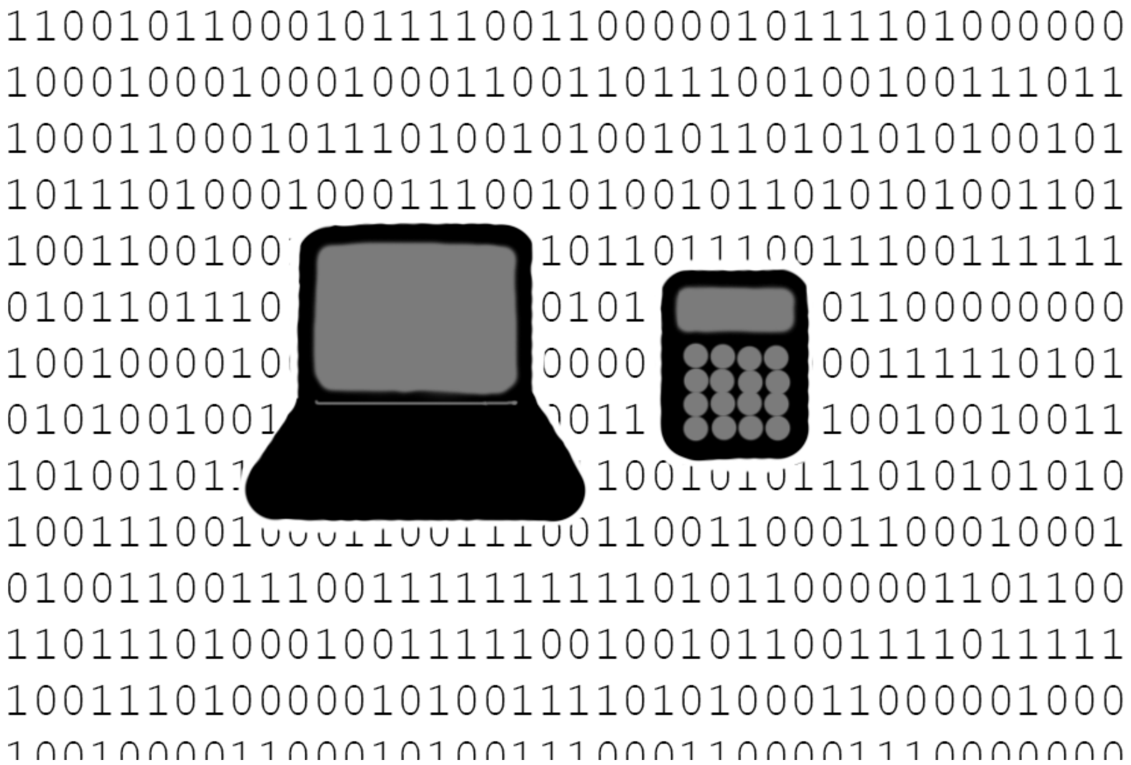
CHALMERS
UNIVERSITY OF TECHNOLOGY



# 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
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

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

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

Johan Ben Mohammad
Adam Fredriksson
Eliot Lagerström 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

## Abstract

The project implements and investigates 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 supports no more than 512-bit RSA and is the greatest area of improvement - since a longer key would be more secure and still 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 decennia to come. Altogether, the prototype created is a basic, yet fully functional, two-factor system with no obvious security flaws if deployed correctly. The whole project is released as open source with the BSD license.

## Sammanfattning

Detta projekt implementerar och undersöker ett två-faktors-autentiseringssystem 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 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, vilket mot vissa attacker, skulle höja systemets säkerhet betydligt. Vidare kan intressanta uppföljningsprojekt vara att undersöka kvantsäker kryptografi, speciellt om projektet skall användas decennier framöver. Sammanfattningsvis är prototypen ett grundläggande, men fullt fungerande, två-faktorssystem 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

Version A - Our first version. Uses the user as communication link
Version B - Our second version. Uses USB for communication
FPGA - Field-Programmable Gate Array
RSA - An asymmetric cryptography scheme
ISE - Development tool for Xilinx FPGA:s
API - Application Programming Interface
PAM - Pluggable Authentication Module
Baud - Bits per second a serial port can transfer
GCC - Gnu Compiler Collection Scrum - An agile software development framework
I/O - Input/Output
ASCII - American Standard Code for Information Interchange
FIFO - First-In First-Out
BRAM - Block Random Access Memory
Pseudo Random - Statisically random
Quantum Computer - A theoretical computer using quantum bits for which factorization is trivial
LUT - Look Up Table
Bitstream File - A file of a binary sequence
Attack Vector - Possible way of attack
MITM - Man In The Middle
APT - A very dedicated attacker
IPS - Intrussion Prevention System
DMZ - Demilitarized Zone
SSH - Secure Shell used for remote login
JTAG - Joint Test Action Group
SOC - System On a Chip
Fuzzing - Finfing Bugs by testing a large amount of random inputs
AFL - American Fuzzy Looper
BSD-license - Open source software license
Key pair - A pair (a private and a public) of cryptography keys

# 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 some kind of information. This is necessary in order to keep information private, restrict permissions and 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 that user only. Unfortunately, passwords are frequently discovered by unauthorized individuals, most often since humans are poor in choosing and storing passwords [38]. 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 a biometric reader. However, since smartphones often are attacked [22] and biometrics are easily faked [39], 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 physical tokens, secure their systems and 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 device on an FPGA will be designed that implements RSA cryptography. The system parts will be evaluated both individually and collectively. 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 only focus on developing a non-portable prototype of the security token as well as the software needed to extend the already existing login authentication functionality in Linux-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 themselves audit and change the code. It is of utmost importance to release the code of any software using cryptography openly - gives transparency its users that possibly no backdoor exists, 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: knowledge, possession, and inherence (e.g. password, credit card info and fingerprint respectively) [6].

When designing a two-factor authentication system, two authentication categories are chosen. It is common to use a password as well as a device that the person possess [40]. Passwords are often poorly chosen and blatantly re-used [38], thus requiring two-factor authentication to access a system greatly increases the authentication security - if done correctly.

## 2.2 Security

Correctly implemented two-factor authentication helps to protect the user against: bad passwords, powerful brute-force attacks, and passwords observed by a third-party. The prevalence of bad passwords in the organizations has prompted a plethora of two-factor authentication solutions. Not all two-factor authentication systems are equal, sadly many solutions are proprietary or connected to the Internet - possibly weakening the security [16] [26]. 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.

Other than brute-force attacks, shared- or otherwise known passwords - the host computer is not protected by a security token. Furthermore, if the host computer has been compromised, somehow infected with malicious software, it is assumed the data is at risk regardless of any two-factor 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 being stored by others. However, one must be careful if the key can be changed by attackers before arriving at the destination regardless of the cryptography type.

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* by the public key can only be *decrypted* by the private key - thus providing confidentiality and integrity [1].

## 2.4 Random-Number-Generator (RNG)

Random-number-generators are extensively used in applications that utilize cryptography and 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 [47].

## 2.5 RSA

RSA (Rivest-Shamir-Adleman) [41] 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. RSA uses modular arithmetic properties of prime numbers to make its algorithm work, the equations below show briefly how to generate keys, encrypt messages as well as decrypt them.

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

According to the NIST foundation RSA-keys shorter than 2048 bits are not 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. I.e. it is assumed that - decades in the future - quantum computers will be able to prime factorize arbitrarily and easily. Thus, the availability of a quantum computer will, in theory, break the security of the RSA cryptography system [44]. There are cryptosystems which in theory are quantum-safe, e.g. Lattice-based algorithms [23]. Nevertheless, many applications still use RSA since it is well known, well understood and well supported.

## 2.6 FPGA

An FPGA (*Field-Programmable Gate Array*) is an integrated circuit that can be reconfigured according to specifications given by a programmer. Such specifications come in the terms of functional code and constraints (e.g. power consumption). An FPGA often comes mounted on a development board, with many different peripherals such as memory, connectors, displays, power supplies and LEDs. Furthermore, FPGA boards often have different buttons and switches for debugging and prototyping purposes [48].

## 2.7 VHDL

A hardware description language is used to program an FPGA - VHDL is one such language. VHDL describes how physical signals interact, are stored and manipulate the program flow, unlike in software descriptive languages where code specify in which order low-level instructions are executed. A hardware descriptive language would describe the hardware itself. To avoid files which contain the whole program, the code is most often 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.

## 2.8 PAM - Pluggable Authentication Modules

PAM is an API used to create and configure different methods for authenticating users and handling user sessions. In this project, the focus has solely been on user authentication. On most systems, the default authentication requires you to enter a password which is checked against the encrypted password stored on the computer since user creation.

It is possible to use multiple layers of authentication in addition to the default password, such as an RFID-chip or fingerprint. PAM enables the use of multiple modules that each handle one layer of the authentication. Any PAM-aware application can be configured to require the user to perform multiple tasks to be authenticated.

A PAM-configuration file is unique for each application. Such files decide what modules are included and of how high priority they are. For example, the configuration for user login can state that it is sufficient for the user to use her fingerprint. If the same user declines to login via fingerprint, she would be prompted to use the standard password login in addition to any other methods declared in the configuration [19].

## 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 utility functions for these. In the case of this project, OpenSSL is only used for the RSA and RNG implementations. In addition to the programming functions, OpenSSL offers command line utilities for most of its functions. This is useful if you, for example, do not want to generate keys inside your program or perform simple testing [36].

## 2.10 Modular Exponentiation

In the RSA algorithm, it is required to calculate $m^e \ mod \ N$, which is done in the two separate steps - $m^e = i$ and $i \ mod \ N$ - is neither memory nor calculation efficient. This because of the product of a number of digits length $x$ to the power $e$, becomes a number with length $x \times e$. 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 is a way to make such calculations more efficient by dividing the whole calculation process into smaller steps by modular arithmetics properties.

Utilizing the fact that:

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

A rather clever 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.

```
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
```

## 2.11 Multiprecision Operations

When operating on big numbers, the bus width of the processor could be exceeded. A multi-precision algorithm can be used to perform any normal operation. An analog to how multi-precision multiplication is done is how many school kids perform manual multiplication. To multiply two two-digit numbers such as 13 and 94, 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 [24] .

Multiprecision operations become very relevant even for 64-bit processors when handling numbers with sizes relevant for RSA encryption, which usually have the bit size of 512, 1024, 2048 or 4096.

## 2.12 CentOS/Red Hat

Red Hat Enterprise Linux is a Linux distribution often used by businesses, 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 Linux distributions. The security token was tested with CentOS and should thus be fully compatible with Red Hat. Furthermore, the security token could work for any Linux recent distribution - but this would require minor changes to the PAM-configuration files [8]. The system was tested with *CentOS 6.8 (Final)*, using Linux kernel *2.6.32-642.15.1.el6.x86_64 SMP*, and *Gnome Display Manager 2.30.4*.

## 2.13 USB

USB is a serial communications interface that operates by sending information one bit at the time. Baud rate is the symbol frequency per second in an electronic circuit. For USB connected devices the baud rate is used as a means to decide how many bits of information that can travel from one unit to another in a 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.

## 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 according to their wish. The 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 [20]. However, most open source licenses are permissive - i.e. there are no limits to its use. Licensing a project as open source makes it possible for other companies and individuals to use the source code, and in theory, improve the code for everyone.

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, in theory, for anyone to analyze software for security issues, backdoors or privacy concerns. The code created in the project 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 [25].

# 3. Method

## 3.1 Planning

The project is to use Scrum [42] in two-week sprints and consequently creating five product iterations labeled Mark I, Mark II, Mark III, Mark IV, and Mark V. Mark I focuses on getting basic I/O functionality working. Mark II aims to implement the RSA cryptography. Mark III focus on improving security from Mark II by implementing pin and password. Mark IV is there as an extra chance to finish the functionality of Mark III. Lastly, Mark V aims to put the finishing touches on the whole design. The project will use git as its version control system and Google Docs to keep a time log. Only three assignments are dealt out to the: project members, meeting convener, git responsible, and logbook writer. Other responsibilities are to be taken collectively.

## 3.2 GCC

GCC will be used to compile all C code, i.e. both the PAM-module as well as the used libraries.

## 3.3 OpenSSL

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

## 3.4 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 used will be the Digilent Nexys-3 housing the FPGA Xilinx Spartan-6 XC6SLX16-CSG324C [13].

## 3.5 QuestaSim

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

## 3.6 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: maps and configures specific pins on the FPGA to our design, optimizes implementation, placement, and timings of the design itself onto the FPGA, and generating bitstream files. Besides providing functionality for realizing code in hardware, Xilinx ISE can also be used for analyzing:g the implementations size requirements, power usage, and max speed. The tool will be used for all listed functionalities besides simulation.

## 3.7 Adept

Adept is a Digilent 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.

## 3.8 Docklight

To test the correct 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, but on Windows systems, it's not as easy. A useful tool that can provide these actions is Docklight, which will be used on systems which are not Linux based.

## 3.9 Mobilefish.com

Most 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` [28] and `Big number equation calculation` [29] will be used, to theoretically calculate what numbers used to test the RSA cryptography should yield as result after encryption. These theoretical numbers will be cross-referenced with the results returned from the token to ensure that it encrypts data correctly.

# 4. Implementation

## 4.1 Design of the security token system

The design of the security token system is based on security derived from asymmetrical cryptography. RSA is the cryptosystem used since it is recommended by multiple standardization bodies and is, for example, ISO certified [9]. The system implemented in this project acts as a challenge-response system where the user 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 security is compromised. Hence the system should depend as little as possible on the user and still enhance security. The computer uses Linux (CentOS) and thus extending the PAM-login module is sufficient to add the security token as a second factor of authentication.

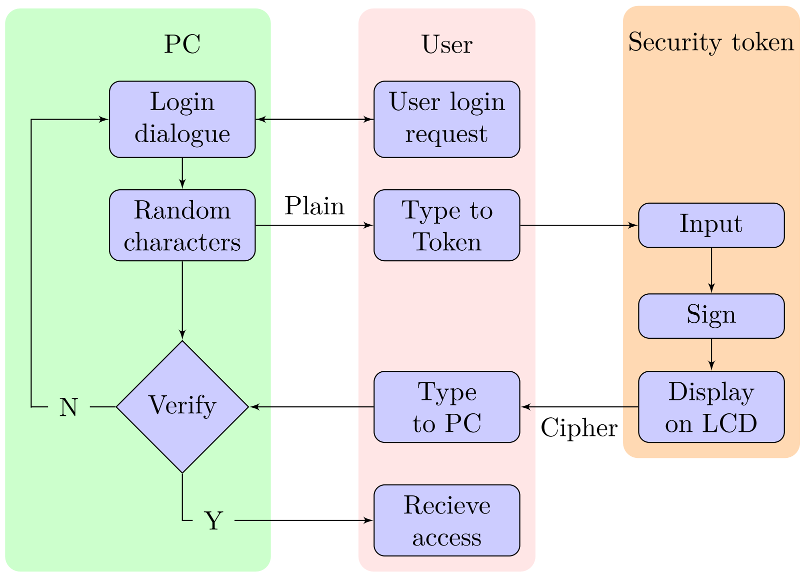


**Figure 4.1:** System Overview

Version A uses a human as the communication link between the security token and the computer and thus gave 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 characters; zero to nine, a to z, A to Z, ! and ", was designed to be mapped to unique six-bit combinations. The character set allowed for data to be transferred densely, and more precisely it enables for 72 bits of information to be transferred in only twelve characters.

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

## 4.2 Implementation of the Security Token

The module's top module is responsible for housing all submodules, designing 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 (4.2) as a series of states. The first state, INIT, waits for the LCD to be initialized, and once the LCD gives a signal that it is ready, the program proceeds to the state PIN.
In the PIN state, characters starting from a fixed address in memory are printed from ROM until the character \0 is encountered. When the message is printed, the token waits for the user to insert the PIN on the keyboard.
If the entered PIN is correct, the program continues to print out a message that prompts the user to enter randomly generated message from the computer. Is it incorrect, however, the program goes back to the PIN state unless the maximum amount of tries has been reached. In the project's case, the user is given three tries. If a number of tries have 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.

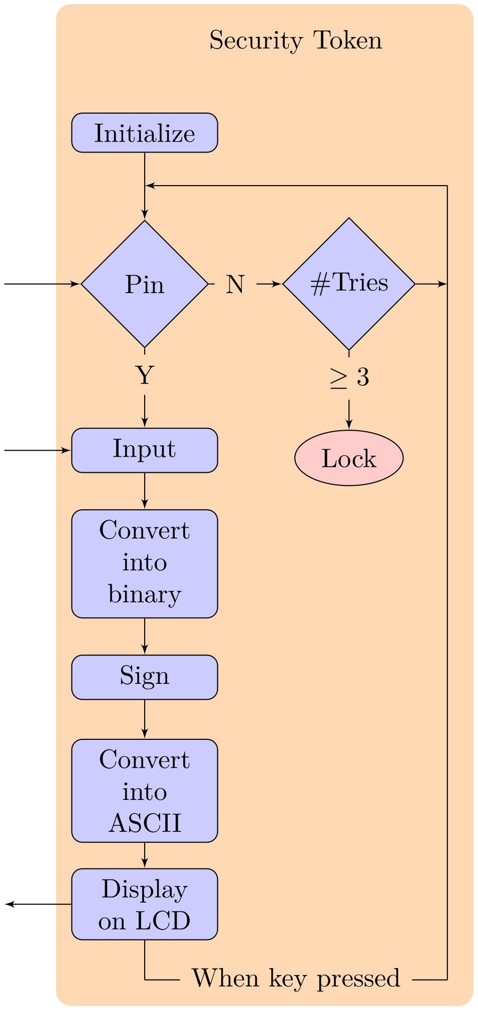


**Figure 4.2:** Flowchart of Security Token

In the PRINT_MSG_X state, characters from ROM starting at STRING_PTR_X are printed to the LCD on the upper row. Characters are printed until a \0 is encountered. After this string is printed to the screen the program moves on to the next state. The exception to this is the state PRINT_MSG_2 where not only the string from ROM is printed, but also the entire signed message from RAM is showed on the lower row.

Once the token has reached the state GET_MSG, it waits for the user to enter an input on the keyboard, which is later put into RAM. The number of buttons the user pushes is defined in the configurable constants. To make the user sure that the correct numbers have been pressed, they will be shown on the lower row of the LCD. Once all values are inserted, whether it is the correct values or not, the program state is changed to RSA.

The RSA state configures the input of the internal signals so the RSA module receives direct access to the RAM and activates the module. The RSA module performs the RSA algorithm. Once it flags that the signed message is in RAM, it is set to inactive by the top module - and the state changes to BYTE_TO_6BIT.

BYTE_TO_6BIT is a state that splits up the signed messages to bytes of which only the 6 least significant bits has values. The top module gives the splitter module access to the RAM and waits until it is done before moving on to the state PRINT_MSG_2.

CLEAR_RAM clears the RAM completely by looping through all the cells and writing zeroes, after which the program loops back to the PIN state.

### 4.2.1 Hexadecimal Keypad

In the project, a hexadecimal keypad that consists of sixteen buttons is used because it simplifies the translation of input messages to bytes and other binary numbers. The keyboard chosen has eight pins, four of which are connected to the rows and the other four are connected to columns. When a button is pressed, current flows 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 seen 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 only if one of the four columns is active, the input is a valid button press. Otherwise, two or more buttons have been pushed at once, and thus resulting in an invalid button press. Once a legal button press has been detected, the token translates the button's value to a hexadecimal value and then waits for the button to be released before signaling to the rest of the hardware which value is to be used.

To give the signals time to propagate, each check is delayed by 500 clock cycles.

### 4.2.2 LCD

The LCD screen used was the DMC16207 with the HD44780 controller [10]. This LCD can display 2 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, more functions than what was needed were not implemented. As another simplicity measure, the screen has a separate clock from the rest of the token because it was in need of longer delays after certain functions, such as clearing the screen. This clock is realized by dividing the token's main clock enough to reach a frequency that allows the LCD to always be ready for the next operation. As a result, the printing to the screen is much slower than it could be - however - still fast enough to be unnoticeable for the user.

### 4.2.3 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.

### 4.2.4 RSA Module

The RSA module controls the signing of the message with the use of the sub-module ModMult. It performs the algorithm described in the program flow described in the Theory section 2.5, executing the iteration through the exponent and leaving out the modular multiplication to the ModMult module. It receives the message by being passed permission to the RAM from the top module. The RSA key, exponent, and modulus are given from the constants in the top module.

When the module is set to inactive, nothing executes and all registers are set to their initial value. Once the module gets activated 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.

### 4.2.5 Byte Splitter

The LCD is capable of printing a total of 256 characters as a result of its data bus of eight bits in width. This was deemed bothersome as standard English/Swedish keyboards do not have easy or obvious ways to write many of the characters (i.e. Japanese katakana, $\alpha$, etc.). As a result, the signed message had to be modified to not result in printing these characters. The signed message bytes was split into segments of 6 bits. A message of three bytes thus translates to a split message of four 6-bit segments.

The module has a local memory to temporarily save the values and a small register to save the, still unused, bits read from the signed message in RAM. The process is as following: the module reads one byte from RAM and takes the six least significant bits and saves to the local memory. The unused two most significant bits are saved in the register. The next 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 the 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. Lastly, the six bits in the register is saved as its own value to the local memory. This process repeats until all values are read from the RAM and converted to the corresponding 6-bit vectors they are written back to the RAM.

### 4.2.6 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 to make translation immediate.

### 4.2.7 ROM and RAM

These modules are modified versions of code implemented by Gustav Örtenberg for the course Digital Design (EDA322) given at Chalmers University[32]. The modules are simple implementations of read-only memory 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

«««< HEAD The security token implemented with serial communications link (USB), instead of the user typing from device to device, differ a bit from the original. Since the information transferred do not have to make sense for a human (readable), all the steps converting and partitioning the information are not needed. In addition 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 the code described above was too weak to handle RSA key lengths of significance, so an attempt to integrate an open source core which was supposed to handle RSA of key-length 512 bits was made. This did not fit on the Nexys-3, however, and as such, instead theDigilent Atlys housing the FPGA Spartan-6 XC6SLX45CSG324C[12] was used. ======= The security token implemented with serial communications link (USB), instead of the user typing from device to device, differ a bit from the original. Since the information transferred do not have to make sense for a human (readable), all the steps converting and partitioning the information are not needed. Additionally, 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 the code described above was too weak to handle RSA key lengths of significance, so an attempt to integrate an open source core which was supposed to support RSA of key-length 512 bits was made. This did not fit on the Nexys-3, however, and as such instead theDigilent Atlys housing the FPGA Spartan-6 XC6SLX45CSG324C[12] was used. »»»> 7b384030725c5ba390556d1e53b2b114cae10042

### 4.2.9 Docklight

To test the correct functionality of the USB interactions data has to be able to be read and written to the port. On Linux systems, this can easily be done in the terminal, but on Windows systems, it's not as easy. A useful tool that can provide these actions is Docklight and will be used on those systems which are not Linux based.

### 4.2.10 RSA_512

This module was published by Emilio and Javier Castillo-Villar at OpenCores.org 2010 under the LGPL [14]. It is a part of a larger product they intended to make, which could handle key lengths up to 4096 bits. The version available only provides 512-bit RSA though. The module, given an RSA key and a message return the message encrypted. Along with the VHDL files, the module includes a PDF-document describing the correct usage of the cryptography core, as well as describing modules needed to accommodate their design. Their design is implemented using Montgomery CIOS (*Coarsely Integrated Operand Scanning*) multiplication [2].

### 4.2.11 USB

The USB communication is implemented using the FTDI FT232 USB-UART bridge on the Nexys-3 board. The USB module consists of three submodules. These are the command parser, the RXD-handler (Recieve data) and the TXD-handler (Transmit data). The communication is very simple and has only a small instruction set. Each command and message have a header consisting of the character * and one command specific character. The code was inspired by a private conversation with a Xilinx employee and a Xilinx Vivado Workshop-lab [45][50]. The existing headers are as follows:

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

**Table 4.1:** USB Instruction Set

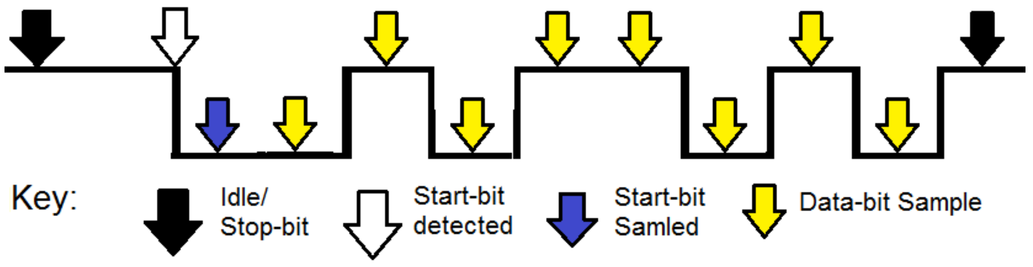


**Figure 4.3:** Example on serial port sampling

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

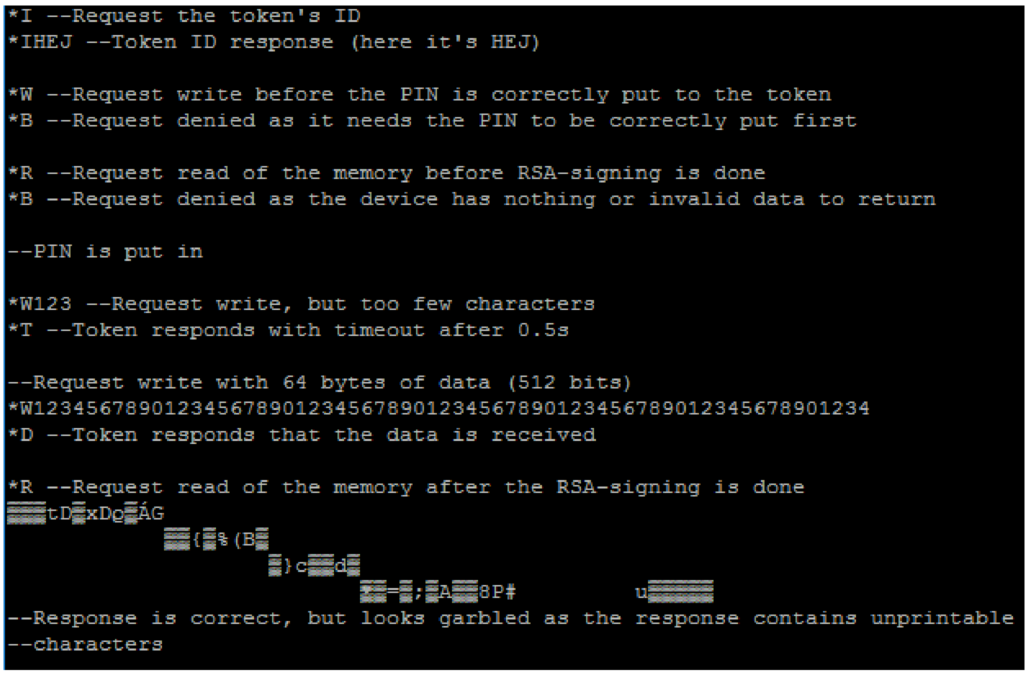
The transmission of data works the same at both the receiving and the transmitting signal, and has four stages: idle, start, data and stop. These are the names given to the different states in the code and the same names will be used in the report. When idle, the signal is resting with a high value, and 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.

### 4.2.12 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.
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 8 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 8 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 this on its output bus and signals the command parser that a new byte is on the bus.

### 4.2.13 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 goes faster to read 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,



```
*I --Request the token's ID
*IHEJ --Token ID response (here it's HEJ)

*W --Request write before the PIN is correctly put to the token
*B --Request denied as it needs the PIN to be correctly put first

*R --Request read of the memory before RSA-signing is done
*B --Request denied as the device has nothing or invalid data to return

--PIN is put in

*W123 --Request write, but too few characters
*T --Token responds with timeout after 0.5s

--Request write with 64 bytes of data (512 bits)
*W1234567890123456789012345678901234567890123456789012345678901234
*D --Token responds that the data is received

*R --Request read of the memory after the RSA-signing is done
▒▒▒tD▒xDϱ▒ÁG
            ▒▒{▒%(B▒
                    ▒}c▒▒d▒
                           ▒▒=▒;▒A▒▒8P#         u▒▒▒▒▒
--Response is correct, but looks garbled as the response contains unprintable
--characters
```

**Figure 4.4:** PC-Token interactions example

the module reads and sends every byte until it is empty again. When a byte is to be sent: 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 the 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.

### 4.2.14 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 sent to 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).

### 4.2.15 Xilinx IP Core Generator

Not all modules used were designed by us nor were parts of open source code fetched from the internet. The memories 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 very 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, but instructions on how to generate IP Cores on one own will be provided and can be found in Appendix. B.1

## 4.3 Implementation of the Linux-PAM system

On CentOS (and RedHat) the out of the box PAM configuration requires only a password for authentication. Since the aim is to have the extra authentication layer to lie 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.

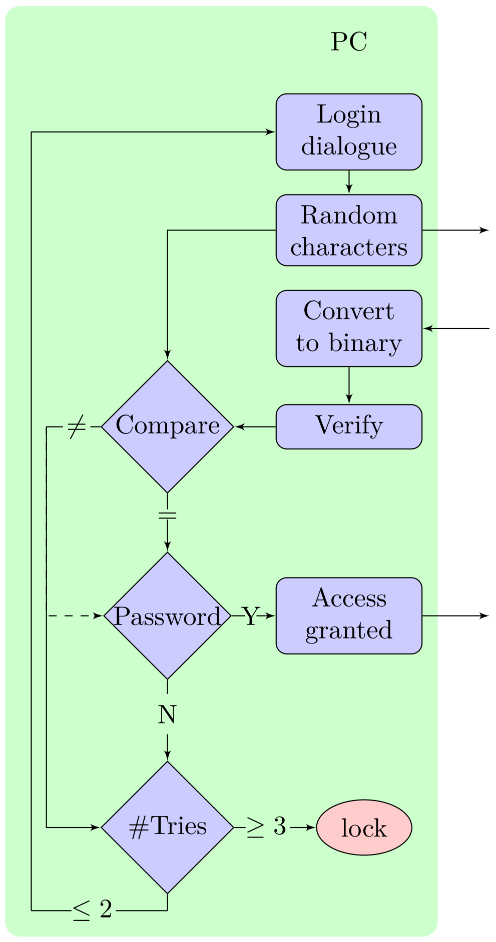


**Figure 4.5:** Flowchart of PAM module

### 4.3.1 The PAM Module

The PAM module first opens a (PAM-)conversation with the user. A PAM conversation is a structure containing a PAM message and a PAM response. The message is the user prompt showing the randomly generated hex string that should be typed on the security token. The response is the output (signed message) from the security token. Code was taken from the existing module `pam_unix.c` [34] which is Unix's standard authentication module and falls under the BSD-3-clause license. This module does the normal `username/password` authentication which is similar to what the new module is doing. 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 entered.

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; 72bit without USB and 512bit with USB. Note that the non-USB version generates a shorter message than possible. This is because we wanted a user friendliness by not making the user type more than six hex characters. OpenSSL uses Linux's built in `/dev/urandom` as entropy source. A check is automatically done verifying that the seed has received a sufficient amount of entropy, meaning that the pseudo-randomness is random enough. These three bytes are split into six hexadecimal characters (four bits each), which are then used as the PAM conversation message to the user.

The user response 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 length specified by the RSA key length. The user input is then parsed in a somewhat complex manner because of how the FPGA handles memory and characters. First, the user response is read as its character's ASCII values. These values are then merged and split into chunks of six bits and put into an array of 72 characters where each is 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.

The verification of the message entered is done by using OpenSSL's `RSA_public_decrypt` function, with the public RSA key available on the computer's storage [37].

### 4.3.2 USB Solution

Since the key and message lengths are global variables and 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 received. 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 parsed as normal ASCII 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, 512 bits when using USB and 72 bits when not [35]. Furthermore, the generation of keys is critical and must be done on a non-compromised machine.

### 4.3.4 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 [18]. With the help of these tools, multiple memory leaks was 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 [17]. Not once did AFL manage to crash the PAM module, which is a good sign.

# 5. Security and System Analysis

## 5.1 Security Token Analysis

The short RSA key used in the project's Version A (keyboard) 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 ciphertext. Thus, it is imperative that 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 discovered, a USB-solution was implemented instead. The big difference with the USB-solution 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. This leads to a design that is small enough to fit onto the Nexys-3. The result was that this implementation took only 1131 out of 9112 available Slice LUTs, had a maximum possible frequency of 134.427MHz and a power usage of 22.02mW. A more detailed summary can be found 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 may be that the USB version uses 50 DSP48A1s (on-chip general purpose FPGA logic) on the bigger Spartan-6 on the Atlys, of which the smaller Spartan-6 on the Nexys-3 only has 32. The USB-version 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. The power usage does not include the power needed for the LCD.

## 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 has been improved and AFL did not manage to adversely affect the program. The analysis was performed quickly and could, therefore, be investigated further for potential flaws.

## 5.3 Complete System Analysis

Without a USB link the key is very small, by default 72 bits, and thus it is required that none of the keys are discovered by an adversary - even the so-called public key. Unlike the recommended RSA usage, the cryptosystem must be thought of as a symmetric one - where the key must be kept secret - in the project's keyboard case 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 easier the shorter the key and the more message transactions are known, also 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.
Thus there are three attack vectors regarding 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 transactions being performed during execution. 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 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 wrongly calculated that a 72-bits would be enough as long as the keys were unknown, however, as soon as this was debunked it became evident that another solution was required. Thus, a modification to the project was made enabling a USB-connection and thus much longer keys, making the system resilient against cryptographic attacks where only the messages are known.

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, especially if only given messages transactions. 512-bit keys are shorter than recommended by institutes such as NIST and is generally not recommended, however, the security is improved since the public key is hidden on a server and the transactions are sent over USB. Using a USB link also prevents people to spy message transaction, on the other hand, it might enable other man-in-the-middle attacks such as the NSA Cottonmouth. However, because of the very limited data to sample, these kinds of attacks will be very inefficient, and a direct attack on the PC itself would be a more logical vector of entry. [31].

If a user requires utmost security it is possible to use this project, however, it is recommended to use replace the RSA_512 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 better. However, using even longer keys would render MTIM-attacks completely useless and removes the burden of securing the public key - which might be a good investment.

# 6. Discussion

## 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 will focus on two-factor solutions specifically.

The project makes it possible for a user to (more) safely log into an organization environment remotely (e.g. via SSH). User mindset is still of utmost importance since an infected computer on their end jeopardizes the 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. However, it is of utmost importance that each sysadmin deploying the security token set these settings as specified in the project files [4] [27].

### 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. Debatably the current design lack enough secure key storage since, even if opened, the key remains synthesized in the FPGA. Hence, if building a custom chip - not done in this project - tamper protection should be implemented. With a custom chip, the program including the key could be stored in volatile memory together with a battery backup - loaded onto the FPGA when turned on. 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* [15].

Even though the key is accessible by opening the device, it is not trivial. 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. Furthermore, the token does not perform random-number-generation or key creation - it 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 the National Institute of Standards and Technology (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 - more than what a user should expect to type in.

A 72-bit key provides some resilience since it requires the attacker to discover the public key or some message transactions. With 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. Furthermore, 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. 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.

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. The greatest weakness of this security token is that even one message transaction could be used to bypass the token, at least in theory.

### 6.1.3 Version B Security Considerations

Instead of having a user enter the clear- and ciphertext 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. 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.

## 6.2 Hardware Programming and FPGA

There were some design choices made early that became core to how the code would be written, and one of them was the usage of generics. Generics are values set at the declaration of a module which can then 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 was also 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 because of this. For example, we reuse a RAM module from a previous course, however, 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 and instead the code written was often sequential.

## 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 laboration with material on USB [45][50]. This helped out tremendously as we received clear instructions on how the protocol was to be interfaced with as well as 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, and as such possibly the internet, 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 way to be able to extract data from the token is if an attacker has physical access to it, knows the PIN and use the predefined commands in a correct way. This is unlikely as if an attacker has physical access, knows the PIN and presumably also knows the password it would be nonsensical not to simply use the token the proper way to log on to the system. Unless an attacker really wants to figure out the encryption key by sampling many signed messages, and physically put in the PIN for each sample, this is scenario is deemed improbable.

## 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, this gives the option to easily replace malfunctioning parts other than the FPGA itself and then reprogram the FPGA to work with them. 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 expansive endeavour.[49] Another solution would be to use a generic ASIC, a SOC such as an Arduino for example. 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.

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.

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 [33] [26]. Security is of utmost importance in society and without it, all would crumble [7].

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, this gives the option to easily replace malfunctioning parts other than the FPGA itself and then reprogram the FPGA to work with them. 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.

## 6.5 Improvements

In the case of the version A 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 functionality. Loading the bitstream file into the FLASH does come with its own security flaws, however. There would be nothing to prevent an attacker to test every possible PIN. The device goes to an unrecoverable state after three incorrect tries are done, but a simple reset the device would reset the counter keeping track of how many incorrect tries has been doneThe counter value should be stored in

the device's FLASH instead of being a volatile signal as it is now. 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 VHDLs 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, 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.

Even though the code is completely open source, 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 [43] [21]. 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 [30].

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 [15]. The code was written 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 [46]. Unfortunately, there was not enough time to perform extensive research and testing of the different flags that 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.

## 6.6 Follow Up

During the projects planning stage[5] it was underestimated how much time different parts of the project would take. The idea was to complete certain functionalities (marks) every two weeks, but this proved hard to succeed with. The first release was postponed by a week because both PAM and FPGA configuration were more difficult and new (i.e. a lot to learn/read) than planned and thus, the second release was, naturally, also delayed. Additionally, the different marks should have been better defined in the planing stage. As we had three marks which essencially only said to improve the previous mark it was impossible to know how long time the improvements would take, as these were decided in the middle of the project.
When version A was completed the security analysis showed that, while the solution worked and did was it was supposed to do, it was not very secure. Therefore, instead of continuing work on the original idea with a user being the communication link, it was found better to use USB. With this switch, we were able to inrease the RSA key length to 512 bits which according to our analysis, was much better. Though this also more space on the FPGA, and the Nexys3 could no longer be used as it proved too small. Instead we used the Digilent Atlys housing the FPGA Spartan-6 XC6SLX45CSG324C[12]. 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, find and use open source modules and solutions.

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 the lack of relevancy. In addition, our decision to use a very flat heirarchy-structure was mostly a success. There were moments when there were 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.

# 7. Conclusion

The first implementation that was created, without USB, is not recommended. 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 long outputs, would work as a replacement for the RSA scheme if one wishes to have a human as the communication channel.

The second implementation 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 serial communication, either one- or two-way, the RSA cryptography will prove to be secure until quantum computing is available - assuming a long enough key is used. The two-factor authentication USB implementation created in the project uses 512-bit RSA, which is sufficient as long as the private key and public key are kept secret.

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

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 and resources 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 this security token more secure than what it is now.

The aim of the project was 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 system can be made quite secure if implemented correctly. However, no system is foolproof but with this two-factor authentication system, a possible point of entry 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* years 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, it quickly enables two-factor authentication on Linux for any PAM-aware application - such as user login, SSH or Samba. 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.

# Bibliography

[1] Will Arthur and David Challener. *A Practical Guide to TPM 2.0.* 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] Mathiesen Christoffer Roxbergh Eliot Örtenberg Gustav Ben Mohammad Johan, Fredriksson Adam. 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].

[6] Duncan Borde. Two-factor authentication. `https://web.archive.org/web/20120112172841/http://www.insight.co.uk/files/whitepapers/Two-factorauthentication(Whitepaper).pdf`. [accessed 9 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] J. Castillo Villar E. Castillo Villar. rsa_512. `https://opencores.org/project,rsa_512`. [accessed 11 May 2017].

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

[16] 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].

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

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

[19] Kenneth Geisshirt. *Pluggable Authentication Modules*. Packt Publishing Ltd, 2007.

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

[21] 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].

[22] 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-targ
html, 28 Mars 2017. [accessed 10 May 2017].

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

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

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

[26] 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].

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

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

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

[31] 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].

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

[33] 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].

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

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

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

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

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

[39] 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].

[40] 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].

[41] 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.

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

[43] 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].

[44] 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].

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

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

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

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

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

[50] 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].

# A. Appendix: Tables and Figures

## 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

## B.1

```
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_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
```