SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor′s Thesis in Informatics

End-to-End On-Chip Encryption to Prevent Physical Attacks

Jonas Zöschg

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

End-to-End On-Chip Encryption to Prevent Physical Attacks

Ende-zu-Ende-Verschlüsselung auf dem Chip, um physische Angriffe zu verhindern

Author:	Jonas Zöschg
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Harshavardhan Unnibhavi, M. Sc.
Submission Date:	15.03.2023

I confirm that this bachelor’s thesis is my own work and I have documented all sources and material used.

Munich, 15.03.2023 Jonas Zöschg

Acknowledgments

First, I would like to thank Harshavardhan Unnibhavi for always being there when I needed his support, reviewing my progress, and guiding me through my bachelor's thesis.

Likewise, I would like to thank Prof. Dr.-Ing. Pramod Bhatotia, for the opportunity to work on this thesis for the Chair of Distributed Systems and Operating Systems.

I am also thankful to the chair staff, especially Sophia Adelmeier, for the help with all the formalities.

Next, I would like to thank all my friends for the time away from my studies to keep me balanced and give me the strength to master all the hurdles on this journey. I especially thank my flatmates and best friends Clemens and Julian for all the moral support in difficult situations.

Most importantly, I would like to thank my family. My parents, Edith and Günther, and my brother Hannes, without whom I would not be where I am now. I want to thank them for always believing in me, being proud of me, and always being there.

Abstract

Root of Trust (RoT) chips play an important role in protecting the security of computer systems, such as data center servers, client devices, IoT devices, and more. One project in this field is OpenTitan [1], an open source project to give high-quality reference designs and integration guidelines for RoT chips.

Since RoT chips are important for security, they are often exposed to attacks. Although OpenTitan has implemented many measurements against physical attacks, vulnerabilities arise when adding multiple CPUs to increase the chip's performance. Those vulnerabilities allow the extraction of information from the communication between CPUs. This thesis, therefore, presents a hardware design that enables end-to-end on-chip encrypted communication between CPUs to prevent those physical attacks.

For this, every CPU uses an AES IP core to encrypt/decrypt messages and a newly developed IP core for the communication between CPUs. After a detailed look into the hardware design and its integration into an existing chip design from OpenTitan, this thesis evaluates the correctness and performance of the design. The thesis shows that the bottleneck for the encryption and sending process is the encryption of the message. For the receiving and decryption process, the reading and writting from/to registers is the bottleneck when using 128-bit and 192-bit keys.

The source code of the implementation of the hardware design, as well as its functionality and performance tests, are available on GitHub [2].

Kurzfassung

Root of Trust (RoT) Chips spielen beim Schutz der Sicherheit von Computersystemen wie Servern in Rechenzentren, Client-Geräten, IoT-Geräten und anderen eine wichtige Rolle. Ein Projekt in diesem Bereich ist OpenTitan [1], ein Open-Source-Projekt, das hochwertige Referenzdesigns und Integrationsrichtlinien für RoT-Chips bereitstellt.

Da RoT-Chips für die Sicherheit wichtig sind, sind sie häufig Angriffen ausgesetzt. Obwohl OpenTitan viele Maßnahmen gegen physische Angriffe implementiert hat, entstehen Schwachstellen, wenn mehrere CPUs hinzugefügt werden, um die Leistung des Chips zu erhöhen. Diese Schwachstellen ermöglichen die Extraktion von Informationen aus der Kommunikation zwischen den CPUs. In dieser Arbeit wird daher ein Hardware-Design vorgestellt, welches eine Ende-zu-Ende verschlüsselte Kommunikation auf dem Chip zwischen den CPUs ermöglicht, um solche physische Angriffe zu verhindern.

Dazu verwendet jede CPU einen AES-IP-Kern zur Ver-/Entschlüsselung von Nachrichten und einen neu entwickelten IP-Kern für die Kommunikation zwischen den CPUs. Nach einer detaillierten Betrachtung des Hardware-Designs und seiner Integration in ein bestehendes Chip-Design von OpenTitan evaluiert diese Arbeit die Korrektheit und Performanz des Designs. Die Arbeit zeigt, dass der limitierende Faktor für den Verschlüsselungs- und Sendeprozess die Verschlüsselung der Nachricht ist. Für den Empfangs- und Entschlüsselungsprozess ist das Lesen und Schreiben von/zu Registern der limitierende Faktor, wenn 128-Bit- und 192-Bit-Schlüssel verwendet werden.

Der Quellcode der Implementierung des Hardware-Designs sowie seine Funktionalität- und Leistungstests sind auf GitHub [2] verfügbar.

Contents

1 Introduction

Back in 2021, 11.28 billion [3] IoT devices were connected worldwide, and it is predicted that by 2030 this number will reach almost 30 billion devices. All those devices are interconnected and exchange private data between them. An essential part of protecting this data from unauthorized access are so-called Root of Trust (RoT) chips. They ensure that a device boots with the correct firmware, provide a cryptographically unique machine identity that can be used to verify a device's legitimacy, protect secrets like encryption keys, and provide cryptographic functionality (e.g., encryption/decryption). Besides IoT devices, such chips can be found in server motherboards, network cards, client devices, and more.

Since RoT chips are essential for computer system security, they are often exposed to attacks. As a result, various attempts have been made to extract secret information from those chips. One type of attack against RoT chips are physical attacks; here, transistor-level vulnerabilities are explored. In those attacks, the attacker either acts actively, for example, by using fault attacks [4, 5] or passively, for example, by using Side Channel Attacks (SCAs)[6, 7, 8], to extract secret information.

One project in the field of RoT chips is OpenTitan [1], an open source project that makes the RoT design and implementation more transparent, trustworthy, and secure. Although OpenTitan has implemented many measurements against physical attacks, vulnerabilities arise when adding multiple CPUs to increase the chip's performance. Those vulnerabilities allow the extraction of information from the communication between CPUs.

Therefore, this thesis presents a hardware design that adds end-to-end on-chip encryption to OpenTitan to prevent such physical attacks. For this, the design uses the existing AES IP core for the encryption or decryption, respectively, and a newly developed IP core for the communication between CPUs. After a detailed look at the hardware design, this thesis explains the implementation of the hardware design and its integration into an existing chip design from OpenTitan. Later, this thesis evaluates the correctness of the implementation and looks at the performance of different parts of the hardware design. Finally, this thesis shows that for the encryption and sending of a message using the design presented in this paper, the encryption of the data is the bottleneck, whereas, for the reception and decryption, the reading and writing to registers is the bottleneck when using 128-bit and 192-bit keys.

2 Background

2.1 OpenTitan

OpenTitan[1] is the first open-source project that builds a transparent reference design and integration guidelines for silicon Root of Trust (RoT) chips. It was launched in 2019 by a consortium of companies, including Google, G+D Mobile Security, Western Digital, and others. The project wants to give an open-source alternative to proprietary RoT chips to increase its trustworthiness.

The lowRISC CIC, a not-for-profit company, stewards the OpenTitan project. The project's goals are to make the implementation of RoT chips for enterprises, platform providers, and chip manufacturers more transparent, trustworthy, flexible, and secure.

Apart from being open-source, which allows anyone to inspect and evaluate the project, it also implements a comprehensive verification process that covers all aspects of the design, including hardware, firmware, and software components.

Furthermore, the project separates the comportable IPs and the design of microcontrollers to make reusability possible. This separation allows a much faster implementation of RoT chips for different use cases, e.g., data center servers, peripherals, storage devices, and other hardware. It therefore also reduces the cost of new System on a Chip (SoC) designs.

2.1.1 Earl Grey

Earl Grey is a low-power microcontroller from the OpenTitan project designed for several use cases requiring hardware security. It combines different comportable IPs from the project and IPs from other vendors; the registers of those IPs are mapped in memory (MMIO) and are, therefore, easily accessible by software. The Earl Grey chip uses the open-source Ibex processor. All the chip components are connected over the TL-UL bus, as seen in the block diagram of the Earl Grey highspeed domain (Figure 2.1).

The Earl Grey design is synthesizable for the CW310 FPGA board. It also can be simulated using the cycle-accurate simulation tool Verilator.

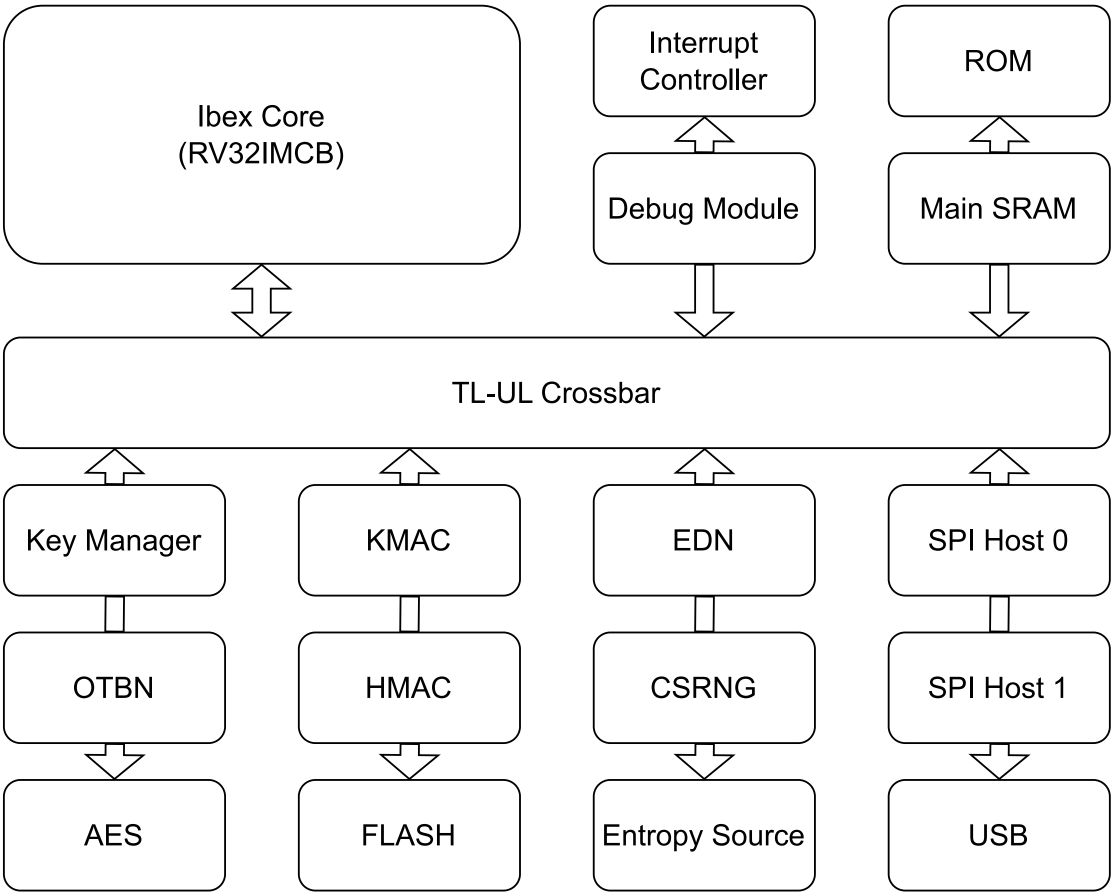


Figure 2.1: A block diagram of the Earl Grey High Speed Domain, adapted from [9].

Ibex Core

As mentioned, the Earl Grey microcontroller uses the Ibex Core [10]. The Ibex Core is an open-source 32-bit RISC-V CPU core written in System Verilog. It is designed to be highly configurable and customizable, allowing users to quickly tailor it to meet their specific requirements. For example, this allows config as to which extensions are used, if the optional third pipeline stage (Writeback Stage) is enabled, if the *Multiplication and Division* Extension is enabled, and others.

The Ibex Core is based on the RISC-V Instruction Set Architecture (ISA), which will be described later in detail, and like the OpenTitan project stewarded by the lowRISC CIC.

Earl Grey uses the Ibex Core with three pipeline stages. It enables both the *Bit Manipulation* and *Multiplication and Division* Extension. In more detail, it adds the single cycle Multiplication Extension. The entire configuration of the Ibex Core used in the Earl Grey design can be seen in Figure 2.2.

```
PMPEnable: "1",
PMPGranularity: "0",
PMPNumRegions: "16",
MHPMCounterNum: "10",
MHPMCounterWidth: "32",
RV32E: "0",
RV32M: "ibex_pkg::RV32MSingleCycle",
RV32B: "ibex_pkg::RV32BOTEarlGrey",
RegFile: "ibex_pkg::RegFileFF",
BranchTargetALU: "1",
WritebackStage: "1",
ICache: "1",
ICacheECC: "1",
ICacheScramble: "1",
BranchPredictor: "0",
DbgTriggerEn: "1",
DbgHwBreakNum: "4",
SecureIbex: "1",
DmHaltAddr: "tl_main_pkg::ADDR_SPACE_RV_DM__MEM␣+␣dm::HaltAddress[31:0]",
DmExceptionAddr: "tl_main_pkg::ADDR_SPACE_RV_DM__MEM␣+␣dm::ExceptionAddress[31:0]",
PipeLine: "0"
```

Figure 2.2: Ibex Core Configuration used in the Earl Grey design.

TL-UL

OpenTitan uses the TileLink [11] standard for chip-level interconnect as its bus protocol, which was built for RISC-V. More specifically, it uses the TileLink Uncached Lightweight (TL-UL) variant, the simplest variant in the TileLink standard but sufficient for OpenTitan.

This variant allows simple memory read and write operations of single words. It supports multiple bus hosts, bus devices, and clock domains. In contrast to the other TileLink variants, the TL-UL variant does not support bursts, meaning a message cannot be spread across multiple clock cycles. In addition to the TL-UL specification, OpenTitan added a user extension for future IPs.

OpenTitan provides bus primitives that can be combined to form a flexible crossbar of any M to any N devices. At the time of writing, these crossbars are generated automatically through the usage of config files.

AES IP Core

The AES IP core is the primary symmetric encryption and decryption accelerator used in OpenTitan. At the time of writing, it supports the following modes of operation: Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR). As specified in the Advanced Encryption Standard (AES) [12], the core encrypts and decrypts the data in blocks of 128 bits using cryptographic keys of 128, 192, or 256 bits. The support of 192-bit keys can be removed to save area on the chip. This can be done by a compile-time Verilog parameter. In order to not expose the key material to the processor and other hosts attached to the system bus interconnect, the AES IP core makes it possible to enable key sideloading from the system's key manager. In addition, the core implements domain-oriented masking [13] against first-order SCAs, which also can be disabled. The AES IP core can be operated in automatic mode, where it automatically starts the encryption/decryption, or in manual mode, where it is triggered by writing to a register.

2.2 RISC-V

RISC-V [14] [15] is a family of related open ISAs with four base ISAs, at the time of writing. RV32I and RV64I are the primary base integer variants. They provide 32-bit and 64-bit address spaces, and are both already ratified. Apart from these two variants, the RV32E and RV128I variants exist; both variants are still drafts. RV32E is a subset variant of RV32I and was added to support small microcontrollers and has half the number of integer registers as RV32I. On the other side, RV128I is another base integer variant, in this case, with a 128-bit address space. Although currently not needed, this variant was added to be present for eventual need in the future.

As RISC-V has been designed to be extensively customizable and specializable, multiple optional instruction-set extensions can be added to the mentioned base variants. The most common ones are listed in Table 2.2. If all of these extensions are present, they usually are combined into the symbol G, which signals a general-purpose ISA.

Base Name	Version	Status
RV32I	2.1	Ratified
RV64I	2.1	Ratified
RV32E	1.9	Draft
RV128I	1.7	Draft

Table 2.1: RISC-V Base Variants, adapted from [14]

Extension Symbol	Description
M	Standard Extension for Integer Multiplication and Division.
A	Standard Extension for Atomic Instructions.
F	Standard Extension for Single-Precision Floating-Point.
D	Standard Extension for Double-Precision Floating-Point.
Zicsr	Control and Status Regsiter (CSR) Instructions.
Zifencei	Instruction-Fetch Fence.

Table 2.2: Most common RISC-V extensions.

2.2.1 Benchmarking

The RISC-V ISA defines some important performance registers as part of the Control and Status Registers (CSRs) that increment on certain conditions. One is the machine cycle counter, which contains the current cycle count. This register is used to measure the performance of our implementation. The results of these measurements are presented in Chapter 6. Since the machine cycle counter is a 64-bit value and the Ibex Core is a 32-bit processor, the value is stored in two 32-bit registers (´mcycle´ and ´mcycleh´). This adds some inconveniences to the measurement. While reading the ´mcycleh´ register (higher 32 bits) the ´mcycle´ register (lower 32 bits) could overflow. This is solved by the ibex_mcycle_read() function we use to read the machine cycle counter. The function uses the assembly code in Figure 2.3, adapted from [14]. This assembly code ensures that a valid 64-bit value is returned by adding an overflow check. This check, however, adds some overhead to the function, which we want to discuss now.

Before the execution of the code under test and after the final readout of the ‘mcycle‘ register, as seen in Figure 2.3, we have an additional CSR read and a branch that is not taken. This leads to an overhead of 2 instructions before the code execution. After the code was executed, we have two possible scenarios. Scenario one is that we have no overflow during the readout of the ‘mcycleh‘ register. This results in one CSR read before the readout of the final ‘mcycle‘ register. In the second scenario, we have an overflow of the ‘mcycle‘ register. If this is the case, we have four CSR reads and one branch taken before the final readout of the ‘mcycle‘ register. In total, we have an overhead of three instructions (2 CSR reads; 1 branch not-taken) in the best case and an overhead of seven instructions (5 CSR reads; 1 branch not taken; 1 branch taken) in the worst case. When looking at Table 2.3, this is three cycles in the best case and at least eight cycles in the worst.

```
1:
csrr cycle_high,  mcycleh;
csrr cycle_low,   mcycle;
csrr cycle_high2, mcycleh;
bne  cycle_high,  cycle_high2, 1b;
```

Figure 2.3: Assembly code to read number of cycles as in [1]. Adapted from [14]

Instruction Type	Cycles	Comments
Integer Computational	1	
CSR Access	1	
Multiplication	1/2	1 for MUL, 2 for MULH
Branch (Not-Taken)	1	
Branch (Taken)	2 - N	Branch Target ALU is enabled

Table 2.3: Numbers of cycles for the Ibex Core as configured in the Earl Grey design, adapted from [16].

2.3 Block Cipher Modes of Operation

The block cipher is only used to encrypt/decrypt one block of 128 bits securely. Additionally, block cipher modes are needed that describe how this block cipher is used to encrypt/decrypt data consisting of multiple blocks. Next, we briefly introduce the different cipher modes the AES IP core supports. Their encryption part can be seen in Figure 2.4.

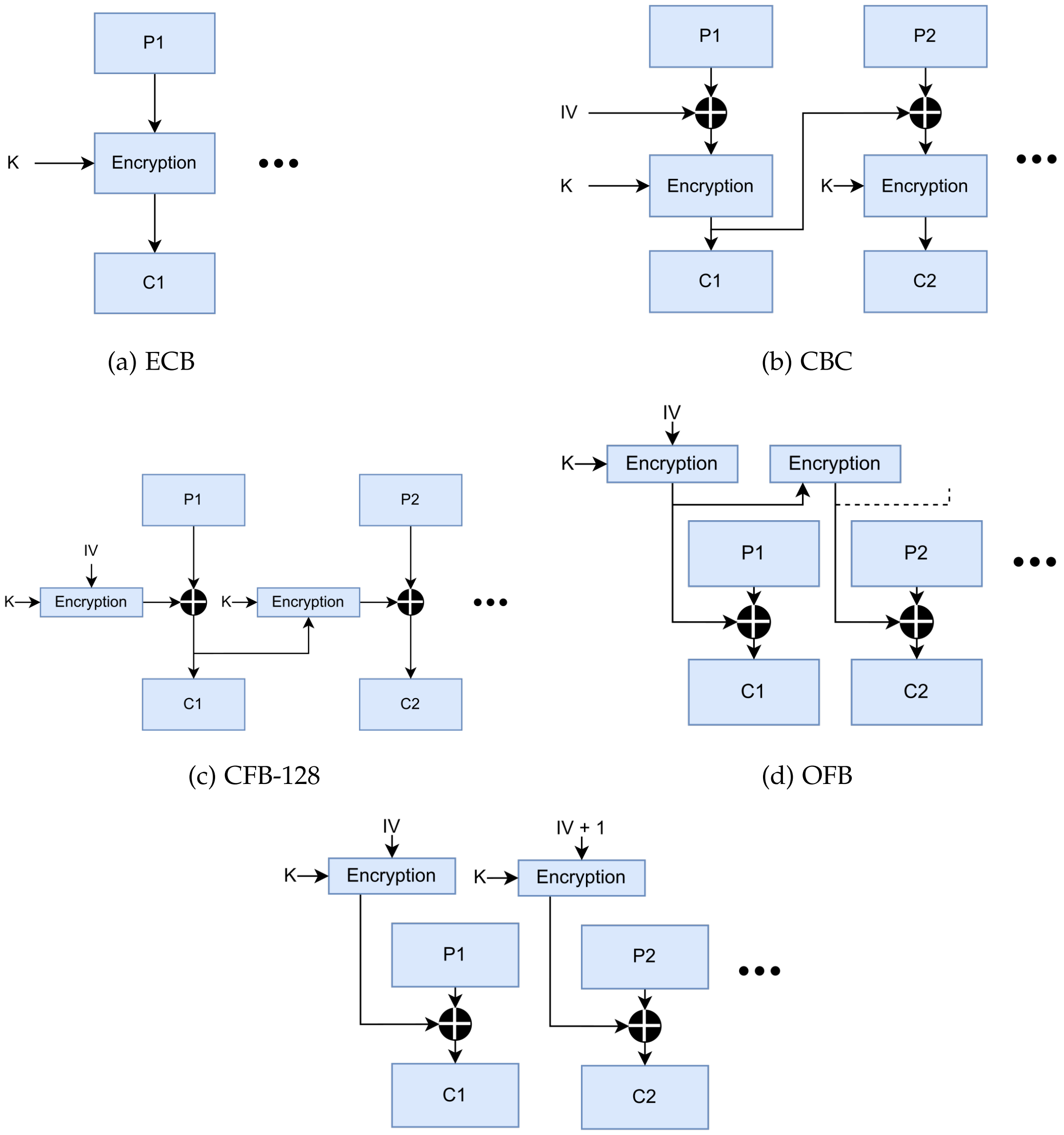
Electronic Code Book (ECB): The ECB mode directly encrypts/decrypts the input block using the block cipher. This results in the same output block for the same input blocks.

Cipher Block Chaining (CBC): When encrypting using the CBC mode, the input block gets XORed with the previous output block (or the IV for the first block). Then the result of this operation is encrypted using the key. For the decryption, these steps are reversed. First, the input block gets decrypted, then the result gets XORed with the previous input block (or IV).

Cipher Feedback Mode (CFB-128): In the CFB-128 mode, during encryption, first, the previous output block (or IV) gets encrypted. Then the result gets XORed with the input block. In the decryption, this mode first encrypts the previous input block (or IV) and afterward XORs it with the input block.

Output Feedback Mode (OFB): The OFB works similarly to the CFB-128 mode. For the first input block, the IV gets encrypted, and the result, apart from being XORed with the input block to produce the output block, takes the place of the IV for the next block, and so on. Both encryption and decryption work in the same way.

Counter (CTR): In the CTR mode, first, the counter value (index of the block (starting with 0) gets added to the IV), gets encrypted then is XORed with the input block.



(a) ECB

(b) CBC

(c) CFB-128

(d) OFB

(e) CTR

Figure 2.4: Block Cipher Modes of Operation

3 Overview

This chapter gives a brief overview of the system, its workflow, and the design goals adhered to by the hardware design.

3.1 System Overview

First, we look at the hardware design for the end-to-end on-chip encryption. There are two CPUs, in our case Ibex cores, which are both connected to their own AES IP core and CMOD IP core. The AES IP cores are used to encrypt/decrypt the messages sent/received. The CMOD IP cores, on the other hand, are used to send and receive those encrypted messages. For this, they are both interconnected with each other.

Next, we look at the setup to run our software, which tests the hardware design. To run the software, we need a way to simulate the hardware it runs on. For this OpenTitan either supports the ChipWhisperer CW310 FPGA board or Verilator [17]. We use Verilator since it is a free tool and does not require any additional hardware. In both cases, a host PC is needed. When using Verilator, the host PC is used to run the simulation, and when using an FPGA board, it is used to communicate with the simulation.

3.2 Design Goals

Our hardware design adheres to the following design goals:

1. Flexibility: Our hardware design should work on different chip designs that are part of the OpenTitan project. This is already promoted by the OpenTitan project due to the separation of the chip design and the design of the accelerators. Also, our communication module is separated from the chip design and, therefore, can be reused for different chip designs.
2. Comportability: Our module follows the Comportability Definition and Specification [18]. This defines a minimum set of functionality that must be implemented to comply with the chip design around it. It also defines the interfaces of peripherals in order to be easily integrable.

3. Performance: We would like to ensure that the slowdown in terms of performance due to both encryption and communication is as low as possible. Additionally, our hardware design for end-to-end encrypted communication should not stall the CPUs unnecessarily. Therefore we added buffers to our communication module, which allows sending even if the other CPU is busy.

3.3 System Workflow

Since we decided to use Verilator to simulate our hardware design, next, we will describe the end-to-end workflow of the system using the Verilator simulator. More information about the FPGA setup can be found at [19].

The workflow using Verilator is as follows:

1. Verilator is used to generate and compile a C++ simulation of the hardware.
2. 2. Next, the software that is run on the simulated hardware in OpenTitan, is built using bazel [20]. Bazel creates images for the ROM, the Flash, and the OTP. The ROM image contains the Test ROM, which in our case, jumps directly to the Flash, which contains the software we want to run. The OTP image contains root secrets, runtime configuration data, and life cycle state.
3. Next, we run the simulation generated by Verilog and provide it with the ROM, Flash, and OTP images as attributes.
4. All the logging is done over the UART IP core. The simulation creates a pseudoterminal to interact with the system. The path to the pseudoterminal is printed out during the simulation.

4 Design

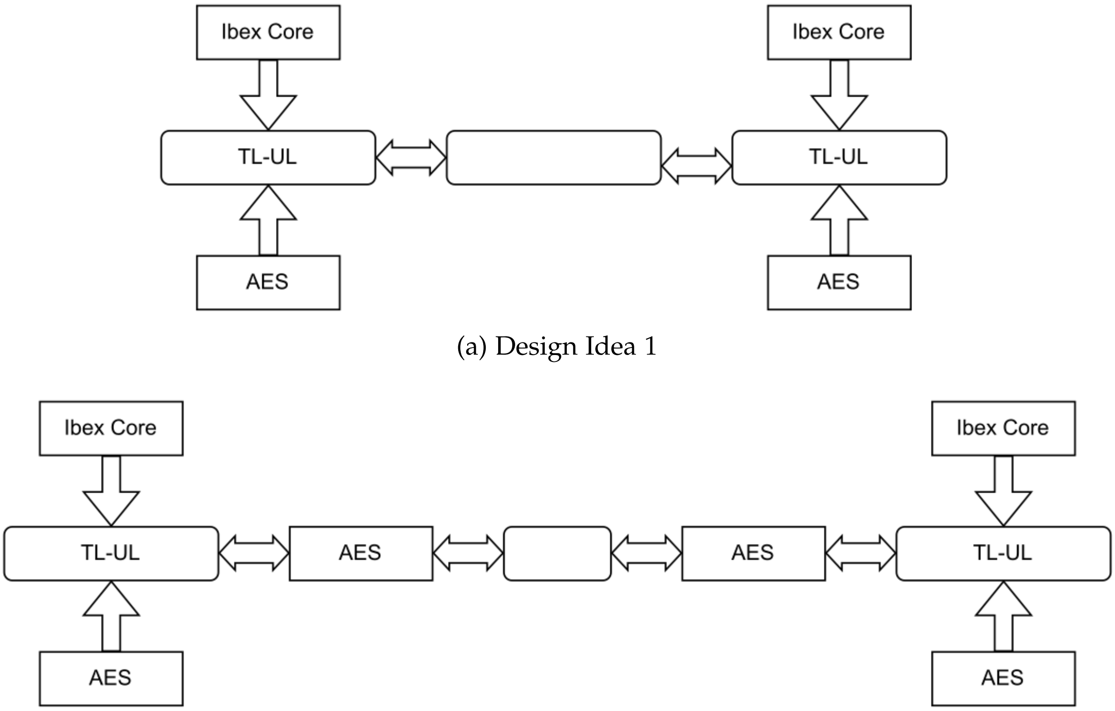
This section describes the design of the end-to-end encrypted communication in more detail. First, the design ideas we elaborated on at the beginning will be presented and compared. Next, the final design will be described in more detail. We will describe the existing IP cores that we use, and the one we newly developed. In the end, the workflow of the used IP cores will be described in more detail.

4.1 Design Ideas

In order to implement end-to-end on-chip encryption, initially, we elaborated on three different design ideas. These three ideas, which can be seen in Figure 4.1 and Figure 4.2, are the following:

1. The first idea, shown in Figure 4.1a, is to manage the communication entirely by software and to connect the communicating CPUs directly. This design has the advantage that no additional hardware is required, decreasing the chip size. However, this design also lowers the performance of the chip. The decrease in performance is because the sending core always needs to wait for the receiving core to be ready to receive the data.
2. In contrast to Figure 4.1a, we have additional hardware in the second design as shown in Figure 4.1b. This idea adds a modified AES IP core for each ibex core. This additional hardware manages everything, from the encryption/decryption to the communication. In this design, the sending CPU only writes the message in defined registers, and then the module encrypts, waits until the receiving module is ready, and then sends the message. When receiving, the CPU can directly read the decrypted message from the module registers. Compared to design one, this design heavily increases the performance because the CPUs do not have to busy wait. However, this design also has some disadvantages. Implementing this design heavily increases the board size and is also a lot more effort due to the complexity of the modifications required on the AES IP core.
3. The final design idea, shown in Figure 4.2, is the one we chose, and also contains additional hardware. In this design, the additional hardware component, however,

only manages the communication. The encryption/decryption is done by the AES IP core. This design performs better than the first design idea as the CPUs do not have to busy wait. It also does not increase the chip size compared to design 2. A downside of this idea is that the AES IP core cannot be used for different applications while it is being used for communication.



(a) Design Idea 1

(b) Design Idea 2

Figure 4.1: First Design Ideas for the End-to-End On-Chip Encryption.

4.2 Final Design

As mentioned earlier, we decided to go with design idea 3, as shown in Figure 4.2. In this design, we have two CPUs. Every CPU has its own AES IP core used for encrypting and decrypting the messages and a newly developed IP core used for communication, which we call the Communication Module (CMOD). This CMOD IP core receives already encrypted datablocks from the CPU, checks whether the CMOD IP core of the receiving CPU is ready, and then sends the data to the other CMOD IP core. The receiving core then buffers the received data blocks and notifies the CPU. The receiving CPU then can read the data block and decrypt it using its AES IP core. The workflow of the AES IP core and the CMOD IP core is described later in this chapter.

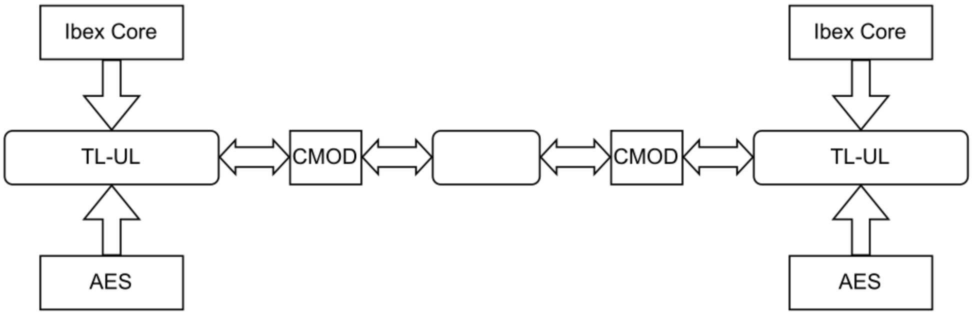


Figure 4.2: Final Design Idea for the End-to-End On-Chip Encryption.

4.2.1 Communication Module (CMOD)

As described above, the CMOD IP core is used to communicate between two cores. It conforms to the Comportability Definition and Specification [18] for Comportable IPs, which mainly regulates their interface. This specification is the minimum functionality that every Comportable IP must adhere to, to comply with the framework around it. In our case this framework is the existing Earl Grey design. Comportable IPs must specify the following features:

- **Clocking:** Each peripheral must specify its primary functional clock. Additional clocks can be added as needed.
- **Bus Interface:** All the bus interfaces supported by the peripheral need to be listed. At the time of writing this thesis, these interfaces only support TL-UL. However, this may change in the future.
- **Registers:** Every peripheral must define its registers connected to the bus.

Those features are defined in a config file explained in more detail in Chapter 5. The CMOD IP core has one main clock, which consists of the clock and reset signal. It is connected to the TL-UL bus and takes the role of a device, which means its registers are available for hosts to write to and read from. We now want to give a brief overview of the registers, their exact description can be seen in the tables of Subsubsection 4.2.1.

The CMOD IP core has fourteen registers with a width of 32 bits. These registers consist of multiple bits and fields. Four of the registers are for the interrupts and alerts defined for the module. Additionally, we have two multi-registers that each consist of four registers, where the received data can be read from and where the data to be sent is written. This results in a total width of 128 bits, which is the size of the data blocks the module works with. This size is chosen because the AES IP core also works with this size, and it, therefore simplifies the workflow and improves performance. Additionally, we have one status register to read the state of the CMOD IP core and one control register to control it. All the registers and their fields/bits are listed and described in Subsubsection 4.2.1.

Apart from the mandatory features, comportable IPs can also specify the following optional ones:

- **Available IO:** Each peripheral can define which signals (input, output, or inouts) are available for chip IO.
- **Interrupts:** Each peripheral can define signals that are used to interrupt the processor.
- **Alerts:** Peripherals have the option of generating signals that are used to indicate potential security threats.
- **Inter Signal:** In addition to being connected over the TL-UL bus, peripherals can also have signals that are directly connected to other peripherals. These signals are called Inter Module Signals in OpenTitan.

In the CMOD IP core, we have all of them except for available IO. A description of the different interrupts and alerts implemented can be seen in Subsubsection 4.2.1. The inter-module signals are used to connect two CMOD IP cores directly. We use them to exchange data blocks, signal availability, and mark the last data block of a message.

Next, we will describe in more detail how the CMOD IP core is used to transmit and receive messages.

Transmission

The transmission of a new message over the CMOD IP core works as follows:

1. Before starting, the software must ensure that the TX bit of the STATUS register is 0, which signals that the core is idle.
2. Once the core is idle, the software sets the TXTRIGGER bit of the CTRL register, which triggers a new transmission. As a result, the TX bit of the STATUS register is set, which signals that data written to the WDATA registers will be transmitted.

3. If the TXINPUT_READY bit of the STATUS register is set, the software writes a data block of 128 bits to the WDATA registers. Whereby every WDATA register needs to be written at least once. The order in which this is done does not matter. Should the TXINPUT_READY bit be unset, the software has to wait until it is set.
4. Step 3 is repeated until the last data block of a message is written to the WDATA registers. After that, the software must set the TXEND bit of the CTRL register to finish the transmission.
5. After the TXEND bit is set, all the additional data written to the WDATA registers will be ignored. Now all the data blocks in the WDATA BUFFER will be sent. Once the buffer is empty, the TX bit of the STATUS register will be unset by the CMOD IP core, and a new message can be transmitted.

Reception

The reception of a new message over the CMOD IP core works as follows:

1. A new message is detected either by reading the RXVALID bit of the STATUS register, which is always set when valid data is in the RDATA registers and, therefore, can be used to detect a new message, or by the rx_watermark interrupt.
2. If the RXVALID bit of the STATUS register is set, the software can read the data from the RDATA register via the TL-UL bus interface. For the CMOD IP core to detect a read, every RDATA register must be read at least once. Also, in this case, the order in which the registers are read does not matter.
3. Step 2 is repeated until the RXLAST bit of the STATUS register is set, which marks the last data block of a message.
4. Once the software has read the whole message, the RXCONFIRM bit of the CTRL register must be set to allow the CMOD IP core to receive a new message.

Register Table

This section lists all the different registers of the CMOD IP core and describes them in more detail.

Name	Offset	Length	Description
cmod.INTR_STATE	0x0	4	Interrupt State Register
cmod.INTR_ENABLE	0x4	4	Interrupt Enable Register
cmod.INTR_TEST	0x8	4	Interrupt Test Register
cmod.ALERT_TEST	0xc	4	Alert Test Register
cmod.CTRL	0x10	4	CMOD control register
cmod.STATUS	0x14	4	CMOD status register
cmod.WDATA_0	0x18	4	Data to be transmitted.
cmod.WDATA_1	0x1c	4	Data to be transmitted.
cmod.WDATA_2	0x20	4	Data to be transmitted.
cmod.WDATA_3	0x24	4	Data to be transmitted.
cmod.RDATA_0	0x28	4	If cmod.RXVALID is 1, data that was received.
cmod.RDATA_1	0x2c	4	If cmod.RXVALID is 1, data that was received.
cmod.RDATA_2	0x30	4	If cmod.RXVALID is 1, data that was received.
cmod.RDATA_3	0x34	4	If cmod.RXVALID is 1, data that was received.

Table 4.1: Summary of Registers of the CMOD HWIP.

cmod.INTR_STATE @ 0x0 Reset default = 0x0, mask 0xf				
Bits	**Type**	**Reset**	**Name**	**Description**
0	rw1c	0x0	tx_watermark	Raised if the transmit FIFO is past the high-water mark.
1	rw1c	0x0	rx_watermark	Raised if the receive FIFO is past the high-water mark.
2	rw1c	0x0	tx_empty	Raised if the transmit FIFO has emptied.

Table 4.2: Interrupt State Register of the CMOD HWIP.

<table>
<tr><th colspan="5">cmod.INTR_ENABLE @ 0x4
Reset default = 0x0, mask 0xf</th></tr>
<tr><th>Bits</th><th>Type</th><th>Reset</th><th>Name</th><th>Description</th></tr>
<tr><td>0</td><td>rw</td><td>0x0</td><td>tx_watermark</td><td>Enable interrupt when INTR_STATE.tx_watermark is set.</td></tr>
<tr><td>1</td><td>rw</td><td>0x0</td><td>rx_watermark</td><td>Enable interrupt when INTR_STATE.rx_watermark is set.</td></tr>
<tr><td>2</td><td>rw</td><td>0x0</td><td>tx_empty</td><td>Enable interrupt when INTR_STATE.tx_empty is set.</td></tr>
</table>

Table 4.3: Interrupt Enable Register of the CMOD HWIP.

<table>
<tr><th colspan="5">cmod.INTR_TEST @ 0x8
Reset default = 0x0, mask 0xf</th></tr>
<tr><th>Bits</th><th>Type</th><th>Reset</th><th>Name</th><th>Description</th></tr>
<tr><td>0</td><td>wo</td><td>0x0</td><td>tx_watermark</td><td>Write 1 to force INTR_STATE.tx_watermark to 1.</td></tr>
<tr><td>1</td><td>wo</td><td>0x0</td><td>rx_watermark</td><td>Write 1 to force INTR_STATE.rx_watermark to 1.</td></tr>
<tr><td>2</td><td>wo</td><td>0x0</td><td>tx_empty</td><td>Write 1 to force INTR_STATE.tx_empty to 1.</td></tr>
</table>

Table 4.4: Interrupt Test Register of the CMOD HWIP.

<table>
<tr><th colspan="5">cmod.ALERT_TEST @ 0xc
Reset default = 0x0, mask 0x1</th></tr>
<tr><th>Bits</th><th>Type</th><th>Reset</th><th>Name</th><th>Description</th></tr>
<tr><td>0</td><td>wo</td><td>0x0</td><td>fatal_fault</td><td>Write 1 to trigger one alert event of this kind.</td></tr>
</table>

Table 4.5: Alert Test Register of the CMOD HWIP.

<table>
<tr><th colspan="5">cmod.CTRL @ 0x10
Reset default = 0x0, mask 0x3f</th></tr>
<tr><th>Bits</th><th>Type</th><th>Reset</th><th>Name</th><th>Description</th></tr>
<tr><td>0</td><td>rw</td><td>0x0</td><td>TXTRIGGER</td><td>Starts the transmission of a new message.</td></tr>
<tr><td>1</td><td>rw</td><td>0x0</td><td>TXEND</td><td>Marks that the last data block of a message was written to the cmod.WDATA registers.</td></tr>
<tr><td>2</td><td>rw</td><td>0x0</td><td>RXCONFIRM</td><td>Confirms that the end of a message was noticed and allows the reception of a new message.</td></tr>
<tr><td>4:3</td><td>rw</td><td>0x0</td><td>TXILVL</td><td>Trigger level for tx_watermark interrupt. If the FIFO depth is smaller to the setting, it raises a tx_watermark interrupt.
<table>
<tr><td>0x0</td><td>txlvl2</td><td>2 data blocks</td></tr>
<tr><td>0x1</td><td>txlvl3</td><td>3 data blocks</td></tr>
<tr><td>0x2</td><td>txlvl4</td><td>4 data blocks</td></tr>
<tr><td>0x3</td><td></td><td>Reserved</td></tr>
</table></td></tr>
<tr><td>6:5</td><td>rw</td><td>0x0</td><td>RXILVL</td><td>Trigger level for rx_watermark interrupt. If the FIFO depth is greater or equal to the setting, it raises a rx_watermark interrupt.
<table>
<tr><td>0x0</td><td>rxlvl1</td><td>1 data block</td></tr>
<tr><td>0x1</td><td>rxlvl2</td><td>2 data blocks</td></tr>
<tr><td>0x2</td><td>rxlvl3</td><td>3 data blocks</td></tr>
<tr><td>0x3</td><td></td><td>Reserved</td></tr>
</table></td></tr>
</table>

Table 4.6: CMOD Control Register.

cmod.STATUS @ 0x14
Reset default = 0x0, mask 0xfff

Bits	Type	Reset	Name	Description
0	ro	x	TX	The CMOD unit is idle (0) or busy(1). This bit is ‘0’ if no message is currently transmitted. This bit is ‘1’ one of the following is the case: i) The transmission has started (TXTRIGGER set), and the last data block of the message was not yet received (TXEND was not set yet), ii) The last data block of a message was received but still needs to leave the outgoing buffer.
1	ro	x	TXFULL	The buffer for outgoing data is full.
2	ro	x	RXFULL	The buffer for incoming data is full.
3	ro	x	TXEMPTY	The buffer for outgoing data is empty.
4	ro	x	TXINPUT_READY	The CMOD unit is ready (1) or not ready (0) to receive new input via the cmod.WDATA registers. If this bit is ‘0’, it means that either the transmission of a new message was not started yet or that the buffer for outgoing data is full.
5	ro	x	RXVALID	The CMOD unit has no valid output (0) or valid output data.
8:6	ro	x	TXLVL	The current fill level of the buffer for outgoing data.
11:9	ro	x	RXLVL	The current fill level of the buffer for incoming data.
12	ro	x	RXLAST	If set, it means that the last data block of a message was already read. To allow the reception of a new message, the cmod.RXCONFIRM bit needs to be set.

Table 4.7: CMOD Status Register.

<table>
<tr><th colspan="5">cmod.WDATA_0 @ 0x18
Data to be transmitted.
If the cmod.TXINPUT_READY bit is unset the data written is ignored.
Reset default = 0x0, mask 0xffffffff</th></tr>
<tr><th>Bits</th><th>Type</th><th>Reset</th><th>Name</th><th>Description</th></tr>
<tr><td>31:0</td><td>wo</td><td>0x0</td><td>WDATA_0</td><td></td></tr>
</table>

Table 4.8: WDATA0 Register of the CMOD HWIP.

<table>
<tr><th colspan="5">cmod.WDATA_1 @ 0x1c
Data to be transmitted.
If the cmod.TXINPUT_READY bit is unset the data written is ignored.
Reset default = 0x0, mask 0xffffffff</th></tr>
<tr><th>Bits</th><th>Type</th><th>Reset</th><th>Name</th><th>Description</th></tr>
<tr><td>31:0</td><td>wo</td><td>0x0</td><td>WDATA_1</td><td></td></tr>
</table>

Table 4.9: WDATA1 Register of the CMOD HWIP.

<table>
<tr><th colspan="5">cmod.WDATA_2 @ 0x20
Data to be transmitted.
If the cmod.TXINPUT_READY bit is unset the data written is ignored.
Reset default = 0x0, mask 0xffffffff</th></tr>
<tr><th>Bits</th><th>Type</th><th>Reset</th><th>Name</th><th>Description</th></tr>
<tr><td>31:0</td><td>wo</td><td>0x0</td><td>WDATA_2</td><td></td></tr>
</table>

Table 4.10: WDATA2 Register of the CMOD HWIP.

<table>
<tr><th colspan="5">cmod.WDATA_3 @ 0x24
Data to be transmitted.
If the cmod.TXINPUT_READY bit is unset the data written is ignored.
Reset default = 0x0, mask 0xffffffff</th></tr>
<tr><th>Bits</th><th>Type</th><th>Reset</th><th>Name</th><th>Description</th></tr>
<tr><td>31:0</td><td>wo</td><td>0x0</td><td>WDATA_3</td><td></td></tr>
</table>

Table 4.11: WDATA3 Register of the CMOD HWIP.

<table>
<tr><th colspan="5">cmod.RDATA_0 @ 0x28
If cmod.RXVALID is 1, data that was received.
Reset default = 0x0, mask 0xffffffff</th></tr>
<tr><th>Bits</th><th>Type</th><th>Reset</th><th>Name</th><th>Description</th></tr>
<tr><td>31:0</td><td>ro</td><td>0x0</td><td>RDATA_0</td><td></td></tr>
</table>

Table 4.12: RDATA0 Register of the CMOD HWIP.

<table>
<tr><th colspan="5">cmod.RDATA_1 @ 0x2c
If cmod.RXVALID is 1, data that was received.
Reset default = 0x0, mask 0xffffffff</th></tr>
<tr><th>Bits</th><th>Type</th><th>Reset</th><th>Name</th><th>Description</th></tr>
<tr><td>31:0</td><td>ro</td><td>0x0</td><td>RDATA_1</td><td></td></tr>
</table>

Table 4.13: RDATA1 Register of the CMOD HWIP.

<table>
<tr><th colspan="5">cmod.RDATA_2 @ 0x30
If cmod.RXVALID is 1, data that was received.
Reset default = 0x0, mask 0xffffffff</th></tr>
<tr><th>Bits</th><th>Type</th><th>Reset</th><th>Name</th><th>Description</th></tr>
<tr><td>31:0</td><td>ro</td><td>0x0</td><td>RDATA_2</td><td></td></tr>
</table>

Table 4.14: RDATA2 Register of the CMOD HWIP.

<table>
<tr><th colspan="5">cmod.RDATA_3 @ 0x34
If cmod.RXVALID is 1, data that was received.
Reset default = 0x0, mask 0xffffffff</th></tr>
<tr><th>Bits</th><th>Type</th><th>Reset</th><th>Name</th><th>Description</th></tr>
<tr><td>31:0</td><td>ro</td><td>0x0</td><td>RDATA_3</td><td></td></tr>
</table>

Table 4.15: RDATA3 Register of the CMOD HWIP.

4.2.2 AES IP core

As mentioned we use the AES IP core to encrypt/decrypt the data blocks of a message, before sending or receiving them using the CMOD IP core. In Chapter 2 we already talked a little bit of the core. However, to understand the complete process of the end-to-end encrypted communication, we now explain the workflow of the AES IP core in more detail:

1. If the IDLE bit of the STATUS register is set, the software starts with initializing the AES unit. For this, it sets the operation (encryption/decryption) of the AES IP core, the AES block cipher mode (ECB, CBC, CFB-128, OFB, or CTR), and the key length (128-bit, 192-bit, or 256-bit) by writing to the respective bits/fields of the CTRL_SHADOWED register. Additionally, the software can enable sideloading, where the key gets loaded from the system key manager, set the reseeding rate of the internal pseudo-random number generator (PRNG), and determine whether the AES unit works in automatic mode (the encryption/decryption starts automatically when new input data is received) or manual mode (the encryption/decryption is triggered by setting the START bit of the TRIGGER register).

2. After initializing the AES IP core, the software loads the initial key. The key is written in shares of two, which XORed together result in the actual key. For this, every KEY_SHARE register needs to be written at least once. The order in which the registers are written does not matter. If sideloading is enabled, the key will be loaded from the key manager via the key sideload interface.

3. Then, if necessary for the selected AES block cipher mode, the initialization vector (IV) is written to the four IV registers. Also, in this case, every register must be written at least once.

4. After the previous steps, the AES unit is ready for encryption/decryption. Now if the INPUT_READY bit of the STATUS register is set, the software loads a 128-bit data block to the DATA_IN registers and, if necessary, sets the START bit as mentioned earlier.

5. Once the OUTPUT_VALID bit of the STATUS register is set, the encrypted or decrypted data can be read from the DATA_OUT register.

6. Steps 4-5 are repeated until the whole data is encrypted/decrypted.

7. Once the encryption/decryption is finished, the AES unit needs to be deinitialized for security reasons by setting the MANUAL_OPERATION bit of the

CTRL_SHADOWED register and clearing all KEY_SHARE registers, IV registers, as well as the DATA_IN and the DATA_OUT registers with pseudo-random data by setting the KEY_IV_DATA_IN_CLEAR and DATA_OUT_CLEAR bits of the TRIGGER register.

5 Implementation

In this chapter, we describe the different parts of the project in terms of their implementation. First, we describe the implementation of the Communication Module, which files it consists and what purpose they have. Then, we explain the changes made to integrate the module into the Earl Grey design.

5.1 Communication Module (CMOD)

The main part of this thesis is the Communication Module, which is used to send data, encrypted or un-encrypted, between cores. In this section, we describe its implementation. We describe the Configuration and Register Definition File and the RTL Design Sources.

5.1.1 Configuration and Register Definition File

As mentioned earlier, the CMOD IP core follows the comportable guidelines. This means some mandatory features, as described in Chapter 4, of a Comportable IP have to be defined.

For the CMOD IP core, this is done in the cmod.hjson file, a snippet of this file can be seen inFigure 5.1. In addition to the mandatory features, this file defines some optional features and a general description of the CMOD IP core.

Apart from the registers, which are described in more detail in Subsubsection 4.2.1, there are two entries we want to highlight: the interrupt_list and the inter_signal_list.

In the list of the interrupts, the following interrupts are defined: **tx_watermark**, **rx_watermark**, and **tx_empty**. They all increase communication performance by preventing the outgoing buffer from being emptied and the incoming buffer from being completely filled-up until the last data block of a message is sent or received, respectively.

The inter_signal_list is for the signals between two CMOD IP cores. These intermodule signals are used to send data blocks, signal availability, and signal available data.

```
{ name: "cmod",
  clocking: [{clock: "clk_i", reset: "rst_ni"}],
  bus_interfaces: [
    { protocol: "tlul", direction: "device" }
  ],
  interrupt_list: [
    { name: "tx_watermark"
      desc: "Raised␣if..."}
    ...
  ],
  alert_list: [
    { name: "fatal_fault",
      desc: "This␣fatal␣alert..."
    }
  ],
  countermeasures: [
    { name: "BUS.INTEGRITY",
      desc: "End-to-end␣bus␣integrity␣scheme."
    }
  ],
  param_list: [
    { name:    "NumRegsData",
      type:    "int",
      default: "4",
      desc:    "Number␣of␣registers␣for␣multiregister␣WDATA␣and␣RDATA.",
      local:   "true"
    }
  ],
  regwidth: "32",
  registers: [
    ...
  ],
  inter_signal_list: [
    ...
  ],
}
```

Figure 5.1: CMOD Configuration and Register Definition File

5.1.2 Autogenerated RTL Sources

This section describes the two SystemVerilog files, cmod_reg_pkg.sv and cmod_reg_top.sv, used, in the next section, to connect the TL-UL system bus to the registers of the CMOD IP core and the registers to the rest of the core.

Both files are generated by using the Register Tool from the OpenTitan project. This tool takes the cmod.hjson file described above and generates the two files. Additionally, the tool can generate header and documentation files.

The first file is an interface between the registers and the rest of the CMOD IP core. It contains a package definition that includes two packed structures. One contains the signals driven from the register to the rest of the core, and the other contains the signals in the opposite direction. Additionally, this file includes the offsets of the different registers. A snippet of the file can be seen in Figure 5.2.

The second file contains a module that instantiates the registers of the CMOD IP core. Additionally, it connects the registers to the TL-UL bus and provides them to the rest of the hardware using the first file's interface.

```
...

typedef struct packed {
  logic [31:0] d;
  logic        de;
} cmod_hw2reg_wdata_mreg_t;

...

// HW -> register type
typedef struct packed {
  cmod_hw2reg_intr_state_reg_t intr_state; // [292:285]
  cmod_hw2reg_ctrl_reg_t ctrl; // [284:273]
  cmod_hw2reg_status_reg_t status; // [272:260]
  cmod_hw2reg_wdata_mreg_t [3:0] wdata; // [259:128]
  cmod_hw2reg_rdata_mreg_t [3:0] rdata; // [127:0]
} cmod_hw2reg_t;

...
```

Figure 5.2: Snippet of the cmod_reg_pkg.sv file

5.1.3 RTL Design Sources

The functionality of the Communication Module itself can be found in the RTL source files. These consist of the following files:

- cmod_core.sv,
- cmod_pkg.sv,
- cmod_reg_pkg.sv,
- cmod_reg_top.sv, and
- cmod.sv

Both the cmod_reg_pkg.sv file and the cmod_reg_top.sv files were already discussed in Subsection 5.1.1. The cmod_pkg.sv file consists of two structs for the signals between two CMOD IP cores. The main work, however, is done in the cmod.sv and the cmod_core.sv file.

In the cmod.sv file the cmod_reg_top module, cmod_core module, and the prim_alert_sender module are initialised. The first module is used to initialize all the registers. Additionally, it connects the registers to the TL-UL bus and the rest of the hardware. The prim_alert_sender module is used to support alerts. The cmod_core module is the core of the CMOD IP core implementation and is defined in the cmod_core.sv file. In it, all the registers are read and written. It instantiates two synced FIFO modules, which buffer the incoming and outgoing data blocks. Once the incoming buffer is full, the reception of additional data blocks is blocked by using an inter-module signal until data is read by software.

Similarly, the outgoing buffer is used; once it is full, the TXINPUT_READY bit of the STATUS register is unset, and writes to the WDATA registers are ignored. The core also instantiates the interrupt hardware, checks when interrupt conditions are met, and raises them. Additionally, the core stores CTRL bits hat were set and resets its state once a complete message is sent.

5.2 Integration into Earl Grey

In order to test the CMOD IP core, we chose to include it in the existing Earl Grey design. The Earl Grey design is mainly configured in three Hjson files:

- top_earlgrey.hjson,
- xbar_main.hjson, and
- xbar_peri.hjson.

The first file is the top configuration file for the Earl Grey design. In it, the details of the design can be found. Some information in it includes the general data width, clock sources, reset sources, and a list of instantiated peripherals. To not have to add an additional Ibex Core with all the necessary peripherals and connections, we decided to use only one core to test the functionality of the CMOD IP core. This decision does not make any difference in testing because we can connect both CMOD IP cores to the existing core and simulate the communication between the two cores by using one CMOD IP core for sending and the other for receiving. The integration of the two modules can be seen in Figure 5.3. The connections of the inter-module signal we described before are connected in this file.

The second and third files are the TL-UL description files. They describe the peripherals connected to the TL-UL bus and the connections between them. Since the communication between two cores (although only simulated here) is essential, we decided to connect the two CMOD IP cores to the high-speed domain of the Earl Grey design by adding them to the xbar_main.hjson file.

From these files and the configuration files of each peripheral added into the design, the Top Generation Tool of the OpenTitan project creates the overall top module, the interrupt controller, the pinmuxes, and the TL-UL crossbars.

```
{ name: "earlgrey",
  type: "top",
  ...
  num_cores: "1",
  module: [
    ...
    {
      name: "cmod0",
      type: "cmod",
      clock_srcs: {clk_i: "main"},
      clock_group: "infra",
      reset_connections: {rst_ni: "lc"},
      base_addr: "0x41190000",
    },
    {
      name: "cmod1",
      type: "cmod",
      clock_srcs: {clk_i: "main"},
      clock_group: "infra",
      reset_connections: {rst_ni: "lc"},
      base_addr: "0x411a0000",
    },
    ...
  ]
  ...
  inter_module: {
    'connect': {
      ...
      'cmod0.tx'        : ['cmod1.rx'],
      'cmod1.tx'        : ['cmod0.rx'],
      'cmod0.rx_wready' : ['cmod1.tx_rready'],
      'cmod1.rx_wready' : ['cmod0.tx_rready'],
    }

    'top': [...],
    'external': {...},
  },
  ...
}
```

Figure 5.3: Earl Grey Configuration File

5.3 Device Interface Functions (DIFs)

To use the CMOD IP cores, we need some higher-level software that works as an API to use the hardware. In OpenTitan, this is done by the Device Interface Functions (DIFs). These functions are a working code reference for interacting with the hardware. However, they should not be seen as drivers; they are used for testing and example code only and want to aid the implementation of non-production firmware.

As described in [21], DIFs must only depend on the following:

- the freestanding C library,
- compiler runtime libraries,
- the bitfield library (sw/device/lib/base/bitfield.h),
- the mmio library (sw/device/lib/base/mmio.h),
- the memory library for non-volatile memory (sw/device/lib/base/memory.h), and
- any IP-specific register definition files.

Additionally, DIFs must not know the location of the hardware blocks they are associated with since they do not know how many hardware blocks of this type are included in the design. Therefore, the DIFs include an init function that creates a new handle of the peripheral, a parameter of every DIF function. Additionally, the autogenerated DIFs include functions to interact with the alerts and interrupts of a peripheral.

Apart from the automatically generated functions, the CMOD DIFs allow reading the bits of the STATUS register, setting bits/fields of the CTRL register, loading 128 bits to the WDATA registers, and reading data of 128 bits from the RDATA registers.

All of these functions return a value of type 'dif_result_t'. This return value is 'kDifOk' if everything worked as expected or 'kDifBadArg' if the CMOD handle is a NULL-pointer for all functions. Additionally, the load data and read data functions return 'kDifUnavailable' if the TXINPUT_READY/RXVALID bit is unset. Lastly, 'kDifError' is returned by the read data function if the last data block of a message was already read but was not confirmed by setting the RXCONFIRM bit of the CTRL register and therefore prevents the reception of a new message.

6 Evaluation

In this section we want to evaluate our implementation. For this we answer the following questions:

1. Is the implementation of the CMOD IP core correct?
2. What are the overheads of using the CMOD IP core?
3. What is the performance of the end-to-end encrypted communication using our core?
4. What is the impact of using different cipher modes?

6.1 Correctness

In order to verify the correctness of the CMOD IP core, we implemented some high-level software in C that tests the functionality of the CMOD IP core. We decided not to include the AES IP core and thereby test the complete encrypted communication since the AES IP core is already functional and gets verified as part of the OpenTitan project.

Besides checking if the receiving module correctly receives the messages sent using the sending module in different scenarios, the tests also continuously check the status registers of the modules used.

We also tested some special cases to ensure the CMOD IP core works in every situation. The following special cases were included in the tests:

1. Sending and receiving, where the receiver is faster than the sender and, as a result, always clears its internal buffer before receiving new data blocks.
2. Send until both the incoming buffer of the sending module and the outgoing buffer of the receiving module are full. Then continue receiving and sending normally.
3. Send a new message when the receiver has yet to receive the previous message fully.

6.2 Performance

Next we want to analyze the performance of the encrypted communication.

6.2.1 Methodology

In order to analyze the end-to-end encryption, we measure the number of cycles taken by different parts of the communication. For this, we use the ibex_mcycle_read() function from the ibex library already present in OpenTitan. As explained in Chapter 2, this function adds an overhead of at least three cycles between the two measurements taken at the beginning and the end of the code execution. Since this overhead is inconsistent, the cycle count between two immediate readouts where between 4-9 cycles, we decided always to give the measurements, including the overhead.

However, this is not the only overhead. Sometimes the software needs to wait until a bit is set. For this, there are two possibilities; the first possibility are interrupts. This however, adds an additional overhead for handling the interrupts. Therefore we chose the second possibility, which are busy waits. This technique has a much smaller overhead compared to the technique using interrupts.

Additionally, we want to point out that the software that measures the cycles does not use the DIFs described before. This decision was taken since DIFs add additional checks not present in production code. Therefore only memory read, memory write, and bit manipulation functions already present in OpenTitan, were used.

6.2.2 Results

Next, we want to discuss the results of the measurements. For the communication, we always used messages of 640 bits. This size was chosen because the CMOD IP core internally has buffers for the incoming and outgoing messages, where it can store 4 data blocks of 128 bits each. As a result, when sending a message of 640 bits, if the receiving ibex core does not read the data blocks, the sender CMOD IP core needs to buffer one data block.

In the following subsections, we first present the measurements of the CMOD IP core, and the AES IP core and afterward, we present the measurements of the end-to-end encrypted communication.

CMOD IP core

First, we look at the communication measurements using the CMOD IP core. In the results illustrated in Figure 6.1, we can see that for sending a message of 640 bits, 313 cycles are needed. When looking at the different parts of the communication, we can see that the most significant part of the communication overhead using our CMOD IP core is writing the data to the registers. Since the reception of a message depends on the sender, we have no measurements taken for it. However, to read a message of 640 bits, we need about 230 cycles.

When looking at the measurements taken, we point out that the TL-UL bus only allows reading and writing of the whole register. Therefore when reading a bit of a register, the whole register is read, and then the bit is extracted from the returned value. On the other hand, when writing a bit, the register needs to be read first, the bit we want to write needs to be updated in the returned value, and afterward, the whole register needs to be written. This results in faster reading and writing from/to a register than reading/writing a bit of a register.

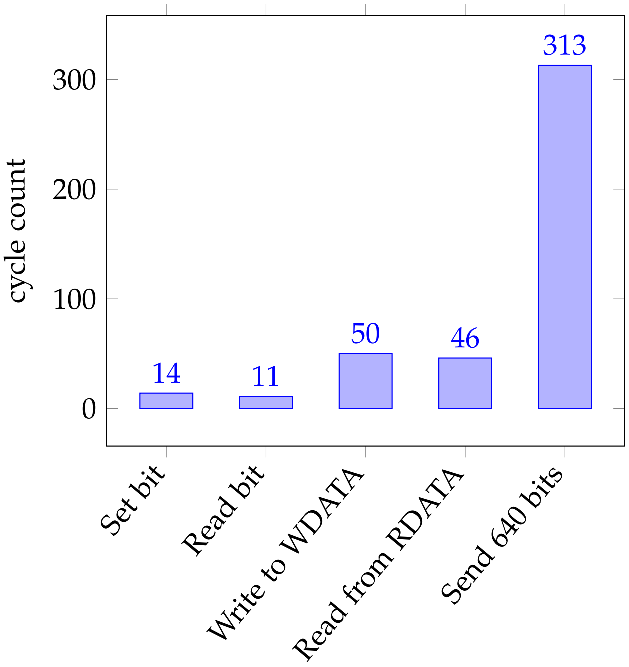


Figure 6.1: Cycles measured for different parts of the communication.

AES IP core

Next, we look at the measurements of the encryption of a message using the AES IP core. We measured the cycle count for all key lengths (128 bits, 192 bits, and 256 bits) and cipher modes (ECB, CBC, CFB-128, OFB, and CTR) that the AES IP core supports. The results can be seen in Figure 6.2. As expected, we can see a difference when using different key lengths. This is because the encryption takes longer than writing and reading data blocks to/from the AES IP core. The top of Figure 6.6 illustrates this case. Would it be the other way around, as shown at the bottom of it, there would be no difference when using different key lengths. These two cases are essential for later.

Additionally, we can see that all the cipher modes have the same execution time except for the ECB mode. This was also expected since the ECB mode does not need an IV, which leads to a much shorter initialization time for the ECB mode in comparison to the other modes since the IV registers do not have to be written. When measuring the initialization time of the different cipher modes (Figure 6.3), the same difference as in Figure 6.2 can be seen. The initialization of the ECB mode takes 138 cycles, whereas, for the other modes, it takes 351. The decision of the key length does not influence the initialization at all because all the KEY_SHARE registers must be written to, also, if not needed, by the AES IP core.

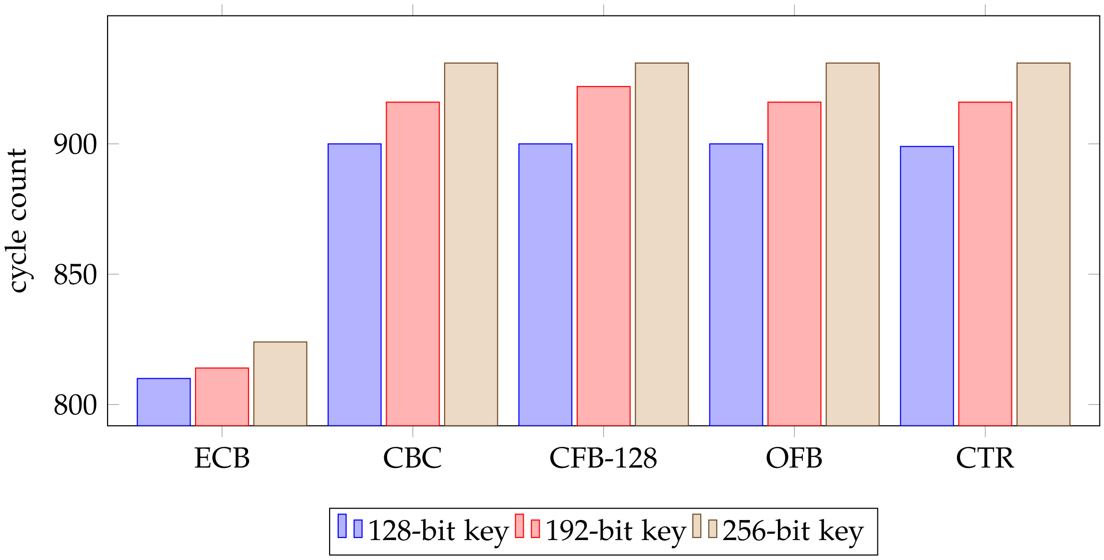


Figure 6.2: Cycles measured for encrypting 5 x 128 bit blocks.

End-to-End On-Chip Encryption

Lastly, we want to examine the measurements of the whole end-to-end encrypted communication. We begin with the measurements for encrypting and sending a message over the CMOD IP core. First, the results presented in Figure 6.4 were unexpected. Since we can see a difference when using different key lengths. Taking the previous assumption, this would mean that the encryption takes longer than the CPU

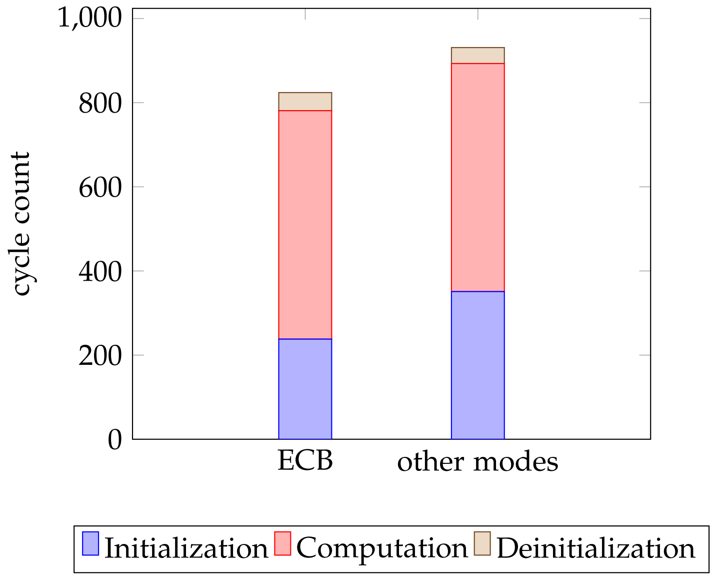


Figure 6.3: Cycles measured for the different parts of the encryption, when encrypting 5 x 128 bit blocks

for reading-writing-sending the data blocks (Figure 6.7(top)). If that is the case, we should not see any difference between the measurements in Figure 6.2 and Figure 6.4. Except for the time to set the TXTRIGGER and TXEND bit and send the last block. We measured that setting a bit takes around 11 cycles, and sending one data block takes about 50 cycles. However, this does not explain the difference of 250 cycles. Therefore the encryption must take less time than a data block's read,write, and send. As a result, we should not see a difference when using different key lengths, which, however, is the case. After we looked into it in more detail, we found the reason for these results. As shown in Figure 6.7(bottom), the encryption takes less time than the rest. However, during the first encryption, the software has to wait for the output, which is where the difference when using different key lengths comes from. An arrow in Figure 6.7 marks this part.

Lastly, we want to look at the measurements of the reception and decryption, which also were unexpected (Figure 6.5). In contrast to the measurements described before, we now can see a difference between the CBC cipher mode and the CFB-128, OFB, and CTR modes. This is, however, due to how the decryption of this mode works. Unlike the other modes (except for the ECB mode), the CBC mode needs the data block before doing the first step of each block's decryption. The other modes, however, only need the IV and the key, which they already get during the initialization of the AES IP core.

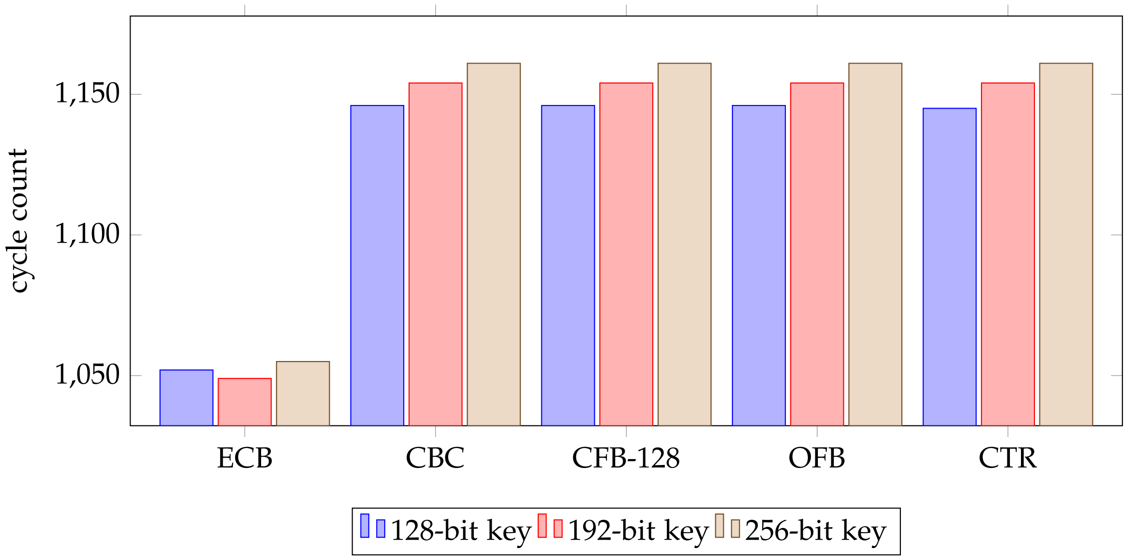


Figure 6.4: Cycles measured for encrypting and sending 5 x 128 bit blocks.

It also can be observed that for the 128-bit key and the 192-bit key, the decryption is faster than receiving, writing, and reading, whereas, for the 256-bit key, this is not the case.

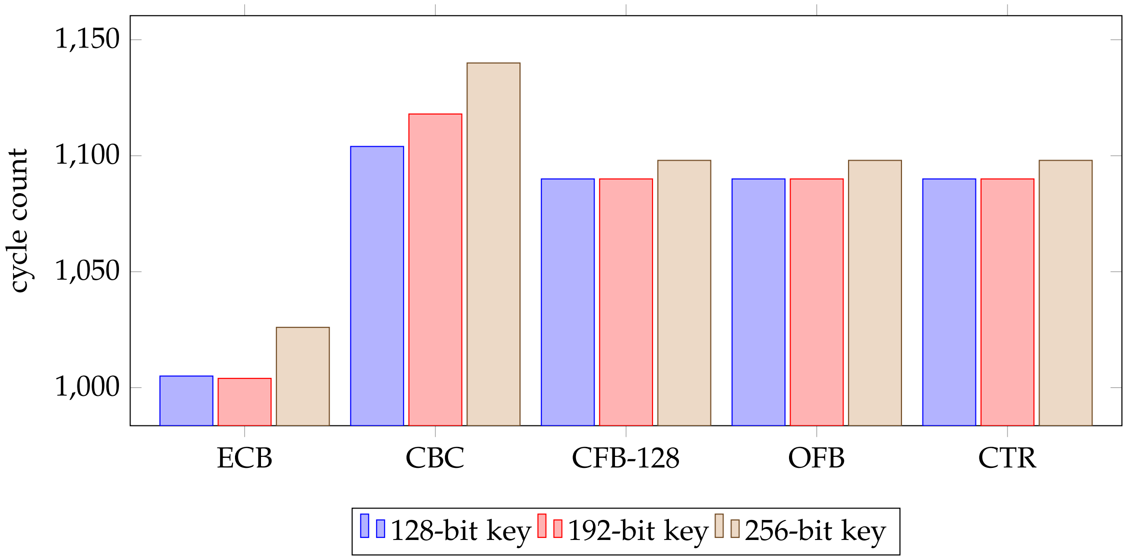


Figure 6.5: Cycles measured for receiving and decrypting 5 x 128 bit blocks.

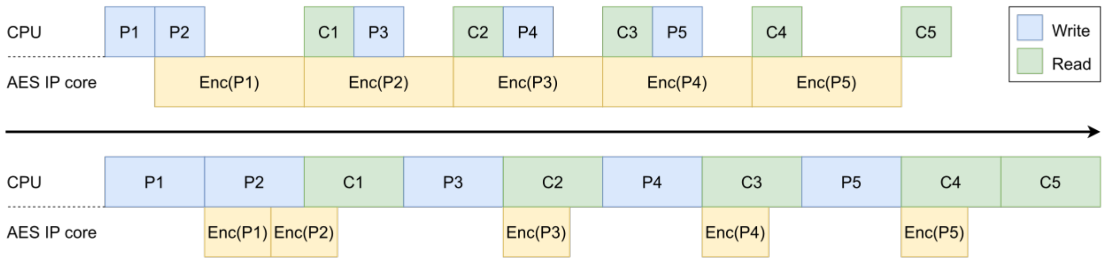


Figure 6.6: Encryption Process (top: dominant encryption; bottom: dominant read-/write)

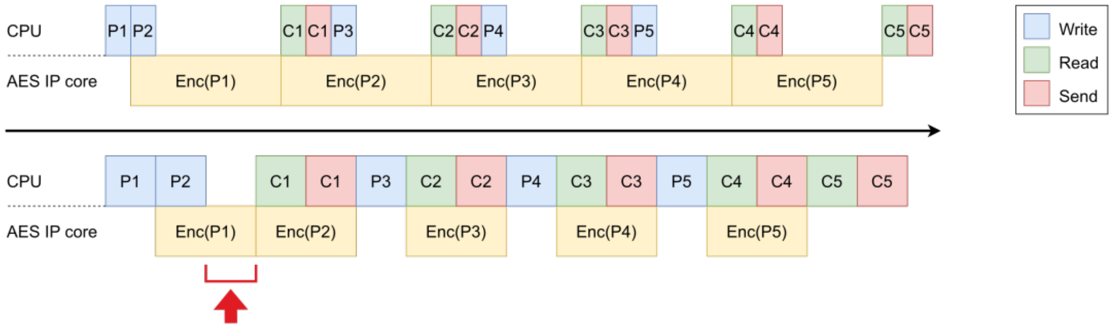


Figure 6.7: Communication Process (top: dominant encryption; bottom: dominant read/write)

6.2.3 Reproducibility

In order to make the results reproducible, we implemented a python script. This script assumes that OpenTitan is up and running.

Once the script is started, it builds the necessary files using a bazel command. These files are the chip simulator, built using Verilator, the test-ROM, the OTP image, and the software for performing the tests, which will be loaded into flash.

After starting the simulation using Popen of the subprocess library, the script reads the output of the simulation via a pipe in order to find the location of the UART file. This file is then used as an interface between the simulation host and the UART IP core, which is used for logging the results. During the simulation, the script reads the results and prints them. Once the benchmark is finished, the script stops the simulation, writes the results into a file, and finishes its execution.

7 Summary and Conclusion

In order to perform end-to-end on-chip encrypted communication, this thesis presented a hardware design using an existing AES IP core and a newly developed CMOD IP core. After giving an overview of the design and its advantages, it described the CMOD IP core in more detail. The thesis explained the registers and how they are used. It also explains, in more detail, how this hardware design can be used in different OpenTitan chip designs.

After the design was discussed, this thesis explained its implementation and gave an overview of its integration into the existing Earl Grey chip design.

After the implementation was presented, this thesis evaluated the correctness and performance of the design by presenting the functionality test in more detail and by looking at the results of the measurements taken by the performance test.

It could be seen that the bottleneck for the encryption and sending process are the writes and reads to/from the registers. On the other hand, for the receiving and decryption process, the reading and writing to registers is the bottleneck when using 128-bit and 192-bit keys.

The source code for the CMOD IP core and its integration into Earl Grey, as well as the functionality and performance tests is freely available at [2].

8 Future Work

This section presents potential future work in the form of some updates to the CMOD IP core that could be implemented.

Currently, the CPU has to manually initiate the encryption operation for every message it wants to send and then write it to the CMOD IP core. A potential improvement in performance could be gained by pipelining the AES IP core and the CMOD IP core. The pipelining can be implemented in different ways. One approach is that the CPU only writes to the CMOD IP core, and the core performs the encryption by using the AES IP core itself. When receiving a message, the CPU directly reads the decrypted message from the CMOD IP core. Another approach is that the CPU informs the CMOD IP core that it wants to send a message and then, in contrast to the first approach, the CPU writes the data directly to the AES IP core, from which the CMOD IP core then sideloads the encrypted data. The reception would work in reverse order. The second approach should perform better since it requires one lesser read/write operation. Pipelining the cores has a high impact on the performance of the end-to-end encrypted communication and, therefore, would be a worthwhile addition.

Another update for potential future work could be the support to communicate with multiple CPUs using the same CMOD IP core. Currently, in the implementation, two CMOD IP cores are hardwired together. This means that for every other CPU that a core wants to communicate with, an additional CMOD IP core has to be added. An update could replace the hardwired connection with a connection to a newly added TL-UL bus and add additional register fields for addressing. Also, this update would be a worthwhile addition because it solves the problem of having multiple CMOD IP cores per CPU by allowing a single CMOD IP core to communicate with an arbitrary amount of other CMOD IP cores.

Abbreviations

AES Advanced Encryption Standard
CBC Cipher Block Chaining
CFB Cipher Feedback
CMOD Communication Module
CTR Counter
DIFs Device Interface Functions
ECB Electronic Code Book
ISA Instruction Set Architecture
OFB Output Feedback
RoT Root of Trust
SCAs Side Channel Attacks
SoC System on a Chip

List of Figures

List of Tables

Bibliography

[1] *OpenTitan*. URL: https://github.com/lowRISC/opentitan (visited on 01/27/2023).

[2] *End-to-End On-Chip Encryption*. https://github.com/TUM-DSE/end2end-encr.

[3] *Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030*.

[4] D. Boneh, R. A. DeMillo, and R. J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults." In: *Advances in Cryptology — EUROCRYPT '97*. Ed. by W. Fumy. Springer Berlin Heidelberg, 1997, pp. 37–51.

[5] E. Biham and A. Shamir. "Differential fault analysis of secret key cryptosystems." In: *Advances in Cryptology — CRYPTO '97*. Ed. by B. S. Kaliski. Springer Berlin Heidelberg, 1997, pp. 513–525.

[6] S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks: Revealing the secrets of smart cards*. Vol. 31. Springer Science & Business Media, 2008.

[7] P. C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems." In: *Advances in Cryptology — CRYPTO '96*. Ed. by N. Koblitz. Springer Berlin Heidelberg, 1996, pp. 104–113.

[8] P. Kocher, J. Jaffe, and B. Jun. "Differential Power Analysis." In: *Advances in Cryptology — CRYPTO' 99*. Ed. by M. Wiener. Springer Berlin Heidelberg, 1999, pp. 388–397.

[9] *OpenTitan Earl Grey Chip Datasheet*. URL: https://opentitan.org/book/hw/top_earlgrey/doc/specification.html (visited on 03/08/2023).

[10] *Ibex RISC-V Core*. URL: https://github.com/lowRISC/ibex (visited on 02/02/2023).

[11] *SiFive TileLink Specification*. Version 1.9.3. SiFive, Feb. 9, 2023. 95 pp.

[12] M. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. Bassham, E. Roback, and J. Dray. *Advanced Encryption Standard (AES)*. en. Nov. 2001. DOI: https://doi.org/10.6028/NIST.FIPS.197.

[13] H. Gross, S. Mangard, and T. Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order." In: *Proceedings of the 2016 ACM Workshop on Theory of Implementation Security*. TIS '16. Association for Computing Machinery, 2016. DOI: 10.1145/2996366.2996426.

[14] *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. Version 20191213. Dec. 13, 2019. 238 pp.

[15] *The RISC-V Instruction Set Manual Volume II: Privileged ISA*. Version 20211203. Dec. 3, 2021. 155 pp.

[16] *Ibex Documentation: Pipeline Details*. URL: `https://ibex-core.readthedocs.io/en/latest/03_reference/pipeline_details.html` (visited on 02/26/2023).

[17] *Verilator*. `https://www.veripool.org/verilator/`.

[18] *Comportability Definition and Specification*. URL: `https://opentitan.org/book/doc/contributing/hw/comportability/index.html` (visited on 03/10/2023).

[19] *FPGA Setup*. URL: `https://docs.opentitan.org/doc/getting_started/setup_fpga/` (visited on 03/03/2023).

[20] *Bazel*. `https://bazel.build/`.

[21] *Device Interface Functions (DIFs)*. URL: `https://opentitan.org/book/doc/contributing/sw/device_interface_functions.html` (visited on 03/10/2023).