Ossi Paldanius

Reference Cryptographic Accelerator

Implementing AES Algorithm on an FPGA

Helsinki Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master's Thesis

27 May 2018

Metropolia

Author(s) Title Number of Pages Date	Ossi Paldanius Reference Cryptographic Accelerator 54 pages 27 May 2018
Degree	Master of Engineering
Degree Programme	Information Technology
Instructor(s)	Ville Jääskeläinen, Head of Programme, Master's program in Information Technology

Securing data in the evermore networked world of today, is a profound necessity that is embraced by a corporate world as well as an increasing awareness of the internet user base. One of the many important mechanisms of securing data is encrypting the data traffic traversing via public networks, which is a core concept behind this thesis. As the amounts of data transmitted through unsecured or public networks increase rapidly all the time, a demand for a low power consuming, yet highly efficient hardware acceleration solutions for encryption and decryption processes grow higher.

Whenever an encryption is involved in a product, designers need to address issues regarding securing the secret components that are involved in the ciphering process, such as the encryption keys. When introducing a hardware acceleration, the overall device design gets more complex. A careful planning and design trade-offs are to be made to get the security right in a cost effective manner.

For this thesis, a simple FPGA - based hardware accelerator device was built which can be placed between public and private networks. The device encrypts all data sent from a private to a public network and decrypts incoming data from a public to a private network. The key aspects of the design were, that the cryptographic acceleration is performed by a separate FPGA logic circuit, and that there is no possibility to breach the device from a public network side such, that e.g. encryption keys and other sensitive information would be compromised. Beyond describing the implemented device, this thesis discusses the difference between the device and real world products, and sheds some light into the problems that are present when designing them.

The hardware acceleration for designs using FPGAs, can be highly effective for increasing the computational performance of the cryptographic algorithms, if the algorithm used supports concurrency. The data transmission between different nodes in the system has to be carefully designed and implemented to be able to provide the increased throughput requirement. Also when a separate cryptographic accelerator nodes are added to the design, the security can be increased if properly designed.

Keywords	AES, FPGA, Cryptography

Contents

Abbreviations

AES	Advanced Encryption Standard.
ASIC	Application Specific Integrated Circuits.
CBC	Cipher Block Chaining.
CLB	Configurable Logic Blocks.
CLI	Command-line Interface.
CPU	Central Processing Unit.
DSP	Digital Signal Processor.
ECB	Electronic Code Book.
FPGA	Field Programmable Gate Array.
GPGPU	General-purpose Computing on Graphics Processing Units.
GPIO	General Purpose Input/Output.
HDL	Hardware Description Language.
HSM	Hardware Security Modules.
HW	Hardware.
IRQ	Interrupt Request.
IV	Initialization Vector.
JTAG	Joint Test Action Group.
LE	Logic Elements.
MAC	Multiply And Accumulate.
NIST	National Institute of Standards and Technology.
OSS	Open-source Software.
PC	Personal Computer.
PLD	Programmable Logic Devices.
RTL	Register Transfer Level.
SBC	Single Board Computer.
SPI	Serial Peripheral Interface.
SSH	Secure Shell.
SSL	Secure Socket Layer.
SW	Software.
TCP	Transmission Control Protocol.
UART	Universal Asynchronous Receiver/Transmitter.
VHDL	Very high speed integrated circuit Hardware Description Language.
VPN	Virtual Private Network.

1 Introduction

Cryptography can be described as study or practices to obfuscate information in such a way, that original contents of the information cannot be understood by people or entities who do not possess the means to decipher the information. It has been used since ancient times to secure written messages between counterparts for example for military purposes or to prevent otherwise sensitive information to fall on wrong hands [1] [2].

This day and age, computerization is involved in everything we do. As part of it cryptography has become an integral part of our everyday life. Whether it is used to protect company confidential information in our work laptops, or to keep our Google searches out of sight of evermore prying eyes lurking on the Internet. Protection of our privacy and intellectual properties are important but only a small part of applications computer based cryptography is applied [3]. One could go as far as to say that all data that is not supposed to be shared publicly, could be worthwhile to be encrypted. This is becoming reality as computing power for any given task increases all the time, lowering the computational cost of encrypting and decrypting user data. Furthermore, many integrated circuits, whether in cell phones, data centers, or in highly specific devices house high-speed accelerator blocks for providing powerful and transparent acceleration for wide variety of computing tasks. This is also true for cryptographic applications.

This thesis describes the implementation of a reference cryptography accelerator using Advanced Encryption Standard (AES) [4], which has very wide adoption at the time of writing. Aim is to provide essentials of a practical approach to implementing AES encryption and decryption algorithms on a Field Programmable Gate Array (FPGA). Important feature is also the ability to enhance and to develop further the design provided during the work of this thesis. To support this requirement, two low cost Linux computer units were used to interface the FPGA on a development board, to provide network access and configuration interface.

Thesis is provided for a case company, which requires a functional reference device with mentioned characteristics for internal study of hardware accelerated cyprographic de-

vice, especially FPGA based one. In addition to the device, Study is done on how to implement FGPA based cryptographic accelerator in an efficient manner and with tight security. Some of the findings are implemented in the device design, and some more are discussed in pages of this document. As the field of cryptography and hardware acceleration is wide, this thesis concentrates on a Virtual Private Network (VPN) like network appliance. In other words a gateway device, which may be placed between public and private network, encrypting all data send to public network. To be even more precise, only algorithm performance, data transmission and intrusion prevention topics are discussed in this thesis.

Following documentation is divided between following chapters. Material and Methods explains device requierements in more detail, as well as how design, implementation and study process is done as a whole. Evaluation criteria for the thesis is also discussed. Background on Used Technologies chapter takes a look into cryptography and its hardware accelereation. AES is explained in simplified manner, to ease understanding device implementation and result evaluation in further chapters. Implemented Reference Device chapter goes into details of the implemented device, its hardware components, overall operation, FPGA design and Linux software architecture. Used development environment is also briefly visited. Results and Analysis explores the implemented device from the requirements point of view, and discusses security and performance related topics for FPGA based cryptographic accelerator design. Conclusion and Summary is a brief overlook and wrap up on topics mostly discussed in Results and Analysis chapter.

2 Material and Methods

This section aims to provide basic look on some of the concepts used as well as introducing requirements for the work. Also used tools and methods are looked into, as well as evaluation criteria.

2.1 Requirements

Original requirement for this thesis is to provide a working, proven reference logic design on an FPGA of a cryptographic algorithm. In addition to that, it is important to open up some of the design topics that concern cryptographic design on a hardware. Like mentioned AES was chosen to be the implemented algorithm due to it's wide usage in the industry. Testability and adaptability were also important factors, so an actual device is needed to be put together. Device needs to have network connectivity so that it can operate autonomously between networks very much like a simplified VPN gateway. Device will have two network interfaces, one towards private or secure network where data can be handled in unencrypted form, and the other towards for example public network to/from data must be handled encrypted. It is important that there is no access to FPGA design internals, such as the key used for encryption/decryption from the public network side. It should not be possible to access the private network even if access to the device is compromised from the public network side. AES Key management and other configuration should be able to be handled from the trusted side. For AES, 256-bit key size is to be used (explained in detail in section 3.4).

Network interfaces are requested to be handled by Linux computer units for simple usability and expandability in future.

Cost of this entire project is to be kept to a minimum. This does not only concern the device itself to be put together, but also all the tools used for the development work have to be cheap or no-cost as well. It is far simpler to continue experimenting with the reference device if the equipment used is readily available and does not require major investment

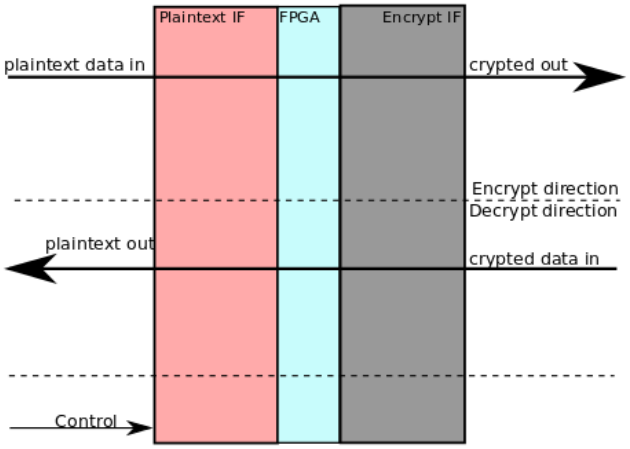

Figure 1: Overall view of the device requirements

of either developer time, Hardware (HW) resources or Software (SW) licenses.

2.2 Design Process

Requirements for the project were rather clear from the beginning. This made it easy to define the work ahead, and they can be divided roughly into following categories.

Study of AES and Relevant Cryptographic Topics

Quite obvious first step is to invest time on studying the actual algorithm and cryptography in general. It was important to scout for e.g. other possible implementations of AES on FPGA:s. That study was important step before selecting FPGA hardware, for which the project were be developed. Besides just AES algorithm itself, other security related requirements or concerns were in play as well. Thus it was important to gain more understanding about those topics.

Hardware Selection & Overall Device Design

After real requirements for AES implementation were better understood from the FPGA point of view, selection of HW components could be done. Core of the design were an FPGA development kit for sure, with possible additional circuit boards to support network access and configuration. This was the stage where overall design was locked in place as well. Key elements of interfaces between FPGA and Central Processing Unit (CPU)

running Linux were to be defined.

Tools Selection

Choice of design and testing tools are not exactly separate from the actual HW selection process. In fact, it was quite crucial that design tools are easily taken into use and provide ways to rapidly ramp up development process. As the actual demands for components used for the device were not that specific, Tools availability did dictate choice of HW to a large extent. Open-source Software (OSS) solutions were favored if at all feasible.

Algorithm Implementation and Testing in Simulator

After previous steps had been completed to a point where the project was quite well defined, actual implementation work could commence. This work stage were separated between implementation and testing while using a simulation SW, and testing in real FPGA. Development and simulation was done as a SW development task entirely in a Personal Computer (PC) environment using Hardware Description Language (HDL) to describe the algorithm logic.

Adapting Design to Hardware

Once tested in simulation environment, AES HDL design were synthesized to a real FPGA and tested. At this point Linux CPU interfaces need to be taken into use as FPGA must be fed with test data, and results needed to be extracted as well. In this phase interfacing all components of the device were to be addressed, and some form of configuration interface needed to be implemented too.

Linux Software Implementation and Testing

Last development phase was writing software for Linux environment. That SW would make configuration and control of the device possible, and it would feed the data to the encryption FPGA.

2.3 Evaluation

First obvious evaluation criteria for this project is whether the device works as intended or not. If the device fails to operate, none of the design work learning or produced source codes matter. However, as a reference device, implementation itself is of even more importance than the absolute functionality. Thus once working, all the source code written especially for the FPGA is of interest and must be clearly implemented and properly documented. Design choices will be clarified in this document, and some effort is put to reflect those choices to cryptographic HW designs in real life products.

3 Background on Used Technologies

This Chapter aims to open up key areas of interest of this thesis for the reader. These include brief overlook into cryptography, FPGA technology, cryptographic HW design and AES cryptosystem itself. There are other technologically relevant parts like Linux application and network programming among others. They are required part of fulfilling the requirements for the produced device, but are not the main focus. Those topics are therefore not discussed here and are only briefly touched in chapter 4.

3.1 Foreword on Cryptography

Cryptography is a very large and complex topic, and it is discussed mostly from AES perspective in section 3.4. This chapter just describes what encryption and decryption mean for this particular thesis so that requirements of it can be understood better. From now on, cryptography is only discussed in digital, or computerized context.

Encryption and Decryption in digital Cryptography

Encryption process is basically converting *plaintext* data to *ciphertext*, by applying a mathematical algorithm that obfuscates the ordinary, readable plaintext information into seemingly unreadable ciphertext. When that process is reversed to convert ciphertext into plaintext, it is called decryption. To make the encrypted ciphertext difficult to decrypt for unwanted parties, proper algorithms (such as AES) use specific secret key in the ciphering process. So to decrypt ciphertext, both the key and used algorithm is needed to be known [5].

Symmetric Key vs. Public Key Algorithm

Symmetric key algorithm is an algorithm that uses the same key for both encryption and decryption. AES is a symmetric key algorithm. Positive aspect of this is the simplicity, since only one key is needed to be used. Downside is the security problem, since all

trusted parties need to know the same secret key in order to decipher the ciphertext. Also it is clear that symmetric key approach is not feasible when there are e.g. multiple clients and one server instance, which are not allowed to access other clients' data. As then it would be necessary to have multiple key pair combinations for each client. This also poses a key management and negotiation problem.

Public key (or asymmetric key) algorithm on the other hand requires a key pair. One public and one secret. Both are generated by the same party, and they are related to each other. Public key is used for encryption and private (secret) is used for decryption. In this way it is possible to distribute the public key to anyone who would want to encrypt data for only the holder of the private key to be able to decipher [5]. Public/private key concept is used today in many applications like digital signing, securing web traffic via Secure Socket Layer (SSL) [6], and many more. Public/private key technology can be also used to trade symmetric keys like is done in many Virtual Private Network (VPN) solutions, thus solving the key management and negotiation problem mentioned .

3.2 Field Programmable Gate Array

FPGAs are reprogrammable integrated digital logic circuits, which belong to Programmable Logic Devices (PLD) [7]. PLD is a digital circuit, which can be programmed in target circuit board multiple times after manufacturing the board. Since the introduction of FPGA technology in mid 80's [8], FPGAs have gained wide adoption in different fields of embedded computing, and is expected to grow in popularity as a market [9] [10].

Fundamentals

Basic building blocks of a traditional FPGA are I/O blocks, Configurable Logic Blocks (CLB) and interconnects between them. CLBs containing logic elements or logic cells are main building blocks which are configured, or programmed to perform logic operations as required by the design. I/O blocks are attached to I/O pads of the chip which connect to external circuitry.

As simplified illustration of FPGA internal structure show, CLBs can be arranged as an

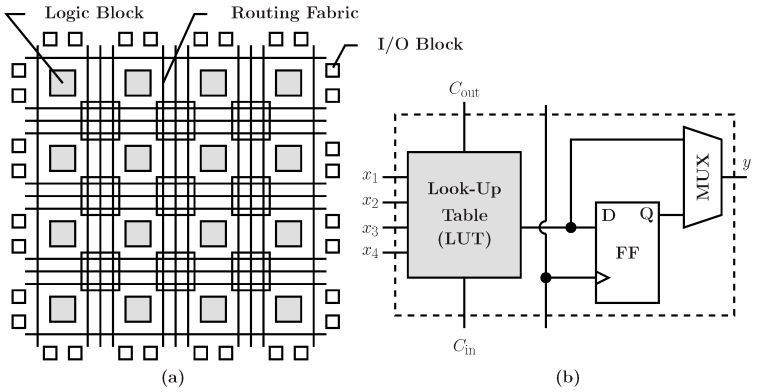

Figure 2: Basic FPGA block diagram & logic element sketch [11]

array separated with interconnects. This is what ”Gate Array” stands for in FPGA. Modern day FPGAs have other types of CLB placement schemes, where multiple CLBs are clustered together for faster connectivity between them. These are utilized by design software to produce more optimized designs.

Modern largest FPGAs can house several million Logic Elements (LE). In addition to that, FPGAs of today often contain hard IP elements in addition to programmable logic. These hard IP elements include fast I/O controllers like PCI Express controllers, Multiply And Accumulate (MAC) accelerators for DSP solutions, external memory controllers and even complete multiprocessor clusters. Hard IPs are built-in functionality which user cannot alter like generic FPGA fabric, but may use to support own FPGA logic design. Larger and more sophisticated FPGAs can be viewed as System On Chips or SoCs due to ever increasing use of hard IP technology [12] [13].

FPGA internal logic is defined traditionally by first writing description of the logic with HDL, such as Verilog or Very high speed integrated circuit Hardware Description Language (VHDL). HDL source code is compiled and synthesized to a binary programming file called bitstream, which is programmed to the target device by separate programmer integrated circuits, a CPU connected to FPGA on same circuit board or via Joint Test Action Group (JTAG) boundary scan connection. HDL compilation, synthesis and JTAG operations are performed with vendor locked-in tools provided by FPGA manufacturers. Writing HDL designs by hand is not the only method of describing logic in FPGAs, as vendors and third parties offer soft IP cores and functions, a HW designers equivalent to software libraries, which can be applied to FPGA without having the need to write them.

3.3 Hardware-based Cryptographic Acceleration

Executing cryptographic algorithms may gain significant performance boost when run on dedicated accelerators, even when compared against DSP based implementations [14]. Other important reason for offloading cryptography onto hardware may be improved security. Hardware acceleration or offloading can be implemented in different ways, which are briefly mentioned here. chapter 4 is more focused on the FPGA solution implemented in this thesis and more detailed comparison to other FPGA based solutions are discussed there.

3.3.1 Implementation methods

Cryptographic hardware acceleration can be done many ways, with different levels of security, feasibility for the application design and performance profile. These include:

- host processor instruction set extensions
- generic purpose HW accelerators
- on-die co-processor core
- separate cryptographic processor
- separate crytpomodule device.

Many general purpose CPU manufacturers offer built in extended instruction sets for accelerating crypto algorithms, especially for AES, such as many x86 and ARM vendors [15] [16] [17]. Extended instruction set offers improved performance for software execution, but do not offer any added security. Usability of instruction set extension approach is thus limited to any environment where software solution is acceptable. Similar security environment is applicable when generic purpose accelerators are used, such as modern graphic cards. They offer massive gains in algorithm performance, especially when parallel execution can be utilized to the fullest [18].

On-die co-processors are common in SoCs, or specific application processors. Common usage for co-processor is a security co-processor which is responsible for security fea-

tures for the device itself. Example of such devices are device managers in high end FPGAs, where they maintain e.g. security of the FPGA bitstream [19]. Separate cryptoprocessors and cryptomodules are similar in the sense that they may connected to a client device (such as a CPU) via separate bus or link. One major example of separate cryptomodule or cryptoprocessor use is Hardware Security Modules (HSM) [20], which may be used in for example secure key management and securing other important assets of the product that are needed to be kept out of hands of the application in case of being compromised [21]. General-purpose Computing on Graphics Processing Units (GPGPU) technology is also feasible platform for hardware acceleration. GPUs offer vast amount of computational resources if solution can be parallelized efficiently, which is the case for many cryptographic applications [22].

3.3.2 Hardware acceleration security

Like many other subsections in this thesis, this topic is too wide to be adequately discussed here. Since this topic has been important for the implementation of the device in this thesis, it is briefly touched to aid in understanding the design.

If HW is used only to accelerate algorithm execution as is in case of instruction set extensions, added security is of no concern. Separate cryptocores, processors or modules, may offer increased security if so designed. Key principle in separate cryptomodules can be seen as a device which completely handles all cryptography related operations without client application interference. That is, application simply uses security module as a service. This way even erroneous, buggy or malicious software cannot access secrets such as encryption keys that reside in cryptomodule. How this requirement is achieved, is entirely based on given application. FIPS Pub 140-2 standard [23] defines levels of security for cryptographic modules, as well as design guidelines which can be utilized.

3.3.3 FPGA vs. ASIC

As mentioned, FPGAs and other PLDs can be reprogrammed after manufacture. It means their internal structure can be rewired to produce different logic circuit each time. Application Specific Integrated Circuits (ASIC)s are integrated circuits manufactured once without

a possibility of reprogramming. FPGA and ASIC as technologies are often viable solutions for same type of applications, but each technology has its own advantage over the other. Main advantages of ASICs over FPGAs are cost per unit produced when manufactured in high volumes, and more optimized chip design make higher performance possible. ASIC has an advantage also when power consumption is considered. On the other hand, FPGAs are easier to develop and have shorter time to market. The most obvious benefit of FPGAs is of course reprogrammability in target and on site (hence the term ”Field Programmable”). This makes design alterations and corrections possible after shipping the product.

Thus FPGAs have been generally used in high end and low volume products in the past. FPGA technology has improved over the years however, offering more capacity and reducing power consumption. This has made it possible for the FPGA technology to not only be used more in traditional ASIC domain, but also to gain foothold over applications run on CPUs or Digital Signal Processor (DSP)s [7] [24] [25].

3.3.4 FPGA vs. Software Implementation

FPGAs excel in areas where custom I/O or a lot of parallelism is required, or simply when product requires high level of custom logic integration. Also when design has really tight real time requirements FPGA based solutions is often more feasible choice. Software based solutions may be more easily portable between products. Development of software based solutions may often be faster due to high availability of development & testing environments. Embedded operating systems are readily available for practically all processors sold with support from the manufacturers. Also many complex algorithms (i.e. with lot of branching) are easier to develop with software. However, different processor types such as DSPs enhance software algorithm performance in suitable scenarios. [26].

Because of the very different nature of using FPGA and purely software solutions, it has been increasingly viable to deploy both methods. This has become more accessible since SoC solutions are available throughout FPGA manufacturers’ product portfolios [12] [13]. SoCs with hard IP CPU cores are not necessarily needed though. FPGA chip manufacturers & third parties provide various soft IP CPUs which can be deployed to any FPGA

with enough capacity, offering decent solution for executing e.g. control code where high clock speeds are not required. OpenCL language which is commonly used for GPGPU programming, can also be applied to FPGA fabric directly [27]. Instead of using FPGA as an accelerator controlled by separate CPUs or those inside a SoC, it is possible to deploy Register Transfer Level (RTL) designs written in HDL alongside openCL instantiations [27] [28].

3.4 Advanced Encryption Standard

AES is a complete crytposystem [5] standardized by U.S. National Institute of Standards and Technology (NIST). It is sometimes referred to as Rijndael, the name which is derived from original designers of the algorithm, Vincent Rijmen and Joan Daemen [29]. Reason for the duplicate naming is that NIST held a selection process for new Advanced Encryption Standard, and slightly reduced version of Rijndael was selected [30].

AES is documented in its standards release [4], but a simplified walk-through is presented here since it's internals are important for understanding FPGA implementation described in chapter 4.

3.4.1 General view

AES is a block cipher algorithm, for which the block size is set to 128 bits. This means that algorithm encrypts or decrypts user data 128 bits (16 bytes) at a time. From algorithm point of view those 16 bytes form a 4 by 4 matrix called the State. Like so:

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

AES cipher and decipher (decrypt) operations are divided into steps, or 'transformations', which all operate on a given State. All steps operate on a row of 4 bytes, on a column of 4 bytes, or on single bytes during a single round of operation. AES supports key sizes

of 128, 192 and 256 bytes. It depends on key size how many rounds of operation are subjected to each State.

Table 1: AES Key size to # of rounds

Key Size	# of rounds
128	10
192	12
256	14

As requirement is to use 256-bit key size, 14 rounds are applied in the implementation of this reference device. AES standard uses pseudo code functions to explain algorithm routines, which are included in this chapter for convenience. In them several variables are used, which are key length, block size and number of rounds to be applied. block size and key length are presented in number of 4-byte words.

Table 2: Variables used in pseudo code blocks

Nk (Key Length)	8
Nb (Block Size)	4
(Nr (# of rounds)	14

3.4.2 Cipher

When State (16 bytes) of plain text data is encrypted, it goes through cipher algorithm. It utilizes repeatedly independent transformation operations called SubBytes, ShiftRows, MixColumns and AddRoundKey. Transformations are executed in # of rounds +1 times (15 rounds for 256-bit key), of which first and last rounds differ from each other and the core rounds in between, which for 256-bit key are run 13 times.

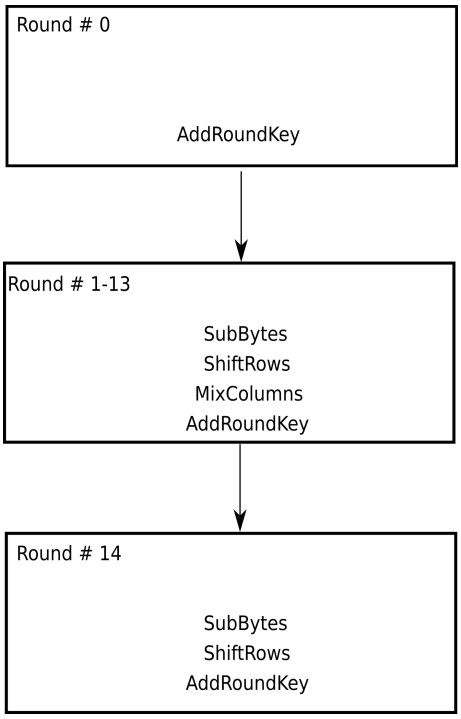

Figure 3: Cipher Transformations during rounds

First round takes the original plaintext state as input, but further rounds each take the output of the previous round as the input data. Logically each transformation function uses output of the previous transformation function as its input. Following subsections discuss transformations in more detail. Cipher algorithm is presented in pseudo code in AES standard, shown in Listing Listing 1, from which Figure 3 is derived from.

```
1  Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
2  begin
3      byte state[4,Nb]
4      state = in
5      AddRoundKey(state, w[0, Nb-1])
6      for round = 1 step 1 to Nr-1
7          SubBytes(state)
8          ShiftRows(state)
9          MixColumns(state)
10         AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
11     end for
12     SubBytes(state)
13     ShiftRows(state)
14     AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
15     out = state
16 end
```

Listing 1: Pseudo Code for Cipher as in AES Standard

SubBytes() Transformation

In SubBytes, each byte of the input State is substituted by a corresponding byte in static 16 by 16 byte matrix table called S-Box, which contains 16*16=256 predefined bytes. S-Box contents is defined in AES standard [4]. Substitution selection works by using 1st half of input byte as row indicator and 2nd half of input byte as column indicator for selecting byte from the S-Box table, which will replace the byte in the input State. Below is an example illustrating SubBytes transformation.

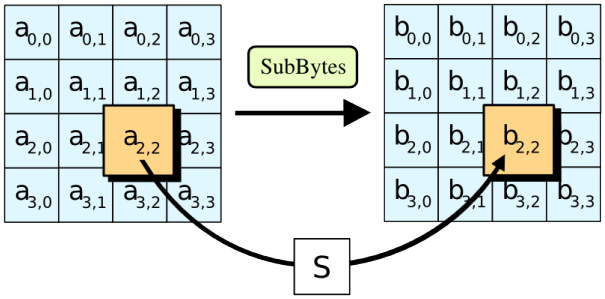

Figure 4: SubBytes() operation [31]

Mathematical background explaining how table entries are derived is explained in Sub-Bytes chapter 5.1.1 in [4]

ShiftRows() Transformation

ShiftRows rotates bytes to left in 2nd, 3rd and 4th row of the input State by one, two and three bytes respectively.

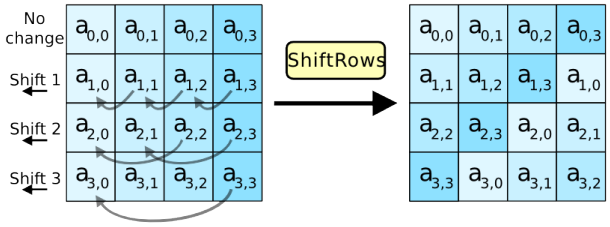

Figure 5: ShiftRows() operation [32]

MixColumns() Transformation

Mathematically, MixColumns is the most involved operation of AES cipher. MixColumns operate on each of the four columns in input State separately, to produce new column to output.

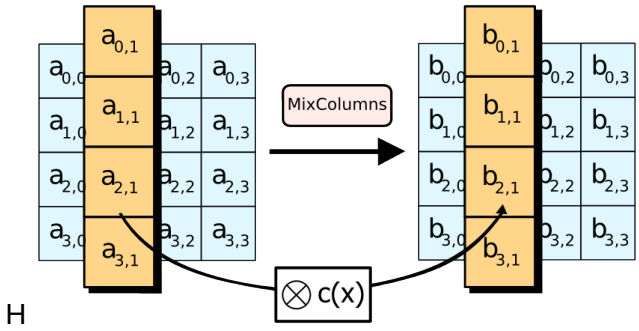

Figure 6: MixColumns() operation [33]

All four bytes in the column are used as operands to calculate contents of the output column, and are treated as four term polynomial as described in chapter 4.3 of the AES specification. The columns are considered as polynomials over finite field (or Galois Field) $GF(2^8)$ and multiplied modulo x^4+1 with a fixed polynomial $a(x)$, defined as:

$$(x) = 03x^3 + 01x^2 + 01x + 02 \tag{1}$$

As the specification describes, it can be seen as matrix multiplication where all bytes in the column b_i (input) are replaced by b_o (output):

$$\begin{bmatrix} b_{i,0} \\ b_{i,1} \\ b_{i,2} \\ b_{i,3} \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} b_{o,0} \\ b_{o,1} \\ b_{o,2} \\ b_{o,3} \end{bmatrix} \tag{2}$$

Which yields according to AES specification:

$$
\begin{aligned}
(02 \bullet b_{i,0}) \oplus (03 \bullet b_{i,1}) \oplus b_{i,2} \oplus b_{i,3} &= b_{o,0} \\
b_{i,0} \oplus (02 \bullet b_{i,1}) \oplus (03 \bullet b_{i,2}) \oplus b_{i,3} &= b_{o,1} \\
b_{i,0} \oplus b_{i,1} \oplus (02 \bullet b_{i,2}) \oplus (03 \bullet b_{i,3}) &= b_{o,2} \\
(03 \bullet b_{i,0}) \oplus b_{i,1} \oplus b_{i,2} \oplus (02 \bullet b_{i,3}) &= b_{o,3}
\end{aligned} \tag{3}
$$

Here $\oplus$ denotes logical exclusive OR operation and $\bullet$ denotes multiplication in $GF(2^8)$ [4]. In short, the specification explains that in byte level, multiplying byte with a constant $x = 02$, can be implemented as a left shift and bitwise exclusive OR with $1b$. Specification uses $xtime()$ to denote this operation:

$$b_{i,x} \bullet 02 = xtime(b_{i,x}) \tag{4}$$

It also states that multiplying bytes by higher powers (of two) of x are implemented by repeating the $xtime()$ procedure, and that any constant can be used in multiplication by adding the intermediate results. As addition in $GF(2^8)$ is handled by exclusive OR operation, multiplication with constant 03 becomes:

$$b_{i,x} \bullet (01 \oplus 02) = b_{i,x} \oplus xtime(b_{i,x}) \tag{5}$$

To summarize, even as MixColumns() is more computationally demanding transformation than the other ones used for cipher, it still boils down to a series of bitwise shift left and XOR operations on bytes in each four columns.

AddRoundKey() Transformation

In AddRoundKey, corresponding round key from the Key Schedule is added to the input State. Round key generation is touched in subsection 3.4.4 As Key Schedule can be interpreted as size 15 array of 4-byte words, key to apply each round is indexed starting from the beginning. Transformation is done by bitwise exclusive ORing each column in input State with corresponding 4-byte word in round key. Or to be precise, corresponding

bytes between those.

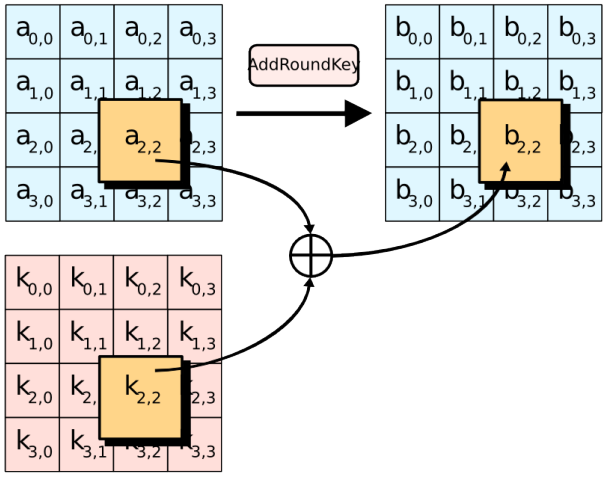

Figure 7: AddRoundKey() operation [34]

3.4.3 Inverse Cipher

When encrypted ciphertext is being decrypted, it is run through inverse cipher algorithm. Process itself is very similar than cipher, but with several changes. SubBytes, MixColumns and Shiftrows transformations have inverse equivalents. AddRoundKey does not, since it is basically a XOR operation. As AES is a symmetric key algorithm, same key will be used for inverse cipher, and same Key Schedule as well. Just that when traversing through the rounds, round keys are applied in reverse order compared to cipher.

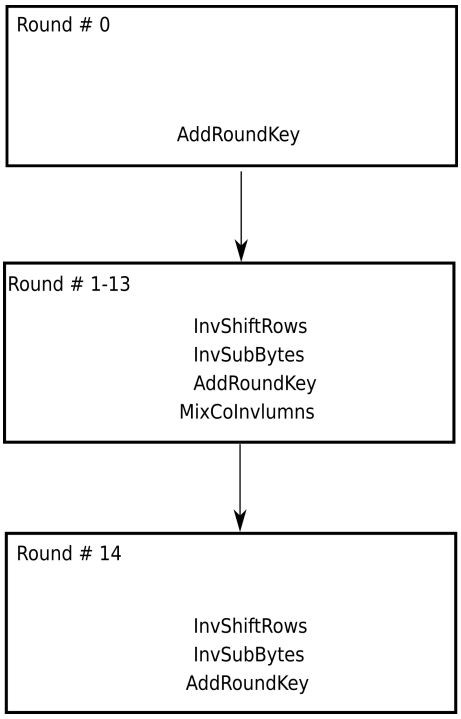

Figure 8: Inverse Cipher Transformations during rounds

Pseudo code for Inverse Cipher from the standard is in Listing Listing 2.

```
1  InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
2  begin
3      byte state[4,Nb]
4      state = in
5      AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
6      for round = Nr-1 step -1 downto 1
7          InvShiftRows(state)
8          InvSubBytes(state)
9          AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
10         InvMixColumns(state)
11     end for
12     InvShiftRows(state)
13     InvSubBytes(state)
14     AddRoundKey(state, w[0, Nb-1])
15     out = state
16 end
```

Listing 2: Pseudo Code for Inverse Cipher as in AES Standard

InvShiftRows() Transformation

InvShiftRows is similar to ShiftRows, but instead of rotating rows to the left, they are rotated to the right.

InvSubBytes() Transformation

InvSubBytes is exactly the same operation as SubBytes, but with a different 16 by 16 bytes static matrix called Inverse S-Box. Static values in Inverse S-Box are designed such, that when SubBytes output is used as an input to InvSubBytes, original input to SubBytes is returned.

InvMixcolumns() Transformation

InvMixColumns() is very similar to MixColumns() transformation, but to produce the inverse, different fixed polynomial is used:

$$ai(x) = 0bx^3 + 0dx^2 + 09x + 0e \tag{6}$$

Now that turns out as as matrix multiplication where all bytes in the column b_i (input) are replaced by b_o (output):

$$\begin{bmatrix} b_{i,0} \\ b_{i,1} \\ b_{i,2} \\ b_{i,3} \end{bmatrix} \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} = \begin{bmatrix} b_{o,0} \\ b_{o,1} \\ b_{o,2} \\ b_{o,3} \end{bmatrix} \tag{7}$$

Which yields according to AES specification:

$$
\begin{aligned}
&(0e \bullet b_{i,0}) \oplus (0b \bullet b_{i,1}) \oplus (0d \bullet b_{i,2}) \oplus (09 \bullet b_{i,3}) \\
&(09 \bullet b_{i,0}) \oplus (0e \bullet b_{i,1}) \oplus (0b \bullet b_{i,2}) \oplus (0d \bullet b_{i,3}) \\
&(0d \bullet b_{i,0}) \oplus (09 \bullet b_{i,1}) \oplus (0e \bullet b_{i,2}) \oplus (0b \bullet b_{i,3}) \\
&(0b \bullet b_{i,0}) \oplus (0d \bullet b_{i,1}) \oplus (09 \bullet b_{i,2}) \oplus (0e \bullet b_{i,3})
\end{aligned} \tag{8}
$$

So in InvMixColumns() higher powers of multiplicands are used than in MixColumns(). This leads to more computationally demanding procedure with more $xtime()$ operations. For example when input byte $b_{i,x}$ is multiplied by $0e$ we see the operation becomes:

$$
\begin{aligned}
b_{i,x} \bullet (02) &= xtime(b_{x,1}) = xt_2 \\
b_{i,x} \bullet (04) &= xtime(xt_2) = xt_4 \\
b_{i,x} \bullet (08) &= xtime(xt_4) = xt_8
\end{aligned}
$$

thus,

$$
b_{i,x} \bullet (0e) = xt_8 \oplus xt_4 \oplus xt_2 \tag{9}
$$

Same logic applies when multiplied by 09, 0b and 0d. Not only does multiplications need more xtime() operations as coefficients are of higher power, but now each member of the column polynomial have to be multiplied with with a higher power coefficient. That is, compared to MixColumns() transformation. This has implications for logic design as is explained in subsection 4.3.5.

3.4.4 Key Expansion

Given 256-bit key is first needed to be expanded to a set of 128-bit round keys called Key Schedule. Each round key in key schedule are interpreted as four 4-byte words. The amount of round keys equal to number of rounds +1. This process in AES is called Key Expansion. Each of the round keys in key schedule are used by cipher and inverse cipher states during different rounds of operation. Key Expansion implementation is described in detail in chapter 5.2 in [4], but here it is described in a bit more simplified manner.

Key Expansion is, like cipher and inverse cipher, illustrated as a pseudo code function in AES standard.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
    word temp
    i = 0
    while (i < Nk)
        w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
        i = i+1
    end while
    i = Nk
    while (i < Nb * (Nr+1))
        temp = w[i-1]
        if (i mod Nk = 0)
            temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if (Nk > 6 and i mod Nk = 4)
            temp = SubWord(temp)
        end if
        w[i] = w[i-Nk] xor temp
        i = i + 1
    end while
end
```

Listing 3: Pseudo Code for Key Expansion as in AES Standard

When using 256-bit key, standard defines Nk, Nb and Nr to be as shown in Table 2. Pseudo code uses two functions, $RotWord()$ and $SubWord()$. $RotWord()$ performs cyclic permutation on the input, producing output as $[a_0, a_1, a_2, a_3]$ to $[a_1, a_2, a_3, a_0]$. $SubWord()$ substitutes each of the four bytes with $S - Box()$ used by cipher as well (see Subbytes in subsection 3.4.2). Rcon[] is a 256-byte array which contents are calculated as described in the standard. It can be precalculated and hard coded for the execution of the algorithm to simplify implementation. For 256-bit Key Schedule only Rcon[] index values 1 to 7 are used.

For 256-bit AES KeyExpansion has one 256-bit input argument *key*, presented as 32-byte array in the pseudocode, which is the original 256-bit AES key. It has one 1920-bit output argument, presented as an array of 60 words (4*byte). In first phase, original input key is copied to in the beginning of the output data array. It will be used as such for first round key for cipher. During second phase, KeyExpansion uses previous word in the output array (starting from the last in the original key), and performs operations on it, adding it to the output array once done. Simply put, second phase performs exclusive OR operation

between previous word in the current output array, and the word located 8 indexes before that, for each word it creates. In addition to that, every 4th word is manipulated before that. For every 8th word (starting from the 1st index after original key), bytes in the word are shifted by *RotWord*, replaced by *SubWord*, and exclusive ORred with incrementing word from the precalculated Rcon table. For Every 8th word (starting from the 4th index after original key), bytes int the word are substituted by *SubWord*. Second phase is run 52 times in total filling the output array, which then consists 1920/128 = 15 round keys.

3.4.5 Mode of Operation: Cipher-block Chaining - CBC

As AES is a block cipher, it encrypts or decrypts data in blocks. When the amount of data to be encrypted or decrypted exceeds block size of the cipher, most simple solution is to divide data to 16-byte blocks and process each separately [5]. This is the most basic mode of operation for block cipher and is called Electronic Code Book (ECB). Since ECB mode not very secure, other modes of operation have been developed, varying in complexity and security. National Institute of Standards and Technology has released recommendations and explanations of several of them [35]. Out of the many available, Cipher Block Chaining (CBC) was selected early on for this project as it is rather simple to implement, but does offer improved security over ECB mode. CBC works, as the name implies, by chaining subsequent processed blocks in such a way, that input of the next state is dependent of the output of the previous one.

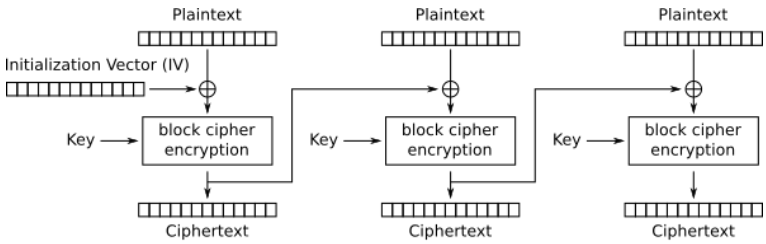

Figure 9: Cipher Block Chaining mode of operation: Encrypt [36]

In encryption, first plaintext 16-byte state is first XORred with a 16-byte data block called Initialization Vector (IV), before it is encrypted with AES. Every subsequent 16-byte plaintext state is in turn XORred with previously encrypted ciphertext 16-byte state. See Figure 9.

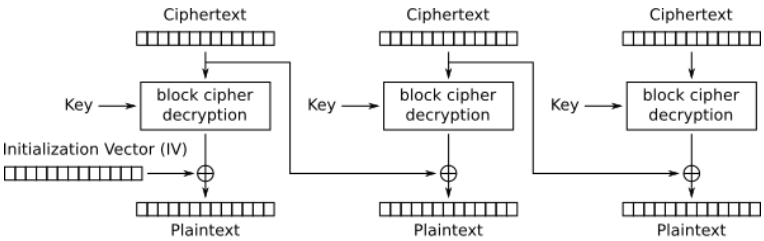

Figure 10: Cipher Block Chaining mode of operation: Decrypt [37]

When data is decrypted, it must be processed in same block order as it was encrypted. After first 16-byte ciphertext state is decrypted, its output is XORred with same IV as was used in encryption. This works since exclusive OR operation is its own inverse. In ciphertext, every following decrypted plaintext 16-byte state is XORred with ciphertext of the previous 16-byte state, and thus CBC process is reversed. Decryption is illustrated in Figure 10.

Counter mode (CTR) of operation was also considered, mainly for its appealing ability to allow ciphering and deciphering of 16-byte blocks in parallel. CBC was selected for this thesis due to simpler implementation.

4 Implemented Reference Device

This chapter describes the device implemented for this thesis. The hardware components and software design is explained, as well as choices that were made during selection process.

4.1 Hardware Components

Even though a HW acceleratior is designed, this thesis can be seen as a software project. There would be enough effort to get everything done without having to work on any specific hardware, so efforts put on HW were to keep as small as possible.

4.1.1 Selection Process

Requirement demands that FPGA is able to house AES encrypt and decrypt data paths, and that both plaintext and ciphertext domains have a network access and are operated under Linux operating system. Evaluation of commercial, off-the-shelf products was conducted in the beginning of the project, to be able to build a device that would satisfy those requirements. Selection was based on these key principles:

- suitability
- availability
- price
- availability of development tools
- ease of use.

List above is only roughly in order, since all of them needed to be fulfilled. Suitability study was begun by first studying available documented AES implementations on FPGA, and estimating the FPGA circuit that was sufficient for the device. It was soon clear that fairly small FPGA would be capable enough, so that left many development boards and

FPGAs available for the project, which would fit the budget and were readily available from multiple vendors. It was important that development tools were available without much of a licensing cost or hassle, and are professional enough for the all development tasks regardless. Ease of use was an overall view on how much effort it would take to take any product into use and tailor all tools to satisfy the needs of developing this device.

The selected target was to have as open source friendly environment as possible, but without sacrificing any of the above mentioned key principles. It became evident very soon that open source design or synthesis tools were not available for FPGAs on sale, if not considering some projects on select devices based on reverse engineered bitstreams built with official tools. Therefore selection for the FPGA was biased on behalf of devices made by Altera, for two major reasons. Author had experience working with several Altera FPGA products in the past, so there was familiarity with the development tools, making for a rapid ramp up in starting of the development. Also Terasic corporation had a small, affordable and yet highly suitable development board on the market called DE0-Nano [38], based on Altera Cyclone® IV [39] FPGA device.

After selecting the FPGA dev board which did not have any CPUs or network interfaces in it, separate Linux computer unit was needed. Raspberry Pi [40] is a very popular ARM based Single Board Computer (SBC), which filled the requirements. It has wide user base and is simple to take into use for rapid development and prototyping, yet capable of running full Linux operating system. There was however, a clear requirement that on ciphertext domain, FPGA internals must be decoupled from the Linux operating system in that domain. This requirement was tackled with adding second raspberry PI for the ciphertext side.

4.1.2 Raspberry Pi

Raspberry Pi is a credit card size development board produced by Raspberry Pi foundation. At the time of implementing the device, Raspberry Pi 3 model B was the current model, which was used for this project. It is based on 4 core 64-bit Broadcomm BCM2837 SoC, which runs at 1,4 GHz clock frequency. Processor itself is easily powerful enough for the purpose of this device. Raspberry Pi 3 has multiple interfacing options, but for

this project 40-pin IO header was used for interfacing the DE0-Nano FPGA board, and Ethernet interface for network connectivity.

4.1.3 DE0-Nano

The DE0-Nano is a FPGA development board slightly smaller than the Raspberry Pi. It is based on Cyclone IV EP4CE22 FPGA which contains 22000 logic elements. DE0-Nano has two 40-pin expansion headers with 72 of EP4CE22s I/O pins available. Apart from generic I/O, DE0-Nano has 8 LEDs, 4-position DIP-switch, 2 push buttons, 3-axis accelerometer, A/D converter, 32MB of SDRAM memory, 2Kb I^2C EEPROM, USB port for powering the board & programming the FPGA, and EPCS64 serial configuration device. External oscillator drives the FPGA clock inputs with 50 MHz clock signal. For this project I/O headers were used to connect the Raspberry Pis, EPCS64 to house created FPGA bitstream, USB for programming and debugging the design. FPGA is clocked by a 50 MHz oscillator.

4.2 Device Operation

Reference device consists of two Raspberry Pi's and one DE0-Nano. Both Raspberry Pi's connect to network via Ethernet, one for plaintext domain, other to ciphertext domain. Ethernet connections are only means of using the device. DE0-Nano FPGA board is placed in between the two Raspberry Pi's, and is connected to them by 40-pin ribbon cables. Only small number of pins are used for data transmission between the boards, but ribbon cable was found to be much more reliable for high clock speed transmission than single poor quality unshielded wires, so for simplicity the whole 40-pin cables were used. Only wires connected to 5V power pins on the Raspberry Pi side were cut. Connections between Plaintext RPi and DE0-Nano are two separate Serial Peripheral Interface (SPI) buses, I^2C bus and Interrupt Request (IRQ), buffer status & reset signals. Buffer status, IRQ and reset signals are implemented on the Raspberry PI side by using General Purpose Input/Output (GPIO) available on the Raspberry Pi. Connections between Ciphertext RPi and DE0-Nano are the two SPI buses and IRQ & buffer status signals. SPI buses are used for transferring user payload data to and from the FPGA, while I^2C and reset are used when configuring the FPGA on the DE0-Nano. Overall view of the device

is shown in Figure 11.

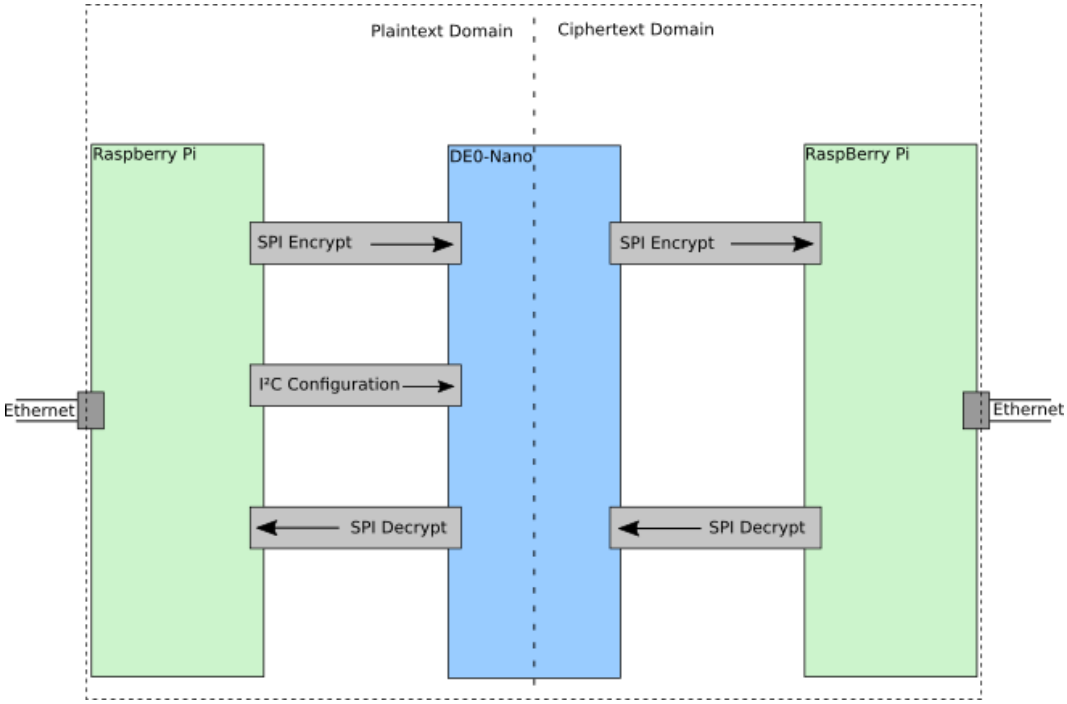

Figure 11: Block diagram of the device operation

For the sake of the device implemented for this thesis, actual separate Raspberry Pi and DE0-Nano devices are not the focus. Rather, from the application operational point of view the device can be seen as an FPGA between two CPUs running Linux OS, as they could be located on a single circuit board as well. Therefore for the simplicity, in this document Plaintext CPU, FPGA and Ciphertext CPU may be used in place of Plaintext RPi, DE0-Nano, and Ciphertext RPi respectively. This results in slightly easier read read in many situations. Detailed connections between CPUs and the FPGA are shown in Figure 12

Encrypt Path

First step on the encryption path is a simple Linux user space software module running on plaintext CPU. It acts as a Transmission Control Protocol (TCP) server, and simply waits for incoming TCP packets over the Ethernet interface. Once packets arrive it checks whether input buffers on the FPGA have enough room for new data, by inspecting buffer status GPIO signal. When enough data is available software initiates SPI transfer of 64 bytes containing four AES States worth of data to the FPGA input buffer. AES Encryption algorithm inside FPGA encrypts those four states and places corresponding ciphertext

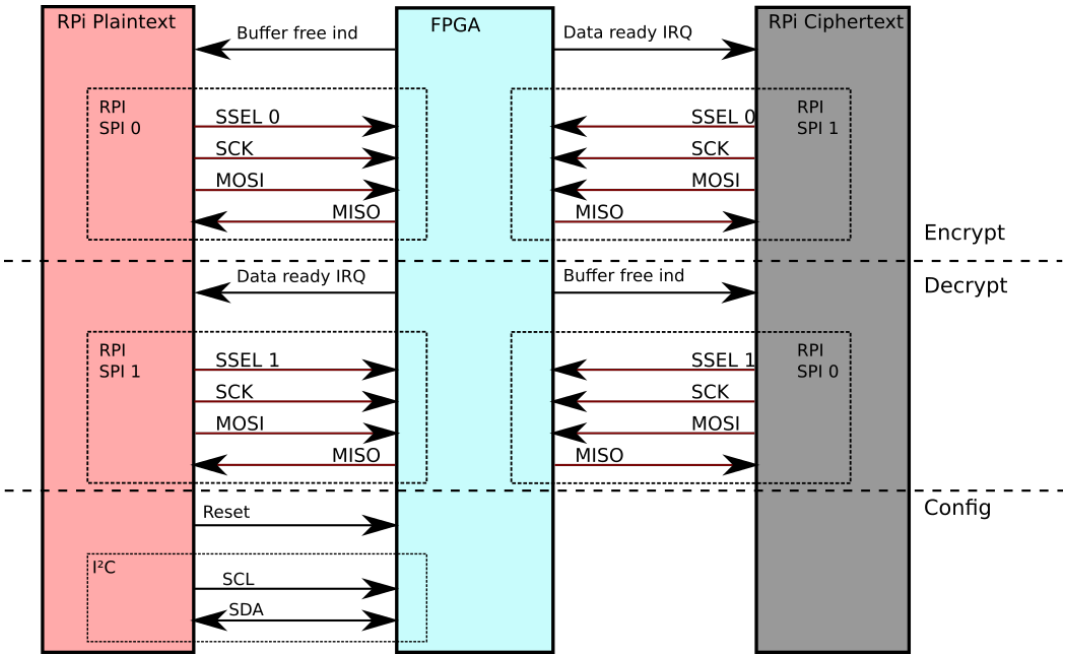

Figure 12: Connections between Raspberry Pi CPUs and the FPGA

result to the output buffers, and notifies ciphertext CPU by activating the IRQ. On the ciphertext CPU, there is similar small Linux application doing the reverse than the one on plaintext side. Once activated by the IRQ, it initiates SPI transfer reading that 64 bytes of ciphertext data, and sends it via TCP connection over the Ethernet interface.

Decrypt Path

Decrypt path is in fact exactly the same as encrypt path, just in reverse direction. Software modules on the CPUs are the same, as they do not interfere with the ciphering or deciphering process in any way. Different physical SPI bus is used for encrypt and decrypt data paths on the Raspberry Pi. Software modules running on the Pi see them just as SPI bus for sending, and SPI bus for receiving. They are mapped to be the same for both boards. IRQ and buffer status GPIO signals are physically the same from RPi point of view. This makes it possible to use same software modules for both Raspberry Pi's.

Management

Device is managed by simply accessing both Raspberry Pi's either locally (by connecting keyboard, mouse & monitor directly to Pi), via Universal Asynchronous Receiver/Transmitter (UART), or by connecting remotely e.g. via Secure Shell (SSH). All software components are either started manually via Command-line Interface (CLI) or

scripted to be started up during Linux boot. No time was spent on lean user interface, as it is not viable for the use case of the device.

Security Aspects of the Design

As a reference device not to be used in any real world application, actual security concerns are non existent for the design. Still, as stated in section 2.1, there can be no access from ciphertext side to any of the FPGA internals, such as encryption key used for AES. In this design that is achieved by having physically separated interfaces for trusted (plaintext), and untrusted (ciphertext) side. Encryption key is delivered to FPGA from trusted side Raspberry Pi, with dedicated signals (I^2C in this case). Only physical connection between ciphertext Raspberry Pi and the FPGA, are the signals used for transferring payload data to and from the FPGA. These signals cannot be used to hack, tamper or in any way interfere with any of the cryptographic logic or relevant data, even if ciphertext Raspberry Pi is compromised and being used maliciously. In worst case, incorrect data can be sent to the FPGA for deciphering which would just end up being gibberish once past the decrypt data path. If this device would be used in real environment, ciphertext side Raspberry Pi should be hardened accordingly, and all maintenance should be done by using UART or locally with keyboard and monitor setup, but not via same network interface which is exposed untrusted domain. If software update or reconfiguration were needed, another network interface could be introduced (i.e. USB Ethernet dongle) for the SW update network access.

4.3 FPGA design

FPGA design is the core of this thesis project, and in this chapter it is described in detail. After overview of the design, each block or section is explained. VHDL implementation of the AES algorithm is also viewed in detail.

4.3.1 Overview

Logically FPGA design can be divided into four parts; Configuration, Key Generator, and Encrypt & Decrypt data paths. as is shown in Figure 13. Configuration block is given the

initial Key, which is delivered for Key Generator to calculate the round keys. Along with the Key, Initialization vector for CBC Block chaining is given. It also distributes external reset signal across the logic. After Key Generator has calculated the round keys, Encrypt and decrypt data paths can begin to process input data. Encrypt Data Path encrypts plaintext input to ciphertext output, and Decrypt Data Path decrytps ciphertext input into plaintext output. Both datapaths operate independently of each other. 50MHz core clock signal is used to clock the design.

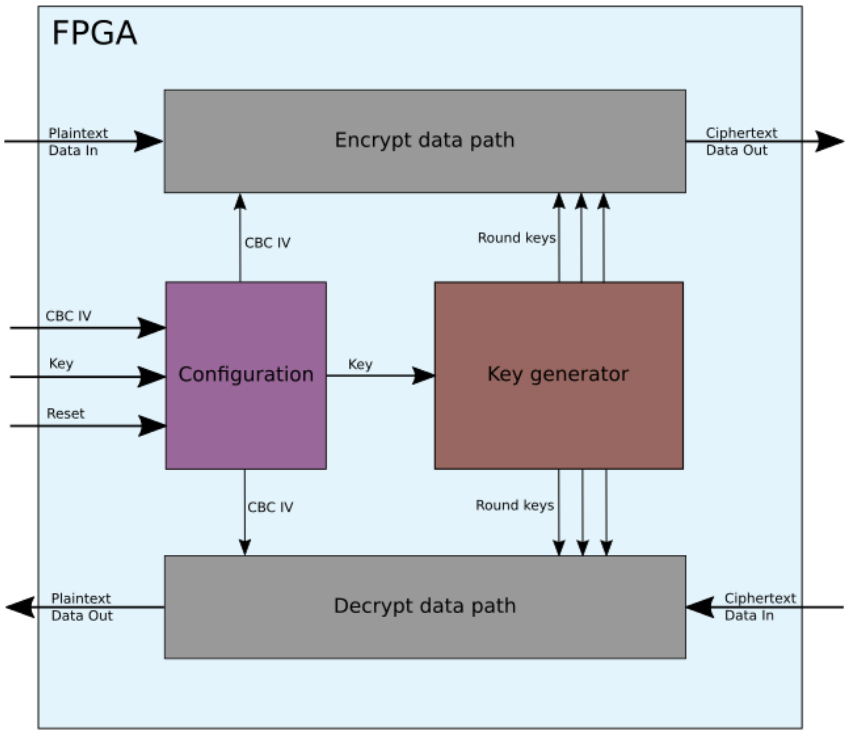

Figure 13: Block diagram of the FPGA design

4.3.2 Configuration

Configuration consists of I^2C slave block, and the actual configuration control block on the FPGA. After Plaintext RPi SW activates I^2C BUS for writing, configuration control block drives the FPGA into configuration mode. In this mode, data path pipelines are stalled and do not process any data even if there would be some stored in FIFOs. Configuration SW on the plaintext RPi sends CBC initialization vector and AES key over the I^2C BUS, being 48 bytes in total. After all data is received, Configuration control block stores both

IV and key to its output registers and asserts internal reset signal for two clock cycles.

Reset clears internal data buffers such as FIFOs and serial/parallel buffers, and resets I^2C and SPI states. When Reset is lifted, Key Generator is triggered to begin calculating round keys.

In addition to receiving configuration data, Configuration control block also receives external reset asserted by Plaintext Raspberry Pi SW, via Pi's GPIO pin.

4.3.3 Data Path

Block diagram in Figure 14 describes the data path in logical level. First block in the

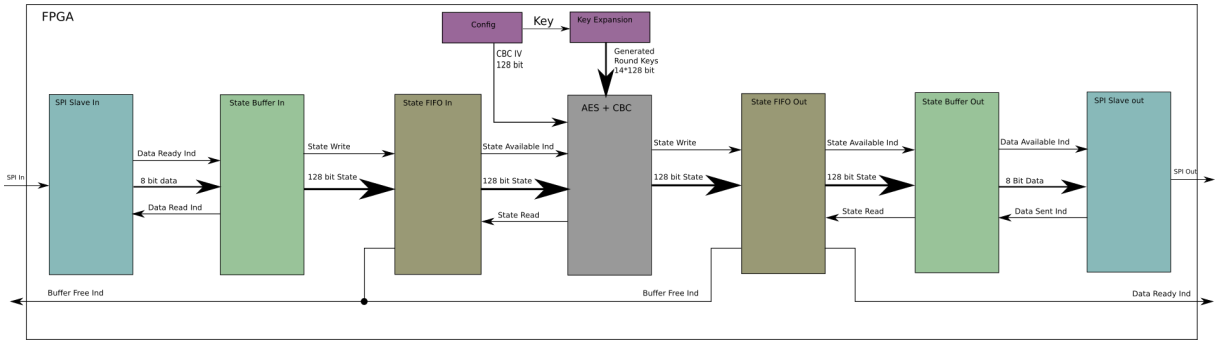

Figure 14: Block diagram of the FPGA data path

pipeline is the SPI slave module, which receives input serial data bits which are sent from the Raspberry Pi. If enough room is available, every received byte is send to *State Buffer In*, in which one AES state (16 bytes) is stored. *State Buffer In* notifies *SPI Slave in* when data has been read by asserting *Data Read Ind* strobe. *State FIFO In* in turn stores several states worth of data for *AES+CBC* block to process. At the time of writing this thesis report the amount of states stored in FIFO is 8, which is two times the CBC block of 4 states. *State FIFO In* registers input state when *State Buffer In* asserts *State Write* strobe signal.

If *State Available Ind* signal notifies availability of new state to be processed, *AES+CBC* block begins to cipher or decipher the state by first issuing *State Read* strobe, which notifies *State FIFO In* that state it had in output port is read. This is done reading the initialization vector *CBC IV* from the *config* output port if this is the first state of CBC block. If not, *AES+CBC* uses internal intermediate value instead of CV. This is described

in subsection 4.3.7. *Key Expansion* output port has precalculated round keys available for AES block to use for calculating round operations.

Once *AES+CBC* block has processed one state of data, output state is stored in *State FIFO Out* by asserting *State Write* strobe signal. If *State FIFO Out* holds at least one state, it has *State Available Ind* signal active. *State Buffer Out* is a parallel to serial buffer functioning as a reverse for *State Buffer In*. If it has empty internal parallel buffer and *State Available Ind* from *State FIFO Out*, new state is read in by asserting *State Read* strobe, after which it activates *Data Available Ind* signal and stores one byte to the 8 bit output port. When receiving end Raspberry Pi issues a SPI read, *SPI Slave Out* reads the data from *State Buffer Out* output port and notifies it by asserting *Data Sent Ind* strobe signal.

All SPI blocks in this FPGA design are SPI slaves. Therefore Raspberry Pi's are always initiating SPI transfers as they are SPI masters. Data path keeps both sending and receiving end Pi's notified whether there is room to send data to FPGA or read from it. This is done by FIFOs. *State FIFO In* and *State FIFO Out* keep *Buffer Free Ind* active as long it has at least one CBC block worth of free buffer space available. On the output side, *State FIFO Out* activates *Data Ready Ind* if it has at least one CBC block worth of data available for reading. SPI transfer is always done by writing or reading one CBC block of data at a time.

4.3.4 AES Cipher Algorithm

In this design, AES Encrypt block is fully responsible for ciphering the 16-byte State input from CBC block. AES cipher is thoroughly explained in subsection 3.4.2. Simplified block diagram of the AES cipher algorithm implementation on the FPGA is presented in Figure 15. When inspecting the actual RTL logic level output of the compilation result, one can see that it is vastly more complex than this block diagram illustration pictures. This level is enough to grasp the logic behind the implementation and to verify its performance.

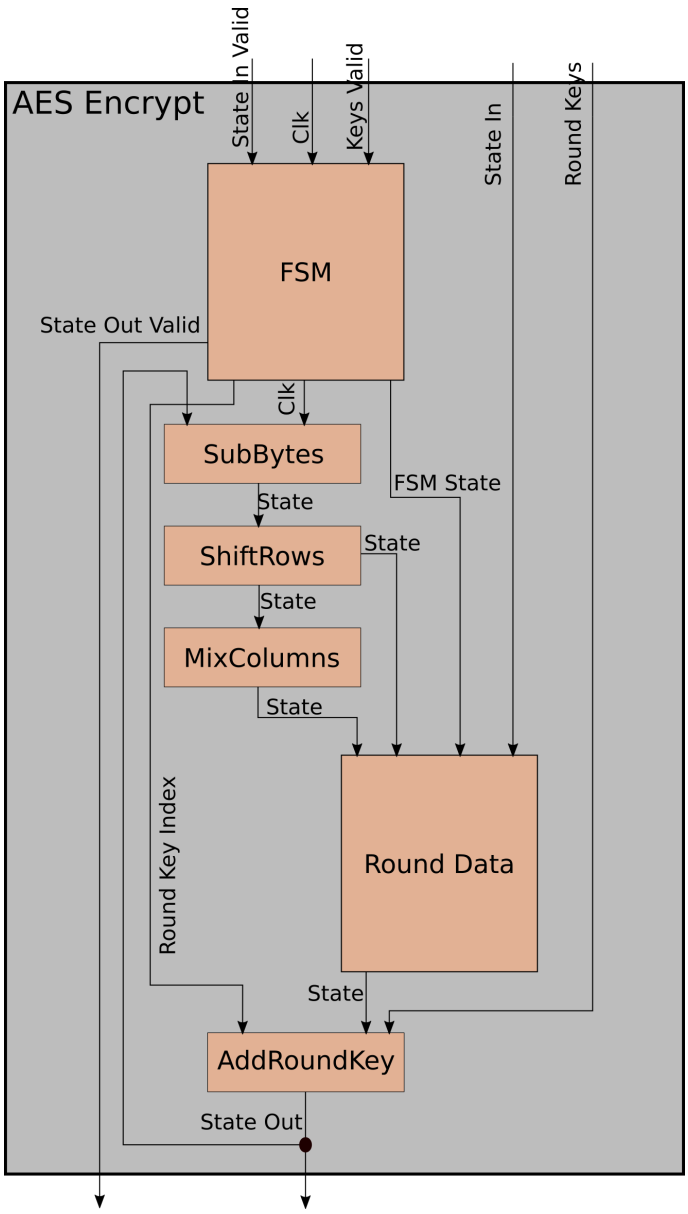

Figure 15: Block diagram of AES cipher FPGA implementation

AES Encrypt has, in this simplified view, five different inputs.

- State In Valid signal
- Clock signal
- Keys Valid signal
- 16-byte input State to be ciphered
- Round Keys generated beforehand by KeyExpansion as 16-bytes key array.

It outputs the ciphered 16-byte output State, as well output validity strobe. AES Encrypt

operation is controlled by Finite State Machine (FSM) block, which keeps track of cipher rounds, and transforms them to internal states, as shown in Table 3. FSM is clocked by the same 50 MHz clock that the rest of the design. FSM begins clocking rounds when 'State Valid In' signal assertion notifies of new data in 'State In' port is valid. Requirement for clocking rounds is that 'Keys Valid' signal is asserted, which it is always after KeyExpansion has created round keys after reset. FSM block routes the input clock through to the *SubBytes* function block, and provides correct 'Round Key Index' for *AddRoundKey* so it is using correct key form 'Round Keys'. Once rounds are complete, it asserts 'State Valid Out' signal to notify completion. *Subbytes*, *MixColumns*, *AddRoundKey* and *ShiftRows* logic blocks implement the functions described in subsection 3.4.2. AES Encrypt internal state is not to be confused with 16-byte State, a synonym for 16-byte data block algorithm operates on.

Table 3: AES Encrypt FSM rounds to internal states

Internal state:	AES round #:
0	-
1	1
2	2-14
3	-

Internal state is labeled as 'FSM State' in Figure 15. Round Data block is in effect a simple multiplexer, which selects which of the 16-byte input States it feeds to *AddRoundKey* function block. Selection is done based on FSM internal state as shown in Table 4.

Table 4: Round Data multiplexer output selection

Internal state:	Selected Round Data output:
0	State In
1	State In
2	output State from *MixColumns*
3	output State from *ShiftRows*

Key thing to note from Figure 15 is, that only FSM and *MixColumns* blocks are clocked. That is because they are the only ones containing register logic. The rest of the blocks are purely combinational designs. This leads to very efficient design, as during rounds 2-

14, all four transformation functions are passed during one clock cycle. *SubBytes* function block comes as a natural place to have single round to wait to be clocked, as it use precalculated substitution box (S-Box) contents to substitute input bytes. These precalculated contents reside in SRAM memory of the Cyclone IV FPGA, which can be accessed via registered read.

Performance wise, 16 bytes pass through AES Encrypt block in 15 clock cycles. When one clock is added from CBC operation, throughput is one 16-byte AES State in ciphered in 16 clock cycles. 50 MHz core clock then yields 50 MB/s throughput for this AES design.

4.3.5 AES Decipher Algorithm

AES Decrypt block is equivalent to AES Encrypt block, but for deciphering ciphertext input, and it is illustrated in Figure 16. AES Decrypt implements AES decipher algorithm described in subsection 3.4.3.

FSM State machine is almost identical to the one used in AES Encrypt, except round key indexes output to *AddRoundKey* as 'Round Key Index' are reversed. Other clear difference is, that instead of using only one multiplexer for selecting intermediate 16-byte State to *AddRoundKey*, other one is needed to select State outputs for *InvShiftRows* from between *InvMixColumns* and *AddRoundKey*. Truth tables for selecting RD Add Key input is shown in Table 5, and for RD InvShift in Table 6.

Table 5: RD Add Key multiplexer selection

Internal state:	Selected Round Data output:
0	State In
1	State In
2	output State from *DecSubBytes*
3	output State from *DecSubBytes*

Same performance logic applies as for AES Encrypt. InvSubBytes uses precalculated

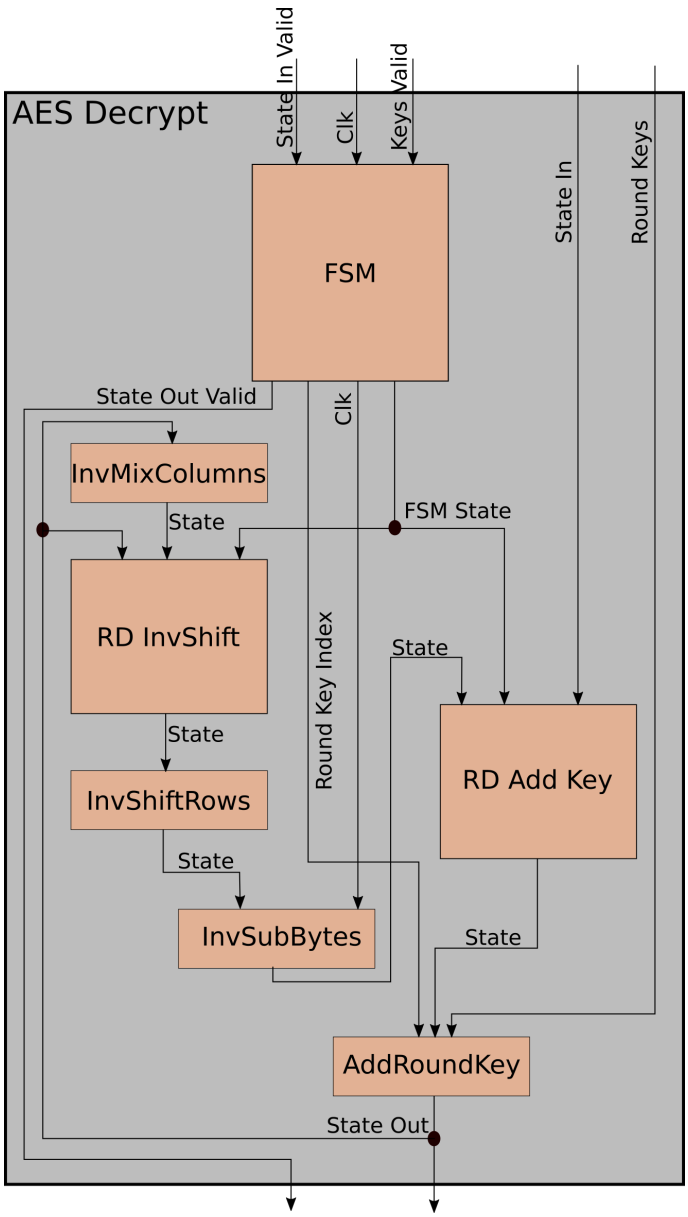

Figure 16: Block diagram of AES decipher FPGA implementation

S-Box tables stored in SRAM, requiring registered read access. This clocks the design same way as is done on the encrypt side.

4.3.6 Key Generator

Key Generator implements KeyExpansion routine described in subsection 3.4.4.

KeyExpansion in fact operates on bytes one word at a time. Each processed word needs output of previous processed word as an input. When implemented in software, KeyEx-

Table 6: RD InvShift Key multiplexer selection

Internal state:	Selected Round Data output:
0	AddRoundKey
1	AddRoundKey
2	output State from *InvMixCol*
3	output State from *InvMixCol*

pansion could be seen as running 52 rounds, with branches every 4th round (as is the case in the pseudocode in the specification). In selected way to implement this on the FPGA, one round can be seen as being 8 word operations long pipeline. In the pipeline, Word 1 and word 5 get special treatments. For word 1, *RotWord*, *SubWord*, and Rcon XOR operations are performed, and *SubWord* for word 5. For each word XOR operation is applied as described in subsection 3.4.4. Each round generates 256 bytes of generated key data to the output. The very first round would be to copy input 256-bit original key to the generated keys array. Simplification of the KeyExpansion FPGA implementation is illustrated in Figure 17.

As is the case with Cipher and Decipher designs, state machine controls the KeyExpansion block execution. Once 'Key In Valid' signal from configuration block is asserted, meaning that configuration block has received new Keys, FSM state machine block starts to clock the design. FSM states translate to 8 'rounds', of which first one is practically used to copy the original key to the beginning of 'Generated Keys' array. This is handled by Write Port block, which is responsible for placing data output to 'Generated Keys'. 'Generated Keys' array is naturally used for the input for Read Port Block as well which, depending on the round or 'Internal State', provides previous round data to all Word processing blocks as shown in Figure 17. Write Port block stores Data output from the Word processing blocks to the 'Generated Keys' output array depending on the 'Internal State'. Word blocks are all clocked and they perform tasks for their corresponding word of the pipeline.

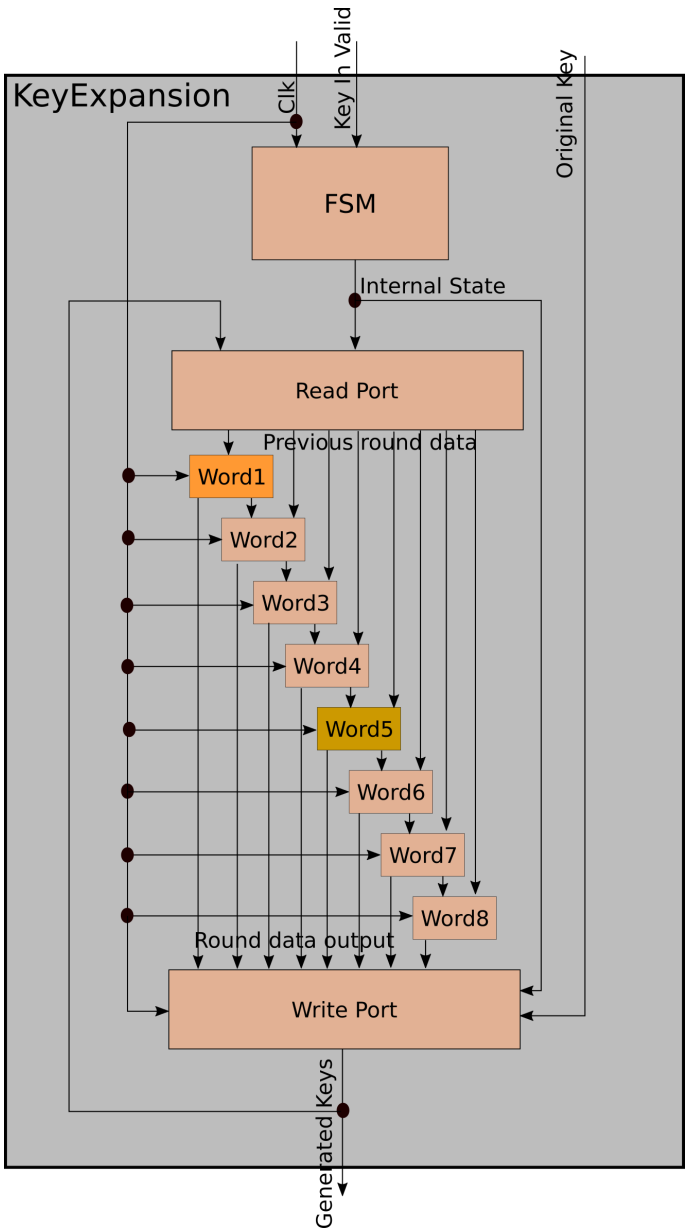

Figure 17: Block diagram of KeyExpansion FPGA implementation

4.3.7 Mode of Operation: CBC

CBC for encrypt was implemented in a straightforward way. In encrypt direction, every fourth input plaintext state, beginning from the first, is XORred with IV available, and read from $config$ module. All the three plaintext states in between, are XORred with the encrypted ciphertext output of the previous state. Operation is managed by simple module between State FIFO In and AES encrypt module as illustrated in Figure 18.

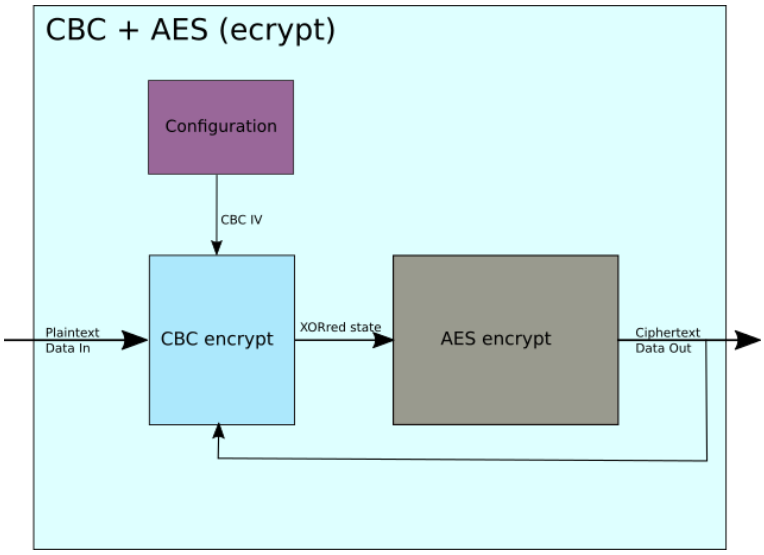

Figure 18: CBC Encrypt block diagram

CBC for Decrypt was implemented in similar manner, except in reverse as described in subsection 3.4.5, and illustrated in Figure 19.

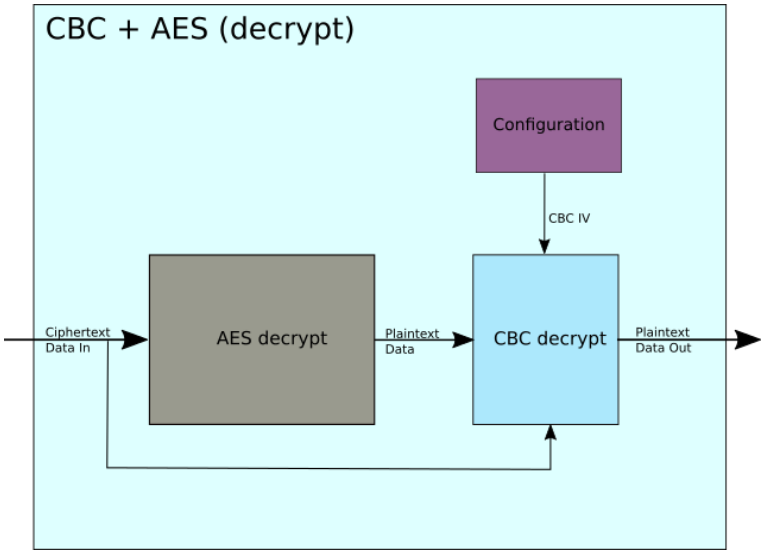

Figure 19: CBC Decrypt block diagram

4.4 Linux Software Architecture

Four simple CLI applications are used for the device operation and are run as command line commands:

- TCP to FPGA data transfer
- FPGA to TCP data transfer
- reset
- configuration.

First two are used by encrypt and decrypt data paths as described in section 4.2. Reset and configuration commands are used only on the plaintext Raspberry Pi. Reset command issues reset for the FPGA circuitry by asserting corresponding GPIO signal. Configuration command is used to deliver new AES key and CBC IV for the FPGA via issuing an I^2C transfer.

4.5 Development Environment

Development environment for the project was a basic Linux PC workstation set up. No special equipment were needed since no own electronics or hardware were produced during implementation. USB and Ethernet were enough to access all the circuit boards which would en up being the target device, and they were powered by wall adapter power supplies or USB ports. Requirement was to use open source software as much as possible, or at least tools that were licensed so that no extra cost would be introduced for the project.

4.5.1 Tools

Development tools used to implement all the SW can be divided following areas:

- VHDL implementation & simulation
- FPGA synthesis & verification
- Linux SW development.

All VHDL code was first compiled in a Linux environment by using open source GHDL tool. GHDL was also used to simulate AES algorithm implementation, including key expansion, and CBC block cipher mode implementation during the development phase. Once simulation proved accurate results, VHDL code was synthesized and programmed onto target FPGA with Alteras Quartus Prime design software, which is a necessity when working with Alteras FPGAs. Quartus Prime combines all the tools needed for the FPGA development, even the ModelSim simulator, so the use of GHDL was an experiment whether it was sufficient enough for the task at hand. It was, so Quartus Prime was used for producing the actual programming file for the target FPGA, and debugging with Signal Tap logic analyzer. Signal Tap works by adding additional logic into the design when it is being built with Quartus. This logic will store states of predefined signals triggered by certain events determined by the user, and clocked by user selected clock signal. After programming the FPGA, user can command Signal Tap to record signal states of those signals that were needed for the debugging. These states are stored into internal SRAM memory of the FPGA. Recorded data is then retrieved by Quartus for the user to verify. Availability of Signal Tap for Quartus was essential for the development of the FPGA design. All the serial/parallel data transmission & configuration related functionality was only tested in target by using Signal Tap, since it was found to be too much effort for simulating it all beforehand. Throughout the entire VHDL coding & debugging process GHDL was used for 1st phase compilation test as it was very rapid to verify VHDL code correctness before attempting synthesis with Quartus Prime.

All Linux SW was written in C programming language. As the software modules were not too complex, no test environment or simulations of any kind were needed. Raspberry Pi based modules were coded in Raspberry Pi Linux environment and tested run time. Linux PC test software modules were written & tested run time as well. Software tools used for developing Linux application modules were made by or at least originated from GNU project.

4.5.2 Development & Testing

Development work was divided between following phases, roughly in order:

- implementation & simulation of AES encrypt VHDL code

- implementing simple data pass through for FPGA between DE0-Nano and Pi's
- testing & Debugging AES encrypt implementation in FPGA
- implementation & simulation of AES Key Expansion VHDL code
- testing & Debugging AES Key Expansion in FPGA
- adding CBC Mode of Operation
- implementation & simulation of AES decrypt VHDL code
- testing & Debugging AES decrypt implementation in FPGA
- improvement of data transfer rate between DE0-Nano and Pi's
- testing & Debugging AES both directions in FPGA
- Linux SW finalization.

Most important aspect of this thesis work was to implement AES in VHDL and successfully deploy it to the selected FPGA platform, so from there the work begun. Implementing a cryptographic algorithm like AES is challenging task as debugging is quite difficult, since the process is designed to obfuscate the input data. It either works and produces the expected result or it does not. In every step of the way, developer is dealing with nothing but hex dumps of 16 bytes in size. To aid the developers, NIST included test vectors in the AES specification, for which correct immediate values have been precalculated through the encryption, decryption and key expansion process. That is, e,g. for encrypt, output state of each transformation during each round is known without having to produce the end result. These test vectors were used in writing automated test scripts using GHDL, and this made possible to rapidly develop and verify VHDL implementation correctness.

After AES encryption produced correct results, Actual data pipeline was tested by implementing SPI data transfer logic into the FPGA towards both Pi's and data buffering in between them. First phase test software was needed for the Pi's, which would utilize the SPI buses of the BCM2837 SoC to transfer data between DE0-Nano and the Pi's. After this phase it was clear that all the tools were in place and that reset of the project would likely succeed as planned.

With AES encrypt VHDL code passing simulation and design which made data throughput possible in place in FPGA, logical next step was to verify that AES encrypt VHDL implementation would work correctly run on the FPGA. It would need several modifications and corrections before operating correctly in real device rather than in mere simulation, but all

the key tools were in place for the following steps to commence. Details of the difficulties and successes are covered more in chapter 5.

FIPS has published a separate document containing multiple test vectors for simplifying AES development called AESAVS [41]. It includes test vectors for several Modes of Operation including CBC. These test vectors were used for verifying the algorithm implementation for both simulation and on hardware. Simple test programs were written for Raspberry Pi's which would send test vector data and on the other end receive it and verify it. This was the cornerstone of lot of the testing done.

After enough stability were gained for the design, test programs that sent files through the FPGA were also written. These SW modules would pad the data with some known data pattern which would be removed in the receiving end. specific tool named openssl was used to decrypt or encrypt files after or before to verify that the design would work against the standard. It would not have been enough to test the device by sending a file through encrypt data path and then decrypt data path, since there could be fundamental flaw in the design that could affect both data paths nullifying the error in the end. However, in the final phases of the testing this method was exactly what was used. Huge amounts (hundreds of megabytes) of data were sent from either ciphertext or plaintext domain and looped back from the adjacent domain to stress the device ensure stability. This was done only after correctness of the algorithm was verified.

5 Results and Analysis

In this chapter finished product is analyzed in terms of performance, success compared to original requirements and current status of the implementation is clarified. Comparison between real world products and the reference device implemented here are also looked into. Few words on the development process is provided.

5.1 Implementation Status

At the time of writing this thesis, device implementation is almost complete. One major issue remains to be solved, and one feature is missing from the implementation. Issue to be fixed is, the FPGA implementation is not properly constrained. In practice, this leads to seldom random incorrectly decrypted states in the decipher pipeline. This is obviously such a fatal problem that it needs to addressed before handing the device over to case company. Luckily the root cause is known, and it is possible to be corrected. Far too long combinational logic chains exist between registered flip-flops, especially in the decipher pipeline. Mostly because more complex *InvMixColumns()* transformation as described in subsection 4.3.5. This can be fixed by adding core clocked flip-flops in some of the logic blocks that are now purely combinational.

The one completely missing feature is configuration interface from plaintext Raspberry Pi to the FPGA, and actual configuration update functionality in the FPGA. In essence, I^2C interface is not yet implemented. This feature was deliberately left to be done last, as it is least significant for the operation of the device. Currently CBC initialization vector and AES key are hard coded into VHDL implementation. This feature will also be added before handing the project over.

FPGA constraining and I^2C configuration interface will be done after finishing this document due to schedule problems. This work has taken much more time to finish than originally intended, and separate deadlines for graduation and device completion force the issue.

5.2 Performance Analysis

This device was never intended to be a highly performing one. Still, this section explains design and implementation choices regarding the performance.

5.2.1 AES+CBC FPGA Implementation

As described in chapter 4, AES+CBC block is able to process one 16-byte data block, or State, in 16 clock cycles. This is true for both encrypt and decrypt directions. This provides 50 MB/s throughput with 50 MHz core clock, which is very satisfactory result. Design is not without problems though. In its current state, clock frequency cannot be increased, as the long combinational signal paths would not meet timing requirements and design will fail to operate correctly. Selected operation mode, CBC, is secure and is easy to implement, but does not support States to be encrypted in parallel. If more throughput were needed from single data path, changes to the design are mandatory.

5.2.2 User Data Transmission

Data transmission in this context refers to the phases in cipher and decipher pipelines, where Linux software module in sending Raspberry Pi unit sends data to the FPGA, up to the point that data is accessible by CBC + AES crypto core inside FPGA. And again after data leaves CBC + AES cryptocore and is received by the Linux software module in the receiving Raspberry Pi unit. This consists of SPI data transfer initiated by BCM2837 SPI subsystem, governed by Linux kernel, and parallel buffers & state FIFOs in FPGA design.

Data transmission is a bottle neck for the data throughput of this device, and was known to be so from the point where used hardware components were selected. However, the magnitude of the problem was not expected to be what it eventually became to be. As stated, CBC+AES crypto core is able to process data at a rate of 50 MB/s per pipeline. Still, due to data transmission issues, actual measured data throughput is around 160 KB/s in both directions. That is, CBC+AES core stands idle for 99.68% of the time. And that result was achieved with test SW modules which send single hard coded data block as

fast as possible. When adding network interface handling to the mix, it gets worse. That amount of overall inefficiency is largely due to better than expected algorithm performance in FPGA, but still major blame goes to SPI usage. Stable data rate between these boards and used parallel cables were measured to be 10 MHz maximum, that is, around 1 MB/S in theory. Linux kernel driver is used to issue a DMA transfer which sends requested data through SPI subsystem, and once on the wire, that 1 MB/s is achieved during that transfer. However, arming the transfer from user space SW module takes a variable very long time to actually start sending data. On the other hand, it also takes long time for SW module to check the state of the I/O pin FPGA uses to signal internal buffer availability, and begin issuing the transfer. Exact profiling study was not done since it was not of interest, but overall data throughput shows that Raspberry Pi SW implementation is spending more time not sending data than it is sending.

5.2.3 Linux Software Modules

All Linux software were written as user space SW modules. Also for reading the states of the GPIO pins which FPGA uses to signal receiving Raspberry Pi application of the available data. First draft implemented was using simple fast infinite loop for reading the input pins and this stuck due to not having time to implement proper level triggered IRQ mechanism in kernel space. Level triggered IRQ approach would be less resource consuming overall, as one ARM core would not be spent entirely on looping the status of GPIO pin. Current design is also plagued with uncertainty scheduling wise, since it is not kernel which raises the IRQ. However, Linux is not a hard real time OS so there is no way developer can guarantee a specific response time to an IRQ, even if properly implemented in the kernel.

5.3 Deviation to Real World Products

Cryptographic HW accelerators do come in many shapes, forms and solve wide variety of application problems. As this reference device is built mostly around the requirement of the FPGA based algorithm implementation, it does not compare well to any actual devices on the market. Closest reference would probably be a VPN gateway or so. Still, several key concepts can be identified which have to be improved in real world devices compared

to the outcome of this thesis project.

One obvious difference between this device and official product would be usability and configurability, which would need to be on a completely different level than they are in this device. Wide variety of different configuration options would have to be in place from cryptographic parameters, networking related configurations and the like. These topics however, go so far beyond the scope of this thesis that they are not discussed further. More relevant topics here are security and performance.

Any real world device should have to have much better overall performance to validate its existence. Simply put, it should be able to encrypt and decrypt at native speed of the network interface into which the data path is connected to. As discussed in section 5.2, performance can roughly be divided between the actual throughput of the crypto core itself, and circuitry feeding user data to it, that is, data transmission.

If any meaningful throughput is required, SPI or similar slow interfaces, especially via generic GPIO headers through unshielded wiring can be forgotten. Data should be fed to the FPGA either directly via Ethernet or other communication medium, or if sent from other processing units like is the case in this reference device, via some high speed bus like PCI Express or RapidIO. Naturally there needs to be accelerator subsystems in place in all other devices on the bus, and either hard or soft IPs could be utilized in the FPGA side. One important aspect would be to have FPGA have much larger data buffers before and after the crypto core blocks, than what this reference design does. This is a necessity as regardless of the selected transmission mechanism or bus, communicating and preparing the data send will be the bottle neck in the data path pipeline.

While looking into crypto core itself, which in this device was the CBC+AES implementation, there are also several deviations expected to be found in real world applications. When implemented in FPGA, rather than ASIC, concurrency is where the real performance gains are to be made. In the most simplistic view, as in the example of a VPN gateway, an FPGA could simply process multiple data paths simultaneously. Also algorithm choices themselves play their part. Implemented reference device processes 50 MB/s per data path. For an Ethernet connected device, this would be sufficient for 100 megabit Fast Ethernet but not 1G or beyond. As simple clock speed increase is not pos-

sible due to design not functioning in higher clock speeds, data path specific concurrent processing needs to be applied. Different algorithms can be used, and in case of AES different operation mode could be utilized. For example, selected CBC operation mode requires each state to be processed sequentially, therefore it is not possible to cipher or decipher states in parallel. If e.g. EBC or CRT operation modes were used, concurrent processing would be possible. More logic elements are required, thus it is highly probable that real FPGA based applications will be deployed on more massive FPGAs than what is used in this design. On top of all that, author is confident that data path pipeline and crypto core implementation can be better optimized, allowing for higher clock speeds to be utilized by real world applications.

Security level requirements are dependent on the given application, but one that was set for this reference device is viable to many different designs. That is the impenetrability of the plaintext domain from the ciphertext domain. The device produced for this thesis does indeed guarantee that, but with a rather cumbersome overall design. Maintenance of ciphertext side Raspberry Pi and software running on it are inconvenient and problematic security wise. Actual product would have to have a single point of configuration and maintenance interface, which would be secure and sufficiently isolated from ciphertext domain. Almost same level of isolation could be achieved by for example connecting ciphertext and plaintext domain Ethernet connections directly to an FPGA. All Ethernet frames would be processed by the FPGA design logic, and no host operating system in private plaintext domain would be vulnerable. As a downside, This approach requires much more complex design, as TCP/IP stacks would be needed to be implemented in FPGA fabric.

Some combination of openCL, traditional HDL design, and third party or vendor provided soft IP cores, could prove to produce efficient and secure networked crypto device with meaningful effort estimations. OpenCL could be used for rapid generation of cryptographic algorithm kernels, whereas soft cores could fill the gap between fast HW interconnects on the circuit board, such as network protocol handling and HW interfacing. Additional application specific HDL may be necessary to fill in the gaps.

5.4 Improvement Ideas

Several modifications could be applied to current design to increase its performance or making usability and modifiability more diverse.

Plaintext Raspberry PI and DE0-Nano could be replaced by e.g. SoC development board with an Ethernet interface. SoC would need to have a hard processor capable of running Linux OS. SoC internal bridge connections between Linux processor and FPGA fabric would replace external SPI and I^2C interfaces used now between plaintext Raspberry PI and DE0-Nano. Benefit would be simpler HW usage, as only two circuit boards would be needed. Also this would allow for wider experimentation surface for the user of the reference device, as SoC FPGAs are gaining more wider adoption in the market. For example, manufacturer of DE0-Nano offers development board containing Cyclone V SoC which would cost even less than DE0-Nano and one Raspberry Pi combined.

5.5 Few Words on the Development Process

Key elements of the device to be implemented were agreed upon very early in the project. Them being the choice of HW platforms, Raspberry Pi's & DE0-Nano, and AES cryptosystem with CBC mode of operation. Already at that time project seemed to be rather involving for the author, as the FPGA/VHDL development with this complexity, and AES were new topics to learn. Customer of the project was also not the primary employer, so this work was done completely during spare time. Still, it was not seen as overwhelming, as a lot of reference material, such as AES HDL implementations, were readily available on the internet. The amount of technical difficulties and the extent of project delay was therefore somewhat surprising. No less than four times more time were sunk in this project that was initially estimated.

Algorithm implementation, and partly adaptation from internet sources, was rather nicely progressing experience. That is, up to the point of simulation proven VHDL design. For instance, it took quite a lot more time and rewriting to make it work on real hardware. Most problematic issue throughout the project was without question the data transfer between Raspberry Pi's and DE0-Nano circuit boards. At first, simple cheap wires were used

for SPI transfer which were not of sufficient quality for the purpose. A lot of time was wasted investigating design problems in either RPi software or FPGA logic, even the real culprit was cabling which caused lot of glitches in transmission and control signals. No electrical measurement devices were available during development, but debugging was based purely on FPGA internal logic analyzer and traditional software debugging methods. This made it difficult to detect electrical issues.

Development of data transmission between Linux SW and AES+CBC blocks did consume much more time as anticipated, even after cabling issues were solved. Even if it was known from the beginning, that data transmission speed would not probably match that of the algorithm performance in the FPGA, it did leave much to hope for.

On the positive side, wrestling with the difficulties faced during the development process, taught the author fair amount of how real world fast interconnects should be designed. And when going beyond the issues faced, author did learn much about cryptographic acceleration device design. That itself made it a worthwhile experience.

6 Conclusions and Summary

The original goal was to put together a networked cryptographic accelerator device, which would have the AES algorithm implemented on an FPGA. The key design goal was to secure sensitive information and data on the plaintext side of the device. In the process, hardware based cryptographic acceleration topics were to be investigated from both performance, and security points of view. What are the major design characteristics that make FPGAs formidable platforms to implement cryptographic acceleration on? What are the major points to consider when security is concerned in this type of a device?

FPGAs of today are powerful platforms to apply many kinds of applications. Due to their rather low clock frequencies, at least when compared to those offered by ASICs, they excel in solutions where a concurrent processing can be utilized. Cryptographic algorithms that can be applied on multiple blocks of data of the same data stream simultaneously can see immense benefit when deployed on FPGA. Modern massive FPGAs can be used to cipher or decipher multiple data streams simultaneously even when each single data stream processing would be processed by multiple encryption blocks. Requirement is, that data is being fed to encryption blocks or cores fast enough to keep them busy. A fast data transfer to the FPGA itself requires either hard or soft IP cores in the FPGA to handle fast enough data transmission protocols or interfaces properly.

When the security is a critical concern, FPGAs are fairly easy to design in a way that at least a remote intrusion to critical data is not possible. The key design aspect is to separate plaintext and ciphertext sides such, that there is no access to a plaintext section of the device from ciphertext side. The FPGA design developed for this thesis offers only user a plane data interface, the SPI in this case, to ciphertext or public network side. It is also important that FPGA itself cannot be tampered remotely by forcing it into reset or reprogram itself or so. This is achievable with a proper care when designing such a device.

Using FPGA as a cryptographic accelerator platform really becomes viable when FPGAs with high capacity are used, to provide the required performance boost and interfacing ca-

pabilities. Especially so with networked solutions. Dividing the hardware design properly between public and protected sides tends to increase an overall device complexity, again increasing the cost. Due to these design considerations it is highly likely that FPGAs are used for cryptographic accelerators in high end solutions.

References

1 David Mertz. 2001. Introduction to cryptology Part 1: Basic cryptology concepts. Network article. IBM Developer Works. <https://www.ibm.com/developerworks/tivoli/tutorials/s-crypto/s-crypto.html>. Accessed February 21, 2018.

2 Prof.Waghmare S.P and Simran Sikhwal and Shreyas Nimje and Tanvi Pawar. History Of Cryptography, International Journal For Technological Research In Engineering, Volume 4, Issue 8. 2017 April;.

3 Faranak Nekoogar. 2001. Digital Cryptography: Rijndael Encryption and AES Applications. Network article. EE Times. <https://www.eetimes.com/document.asp?doc_id=1275908>. Accessed February 21, 2018.

4 Federal Information Processing Standards Publications. Advanced Encryption Standard (AES) (FIPS PUB 197). 2001;<https: //csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf>.

5 Alfred J. Menezes , Paul C. van Oorschot and Scott A. Vanstone. Handbook of Applied Cryptography. CRC-Press. 1996.

6 RFC 6101: The Secure Sockets Layer (SSL) Protocol Version 3.0. 2011. Network article. Internet Engineering Task Forse. <https://tools.ietf.org/html/rfc6101>. Accessed April 01, 2018.

7 J. Serrano. Introduction to FPGA design;<https://cds.cern.ch/record/1100537/files/p231.pdf>.

8 Introduction to FPGA Technology: Top 5 Benefits. 2012. Network article. National Instruments. <http://www.ni.com/white-paper/6984/en/>. Accessed March 19, 2018.

9 Field Programmable Gate Array Market (By Product Type: SRAM, Flash Based, and Antifuse; By Application: Industrial, Automotive, Consumer Electronics, Military & Aerospace, Telecom, Data Processing; By Geography: North America, Europe, Asia-Pacific, RoW) Global Scenario, Market Size, Outlook, Trend and Forecast, 2015-2024. 2015. Network article. Variant Market Research. <https://www.variantmarketresearch.com/report-categories/semiconductor-electronics/field-programmable-gate-array-market>. Accessed March 19, 2018.

10 Andrew Moore, Ron Wilson. FPGAs For Dummies, 2nd Intel Special Edition. John Wiley & Sons. 2017.

11 Optimally Fortifying Logic Reliability through Criticality Ranking. 2015. Network article. MDPI. <http://www.mdpi.com/2079-9292/4/1/150/htm>. Accessed

March 31, 2018.

12 UltraScale Architecture and Product Data Sheet: Overview. 2018. Network article. Xilinx. <https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf>. Accessed March 31, 2018.

13 intel® USER-CUSTOMIZABLE soc FPGAs. 2017. Network article. Intel Corporation. <https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/br/br-soc-fpga.pdf>. Accessed March 31, 2018.

14 Guerric Meurice de Dormale and Jean-Jacques Quisquater. High-speed hardware implementations of Elliptic Curve Cryptography: A survey. 2007;<url="https: //www.sciencedirect.com/science/article/pii/S1383762106001044">.

15 Intel® 64 and IA-32 Architectures Software Developer's Manual. 2016. Network article. Intel Corporation. <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf>. Accessed March 31, 2018.

16 AMD64 Architecture Programmer's Manual Volume 4: 128-Bit and 256-Bit Media Instructions. 2017. Network article. Advanced Micro Devices. <https://support.amd.com/TechDocs/26568.pdf>. Accessed March 31, 2018.

17 ARM® Cortex® -A57 MPCore Processor Cryptography Extension. 2015. Network article. ARM. <http://infocenter.arm.com/help/topic/com.arm.doc.ddi0514g/DDI0514G_cortex_a57_mpcore_cryptography_trm.pdf>. Accessed March 31, 2018.

18 CUDA COMPATIBLE GPU AS AN EFFICIENT HARDWARE ACCELERATOR FOR AES CRYPTOGRAPHY. 2007. Network article. Institute of Electrical and Electronics Engineers. <http://ieeexplore.ieee.org/abstract/document/4728256/>. Accessed March 31, 2018.

19 Secure Device Manager for Intel® Stratix® 10 Devices Provides FPGA and SoC Security. 2014. Network article. Intel Corporation. <https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01252-secure-device-manager-for-fpga-soc-security.pdf>. Accessed March 31, 2018.

20 Peter Smirnoff. 2017. Understanding Hardware Security Modules (HSMs). Network article. Cryptomathic. <https://www.cryptomathic.com/news-events/blog/understanding-hardware-security-modules-hsms>. Accessed March 31, 2018.

21 Marko Wolf and Timo Gendrullis. Design, Implementation, and Evaluation of a Vehicular Hardware Security Module. 2007;<url="http://www.marko-wolf.de/files/WoGe12_Automotive_HSM.pdf">.

22 Owen Harrison and John Waldron. Practical Symmetric Key Cryptography on Modern Graphics Hardware. 2008;<url="https://www.scss.tcd.ie/publications/tech-reports/reports.08/TCD-CS-2008-20.pdf">.

23 Security Requirements Fof Crytpographic Modules. 2001. Network article. NIST. <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-2.pdf>. Accessed March 31, 2018.

24 Q Sedeeq, Mays and Salih, Muataz and F Yousif, Omar and Q Mohammed, Nada. THE VANTAGE OF UTILIZING FPGA IN THE DESIGN OF AN EMBEDDED MULTIPROCESSOR. 2016;<url="https://www.researchgate.net/publication/312017323_THE_VANTAGE_OF_UTILIZING_FPGA_IN_THE_DESIGN_OF_AN_EMBEDDED_MULTIPROCESSOR">.

25 Mukul Shirvaikar and Tariq Bushnaq. A Comparison between DSP and FPGA Platforms for Real-Time Imaging Applications. 2009;<url="https://www.researchgate.net/publication/228453392_A_comparison_between_DSP_and_FPGA_platforms_for_real-time_imaging_applications">.

26 Choosing the Right Architecture for Real-Time Signal Processing Designs. 2002. Network article. Texas Instruments. <http://www.ti.com/lit/wp/spra879/spra879.pdf>. Accessed March 31, 2018.

27 Will OpenCL open the gates for FPGAs?. 2015. Network article. Scientific Computing World. <https://www.scientific-computing.com/feature/will-opencl-open-gates-fpgas>. Accessed April 30, 2018.

28 Intel FPGA SDK for OpenCL, Programming Guide. 2017. Network article. Intel. <https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf>. Accessed April 30, 2018.

29 Joan Daemen and Vincent Rijmen. The Rijndael Block Cipher. 1999;<https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf>.

30 Kevin Lee. Advanced Encryption Standard (AES) Selection Process - How Rijndael Won. 2015;<https://www.usna.edu/Users/math/wdj/_files/documents/sm473-capstone/Rinjdael-16WeekFinalDraft-KevinLee.pdf>.

31 AES-SubBytes.svg. 2006. Network article. Wikimedia Commons. <https://commons.wikimedia.org/wiki/File:AES-SubBytes.svg>. Accessed May 14, 2018.

32 AES-ShiftRows.svg. 2006. Network article. Wikimedia Commons. <https://commons.wikimedia.org/wiki/File:AES-ShiftRows.svg>. Accessed May 14, 2018.

33 File:AES-MixColumns.svg. 2006. Network article. Wikimedia Commons. <https://commons.wikimedia.org/wiki/File:AES-MixColumns.svg>. Accessed May 14, 2018.

34 File:AES-AddRoundKey.svg. 2006. Network article. Wikimedia Commons. <https://commons.wikimedia.org/wiki/File:AES-AddRoundKey.svg>. Accessed May 14, 2018.

35 Recommendation for Block Cipher Modes of Operation - Methods and Techniques. 2001. Network article. National Institute of Standards and Technology. <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf>. Accessed March 22, 2018.

36 File:CBC encryption.svg. 2013. Network article. Wikimedia Commons. <https://commons.wikimedia.org/wiki/File:CBC_encryption.svg>. Accessed May 14, 2018.

37 File:CBC decryption.svg. 2013. Network article. Wikimedia Commons. <https://commons.wikimedia.org/wiki/File:CBC_decryption.svg>. Accessed May 14, 2018.

38 DE0-Nano User Manual. 2013. Network article. Terasic Technologies Inc. <https://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=593&FID=75023fa36c9bf8639384f942e65a46f3>. Accessed Jan 31, 2018.

39 Cyclone IV Overview. 2018. Network article. Intel Corporation. <https://www.altera.com/products/fpga/cyclone-series/cyclone-iv/overview.html>. Accessed March 31, 2018.

40 Raspberry Pi website. 2018. Network article. Raspberry Pi Foundation. <https://www.raspberrypi.org/>. Accessed March 31, 2018.

41 Federal Information Processing Standards Publications. The Advanced Encryption Standard Algorithm Validation Suite (AESAVS). 2002;<https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/aes/AESAVS.pdf>.