of characters long. The keyboard chosen has eight pins, four of which are connected to the rows while the other four are connected to the columns. When a button is pressed, a signal goes from one of the row pins into one of the column pins. The pins can be connected in sixteen different ways, just like the number of physical buttons on the keypad.

To parse the keyboard input, the token has to scan which button is pressed. Initially, the token provides a signal to all four column pins until a signal is detected on the row pins (i.e. a button is pressed). Once this happens, the token provides a signal to only one column at a time and reads the row pins. This is done for all the four column pins. If and only if one row-pin gives a signal and one of the four columns is active, the input is a valid button press. Otherwise, two or more buttons have been pushed at once, thus resulting in an invalid button press. Once a legal button press has been detected, the token translates the button's value to hexadecimal and waits for the button to be released - before signaling to the rest of the hardware which value is to be used.

### 4.2.2 LCD

The LCD screen used was the DMC16207 with the HD44780 controller [10]. This LCD can display two rows of 16 character each. There are more possible instructions to give to the LCD than the ones implemented, but for the sake of simplicity, only the essential instructions were chosen. These are: printing a character, clearing the screen, changing row, as well as the instructions required for initialization. To simplify the system, the screen was given a separate clock since it was in need of long delays after certain functions, such as clearing the screen. This clock was realized by dividing the system's main clock to a frequency that allows the LCD to always

be ready for the next operation, and not being busy with a previous. However, printing to the screen is much slower than it could be since the same clock is used regardless of instruction. The printing could be faster if instruction specific delays were used instead. Nevertheless, the implemented solution is still fast enough to be almost unnoticeable.

### 4.2.3 Byte Splitter

The LCD is capable of printing a total of 256 different characters. This would be fine for printing back the signed message, as the bytes of the signed message all can have a value between 0 and 255. However, this was deemed bothersome as standard English/Swedish keyboards do not have easy or obvious ways to write that many different characters. As a result, the signed message had to be modified to print only writable characters. The signed message bytes are split into segments of 6 bits. A message of three bytes thus translates to a split message of four 6-bit segments. The characters were restricted to six bits since a larger character set would require at least 128 ($2^7$) different characters - more than what the average user can type with ease.

Byte Splitter has a local memory to temporarily save the values as well as a small register to save the, still unused, bits read from the signed message in RAM. The process is as follows: the module reads one byte from RAM and takes the six least significant bits and saves to the local memory. While the unused two most significant bits are saved in the register. The following cycle, the module reads the next cell of RAM and combines the two saved bits in the register with the four least significant bits from RAM (where the RAM bits are the most significant in the resulting six-bit vector), and again the unused bits are saved in the register. The third cycle the next RAM cell is read and the two least significant bits are combined with the saved register (again the RAM bits are most significant), and the unused six bits are saved in the register. Finally, the six bits in the register is saved to the local memory. This is visualized in Figure 4.3. This process repeats until all values are read from RAM, converted to the corresponding 6-bit vectors, and written back to the RAM.

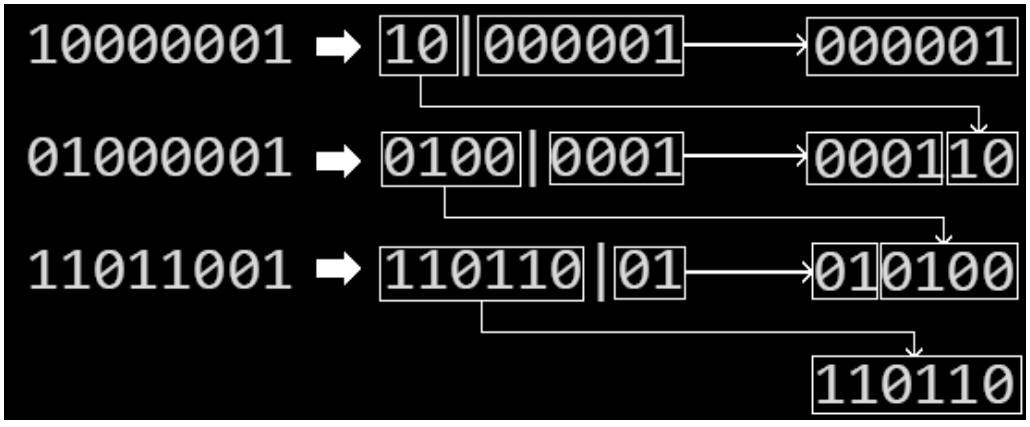



**Figure 4.3:** Demonstration of the Byte Splitter

### 4.2.4 ASCII Converter

An extended alphanumerical character-set containing 0-9, A-Z, a-z, ! and " was invented and named alphanumerical++. The characters in alphanumerical++ can be mapped from 0 to 63, but the LCD requires the characters in standard ASCII encoding. This makes the translation of a value between 0 to 63 to the corresponding ASCII value needed. The ASCII converter module performs the functionality implemented as a Lookup-Table, furthermore, it is asynchronous (i.e. does not wait for clock pulse) to make the translation immediate.

### 4.2.5 ModMult Module

The ModMult module was created by Steven R. McQueen who published it at OpenCores.org 2009 under the LGPL license. The module performs modular multiplication used in the RSA algorithm and takes three parameters: multiplicand, multiplier, and modulus. The module returns the modulus of the multiplication.

### 4.2.6 RSA Module

The RSA module controls the signing of the message with the use of the sub-module ModMult. The RSA module performs the algorithm described in the program flow as described in Section 2.4, executing the iteration through the exponent and leaving out the modular multiplication to the ModMult module. It receives the message via RAM, which it is given permission to by the top module. The RSA key, exponent, and modulus result are read from the constants located in the top module.

When the RSA module is set to inactive, nothing executes and all registers are set to their initial value. Once the module is activated, however, the message is fetched from RAM and saved locally. The RSA algorithm is later executed by using the ModMult module to sign the message. The signed message is then printed back to memory, overwriting the original message.

### 4.2.7 ROM and RAM

The ROM and RAM modules are based on code implemented by Gustav Örtenberg for the course Digital Design (EDA322) given at Chalmers University[35]. The modules are simple implementations of read-only and random-access memory, in which an address vector is used to choose the active cell to function as the output port. In the RAM module, there is also a write enable signal for replacing the active cell's value with that on an input bus.

### 4.2.8 Version B Top Module

Version B of the security token was implemented with a serial communications link (USB), instead of the user typing between devices. Since the information transferred do not have to make sense for a human, all the steps converting and partitioning the information are not needed. More specifically, writing the ciphertext to the LCD is not necessary since it is transferred via the USB connection to the computer, using the USB modules described below. The cryptography core used in Version A, described above, was too weak to handle RSA key lengths of significance. Thus another core, an existing open source one, was used to handle RSA of key length

512 bits. However, the new core did not fit on the Nexys-3, instead the Atlys was used.

### 4.2.9 USB

The USB communication is implemented using the EXAR XR21V1410 USB-UART bridge on the Atlys board. The USB module consists of three submodules: the command parser, the RXD-handler (Recieve data) and the TXD-handler (Transmit data). The USB communication is very simple and has only a small instruction set. Each transaction has a header consisting of the character * and one command specific character. The code was inspired by a private conversation with a Xilinx employee, and is based on a Xilinx Vivado Workshop-lab [50, 59]. The existing headers are mention in Table 4.1.

| | |
|---|---|
| *I | (Identification) |
| *B | (Busy) |
| *R | (Request signed message) |
| *D | (Done receiving) |
| *T | (Timeout, not all data received) |
| *M | (Signed message from FPGA to PC) |
| *W | (Write message to FPGA) |

**Table 4.1:** USB Instruction Set

In the cases of *W and *M, they are immediately followed by 64 bytes of information.

The transmission of data works the same at both the receiving and the transmitting end, and has four stages: *idle*, *start*, *data* and *stop*. When *idle*, the signal is resting with a high value, but when the transmission starts the signal is set to a low value. After the *start* bit is handled, the *data* follows from the least significant to the most significant bit. Finally, a *stop* bit (high value) is put on the signal.

### 4.2.10 RXD-handler

The RXD-handler is the submodule that receives and translates the serial bitstream from the PC to bytes. These bytes can be part of headers or data to be written to the token's RAM. To avoid incorrect sampling the RXD-handler needs to know the exact clock-speed it uses, as well as the baud rate of the transmission. With these numbers, the middle of each bit is sampled as shown in Figure 4.4 The receiving process follows the stage flow described above, and as such, the RXD-handler first listens to the input for the start bit. Once the input goes low, it starts counting to the middle of the start bit and double-checks that the input is still low. If that is the case, then it is confirmed that it is the start bit, and the following eight bits are the input. If the input were high during the double-check of the start bit, however, this was a false start bit (an anomaly) and is thus ignored. After taking the eight bits, if the input is low, then it waits for the stop-bit/idle state on the input. Once a complete byte is received the module puts it on its output bus and signals the command parser that a new byte is on the bus.

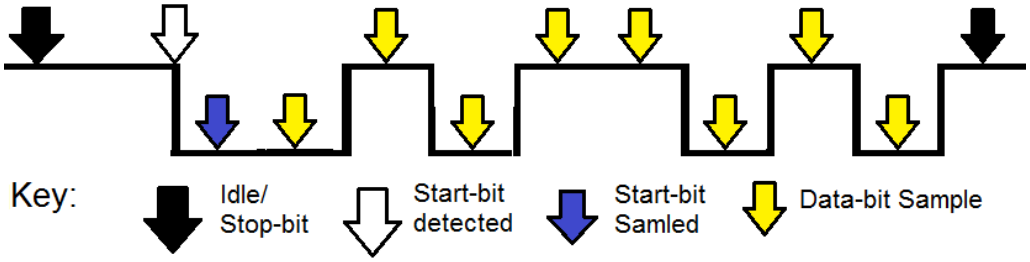



**Figure 4.4:** Example on Serial Port Sampling

### 4.2.11 TXD-handler

The TXD-handler is the submodule that translates bytes to a serial bitstream which is then sent to the PC, much like the RXD-handler reversed. The TXD-handler also needs to know both the baud rate and the clock frequency to be able to send data at a correct pace. However, unlike the RXD-handler, this module needs a FIFO-memory to store active transmission jobs. This is needed as a consequence of the token working much faster than the serial link, meaning that it is faster to execute internal commands than to transmit data to an external device. When the FIFO-memory is empty, the TXD-handler is in its idle state, but as soon as there is data in the memory, the module reads and sends every byte until it is empty again. To send a byte: firstly the module sets the output to low for the start bit, and secondly follows up with the data bitwise in the least significant to most significant bit order. After data is sent, the stop-bit (output high) is set and the module checks if the FIFO-memory is empty. If it is, the module goes to the idle state, otherwise, it begins another transmission with the next byte to be sent.

### 4.2.12 CMD-parser

The command parser does what the name entails and parses commands from the computer. It is also responsible for setting flags to other parts of the token, reading from and writing to the RAM as well as deciding what to send back to the PC. When idle, the module waits for a new byte from the RXD-handler, and this byte must be the first character in a header ('*'). If a received byte does not match this case, it is ignored and the module remains in idle mode. However, if the byte is correct the next byte decides what command it is (see Table 4.1). If the header does not match one of the defined commands, it is ignored. Depending on the token's state, the command parser may respond with either a busy header or the data requested. When headers or data is to be transmitted, they are put into the TXD-handler's FIFO-memory. If invalid headers or *W commands are being sent from the PC, or if the timeout message (*T) is received on the PC, then it verifies that something went wrong in transmission, and the PC should retry to send the command again. If the request (*R) command is received, and there is an active job in the FIFO, the command parser will not send another 64-byte message, but it will instead give a busy response command (*B). An example as shown in Figure 4.5.

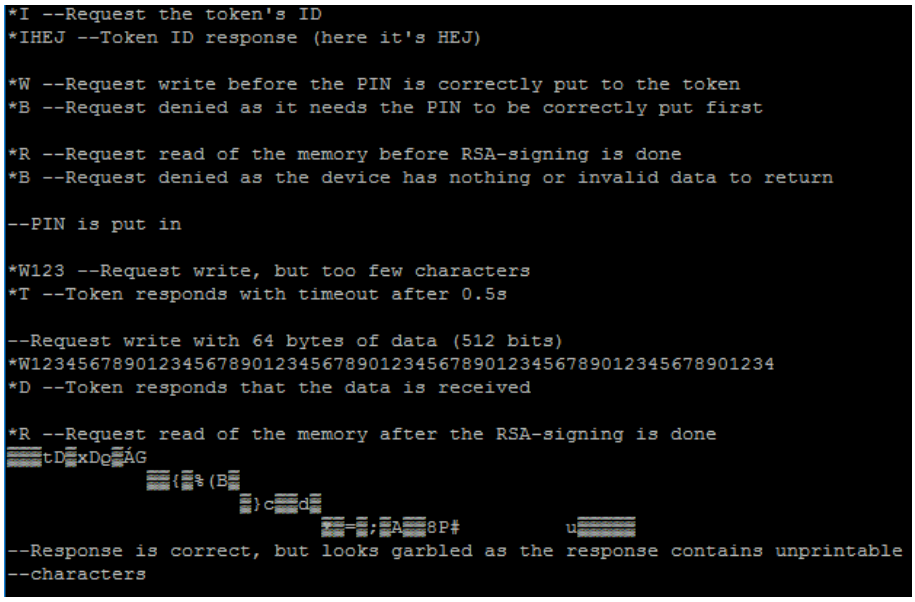



**Figure 4.5:** PC-Token Interactions Example

### 4.2.13 RSA_512

The *RSA_512* module was published by Emilio and Javier Castillo-Villar at OpenCores.org 2010 under the LGPL license [53]. The module is a part of a larger product they intended to make, which could handle key lengths up to 4096 bits. However, the version available only provides 512-bit RSA cryptography. The module, given an RSA key and a message, returns the message encrypted. Along with the VHDL files, the module includes a PDF-document describing the correct usage of the cryptography core and the other modules needed to accommodate their design. The open source module is implemented using Montgomery CIOS (*Coarsely Integrated Operand Scanning*) multiplication [2].

### 4.2.14 Xilinx IP Core Generator

Not all modules used are designed by us nor are open source code fetched from the internet. The memory in the RSA_512 (BRAMs and FIFOs), as well as the FIFO in the TXD-controller, made use of the built-in tool IP Core Generator in Xilinx ISE. The IP Core Generator has an extensive library of complete and customizable modules. The modules are available to use, but only in combination with a valid Xilinx ISE Licence. As a result, they can not be included in the final open source repository, however, instructions on how to create these IP Cores will be provided and can be found in Appendix B.

## 4.3 Implementation of the Linux-PAM System

On CentOS and most Linux distributions, the out of the box PAM configuration requires only a password for authentication (as shown in Figure 4.6). Since the aim of the project is to have an extra authentication layer to lay on top of the already existing password login, the `password-auth` is the only configuration file that needs to be edited. The authentication has to go through this projects PAM-module in addition to the standard password login. Furthermore, because the module only should handle authentication the other management groups (account, session, and password) are left untouched. However, more applications could utilize the security token, by setting the requirement in their PAM configuration files.

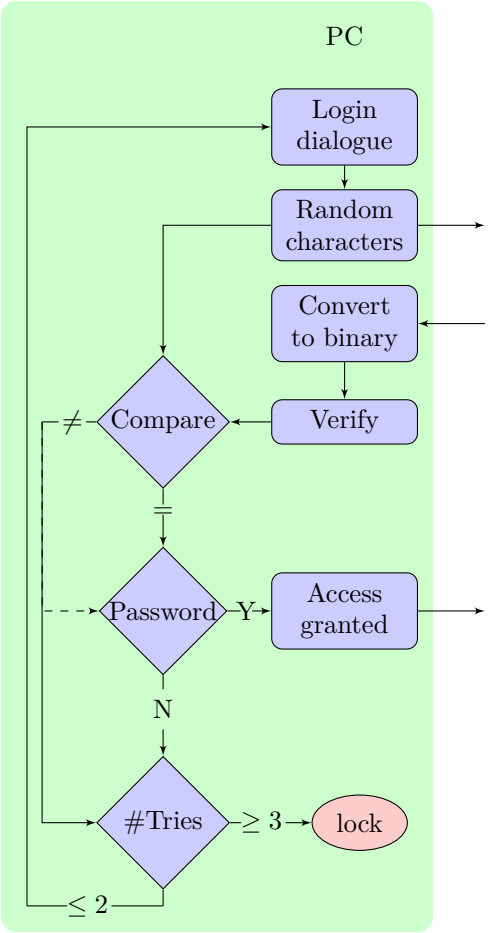



**Figure 4.6:** Flowchart of PAM Module

### 4.3.1 The PAM Module

The PAM module begins by opening a (PAM-)conversation with the user. A PAM conversation is a structure containing a PAM message and a PAM response. The message, in version A, is the user prompt showing the randomly generated hex string that should be typed on the security token. Furthermore, the response is the output (signed message) from the security token. To create the PAM module, code was taken from `pam_unix.c` [37] which is Unix's standard authentication module and falls under the BSD-3-clause license. However, instead of waiting for a password to be entered, the module generates a random message and waits for the token's response to be provided.

To generate the pseudo-random message, OpenSSL's `RAND_bytes`-function is used. In version A three random bytes are generated, but in version B (with USB) this is increased to 63 bytes. These lengths are determined by the size of the RSA keys: <72-bit for version A and <512-bit for version B. Note that version A could generate a longer message with this configuration, however, due to user friendliness being

concern - this was limited to six hexadecimal characters (three bytes). OpenSSL uses the Linux built in `/dev/random` as entropy source to generate cryptographically strong data [56, 39].

In version A, the user response requires multiple steps of parsing before interpreted by the token, as described in this section. The user response - the signed message (ciphertext) - is run through a sanitizer to verify that the string is formatted correctly, i.e. correct length (default: twelve characters) and only legal characters (0-9, A-Z, a-z, ! and "). If not, the string is either filled with zeros to make it longer or null terminated to make it shorter - to the same length as the RSA key. The reason the user response length is important is because RSA without padding is used, and thus the ciphertext must match the key length. Furthermore, by adding zeros before the data if needed, the user does not need to enter leading zeros. After cleaning the user input, the input is parsed in multiple steps because of how the FPGA handles memory and characters. First, the user response is read as its characters' ASCII values. These values are then merged and split into chunks of six bits and put into an array of 72 characters where each holds a binary value. Finally, the `bit-string` is parsed as nine 8-bit characters which are equal to the original message signed with the private RSA key.

Verification of the specified message is done by using OpenSSL's `RSA_public_decrypt` function, with the public RSA key available on the computer's storage [41]. OpenSSL has a `verify` function, however, it works on a too high level requiring trusted certificates and it is meant to be used for communication with a message content - while this project only cares about authentication [55].

### 4.3.2 USB Solution

Since the key and message lengths are global variables, and because the code is fairly generic, the switch to USB was straight forward. With USB, the module no longer needs to initiate a conversation with the user but instead communicates directly with the security token. The communication now takes place in the background and the user will, after entering the PIN-code on the token, not be able to notice it. To read and write to a USB port in Linux, the module has to configure it to be compatible with the token. The USB port requires a correctly set baud rate, the size of each data segment, and enable possible control or parity bits.

The module starts the transmission by writing a pseudo-random message preceded by `*W`, to let the FPGA know it is a write operation of 64 bytes. The message sent over USB does not require any parsing or reformatting since the user does not have to manually read or write to the token. Therefore, the message can be whichever random values originally created, even if they can not be printed as readable characters. The module then continuously reads from the USB port until the FPGA has sent a `*D`, signaling that the message has been received. Lastly, the module sends the request code `*R` to ask the FPGA for the signed message and waits until the FPGA no longer sends its busy code, `*B`. The signed message is then received and is to be verified and compared to the original pseudo-random message. If the

messages match, the module returns that the first authentication step was a success and PAM can move on to test the standard password. On the other hand, if the authentication fails, the user is still asked for a password. However, access will be denied.

### 4.3.3 RSA Key Generation

Generation of the RSA keys is done with OpenSSL's command line utilities. `Genrsa` is used to create a private and public key pair of a length specified in the command call, 72 bits for version A and 512 bits for version B [38]. Furthermore, the generation of keys is critical and must be done on a non-compromised machine. Scripts for key generation is provided in the project code (link can be found in Appendix C).

### 4.3.4 PC-Environment

The security token was tested with CentOS and should thus be fully compatible with Red Hat Enterprise Linux, if using the same versions. Furthermore, the security token could work for any recent Linux distribution - but this might require minor changes to the PAM-configuration files [8]. Since PAM is fairly standard for Unix systems [49] - e.g. Linux, FreeBSD, or Mac OS - with some configuration, the created module could be ported.

### 4.3.5 Analysis

Some tools for analysis were used during development of the PAM module: American Fuzzy Lop (AFL), Valgrind and Flawfinder. Valgrind and Flawfinder are static analysis tools used to detect memory leaks and safety errors, important when coding in an unsafe language such as C [17]. With the help of these tools, multiple memory leaks were resolved and unsafe function calls were replaced. Furthermore, the fuzzer, AFL, was used to test the compiled binary file with millions of different input values [16]. Not once did AFL manage to crash the PAM module, which is a good sign. To analyze the binary, a separate test module was created since it was not possible to analyze the regular authentication program itself. The test module performs the same functions as the project's `pam_module.c`, however, it does not utilize or examine the external PAM functions - which correctness was assumed.

# 5 Security and System Analysis

In this chapter the results of the security and system analysis will be presented.

## 5.1 Security Token Analysis

The short RSA key used in the project's Version A makes it trivial for an advanced threat to *break* the device - i.e. calculating the key-pair - if the host is infected. It is as if a symmetric cryptosystem was used since either key is enough to gain full insight into the other key, and thus perform any cryptographic operation. Therefore, it is imperative that the host computer is not compromised by an adversary - and that each user is aware of this weak link.

At first, it was incorrectly calculated that a 72-bit key would suffice when both keys are secret. When the mistake was discovered, Version B (USB) was implemented instead. The big difference with version B is that the 512-bit key provides safety even if message transactions are known. Furthermore, brute-forcing the private key knowing the public key becomes less trivial.

### 5.1.1 Physical Realities

Version A of the token uses the smaller key size of 72 bits, a design that is small enough to fit onto the Nexys-3. More specifically, this implementation took only 1131 out of 9112 available Slice LUTs, had a maximum possible frequency of 134.4MHz and a power usage of 22mW. A more detailed summary can be seen in Appendix A.1.

Version B of the token is considerably larger, as it needs much more space to house the RSA_512. It uses 9016 Slice LUTs but is not able to fit onto the Nexys-3 regardless. The reason for this is that this version uses 50 DSP48A1s (on-chip general purpose FPGA logic) on the bigger Atlys Spartan-6, of which the smaller Spartan-6 on the Nexys-3 only has 32. Version B has the maximum possible frequency of 93.941MHz and the power usage of 144.57mW. A more detailed summary can be found in Appendix A.2.

Worth noting is that all these values were found without any deep optimization effort due to a lack of time. Only the default settings and a clock constraint of 10ns (100MHz) were used. It is possible that with further effort one or more numbers can be improved if optimization targeting specific aspects (space, speed, power) is done. Furthermore, the power usage does not include the power needed for I/O.

In both version A and B the signing of the message takes so little time to perform, less than one millisecond, that the user does not notice the execution. The signing is the most computationally demanding step performed while the only visually noticeable delay is that of the message being printed onto the LCD. This

because of how the LCD code was implemented. However, even though this delay is noticeable, it is so fast that this delay should not be of any annoyance.

## 5.2 Linux PAM Analysis

Using the tools for static analysis and fuzzing, the correctness of the code was improved. Valgrind shows no memory leaks, Flawfind's results were used to remove unsafe functions, and AFL did not manage to adversely affect the program.

## 5.3 Complete System Analysis

In both of the project's versions, the keys are relatively short, and thus it is required that none of the keys are discovered by an adversary, even the so-called public key. Unlike the typical RSA usage, the cryptosystem must be thought of as a symmetric one. This means both keys must be kept secret, especially in version A, to prevent calculation of the private key. Furthermore, if message transactions are known - i.e. cleartext-ciphertext pairs - the keys can, in theory, be calculated. The keys can be determined faster the shorter the key and the more message transactions that are known. Additionally, knowing the key length makes this processes even easier. In the case of 72-bit keys, it is to be assumed that an attacker can perform a brute-force attack in reasonable time, especially since it cannot be assumed that the key length is secret. However, in version B the 512-bit cryptography should protect against attacks where only message transactions are known.

There are three attack vectors concerning the cryptosystem with a short key: the private key stored in the FPGA, the public key on a local server and in the RAM, and the message transactions taking place. Regardless of the weaknesses, the two-factor authentication solution protects from the most attackers. A majority of attacks against the system must be performed in real life on the premises, and if not, it must be a very targeted malware reading the data or keys when accessed. The two-factor authentication system with keys of 72 bits provides enhanced security compared to not using any two-factor authentication at all, however, it has multiple unnecessary attack vectors.

At first it was incorrectly calculated that a 72-bits would be enough as long as the keys were unknown. However, it became evident that another solution was required. Thus, a modification to the project was made enabling a USB-connection and much longer keys, making the system resilient against cryptographic attacks.

With a USB link, the security is improved in a number of ways. Firstly, the system uses 512-bit keys which make the private key *much* harder to calculate given a public key. Secondly, it makes the keys extremely difficult to calculate given only message transactions. Consequently making man-in-the-middle attacks futile, the biggest improvement compared to version A. Finally, in both versions, the keys are shorter than recommended by institutes such as NIST and is therefore not generally recommended. However, the security is improved since the (so-called) public key is hidden on a server, and not openly shared as in NIST's model.

# 6 Discussion

In this chapter the two versions of the security token system will be evaluated and compared. Furthermore, different approaches as well as security considerations will be discussed.

## 6.1 Threat Landscape

The security token aims to protect against a dedicated adversary, an adversary which in this text is referred to as an Advanced Persistent Threat (APT). However, no solution is impenetrable and the security token does not provide protection for malware if executed locally. Thus one cannot emphasize enough the importance of other protection mechanisms such as firewalls, IPS, and DMZ in the network. Furthermore, user awareness training should be provided within the organization and critical data should be centralized and encrypted. However, the report focuses on two-factor solutions specifically.

The project makes it possible for a user to (more) safely log into an organization environment locally as well as remotely (e.g. via SSH). However, user mindset is still of utmost importance since an infected computer on their end jeopardizes any accessible files. Furthermore, a risk is that users try to circumvent the security token because they find them all too time-consuming. To limit brute-force attempts, invalid login tries is limited as set in the PAM config files [4, 27]. However, it is of utmost importance that each sysadmin deploying the security token set these settings as specified in the project files.

### 6.1.1 FPGA Security Considerations

Even though the security token is protected by a password, one must assume that an APT can obtain the private key if enough resources are utilized assuming full control of the token. In theory, there are multiple points of attack: brute-forcing the key, observing the local buses used to transfer the key when authorized, accessing JTAG or other interfaces, and monitoring register values. If the FPGA used is not open source, or even if it is, one must acknowledge potential backdoors or security bugs present.

If the current designs' bitstream files were to be loaded into the FLASH, there is a lack of enough secure key storage since, the programming file, and thus the key, remains in the FLASH memory. Hence, if building a custom chip - not done in this project - tamper protection should be implemented. With a custom FPGA board, the program including the key could be stored in volatile memory together with a battery backup. Thus, if the device is opened the idea is to cut power to the memory, removing the program, rendering the token useless. Protection measurements as described in this section are explained more in depth in the ENISA *Hardware Threat Landscape and Good Practice Guide* [14].

Although the key might be accessible by opening the device, it is extremely difficult. Furthermore, it is easy to revoke a token's access - i.e. change its public key on the server. Security tokens are going to get lost or stolen eventually, but the ease of invalidating their access to the systems makes them more resilient to attacks. Additionally, the token does not perform random-number-generation or key creation - it only encrypts data and stores the private key - making potential bugs less critical.

### 6.1.2 Version A Security Considerations

Without a USB link, there is a significant trade-off between key strength and usability, i.e. the key and thus the resulting ciphertext has been greatly limited to make the time-loss for the user acceptable. The minimum key size as recommended by NIST (2015) [3] is 2048 bits. However, since the key is as long as the ciphertext in RSA, each 6-bit increase of the key length increases the output by one (6-bit) character. Thus 2048 bits translates to 342 characters on the project's security token without USB, well beyond what is acceptable to type manually. Instead, a key of 72 bits was chosen, resulting in a user input of twelve characters - reaching the limit of what a user should be expected to type in.

A 72-bit key provides some resilience since it requires the attacker to discover the public key or some message transactions. Furthermore, with a key this short it is impractical, yet possible, to calculate the private key provided at least a message transaction coupled with offline brute-force attempts. The more transactions that are known the easier the brute-force, however, if the message length is unknown there will be multiple possible solutions found during brute-force and would thus require multiple tries on the security token. Hence, a key length kept secret would increase the security of the token, but in practice, it would be difficult to enforce. The ciphertext gives hints about the key length: it is known that the key length and ciphertext is the same, but the ciphertext can go from all zeroes to the maximum value, making the possible key length somewhat ambiguous. However, it cannot be assumed that each entity using the token will change the default key length, which is known, and if the key length is changed it can be assumed to be around 72 bits. Thus, since security by obscurity is nothing to rely on and observing one message transaction could, in theory, compromise the token - it was evident that the system needed to be improved and as such a USB implementation was begun.

All in all, the resilience of the security token relies on the difficulty for an adversary to: observe authentication transactions and guess the key length, or get their hands on the public or private key. Ergo, there are multiple vectors of attack. However, the attacks require physical presence or advanced malware, making the token useful.

In both version A and B of the token, the private key was placed in the FPGA while the public key was stored in the computer. The choice was made since it is both harder and requires physical access to attack the token directly. Contrary to attacking the computer, which can generally be done online with less sophisticated tools - since it is a more generic and connected device. Nevertheless, it would be trivial to switch the place of these keys if needed, see section 2.4.

The reason to aim for a longer key in our case would be to protect the private key even if the host is compromised or the visual input copied. Nevertheless, it is possible to accept a short key - e.g. 72-bit - because it is assumed that if the host is compromised or physical access is gained, the secret data may be compromised as well. However, if a short key is used and the host is compromised by a possible APT - it is recommended to replace the user's security token and thus the key-pair. Therefore it is inadvisable to employ short keys if the cost of replacing a user's security token comes with a considerable cost. However, this would only really be relevant if the tokens were implemented on one-time programmable FPGA:s. To conclude the greatest weakness of version A is that even one message transaction could be used to bypass the token, at least in theory.

### 6.1.3 Version B Security Considerations

In version B, instead of having a user enter the clear- and ciphertext manually, it is done over a USB link, enabling longer keys. Currently, the length is 512 bits, since a restriction in the RSA code used on the FPGA. A 512-bit key is shorter than recommended, however, it has the major benefit of resilience - compared to a key of 72 bits - against attacks where only message transactions are known. Thus, an adversary observing the user and recording all message transactions still has a long way to go before compromising the system. Furthermore, using a USB link prevents people to espy message transactions, on the other hand, it might enable other man-in-the-middle attacks such as the NSA *Cottonmouth* [34]. However, because of the limited data gained, these man-in-the-middle attacks will be very inefficient, and a direct attack on the PC itself would be a more logical vector of entry.

The reason 512-bit RSA keys are no longer recommended is because, with known public key and message transactions, it is possible to calculate the private key - in that case making the token useless. Thus the security depends on the public key kept hidden, perhaps on the organization's server. If the organization using the security token is unsure about the secrecy of the public key and its traversal to the user's computer, they are recommended to change the RSA module and increase the key size to the recommended 2048 or 4096 bits. With a key of at least 2048 an adversary would be prevented to *break* the security token for a good number of years - regardless if the public key or message transactions are known [3]. Provided a key of at least 2048-bits, the only *known* weakness of the token would be the private key stored in the FPGA. However, even if theoretically possible, there would be needed much specialized knowledge and hardware to extract a private key from the bit-files in FLASH memory on an FPGA-device - not least the token itself.

To conclude, if a user requires high security it is possible to use this project. However, it is recommended to replace the `RSA_512` module in order to allow keys of 2048-bits or longer. Nevertheless, with a USB link and a key length of at least 512 bits, a skilled adversary must have physical access to the token or access to the server/end-user's computer. Thus the two-factor authentication device has shifted the weakest link to the stronger one.

## 6.2 Hardware Programming and FPGA

There were some design choices made early that became the foundation to how the code would be written, and one of these were the usage of generics. Generics are values set at the declaration of a module which can be used to describe parts of logic and sizes of vectors. Thus, the code became easier to manipulate even with little or no understanding of the language itself. This not only made the reusage and manipulation of the whole project structure simpler, but it also made the code easier to understand, as the names chosen for the generic constants were of a descriptive nature.

When starting this project the group had very rudimentary knowledge of VHDL, the simulation and the Xilinx tools (i.e. QuestaSim and ISE). The lack of FPGA experience resulted in a big time investment just to familiarize with the tools and get the tools working correctly, furthermore, there were also some suboptimal choices made as a result. For example, we re-used a RAM module from a previous course, but a simpler solution would have been to either uses the Xilinx Core Generator to make one or simply define an array which would have similar functionality. Additionally, the available parallelism was rarely utilized fully, and instead the code written was often sequential - limiting the performance.

## 6.3 USB-connection

Once it became clear that the first implementation (version A) with only a keyboard and LCD had inadequate security we needed a way to interface the FPGA with the target computer. USB was the obvious choice as it is a very common port and easy to use in C-programming. But at this stage in the project, there was very limited time left for a complete from-scratch implementation of USB interaction. A search for open source devices handling USB was conducted, but before this search was concluded a Xilinx employee talked with us and helped out by supplying a guide with material on USB [50, 59]. This helped out tremendously as we received clear instructions on how the protocol was to be interfaced with as well as how the module could be implemented. Without this help, there was probably not enough time for us to figure out the USB functions by ourselves, let alone implement it.

Even though the token has a physical link to a computer by using USB, there is very little an external attacker can do with it. The implemented communication protocol only responds to the predefined commands described in Table 4.1. Furthermore, for the token to sign messages the PIN has to be physically put on the device for each signing. Even if a user forgets his device on, connected to the computer and put in the PIN, the built-in timeout of 5 seconds would prevent an attacker to interface with it. The only conceivable way to be able to extract data from the token is if an attacker has physical access to it, knows the PIN and uses the predefined commands in a correct way - thus bypassing the token completely. However, based on the group's limited knowledge of embedded systems and security, a professional opinion could be warranted - even though the USB interface was deemed secure by the group.

## 6.4 Sustainability

When designing hardware solutions for security, the conclusion to go with FPGA has become clear to the project group. The reconfigurability of an FPGA provides a lot of freedom when designing a product, as well as environmental benefits. An FPGA is reprogrammable, enabling easy replacement of peripheral parts by reprogramming the FPGA. Also, if unknown bugs are discovered, or an attacker has compromised the security in any way (e.g. found the key-pair of a specific token), the FPGA can be reconfigured with a new set of keys without having to replace the entire unit.

The product could have been implemented in a customized ASIC which would make the product more power and size efficient, however, this would be a very expensive endeavor [58]. Another solution would be to use a generic ASIC, e.g. a SOC such as an Arduino. However, these solutions are most often proprietary and a separate RSA module would be needed to efficiently perform the required RSA calculation. Thus, an FPGA was used - enabling cheap and fast deployment of the two-factor system.

Two-factor authentication is important today, where more and more critical systems and services are put online. Humans are bad at storing, creating and using good unique passwords. Thus, critical applications such as banking require two-factor authentication if the transaction is to be made online. Moreover, various websites and cloud services have also begun to offer two-factor authentication - to prevent the loss of monetary or sensitive material [36, 25]. Security is of utmost importance in society and without it, our modern society would crumble [7].

## 6.5 Improvements

In both version A and B, the token is performing its task as it should, but there are some improvements that could be made. For the device to become usable for its designed purpose it has to be able to be shut off. In the prototype, the programming file is not put on the on-chip FLASH memory, and as such the token program does not remain on the device after power is cut.

Programming the FLASH memory to instantiate the code at startup should be a very streamlined process using Xilinx ISE and Digilent Adept, but was not done during this project.
As a consequence, the device can not be shut off without losing the entire program and thus all functionality. Loading the bitstream file into the FLASH does come with its own security flaws. Theoretically, the program could be extracted from the FLASH, but this needs specialized knowledge and equipment. Also, with the current implementation the PIN, storing the program in the FLASH would make it so that the token's lock function would be bypassed by a reset. With the program loaded on FLASH, the counter should, in that case, be stored in non-volatile memory instead. Furthermore, depending on how critically the security of the data protected by the

token, it is possible to implement functionality on the device that overwrites the data in the FLASH when too many incorrect tries have been made.
The result would be that the device would be not only unusable after three incorrect tries but also makes probing of the FLASH futile after such a state is reached.

Since this was many of the group members second time programming in VHDL there are much to be improved in both performance and structuring of the code. Many parts do not utilize VHDL's ability for parallelism to its fullest extent, and as such, the code is slower than it might otherwise be. Later on in the project, efforts were made to improve in these regards but with a lack of time for a full rewrite, much of the code was kept as its initial version.

Another issue to consider is what if working quantum computers are made in the future, *breaking* RSA. RSA is listed as a quantum unsafe cryptography system, because of its heavy reliance on the traditional difficulty in prime factorizing large numbers. Thus, exploring the realm of quantum-safe cryptography systems is appealing - and there are such systems available today - but these cryptosystems are not used as much as RSA and thus are less tested and available online. Quantum safe cryptosystems, such as lattice-based ones, are most probably safe to use today - however since they lack a lot of documentation and open source implementations, more work would be required compared to RSA. Changing to a quantum-safe cryptosystem is a recommended improvement if protecting sensitive systems for decades to come.

Even though the code is completely open source and licensed under LGPL and BSD, some elements, as well as the synthesis, were done with the proprietary Xilinx solution. Most would argue using a proprietary FPGA suite is acceptable, however, if the threat perceived is very advanced one should acknowledge possible backdoors. Backdoors are easier to hide in proprietary solutions and thus should be avoided if possible, especially since backdoors in FPGAs are not unheard of [20, 47]. On the other hand, backdoors in an offline two-factor authentication token are for most uses extremely unlikely to be a problem - even if present in the device. The reality is that open-source FPGA solutions suitable for the project are rare, but depending on the threat model and investment, this improvement could be considered [33].

The programming language C was used for the PAM module since it is the native language for PAM. However, as recommended by ENISA, C should be avoided if possible since it is more prone to result in software vulnerabilities. Unlike C, programming languages like Go and Rust introduce type-safety, garbage collection, and more security relevant features [14]. The written code is not long and has been analyzed, nevertheless, an improvement could be to re-write all C code in a safer language. Furthermore, PAM modules can be written in any language which is able to call C functions, such as but not limited to Rust [11].

Compilation flags could be investigated further to harden against memory corruption attacks [54]. Unfortunately, there was not enough time to perform extensive

research and testing of the different flags, which could potentially increase security. Furthermore, before the token is used in critical applications it is also advisable to perform more security analysis of the system (e.g. fuzzing and static analysis) because of the group's limited skillset in this area. Due to time constraints, fuzzing the source code was not possible, and only the binary was examined, it is recommended to fuzz the source code to ensure security flaws are even more improbable. Additionally, further testing of the FPGA interface could be done by fuzzing - to ensure no unwanted response can be triggered.

During development there were attempts to improve the security of version A while not changing the user experience. However, the difficulty lies in the fact that the ciphertext is the only thing to verify the user and it needs to be decryptable and short. Regardless, a possible improvement could be to perform the RSA algoritm multiple times with different keys - similar to the 3DES algoritm [26]. This would not increase the ciphertext length, but it would increase the security we believe. More specifically, it would presumably make it harder to calculate the keys from message transactions alone - which is the biggest weakness in version A.

## 6.6 Follow Up

During the projects planning stage[24] it was underestimated how much time the parts of the project would take. The idea was to complete certain functionalities every two weeks, but this proved hard to accomplish. The first release was postponed by a week because both the PAM and FPGA configuration were more difficult than planned and thus the second release was also delayed. Additionally, the different marks should have been better defined in the planning stage. As we had three marks, which only set out to improve the previous mark, it was impossible to know how much time the improvements would take, as these were yet to be defined.

When version A's basic functionality was completed, the security analysis showed that while the solution worked and did what it was supposed to do, it had security flaws. Therefore, instead of continuing work on the original idea, with a user being the communication link, it was decided to use USB instead. With this switch, we were able to increase the RSA key length to 512 bits, while re-using much existing code. However, this version also required more space on the FPGA and the Nexys-3 could no longer be used, as it proved too small. Instead, we used the Atlys development board. The change was seamless, as the only needed changes was a setting in Xilinx ISE from using the old FPGA to the new one, as well as generate the new specific pin-outs for the Atlys.

Even though it might seem obvious, one should not reinvent the wheel, so to speak. At the start of this project everyone started to think of ways to solve the problems from the ground up, both on the software and hardware side. When we noticed that we didn't have enough time to be so ambitious we started to search for and use open source modules and solutions. Especially were the cryptography done by external open source libraries, partly because it is good from a security perspective - i.e. to use recognized and well-tested code.

We had two designated group roles which were kept throughout the duration of the project (meeting convener and logbook writer), but the third role (git responsible) was dropped because of the lack of relevancy. In addition, our decision to use a very flat hierarchy structure was mostly a success. There were moments when there was uncertainty who was supposed to do what, but generally it worked fine. It will probably not work in projects with a larger work group or longer time spans, but in our case this provided sufficient structure as well as a friendly environment - motivating each member to contribute with their respective skills.

# 7 Conclusion

The aim of the project was (as mentioned in Section 1.1) to discover if the two-factor token could be made secure enough by the use of RSA cryptography. The conclusion, based on the aim, is that a two-factor system can be made quite secure. However, no computer system is foolproof but with this two-factor authentication system, a possible point of entry (i.e authentication) is patched. RSA cryptography worked well and showed to be well suited for the application, however, only 512-bit keys are supported by the system created - thus weakening the security in some respects. Furthermore, the RSA cryptosystem is likely to be *broken* decennium in the future with the dawn of quantum computing, thus, making a poor choice against a very advanced adversary.

The challenge-response system created, regardless of its flaws, quickly enables two-factor authentication on Linux for any PAM-aware application - such as user login or SSH. The current prototype system is not user-friendly but would be if deployed in a more suitable FPGA. For a skilled adversary to bypass the two-factor system it is needed to have extended access to the token or the device used to log in, ergo, the token serves its purpose.

The most reasonable attack against the whole system would be to social engineer the user to execute the malware, thus getting access to their public key, data and transactions. Thus the private key can be computed in theory, however, this data exfiltration and malware execution are very serious in of itself - regardless of any two-factor system.

The first implementation that was created, without USB (Version A), is not recommended for serious use - even though it provides some resilience. The main flaw is that the key length is restricted to ensure user-friendliness since it is needed to be manually entered. A larger key length would indeed result in a more secure system, but a much larger key is impossible when the user is forced to enter it into the token. Other cryptography schemes, which does not generate as long outputs, would work as a replacement for the RSA scheme if one wishes to have a human as the communication channel. This could prevent the currently devastating man-in-the-middle attack, however, keys would still need to be secret and finding a quantum-safe scheme could be hard.

The second implementation (Version B) used USB to communicate and could, therefore, enable arbitrarily long keys. Systems implemented with an electronic link between the devices, such as these, can be very powerful security solutions. By replacing the human as the communicator between the devices, with either one- or two-way serial communication, the RSA cryptography will prove to be secure until quantum computing is available - assuming a long enough key is used. The two-factor authentication token with USB created in the project uses 512-bit RSA,

which is sufficient as long as the private key and public key are kept secret.

The security token has been implemented in a way that should make the source code easily understandable and adjustable for others. A couple of downsides and alternatives has been taken into account in the Discussion chapter. Furthermore, anyone with interest is free to modify the project according to these alternatives or any idea of their own. The intention of leaving this project as an open source software is, after all, to make the security token better and more secure.

# Bibliography


[1] Will Arthur and David Challener. *A Practical Guide to TPM 2.0, p.18*. Apress, 28 January 2015.

[2] Selçuk Baktir and Erkay Savaş. Highly-parallel montgomery multiplication for multi-core general-purpose microprocessors. In *Computer and Information Sciences III*, pages 467–476. Springer, 2013. [accessed 10 May 2017].

[3] Elaine Barker. Recommendation for key management. `http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf`, January 2016. [accessed 10 May 2017].

[4] Tim Baverstock and Tomas Mraz. pam_tally2 - the login counter (tallying) module. `http://www.unix.com/man-page/linux/8/pam_tally2/`. [accessed 10 May 2017].

[5] Duncan Borde. Two-factor authentication. `https://web.archive.org/web/20120112172841/http://www.insight.co.uk/files/whitepapers/Two-factor authentication (White paper).pdf`. [accessed 9 May 2017].

[6] Andrew Butterfield and Gerard Ekembe Ngondi. baud rate. `//www.oxfordreference.com/10.1093/acref/9780199688975.001.0001/acref-9780199688975-e-343`. [accessed 25 May 2017].

[7] Berkeley Center for Long-Term Cybersecurity, University of California. Cybersecurity futures 2020. `https://cltc.berkeley.edu/scenarios/`. [accessed 12 May 2017].

[8] CentOS. About centos. https://www.centos.org/about/. [accessed 10 May 2017].

[9] EMC Corp. What are iso standards? `http://isiloniq.com/emc-plus/rsa-labs/standards-initiatives/iso-standards.htm`. [accessed 10 May 2017].

[10] OPTREX CORPORATION. Lcd module specification. `http://pdf1.alldatasheet.com/datasheet-pdf/view/92672/OPTREX/DMC16207.html`. [accessed 11 May 2017].

[11] Alex Crichton. Rust once, run everywhere. `https://blog.rust-lang.org/2015/04/24/Rust-Once-Run-Everywhere.html`, 2015. [accessed 10 May 2017].

[12] Digilent. Atlys™ fpga board reference manual. `https://reference.digilentinc.com/_media/atlys:atlys:atlys_rm.pdf`, 2016. [accessed 12 May 2017].

[13] Digilent. Nexys 3™ fpga board reference manual. `https://reference.digilentinc.com/_media/nexys:nexys3:nexys3_rm.pdf`, 2016. [accessed 2 Feb 2017].

[14] ENISA. Hardware threat landscape and good practice guide. Technical report, ENISA, 2017.

[15] Jim Fenton. 5 myths of two-factor authentication. `https://www.wired.com/insights/2013/04/five-myths-of-two-factor-authentication-and-the-reality/`, 2013. [accessed 11 May 2017].

[16] OWASP Foundation. Fuzzing tools. `https://www.owasp.org/index.php/Fuzzing#Fuzzing_tools`. [accessed 02 May 2017].

[17] OWASP Foundation. Static code analysis tools. https://www.owasp.org/index.php/Source_Code_Analysis_Tools. [accessed 02 May 2017].

[18] Kenneth Geisshirt. *Pluggable Authentication Modules, chapter 2*. Packt Publishing Ltd, 2007.

[19] GNU.org. What is copyleft? https://www.gnu.org/licenses/copyleft.en.html. [acessed 10 May 2017].

[20] Andy Greenberg. This 'demonically clever' backdoor hides in a tiny slice of a computer chip. https://www.wired.com/2016/06/demonically-clever-backdoor-hides-inside-computer-chip/, 2016. [accessed 10 May 2017].

[21] Tim Greene. Malware infection rate of smartphones is soaring – Android devices often the target. http://www.networkworld.com/article/3185766/security/malware-infection-rate-of-smartphones-is-soaring-android-devices-often-the-target.html, 28 Mars 2017. [accessed 10 May 2017].

[22] Randall Hyde. *The art of assembly language, chapter 8*. No Starch Press, 2010.

[23] Open Source Initiative. The 3-clause bsd license. https://opensource.org/licenses/BSD-3-Clause. [accessed 10 May 2017].

[24] Ben Mohammad Johan, Fredriksson Adam, Mathiesen Christoffer, Roxbergh Eliot, and Örtenberg Gustav. Open-source koddosa; ett säkrare login på linux-system. https://github.com/GustaMagik/RSA_Security_Token/blob/master/planerings_rapport.pdf, 2017. [access 12 May 2017].

[25] Brad Jones. Two-factor security is the best lock for your digital life, but it's not perfect. https://www.digitaltrends.com/computing/why-2-factor-security-is-flawed/. [accessed 12 May 2017].

[26] Muruganandam .A Karthik .S. Data encryption and decryption by using triple des and performance analysis of crypto system. http://www.ijser.in/archives/v2i11/SjIwMTM0MDM=.pdf. [accessed 28 May 2017].

[27] Juliet Kemp. Setting password policy with pam. http://www.serverwatch.com/tutorials/article.php/3771431/Setting-Password-Policy-With-PAM.htm. [accessed 10 May 2017].

[28] Schwaber Ken and Sutherland Jeff. The scrum guide. http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf, 2016. [access 12 May 2017].

[29] Robert Lemos. Fake fingerprint fools iphone 6 touch id. https://arstechnica.com/security/2014/09/fake-fingerprint-fools-iphone-6-touch-id-why-its-not-so-serious/. [accessed 23 May 2017].

[30] Mobilefish.com. Big number converter. http://www.mobilefish.com/services/big_number/big_number.php. [accessed 12 May 2017].

[31] Mobilefish.com. Big number equation calculation. http://www.mobilefish.com/services/big_number_equation/big_number_equation.php. [accessed 12 May 2017].

[32] Heger Monica. Cryptographers take on quantum computers. http://spectrum.ieee.org/computing/software/cryptographers-take-on-quantum-computers, 2009. [accessed 11 May 2017].

[33] Kevin Morris. The fpga tool problem. http://www.eejournal.com/article/20161004-opensource, 2016. [accessed 10 May 2017].

[34] Edward NSA leak via Snowden. Cottonmouth-i. https://www.spiegel.de/static/happ/netzwelt/2014/na/v1/pub/img/USB/S3223_COTTONMOUTH-I.jpg. [accessed 10 May 2017].

[35] Chalmers University of Technology. Eda322. https://www.student.chalmers.se/sp/course?course_id=21227. [accessed 12 May 2017].

[36] Florida Tech Online. The rise of two-factor authentication. https://www.floridatechonline.com/blog/information-technology/the-rise-of-two-factor-authentication/. [accessed 12 May 2017].

[37] openpam.org. Pam_unix.c. https://www.openpam.org/browser/openpam/trunk/modules/pam_unix/pam_unix.c. [accessed 12 May 2017].

[38] openSSL. genrsa. https://www.openssl.org/docs/man1.0.2/apps/genrsa.html. [accessed 10 May 2017].

[39] OpenSSL. Rand_bytes. https://www.openssl.org/docs/man1.1.0/crypto/RAND_bytes.html. [accessed 28 May 2017].

[40] OpenSSL.org. Openssl, cryptography and ssl/tls toolkit. https://www.openssl.org/. [accessed 10 May 2017].

[41] openssl.org. Rsa_private_encrypt. https://www.openssl.org/docs/man1.1.0/crypto/RSA_public_decrypt.html. [accessed 12 May 2017].

[42] Andrew Patrick. Human factors of security systems: A brief review. *National Research Council of Canada*, 2002. [accessed 10 May 2017].

[43] phys.org. Biometric expert shows an easy way to spoof fingerprint scanning devices. https://phys.org/news/2005-12-biometric-expert-easy-spoof-fingerprint.html, 11 December 2005. [accessed 10 May 2017].

[44] Eric Ravenscraft. Which form of two-factor authentication should i use? https://lifehacker.com/which-form-of-two-factor-authentication-should-i-use-1784769336. [accessed 10 May 2017].

[45] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[46] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition (2nd ed.), p.244*. Wiley, 1996. ISBN 978-0-471-11709-4.

[47] The H Security. Backdoor found in popular fpga chip. http://www.h-online.com/security/news/item/Backdoor-found-in-popular-FPGA-chip-1585579.html, 2012. [accessed 10 May 2017].

[48] Tom Simonite. Nsa says it "must act now" against the quantum computing threat. https://www.technologyreview.com/s/600715/nsa-says-it-must-act-now-against-the-quantum-computing-threat/, 3 February 2016. [accessed 10 May 2017].

[49] JT Smith. Understanding pam. https://www.linux.com/news/understanding-pam. [accessed 26 May 2017].

[50] Mathiesen Tryggve. Private Communication. 20 April 2017.

[51] Shane Tully. Simple public key encryption with rsa and openssl. https://shanetully.com/2012/04/simple-public-key-encryption-with-rsa-and-openssl/. [accessed 26 May 2017].

[52] Minh Van Nguyen. Number theory and the rsa public key cryptosystem. http://doc.sagemath.org/html/en/thematic_tutorials/numtheory_rsa.html. [acceseed 25 May 2017].

[53] E. Castillo Villar and J. Castillo Villar. rsa_512. https://opencores.org/project,rsa_512. [accessed 11 May 2017].

[54] Debian Wiki. Hardening. https://wiki.debian.org/Hardening. [accessed 10 May 2017].

[55] OpenSSL Wiki. Manual: Verify. https://wiki.openssl.org/index.php/Manual:Verify%281%29. [accessed 26 May 2017].

[56] OpenSSL wiki. Random numbers. https://wiki.openssl.org/index.php/Random_Numbers, 2 January 2017. [accessed 10 May 2017].

[57] Xilinx. Fiel programmable gate array (fpga). https://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm. [accessed 9 May 2017].

[58] Xilinx. Fpga vs. asic. https://www.xilinx.com/fpga/asic.htm. [accessed 12 May 2017].

[59] Xilinx. Fpga design flow using vivado. https://www.xilinx.com/support/documentation/university/vivado/workshops/vivado-fpga-design-flow/materials/2016x/2016_2_zynq_labdocs_pdf.zip, 2016. [accessed 18 April 2017].

[60] Zen. Openssl: decrypt with rsa private key in c++. http://www.toptip.ca/2010/06/openssl-decrypt-with-rsa-private-key-in.html. [accessed 26 May 2017].

# A Appendix: ISE Result

## A.1 ISE FPGA Reports, Keyboard Version

### A.1.1 Device Utilization

Device Utilization Summary:

| | | | | |
|---|---|---|---|---|
| Slice Logic Utilization: | | | | |
| Number of Slice Registers: | 756 | out of | 18,224 | 4% |
| Number used as Flip Flops: | 756 | | | |
| Number used as Latches: | 0 | | | |
| Number used as Latch-thrus: | 0 | | | |
| Number used as AND/OR logics: | 0 | | | |
| Number of Slice LUTs: | 1,131 | out of | 9,112 | 12% |
| Number used as logic: | 1,112 | out of | 9,112 | 12% |
| Number using O6 output only: | 733 | | | |
| Number using O5 output only: | 130 | | | |
| Number using O5 and O6: | 249 | | | |
| Number used as ROM: | 0 | | | |
| Number used as Memory: | 6 | out of | 2,176 | 1% |
| Number used as Dual Port RAM: | 0 | | | |
| Number used as Single Port RAM: | 6 | | | |
| Number using O6 output only: | 3 | | | |
| Number using O5 output only: | 1 | | | |
| Number using O5 and O6: | 2 | | | |
| Number used as Shift Register: | 0 | | | |
| Number used exclusively as route-thrus: | 13 | | | |
| Number with same-slice register load: | 11 | | | |
| Number with same-slice carry load: | 2 | | | |
| Number with other load: | 0 | | | |
| Slice Logic Distribution: | | | | |
| Number of occupied Slices: | 348 | out of | 2,278 | 15% |
| Number of MUXCYs used: | 336 | out of | 4,556 | 7% |
| Number of LUT Flip Flop pairs used: | 1,201 | | | |
| Number with an unused Flip Flop: | 547 | out of | 1,201 | 45% |
| Number with an unused LUT: | 70 | out of | 1,201 | 5% |
| Number of fully used LUT-FF pairs: | 584 | out of | 1,201 | 48% |
| Number of slice register sites lost to control set restrictions: | 0 | out of | 18,224 | 0% |

A LUT Flip Flop pair for this architecture represents one LUT paired with one Flip Flop within a slice. A control set is a unique combination of clock, reset, set, and enable signals for a registered element.

```
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.

IO Utilization:
  Number of bonded IOBs:                          20 out of     232    8%
    Number of LOCed IOBs:                         20 out of      20  100%

Specific Feature Utilization:
  Number of RAMB16BWERs:                           0 out of      32    0%
  Number of RAMB8BWERs:                            0 out of      64    0%
  Number of BUFIO2/BUFIO2_2CLKs:                   0 out of      32    0%
  Number of BUFIO2FB/BUFIO2FB_2CLKs:               0 out of      32    0%
  Number of BUFG/BUFGMUXs:                         1 out of      16    6%
    Number used as BUFGs:                          1
    Number used as BUFGMUX:                        0
  Number of DCM/DCM_CLKGENs:                       0 out of       4    0%
  Number of ILOGIC2/ISERDES2s:                     0 out of     248    0%
  Number of IODELAY2/IODRP2/IODRP2_MCBs:           0 out of     248    0%
  Number of OLOGIC2/OSERDES2s:                     0 out of     248    0%
  Number of BSCANs:                                0 out of       4    0%
  Number of BUFHs:                                 0 out of     128    0%
  Number of BUFPLLs:                               0 out of       8    0%
  Number of BUFPLL_MCBs:                           0 out of       4    0%
  Number of DSP48A1s:                              0 out of      32    0%
  Number of ICAPs:                                 0 out of       1    0%
  Number of MCBs:                                  0 out of       2    0%
  Number of PCILOGICSEs:                           0 out of       2    0%
  Number of PLL_ADVs:                              0 out of       2    0%
  Number of PMVs:                                  0 out of       1    0%
  Number of STARTUPs:                              0 out of       1    0%
  Number of SUSPEND_SYNCs:                         0 out of       1    0%
```

### A.1.2 Device Clock Timing

```
Data Sheet report:
------------------
All values displayed in nanoseconds (ns)

Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |    7.439|         |         |         |
---------------+---------+---------+---------+---------+


Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 4326163 paths, 0 nets, and 5297 connections

Design statistics:
   Minimum period:   7.439ns{1}   (Maximum frequency: 134.427MHz)
```

### A.1.3 Device Power Summary

```
2.  Summary
2.1.  On-Chip Power Summary
-------------------------------------------------------------------------------
|                            On-Chip Power Summary                            |
-------------------------------------------------------------------------------
|        On-Chip        | Power (mW) |  Used  | Available | Utilization (%) |
-------------------------------------------------------------------------------
| Clocks                |       0.50 |      2 |    ---    |       ---       |
| Logic                 |       0.00 |   1131 |      9112 |              12 |
| Signals               |       0.00 |   1486 |    ---    |       ---       |
| IOs                   |       0.00 |     20 |       232 |               9 |
| Static Power          |      21.51 |        |           |                 |
| Total                 |      22.02 |        |           |                 |
-------------------------------------------------------------------------------
```

# A.2 ISE FPGA Reports, USB Version

## A.2.1 Device Utilization

```
Device Utilization Summary:
Slice Logic Utilization:
  Number of Slice Registers:                 8,653 out of  54,576   15%
    Number used as Flip Flops:               8,205
    Number used as Latches:                      0
    Number used as Latch-thrus:                  0
    Number used as AND/OR logics:              448
  Number of Slice LUTs:                      9,016 out of  27,288   33%
    Number used as logic:                    8,077 out of  27,288   29%
      Number using O6 output only:           6,323
      Number using O5 output only:             530
      Number using O5 and O6:                1,224
      Number used as ROM:                        0
    Number used as Memory:                     648 out of   6,408   10%
      Number used as Dual Port RAM:              0
      Number used as Single Port RAM:            8
        Number using O6 output only:             8
        Number using O5 output only:             0
        Number using O5 and O6:                  0
      Number used as Shift Register:           640
        Number using O6 output only:             0
        Number using O5 output only:             0
        Number using O5 and O6:                640
    Number used exclusively as route-thrus:    291
      Number with same-slice register load:    255
      Number with same-slice carry load:        36
      Number with other load:                    0

Slice Logic Distribution:
  Number of occupied Slices:                 3,034 out of   6,822   44%
  Number of MUXCYs used:                     1,772 out of  13,644   12%
  Number of LUT Flip Flop pairs used:       10,165
    Number with an unused Flip Flop:         2,385 out of  10,165   23%
    Number with an unused LUT:               1,149 out of  10,165   11%
    Number of fully used LUT-FF pairs:       6,631 out of  10,165   65%
    Number of slice register sites lost
      to control set restrictions:               0 out of  54,576    0%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
```

```
IO Utilization:
  Number of bonded IOBs:                          23 out of     218   10%
    Number of LOCed IOBs:                         23 out of      23  100%

Specific Feature Utilization:
  Number of RAMB16BWERs:                           0 out of     116    0%
  Number of RAMB8BWERs:                           10 out of     232    4%
  Number of BUFIO2/BUFIO2_2CLKs:                   0 out of      32    0%
  Number of BUFIO2FB/BUFIO2FB_2CLKs:               0 out of      32    0%
  Number of BUFG/BUFGMUXs:                         1 out of      16    6%
    Number used as BUFGs:                          1
    Number used as BUFGMUX:                        0
  Number of DCM/DCM_CLKGENs:                       0 out of       8    0%
  Number of ILOGIC2/ISERDES2s:                     0 out of     376    0%
  Number of IODELAY2/IODRP2/IODRP2_MCBs:           0 out of     376    0%
  Number of OLOGIC2/OSERDES2s:                     0 out of     376    0%
  Number of BSCANs:                                0 out of       4    0%
  Number of BUFHs:                                 0 out of     256    0%
  Number of BUFPLLs:                               0 out of       8    0%
  Number of BUFPLL_MCBs:                           0 out of       4    0%
  Number of DSP48A1s:                             50 out of      58   86%
  Number of ICAPs:                                 0 out of       1    0%
  Number of MCBs:                                  0 out of       2    0%
  Number of PCILOGICSEs:                           0 out of       2    0%
  Number of PLL_ADVs:                              0 out of       4    0%
  Number of PMVs:                                  0 out of       1    0%
  Number of STARTUPs:                              0 out of       1    0%
  Number of SUSPEND_SYNCs:                         0 out of       1    0%
```

### A.2.2 Device Clock Timing

```
Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |   10.645|         |         |         |
---------------+---------+---------+---------+---------+


Timing summary:
---------------

Timing errors: 32  Score: 11137  (Setup/Max: 11137, Hold: 0)

Constraints cover 405776 paths, 0 nets, and 45275 connections

Design statistics:
   Minimum period:  10.645ns{1}   (Maximum frequency:  93.941MHz)
```

### A.2.3 Device Power Summary

```
------------------------------------------------------------------------------
|                            On-Chip Power Summary                           |
------------------------------------------------------------------------------
|        On-Chip        | Power (mW) |  Used  | Available | Utilization (%) |
------------------------------------------------------------------------------
| Clocks                |      67.11 |      2 |    ---    |       ---       |
| Logic                 |       8.74 |   9016 |     27288 |              33 |
| Signals               |      19.62 |  14193 |    ---    |       ---       |
| IOs                   |       0.85 |     23 |       218 |              11 |
| BlockRAM/FIFO         |       5.18 |  ---   |    ---    |       ---       |
|   8K BlockRAM         |       5.18 |     10 |       232 |               4 |
|   16K BlockRAM        |       0.00 |      0 |       116 |               0 |
| DSPs                  |       3.31 |     50 |        58 |              86 |
| Static Power          |      39.75 |        |           |                 |
| Total                 |     144.57 |        |           |                 |
------------------------------------------------------------------------------
```

# B Appendix: Required Memory Cores

```
Needed FIFOs and BRAMs:

Type: Single Port BRAM
Name: Mem_b

Signals:

CLKA : in STD_LOGIC;
ADDRA : in STD_LOGIC_VECTOR (5 downto 0);
DINA : in STD_LOGIC_VECTOR(15 downto 0);
WEA : in STD_LOGIC_VECTOR(0 downto 0);
DOUTA : out STD_LOGIC_VECTOR(15 downto 0);

Internal sizes:
Write Width: 16
Write Depth: 40

Operating Mode: Write First
Always Enabled



Type: Standard FIFO
Name: FIFO_TXD
Signals:

CLK : in STD_LOGIC;
RST : in STD_LOGIC;
WR_EN : in STD_LOGIC;
RD_EN : in STD_LOGIC;
FULL : out STD_LOGIC;
EMPTY : out STD_LOGIC;
DIN : in STD_LOGIC_VECTOR(7 downto 0);
DOUT : out STD_LOGIC_VECTOR(7 downto 0);

Internal sizes:
Write width: 8
Write Depth: 128
```

```
Type: Standard FIFO
Name: fifo_256_feedback
Signals:

CLK : in STD_LOGIC;
RST : in STD_LOGIC;
WR_EN : in STD_LOGIC;
RD_EN : in STD_LOGIC;
FULL : out STD_LOGIC;
EMPTY : out STD_LOGIC;
DIN : in STD_LOGIC_VECTOR(48 downto 0);
DOUT : out STD_LOGIC_VECTOR(48 downto 0);

Internal sizes:
Write width: 49
Write Depth: 32



Type: Standard FIFO
Name: fifo_512_bram
Signals:

CLK : in STD_LOGIC;
RST : in STD_LOGIC;
WR_EN : in STD_LOGIC;
RD_EN : in STD_LOGIC;
FULL : out STD_LOGIC;
EMPTY : out STD_LOGIC;
DIN : in STD_LOGIC_VECTOR(15 downto 0);
DOUT : out STD_LOGIC_VECTOR(15 downto 0);

Internal sizes:
Write width: 16
Write Depth: 64



Type: Standard FIFO
Name: res_out_fifo
Signals:

CLK : in STD_LOGIC;
RST : in STD_LOGIC;
WR_EN : in STD_LOGIC;
RD_EN : in STD_LOGIC;
```

```
FULL : out STD_LOGIC;
EMPTY : out STD_LOGIC;
DIN : in STD_LOGIC_VECTOR(31 downto 0);
DOUT : out STD_LOGIC_VECTOR(31 downto 0);

Internal sizes:
Write width: 32
Write Depth: 64
```

# C Appendix: Source Code

All code is licensed under BSD-3, and can be found at:
`https://github.com/GustaMagik/RSA_Security_Token`