LOW LATENCY TRANSPARENT MEMORY ENCRYPTION ENGINE BASED ON LIGHTWEIGHT CRYPTOGRAPHY FOR IOT DEVICES.

by

Jesús Isaac Soriano Pineda

Ing., UNAM

A Dissertation
Submitted to the Program in Computer Science,
Computer Science Department
In partial fulfillment of the requirements for the degree of

MASTER IN COMPUTER SCIENCE

at the

National Institute for Astrophysics, Optics and Electronics
January 2024
Tonantzintla, Puebla

Advisor:

Dr. René Armando Cumplido Parra
Principal Researcher Scientist
Computer Science Department
INAOE

Thesis advisor: PhD. Rene Cumplido Parra

Jesus Isaac Soriano Pineda

Low Latency Transparent Memory Encryption Engine Based on Lightweight Cryptography for IoT Devices

ABSTRACT

This thesis presents the development of a low-latency, transparent memory encryption engine utilizing lightweight cryptography for IoT devices. The research addresses the increasing vulnerability of IoT devices to physical memory attacks, such as cold boot and snooping attacks, which threaten the security of sensitive information. Focusing on the lightweight ASCON algorithm, the project entails a thorough analysis of lightweight cryptography algorithms, hardware implementation considerations, and a review of existing memory encryption frameworks. The design and implementation are carried out using Hardware Description Languages on a Zibo z7 SoC, with subsequent behavioral simulation in Vivado 2022.2. The thesis evaluates the system's performance and IoT-specific applications, employing tools like Petalinux and Tinymembench for Full Memory Encryption (FME) testing. The analysis of data collected across all phases demonstrates the efficiency of the proposed solution in enhancing IoT device security against physical memory attacks, while maintaining low latency and transparency, marking a significant contribution to the field of lightweight cryptography in IoT security.

Acknowledgements

I would like to extend my heartfelt gratitude to Dr. Rene Cumplido Parra, whose guidance and wisdom have been the north star of my research journey. Dr. Cumplido Parra has not only been a beacon of knowledge but also an incredible mentor who has shown immense patience and dedication to my development as a scholar. His consistent encouragement and belief in my abilities have been pivotal in my ability to meet and surmount the challenges of this research, ensuring its timely completion.

For the countless hours spent reviewing my work, offering critical feedback, and pushing me to refine my ideas, I am eternally grateful. His support has been a cornerstone of my academic and personal growth, and it is with great respect and appreciation that I acknowledge his role in my achievements.

I would also like to express my sincere appreciation to all the members of the Computer Science Department. Each researcher has contributed to an enriching environment of learning and curiosity, sharing their invaluable expertise and igniting the spark of inspiration within me.

My time at the National Institute of Astrophysics, Optics, and Electronics (INAOE) has been instrumental in providing me with the tools and opportunities to pursue my research. The institution's commitment to fostering a culture of excellence is something I have been fortunate to be a part of.

I am also immensely grateful to CONACyT for the scholarship that afforded me the opportunity to devote myself to my research without financial burdens. This support has been a cornerstone of my success and for that, I am eternally indebted.

Special thanks are due to my esteemed reviewers, Dr. Alejandro Gutiérrez Giles, Dr. Alejandro Medina Santiago, and Dr. Gustavo Rodríguez Gómez. Their valuable comments and constructive critiques have significantly contributed to the enhancement of my thesis. Their willingness to share their expertise and their meticulous attention to detail have been crucial in refining my work.

To everyone who has offered their guidance, support, and belief in my potential, I extend my heartfelt thanks. Your collective wisdom has been the bedrock upon which this thesis was constructed.

Dedication

To my beloved family, for their unwavering faith in my dreams and their unconditional love and support. To my mother, Estela, and my father, Isaac, whose love and guidance light my way, this thesis is a testament to your belief in me.

To my sisters Ivonne and Iveett, pillars of inspiration, encouragement and support. To Iveett, not only for your emotional support but also for your generous investment in my dreams. Your contribution was not just financial but also a symbol of faith in my potential. To you, I owe a debt of gratitude that I am delighted to acknowledge.

And to my fiancée, my Liz, my companion and confidante, who has walked with me every step of this journey. Your love and unwavering support have made all the difference. As we stand on the brink of a new chapter in our lives, I dedicate this milestone to you and the future that awaits us.

Contents

Chapter 1

Introduction

In recent years, the proliferation of Internet of Things (IoT) devices has created a pressing need for memory encryption engines that can secure data transmission and storage in constrained devices. However, traditional software-based encryption solutions can result in significant performance overhead, particularly for devices with limited processing power and memory resources. This situation has led to an increasing demand for hardware-based encryption solutions that can provide high levels of security while minimizing latency and resource consumption. In this thesis, we present a low-latency memory encryption engine that provides transparent encryption for constrained devices, addressing the performance and resource challenges of existing encryption solutions.

1.1 Motivation

The increasing prevalence of physical memory attacks such as cold boot attacks, snooping attacks, and tampering attacks pose a significant threat to the security of Internet of Things (IoT) devices. These attacks can compromise sensitive information, such as cryptographic keys, leading to serious security breaches. Existing memory encryption and protection solutions often come at the cost of significant resource consumption and latency, making them unsuitable for resource-constrained IoT devices.

This project proposes the development of a memory encryption engine based on the lightweight ASCON algorithm to address the challenges mentioned before. ASCON is a promising candidate due to its superior performance and low resource consumption, making it a suitable choice for IoT devices. Moreover, ASCON is set to become a NIST standard, further validating its potential as a robust and secure encryption solution. By developing a memory encryption engine based on ASCON, this project aims to provide an efficient and effective solution for protecting IoT devices against physical memory attacks, ensuring the confidentiality and integrity of sensitive information.

1.2 Problem Statement

The widespread adoption of IoT devices has increased the risk of physical memory attacks, such as cold boot attacks, snooping attacks, and tampering attacks. Therefore, there is a need for an efficient and secure memory encryption engine that can protect the confidentiality, integrity, and freshness of CPU-DRAM traffic over some memory range while consuming fewer resources and outperforming existing implementations in terms of latency. This project aims to develop a memory encryption engine based on the lightweight ASCON algorithm, suitable for deployment on resource-constrained IoT devices. It provides protection against physical memory attacks and evaluates its performance using hardware and software-based testing.

1.3 Objectives

1.3.1 General

- Design a lightweight, low-latency memory encryption engine suitable for deployment on constrained devices based on lightweight cryptography.

1.3.2 Specific

- Identify within lightweight cryptography algorithms the best candidates to be implemented as a primitive for a low-latency memory encryption engine.

- Explore the design space for memory encryption engines.
- Design and implement a memory encryption engine based on lightweight cryptography algorithms on a SoC.
- Design and implement test scenarios with a prototype SoC with the memory encryption engine.

1.4 Methodology

This chapter outlines the methodology used in the research and development of a memory encryption engine leveraging lightweight cryptography algorithms. The approach is systematic, covering algorithm analysis, hardware implementation, existing frameworks, design and development, simulation, testing, application, and data analysis.

Algorithm Analysis

- **Objective:** In-depth analysis of lightweight cryptography algorithms for IoT security.
- **Approach:**
  - *Research Phase:* Extensive review of algorithms with a focus on ASCON, Skinny, Prince, and AES.
  - *Evaluation Metrics:* Analysis based on latency, resource efficiency, and standardization.

Hardware Implementation

- **Objective:** Explore practical hardware implementations for selected algorithms.
- **Approach:**
  - *Hardware Selection:* Investigation of low latency hardware platforms.
  - *Feasibility Study:* Assessment of ASCON Fast Core and Skinny versions.

Existing Frameworks and Schemes Analysis

- **Objective:** Review of current memory encryption engines and schemes.
- **Approach:**
  - *Comparative Analysis:* Evaluation of frameworks like MEMSEC and MemEnc.

Memory Encryption Engine Design

- **Objective:** Design a memory encryption engine for IoT applications.
- **Approach:**
  - *Design Principles:* Focus on scalable and efficient design integration.

HDL Development

- **Objective:** Develop the engine using HDL (Verilog and VHDL).
- **Approach:** Precision and efficiency in development using HDL.

Behavioral Simulation

- **Objective:** Validate and debug the design through simulation.
- **Approach:** Use of Vivado 2022.2 for simulation and debugging.

Synthesis, Implementation, and Testing

- **Objective:** Implement and test the design on Zibo z7 SoC.
- **Approach:**
  - *Testing Metrics:* Data collection on timing, frequency, power, and FPGA utilization.

Running Applications

- **Objective:** Test system functionality with write and read applications.
- **Approach:** Application testing using Xilinx libraries.

Full Memory Encryption Testing

- **Objective:** Evaluate Full Memory Encryption capabilities.
- **Approach:** Use of Petalinux and Tinymembench for performance assessment.

Data Analysis

- **Objective:** Analyze data to evaluate system performance.
- **Approach:**
  - *Performance Review:* Analysis of data from all project phases and formulation of future recommendations.

1.5 Main Contributions

This thesis introduces a novel memory encryption engine using lightweight cryptography, tailored for IoT devices, focusing on addressing their susceptibility to physical memory attacks. Key contributions include:

1. In-depth analysis of lightweight cryptography algorithms, culminating in the selection of the ASCON algorithm.
2. Development and hardware implementation of the encryption engine on Zibo z7 SoC.
3. Extensive behavioral simulations and testing, including application and Full Memory Encryption (FME) testing with tools like Petalinux and Tinymembench.

4. Comprehensive performance evaluation under various conditions, demonstrating enhanced security for IoT devices.

These contributions are significant to the field of lightweight cryptography in IoT security, offering an efficient solution for protecting sensitive data against physical attacks while maintaining low latency.

1.6 Document Organization

The rest of the document is organized in the following way: Chapter 2 is the background of the project, where we provide context to the reader and establish the rationale for the research topic; we talk about the memory threats in IoT devices and the challenges to implementing countermeasures against memory threats in IoT devices. We introduce memory encryption as a countermeasure to different memory threats and hardware memory encryption engine MEE as a feasible option to give protection to constrained devices against physical memory attacks. We also talk about some term that will help the reader to understand the platform that will be used to design and implement the proposal. Chapter 3 shows the most recent implementations of memory encryption engine highlighting works like the MEE implemented in Intel SGX, the open source framework MEMSEC and the lightweight scheme MemSec, mentioning the main features of each work. Chapter 4 details the design challenges, optimizations, and architecture of the proposed system, the in Chapter 5 we decribe the platform, hardware implementation, simulations, resource utilization, and analysis of the results, and finally in Chapter 6 are summarized the findings and suggests areas for future research.

Chapter 2

Background

2.1 IoT

IoT, or the Internet of Things, refers to the network of physical objects connected to the Internet that can collect and exchange data. These objects can be anything from intelligent thermostats to self-driving cars. IoT can revolutionize many industries, including healthcare, manufacturing, and transportation.

There are many benefits to using IoT. For example, IoT can help businesses to improve efficiency and productivity, reduce costs, and improve customer service. IoT can also help to improve safety and security. For example, smart sensors can be used to monitor equipment in factories and warehouses, and self-driving cars can help to reduce traffic accidents [1].

IoT is still in its early stages but can change the world. As more and more devices become connected to the internet, we can expect to see even more innovative and groundbreaking applications of IoT in the years to come.

Here are some examples of how IoT is being used today[1]:

- Smart homes: Smart homes use IoT devices to control lights, thermostats, and security systems. These devices can make homes more comfortable, energy-efficient, and secure.

- Smart cities: Smart cities use IoT devices to collect data about traffic, public transportation, and other infrastructure. This data can be used to improve the efficiency of city services and make cities more livable.
- Smart agriculture: Smart agriculture uses IoT devices to monitor crops, livestock, and other agricultural assets. This data can improve crop yields, reduce water usage, and prevent pests and diseases.
- Smart manufacturing: Smart manufacturing uses IoT devices to monitor and control manufacturing processes. These devices can help to improve quality, reduce costs, and increase productivity.
- Smart logistics: Smart logistics use IoT devices to track and manage the movement of goods. These devices can help to improve efficiency and reduce costs.

These are just a few examples of how IoT is being used today. We expect to see even more innovative and groundbreaking applications as IoT technology develops.

2.1.1 IoT environment features

IoT devices are physical objects connected to the internet or other networks, allowing them to communicate with other devices and services. They are typically small, low-power devices designed to be deployed in large numbers in various environments and use cases.

Some common features of IoT devices include[2]:

- Sensors: Many IoT devices are equipped with sensors to collect data about their environment or operation. For example, an intelligent thermostat might have temperature and humidity sensors to measure the conditions in a room. In contrast, a wearable fitness tracker might have sensors to measure heart rate, steps taken, and other biometric data.

- Connectivity: IoT devices are designed to be connected to the internet or other networks, allowing them to communicate with other devices and services. This connectivity can be wired or wireless, depending on the device. Standard wireless technologies used in IoT devices include Wi-Fi, Bluetooth, and cellular networks.
- Remote control: IoT devices can often be controlled remotely using a smartphone app or web interface. This remote control allows users to monitor and adjust the device's operation from anywhere with an internet connection. For example, a smart home security camera can be viewed and controlled from a smartphone app.
- Automation: IoT devices can be programmed to perform specific actions automatically based on pre-defined rules or sensor inputs. For example, an intelligent irrigation system might automatically water plants when the soil moisture level falls below a certain threshold.
- Data storage and analysis: IoT devices can collect and store large amounts of data, which can be analyzed to gain insights into their operation or the deployed environment. This data can also improve the device's performance or inform other systems or services.
- Machine learning: Some IoT devices incorporate machine learning algorithms that allow them to learn from the data they collect and make more accurate predictions or recommendations over time. For example, a smart home thermostat might learn the user's schedule and adjust the temperature automatically to save energy.
- Security: IoT devices are often designed to be secure, with encryption, secure boot, and secure firmware updates to protect against attacks. However, security is still a significant concern in the IoT industry, and many devices are vulnerable to hacking or other attacks.

IoT devices are designed to be flexible and adaptable, with features and capabilities that can be customized to suit a wide range of applications and use cases. As the IoT industry grows, we expect to see even more innovative and powerful devices with new and exciting features.

2.2 Memory Vulnerabilities

A memory vulnerability in a computer system refers to a flaw or weakness in the system's memory management that an attacker can exploit to compromise the system's security. Various factors, including programming errors, design flaws, or misconfigurations, can cause memory vulnerabilities.

Memory vulnerabilities can be difficult to detect and exploit but can be very dangerous when successfully exploited. Attackers can use them to steal sensitive data, gain access to privileged information or execute unauthorized code. Therefore, system designers, software developers, and security professionals must be aware of memory vulnerabilities and take steps to prevent them [3].

If we try to classify those vulnerabilities, we can find two big groups, physical or non-physical vulnerabilities, depending on whether or not they can be exploited with physical access to the system.

Physical memory vulnerabilities are those that require physical access to the system's memory in order to exploit them. These vulnerabilities typically involve direct hardware manipulation, such as probing memory chips, intercepting signals between the processor and memory, or modifying memory contents using external devices. Physical memory vulnerabilities include cold boot attacks, row hammer attacks, and glitching attacks.

On the other hand, non-physical memory vulnerabilities do not require physical access to the system to exploit them. Instead, these vulnerabilities are typically exploited by sending malicious input to the system, such as specially crafted data or network packets. Non-physical memory vulnerabilities include buffer overflows, use-after-free vulnerabilities, and format string vulnerabilities.

It is important to note that while physical memory vulnerabilities are generally more difficult to exploit than non-physical vulnerabilities in typical or traditional computer systems due to the difficulty of having physical access to the equipment, in IoT devices, this limitation may disappear and make physical attacks more feasible becoming a severe threat to system security. In some cases, physical access to a system's memory may be obtained through social engineering, theft, or other means, making it essential for system designers and security professionals to consider physical security when designing and securing computer systems.

2.2.1 Physical Attacks to Memory

Physical attacks on memory RAM in computer systems are relatively rare, but they are a known security risk exploited by some attackers in certain circumstances. These attacks are generally more sophisticated and require a higher level of expertise than software-based attacks and are often carried out by skilled and determined attackers with access to the physical hardware.

Although these attacks are relatively rare, they can be challenging to detect and defend. Therefore, computer systems must employ appropriate security measures to protect against physical attacks on memory RAM, in addition to other types of attacks.

While physical memory vulnerabilities may be more challenging to exploit on more extensive and complex systems, they can still pose a significant threat to smaller devices if they are not adequately secured. IoT and mobile device manufacturers must implement physical solid security measures, such as tamper-resistant packaging, secure boot processes, and hardware-based encryption, to mitigate the risk of physical memory vulnerabilities.

Cold Boot Attack

One example of a physical attack on RAM is the ”cold boot attack,” in which an attacker quickly cools down the computer’s RAM after turning off the power to preserve the data stored in memory. The attacker can then access the memory contents and potentially retrieve sensitive information such as encryption keys or passwords.

The cold boot attack was described by Halderman et al. in [4], is a type of physical memory attack that exploits the data remanence property of DRAM (Dynamic Random Access Memory). When a computer is turned off, the contents of its memory are expected to be erased. However, the data stored in DRAM chips may persist for a short period, even after removing power. In this short period, an attacker can retrieve sensitive information, such as encryption keys or login credentials, from memory even after a system has been shut down.

A cold boot attack typically involves cooling the DRAM chips to preserve the data for a more extended period. The DRAM can be cooled through a cooling spray or even by freezing the memory chips. Once the memory has been cooled, it is quickly removed from the original system and inserted into a different system, which is used to read and analyze the contents of the memory. The attacker can then use various techniques to recover the encryption keys or other sensitive information that may have been stored in memory.

Cold boot attacks can be particularly effective against systems that use software-based full-disk encryption, as the encryption keys are stored in unencrypted memory during system operation [4]. However, cold boot attacks can also recover sensitive information, such as passwords or other authentication credentials.

Various techniques can be employed to protect against cold boot attacks, such as overwriting memory with random data before shutting down the system or encrypting the contents of memory while the system is in operation. Hardware-based memory encryption, such as Intel's Memory Encryption Engine [5], can protect against cold boot attacks. Existing physical protection measures, like blocking access to the system memory or preventing the removal of memory chips, can also help mitigate the risk of cold boot attacks [4].

Talking in the context of IoT devices is relatively easy for attackers with physical access to them, so it is possible to perform cold boot attacks to extract encryption keys or other sensitive information from an IoT device's memory.

The danger of a cold boot attack on an IoT device depends on the type of information stored in memory. For example, if the device contains encryption keys, the attacker could use those keys to decrypt and steal sensitive data. If the device is part of a more extensive system, the attacker could use the compromised device as a gateway to access other parts of the system.

Furthermore, if the device is a critical infrastructure component, such as an intelligent grid controller, a cold boot attack could have severe consequences, including service disruption, data theft, or physical harm. Therefore, protecting IoT devices against cold boot attacks is essential for ensuring the security and reliability of the entire system.

Bus-Snooping Attack

Bus-snooping attacks, also known as bus sniffing or bus monitoring attacks, are hardware-based attacks involving eavesdropping on the communication between devices on a shared bus. In computer architecture, a bus is a communication system that transfers data between different computer components, such as the CPU, memory, and input/output devices.

The basic idea behind a bus-snooping attack is that an attacker intercepts data transmitted on the bus and then analyzes the data to obtain sensitive information. This attack can be done by connecting a device to the bus, such as a logic analyzer or a specialized hardware device, to monitor the signals and record the transmitted data.

One of the most common bus-snooping attacks is a memory bus-snooping attack, which targets the communication between the CPU and the system memory. In this type of attack, as described in [6], the attacker intercepts the signals transmitted on the memory bus and then extracts the data from the memory. This kind of attack can be hazardous, as the data in memory can contain sensitive information such as passwords, encryption keys, and other confidential data [7].

Bus-snooping attacks can also intercept communication between other components on a computer's bus, such as input/output devices and the CPU. With this attack, someone can steal data transmitted to or from the computer, such as credit card numbers, login credentials, and other sensitive information.

In order to prevent bus-snooping attacks, it is necessary to implement countermeasures such as encryption, which can protect sensitive data even if it is intercepted [8]. In addition, physical security measures such as shielding the bus or physically isolating the components can help prevent attacks. It is also essential to ensure that any devices connected to the bus are trusted and secure to prevent attackers from gaining access to the system.

Memory Tampering

Memory tampering attacks refer to a class of security threats where an attacker attempts to modify the contents of memory in an unauthorized way to gain access to sensitive information, modify program behavior, or exploit a vulnerability in a system. Such attacks are generally carried out by exploiting software vulnerability, such as a buffer overflow, a race condition, or a logic error, to gain control over the execution of a program or the operation of a system.

Once the attacker has gained control, they can modify the contents of the memory to allow them to achieve their goals. For example, an attacker may attempt to overwrite the memory used to store sensitive data such as passwords, encryption keys, or other confidential information. Alternatively, they may modify the behavior of a program in order to bypass security checks or gain elevated privileges.

Memory tampering attacks can be carried out using various techniques, including injection of malicious code, debugging tools, manipulation of system calls, and exploitation of memory vulnerabilities. In some cases, the attacker may use side-channel attacks to obtain information about the contents of memory, which can then be used to plan and execute more sophisticated attacks[9].

One of the main challenges in defending against memory tampering attacks is that the attacker has complete control over the contents of memory and can modify them in any way they see fit[9]. This condition makes it difficult to detect and prevent such attacks, as there is no reliable way to distinguish between legitimate and malicious changes to memory.

Several defensive techniques can be used to mitigate the risk of memory tampering attacks. These include code signing and verification techniques, data encryption, and memory access control. Additionally, many modern operating systems and hardware platforms include various security features, such as memory randomization, address space layout randomization, and sandboxing, which can help to prevent memory tampering attacks and other types of security threats.

Side-Channel Attack

A side-channel attack is a type of security exploit that takes advantage of vulnerabilities in a system's physical or implementation characteristics rather than its software or hardware components. These attacks can extract sensitive information from a system, such as cryptographic keys or passwords, by analyzing patterns in the power consumption, electromagnetic emissions, or timing of the system's operation.

Side-channel attacks can be classified into several categories: power analysis, electromagnetic analysis, and timing attacks. In a power analysis attack, an attacker monitors the power consumption of a device during cryptographic operations and uses this information to determine the cryptographic key. Similarly, in an electromagnetic analysis attack, an attacker monitors the electromagnetic emissions of a device and uses this information to extract the key [10].

Timing attacks involve analyzing the timing behavior of a system, such as the time it takes to perform certain cryptographic operations, and using this information to deduce the key [10]. Other side-channel attacks include acoustic attacks, which involve analyzing sound emitted by a device during operation, and cache-based attacks, which exploit the behavior of a system's cache memory [10].

Side-channel attacks can be difficult to defend against, as they exploit vulnerabilities inherent in a system's design or in the physical environment in which it operates. However, several countermeasures can be employed to reduce the risk of side-channel attacks, including masking techniques that obscure a system's power consumption or timing behavior and differential power analysis (DPA) resistance techniques that reduce the correlation between the key and the power consumption.

2.2.2 Constrains and challenges in the memory safety for IoT

The general population widely adopts Internet of Things devices. People today are more connected than ever before. The widespread use and low-cost-driven construction of these devices in a competitive marketplace render Internet-connected devices a more accessible and attractive target for malicious actors [11].

Protecting IoT devices against attacks, especially talking physical memory attacks, can be challenging for a few reasons [12]:

- Limited resources: Many IoT devices are designed to be low-cost and low-power, which means they need more resources for security measures. These features can make implementing robust security features such as encryption or secure boot challenging.
- Remote deployment: IoT devices are often deployed in remote or hard-to-reach locations, making it difficult to protect them from physical attacks. For example, an intelligent water meter located on the side of a building may be vulnerable to physical attacks such as tampering or theft.
- Lack of standardization: The IoT industry is still relatively new and needs more standardization so that security measures can vary widely between devices and manufacturers. These variations can make it challenging to ensure that all IoT devices are adequately protected against physical memory attacks.
- Diverse hardware: IoT devices can use various hardware, including microcontrollers, microprocessors, and field-programmable gate arrays (FPGAs). Each of these hardware platforms has unique security challenges, making it difficult to implement a standardized approach to physical memory security.
- Complexity: IoT devices often have complex software stacks, with multiple layers of software running on different hardware platforms. This heterogeneity can make

it difficult to implement security measures that cover all layers of the software stack and hardware platforms.

Protecting IoT devices against physical memory attacks requires a combination of hardware and software security measures and a strong focus on standardization and best practices in the industry. As IoT devices continue to proliferate and become more critical to our daily lives, ensuring they are adequately protected against all types of security threats will be increasingly important.

2.3 Memory Encryption as a Countermeasure

Memory encryption is a countermeasure that can be used to protect against memory-related security threats. Memory encryption involves encrypting the contents of a computer's memory so that unauthorized parties cannot read or modify it.

There are two main types of memory encryption as showed in Figure 2.1:

- Full memory encryption: Full memory encryption involves encrypting the entire contents of a computer's memory. This approach provides the most substantial level of security, but it can be resource-intensive and potentially introduce performance overhead.
- Partial memory encryption: Partial memory encryption involves encrypting only specific parts of a computer's memory, such as sensitive data or code. This approach can provide some level of security while minimizing performance overhead, but it may be less effective against certain types of attacks.

Memory encryption can be used to address several physical memory threats in IoT devices, including:

- Tampering: Memory encryption can protect against tampering attacks, which involve physically modifying the device's memory to bypass security measures or

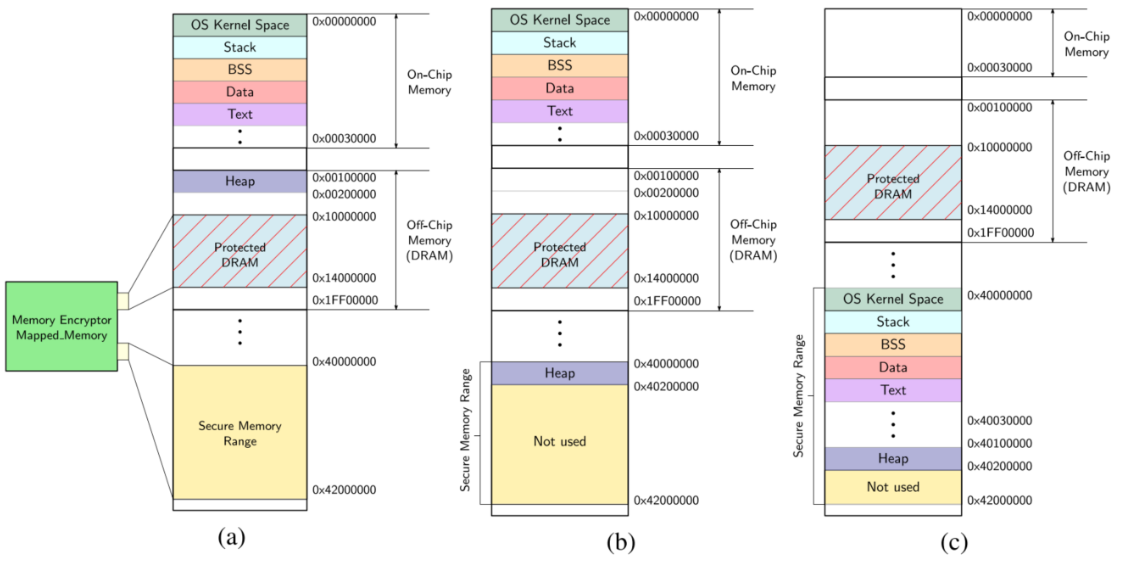


Figure 2.1: Memory map. (a) Unprotected (protected DRAM is not being used). (b) PME. (c) FME [13].

extract sensitive data. By encrypting the memory, even if an attacker can access it, the data will be unreadable without the proper decryption key.

- Side-channel attacks: Side-channel attacks involve measuring the device's physical characteristics, such as power consumption or electromagnetic emissions, to extract information from the memory. Memory encryption can help protect against these attacks by making it more difficult for an attacker to extract helpful information from the memory.

- Physical memory dumping: Physical memory dumping involves using specialized equipment to read the contents of the device's memory. Memory encryption can help protect against this type of attack by ensuring that the data in the memory is unreadable without the proper decryption key.

- Data theft: Memory encryption can also help protect against data theft if an attacker can physically steal the device or its memory. By encrypting the data, the attacker can only read it with the proper decryption key.

Memory encryption can help protect against these attacks by making it more difficult for an attacker to extract helpful information from memory, ensuring the data is unreadable without the proper decryption key.

Memory encryption can be implemented at various levels of the computing stack, including the hardware level, the operating system level, or the application level. Hardware-based memory encryption is generally considered the most secure approach, providing the most substantial isolation between the encrypted memory and other system components. It can also be used with other security measures, such as access controls and intrusion detection systems, to provide a multi-layered defense against memory-related security threats.

2.3.1 Different approaches for memory encryption: software, and hardware.

The standard threat model in the memory encryption literature involves attacks on hardware and software. It is assumed in many scenarios that the attackers have physical access to the system, which is vulnerable to various methods of capturing sensitive information. The attackers' primary objective is to steal confidential information or code, and memory modification is occasionally discussed as a tactic to force a system to reveal confidential information. These attackers may have diverse motivations, ranging from financial gain (such as bank employees capturing ATM PINs or criminals reproducing and disseminating software with digital rights management) to attempts to reverse engineering or extract intellectual property [14].

As we mentioned before, memory attacks could be performed by the physical intervention of the device or by software, so also memory encryption can be implemented in different approaches. The literature on memory encryption is concerned mainly with three core approaches based on 1) hardware enhancements, 2) operating system enhancements, and 3) specialized industrial applications [14]. The characteristics of these three approaches are mentioned below, as well as their main advantages and disadvantages of each.

Hardware-based memory encryption

Hardware enhancements can include several approaches, such as adding specialized encryption units and fundamental storage mechanisms to existing processor designs.

Researchers are exploring using specialized encryption hardware outside the CPU to enhance memory security. This hardware can be placed in the memory bus, between the system memory and CPU, or within the RAM. The main objective of this approach is to increase the likelihood of its adoption without requiring significant modifications to commodity processors.

Encryption and decryption are handled by a dedicated hardware component, such as Intel's Memory Encryption Engine (MEE) or AMD's Secure Memory Encryption (SME). Hardware-based memory encryption provides the highest level of security, as the encryption is performed independently of the CPU and memory and is resistant to attacks against the operating system or applications. However, it requires specialized hardware support, which may limit its adoption.

Advantages:

- It provides the most substantial level of security, as it is isolated from other system components.
- It can be challenging for attackers to bypass or circumvent.
- Generally incurs less performance overhead than other approaches.

Disadvantages:

- It may require specialized hardware, which can be expensive.
- It may be more challenging to implement than other approaches.

- It may require significant changes to existing systems or software.

Operating system-based memory encryption

Like the bus insert technique used to enable memory encryption, software-based methods aim to offer solutions that require minimal modifications to existing applications and hardware, making them more likely to be widely adopted.

This approach involves integrating memory encryption functionality into the operating system. The operating system manages the encryption and decryption of memory contents, including allocating keys and managing encrypted memory pages. This method provides high security and flexibility, as the OS can manage encryption for all applications running on the system. However, it can introduce some overhead due to the additional processing the operating system requires.

Advantages:

- It can provide a high level of security without requiring specialized hardware.
- It may be easier to implement than hardware-based approaches.
- It can be used with existing software and systems.

Disadvantages:

- It may be less secure than hardware-based approaches, as it is separate from other system components.
- May introduce performance overhead.
- They may be vulnerable to certain types of attacks, such as those targeting the operating system.

Application-based memory encryption

This approach involves implementing memory encryption functionality directly within the application. The application developer is responsible for managing the encryption and decryption of memory contents. This method provides the highest level of flexibility, as the application can selectively encrypt sensitive data and optimize encryption based on its specific requirements. However, it requires significant development effort and may only be feasible for some applications.

Advantages:

- It can be tailored to the specific needs of individual applications.
- It may perform better than other approaches, which can be optimized for specific use cases.
- It can be used with existing software and systems.

Disadvantages:

- It may be less secure than other approaches, as it is not implemented at a lower level of the computing stack.
- It may be more challenging to implement and maintain than other approaches.
- It may require significant changes to existing software or systems.

The choice of which level to implement memory encryption depends on various factors, such as the level of security required, the performance overhead that can be tolerated, and the resources available for implementation. A multi-layered approach that uses memory encryption at multiple levels can provide the most substantial security level while minimizing performance overhead and other disadvantages.

2.4 Memory Encryption Engine

A Memory Encryption Engine (MEE) is described in [5] as “as an autonomous hardware unit in charge of protecting the confidentiality, integrity, and freshness of the CPU-

DRAM traffic over a defined memory range." In other works like [15] or [13] is described as a technology that is designed to provide strong protection against memory-based attacks. It works by encrypting all memory contents, including system memory, cache memory, and memory used by peripherals, using a dedicated encryption engine integrated into the processor.

The design of an MEE depends on the security services that want to be provided to the entire system; for example, a basic MEE like the one showed in [13] performs encryption and only provides confidentiality, unlike the designs presented in [15] or [5] where the MEE also has an integrity tree that provides integrity and authentication.

The MEE generally works as follows: it encrypts all memory contents on the fly as they are written to memory and decrypts them as they are read back from memory. In some MEE designs where integrity is added, the MEE generates this integrity tree and saves the root. Adding an MEE ensures that sensitive data is protected against unauthorized access, even if an attacker gains physical access to the system or intercepts data as it is transferred over a bus or network.

A vital point of implementing a memory encryption engine is that this unit's work should be transparent to the rest of the system. It means that no extra instruction from the operative system or the application will be required to encrypt and store the data; the MEE must work with the memory controller (MC) to handle any transaction between the CPU and the DRAM. Figure 2.2 is a diagram of an implementation of an MEE in a SOC; as we can see, the MEE pipeline is placed in the programmable logic part (PL), and the CPU and memory controller is placed in the processor system (PS) part. The DDR memory is out of what is considered the secure boundary; Figure 2.2 tries to show that any transaction between the CPU and memory controller must be handled or passed through the memory encryption pipeline.

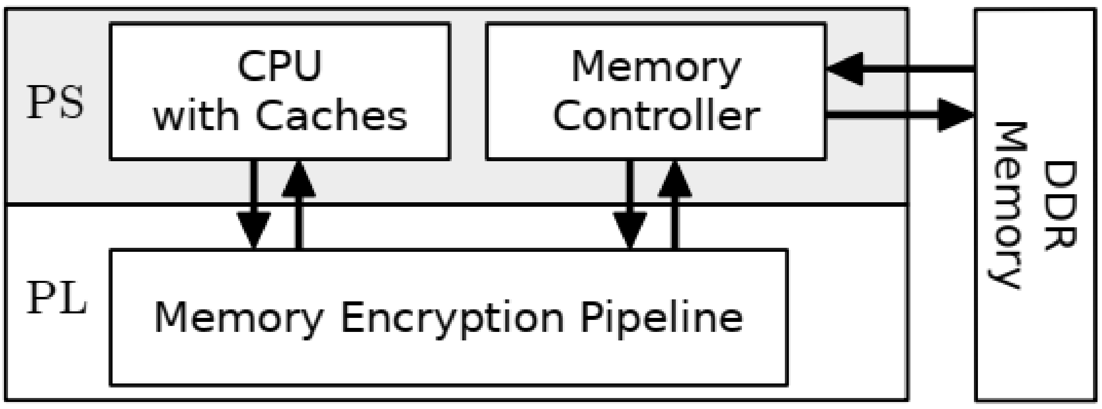


Figure 2.2: Diagram of a Memory Encryption Engine implemented in [15].

There are listed some of the key benefits of using a Memory Encryption Engine:

- Strong security: The use of hardware-based encryption provides more robust security than software-based encryption since it is much harder for attackers to bypass or exploit the encryption.
- Minimal performance overhead: The encryption and decryption operations are offloaded to the hardware, which reduces the impact on system performance. This offload is especially important in high-performance computing environments, where even a small performance overhead can significantly impact.
- Transparent operation: The Memory Encryption Engine is designed to operate transparently to the software running on the system; it means that software does not need to be modified to take advantage of the encryption, and there is no impact on system performance or compatibility.

However, adding an MEE unit to the architecture of a general-purpose processor could be a very challenging design process due to all the strict engineering constraints, like the careful combination of the implemented cryptographic primitive operating with a customized integrity tree that primarily resides on the DRAM while relying only on a small internally stored root [5]. In addition, the limitations of IoT devices should also be considered, such as that the MEE should occupy a small space on the chip, consume little energy and have a good performance in terms of latency.

The Memory Encryption Engine is a powerful memory encryption technology that provides strong security with minimal impact on system performance. It is available on various processors, including those from Intel and AMD, and is supported by many operating systems, including Windows and Linux.

2.5 Lightweight Cryptography

The idea of introducing this Section is to show Lightweight Cryptography primitives as cryptographic core options to be implemented in designing an MEE suitable for IoT devices due to their features of low resource occupancy, low power consumption, and low latency.

Lightweight cryptography (LWC) refers to cryptographic algorithms and protocols designed specifically for use in resource-constrained environments such as embedded systems, IoT devices, and other low-power devices [16]. These algorithms are typically designed to be computationally efficient, use minimal memory, and have low power consumption.

The need for LWC arises because traditional cryptographic algorithms such as AES or RSA were designed with desktop computers in mind, and they need to be better suited to resource-constrained environments. For example, these algorithms may require too much memory or computational power to be implemented on a low-power IoT device.

LWC algorithms typically use various techniques to reduce their computational and memory requirements. For example, some LWC algorithms use lightweight block ciphers such as PRESENT [17], which use smaller block sizes and key lengths than traditional ciphers like AES [18]. Other LWC algorithms use stream ciphers, which generate a stream of pseudorandom bits that can be XORed with the plaintext to produce ciphertext [12].

In addition to their computational and memory requirements, LWC algorithms must also be designed with security in mind. Because these algorithms are often used in critical applications such as medical devices or industrial control systems, they must resist a wide range of attacks, including side-channel attacks, fault attacks, and other forms of cryptanalysis.

Some examples of the mos import an LWC algorithms that we can find in the literature are:

- ASCON [19] is a family of authenticated encryption with associated data (AEAD) and hashing family of lightweight algorithms selected as a new lightweight cryptography standard by the National Institute of Standards and Technology (NIST) and the primary option for lightweight AEAD in the final portfolio of the CAESAR competition.
- Prince [20] a very efficient lightweight algorithm in software and hardware.
- PRESENT [17] an ISO/IEC(29192-2P:2012) approved algorithm, a block cipher based in a substitution-permutation network (SPN).

LWC is an essential area of research in cryptography because it enables secure communication and data protection in resource-constrained environments. As the number of IoT devices grows, the need for efficient and secure cryptographic algorithms will only become more critical.

2.5.1 NIST LWC Competition

The National Institute of Standards and Technology (NIST) is a non-regulatory agency of the United States Department of Commerce. It is responsible for developing standards and guidelines that promote technological innovation and industrial competitiveness.

NIST has been involved in the development of LWC. In 2018, NIST launched a competition to evaluate and standardize lightweight cryptographic algorithms that are secure and efficient for use in resource-constrained environments. The competition aimed to identify a portfolio of cryptographic algorithms suitable for low-power devices such as smart cards, wireless sensors, and other IoT devices.

The NIST received 57 submissions from researchers worldwide, and after a rigorous evaluation, NIST selected ten algorithms as finalists. These finalists were then subjected to further analysis and testing, and in February 2023, NIST announced the selection of the ASCON family as the winner of the competition. This algorithm is now considered the standard lightweight cryptographic algorithm and is recommended for use in low-power devices.

The NIST has played a vital role in developing LWC, as it has helped identify and promote cryptographic algorithms that are secure, efficient, and suitable for use in resource-constrained environments. By providing a standard set of LWC algorithms, NIST has also helped to promote interoperability and compatibility among different LWC implementations.

2.5.2 ASCON

ASCON is a family of authenticated encryption algorithms that won the NIST Lightweight Cryptography Standardization Process in February 2023 and also is the first choice of the CAESAR contest [19]. Researchers from the Graz University of Technology and the Technical University of Austria designed it.

ASCON is a high-performance, authenticated encryption algorithm suitable for many devices, including embedded systems and low-power devices. It is designed to be secure, efficient, and easy to implement. It uses a sponge construction, a type of hash function that can be used to create a block cipher. It also uses a permutation-based design, a technique used to create secure and efficient cryptographic primitives.

The ASCON suit consists of the authenticated ciphers Ascon-128 and Ascon-128a, a new variant Ascon-80pq with increased resistance against quantum key-search; additionally, the hash functions Ascon-Hash and Ascon-Hasha, and the extendable output functions Ascon-Xof and Ascon-Xofa. All schemes provide 128-bit security and internally use the same 320-bit permutation (with different round numbers) so that a single lightweight primitive is sufficient to implement both AEAD and hashing [19].

One of the critical features of ASCON is that it has a small memory footprint, making it suitable for use in resource-constrained environments. It also has a high level of security, with a conservative design resistant to a wide range of attacks. It is a high-performance, secure, and efficient authenticated encryption algorithm well-suited for use in many devices, including those with limited computational and memory resources.

Authenticated Encryption

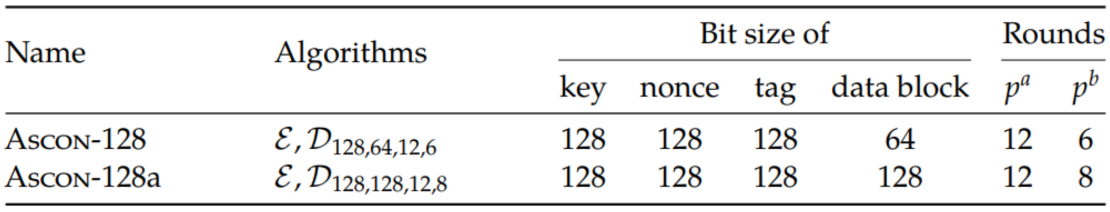
The encryption process $E_{k,r,a,c}$ receives as an input the key (K), a nonce (N), the associated data (A) and the plain text (P) and we obtain as an output the cipher text (C) and a tag (T).

$$E_{k,r,a,b}(K, N, A, P) = (C, T).$$

On the other hand, for the decryption process, we have as input the key (K), a nonce (N), the associated data (A), the cipher text (C), and the tag (T). We obtain the plain text (P) or an error $(\perp)$ depending if the tag (T) verification fails or not. The recommended parameters are shown in Figure 2.3.

$$D_{k,r,a,b}(K, N, A, C, T) \in (P, \perp).$$

The diagram of Figure 2.4 shows how the algorithm is divided into four stages, the



Name	Algorithms	Bit size of				Rounds	
		key	nonce	tag	data block	p^a	p^b
Ascon-128	$\mathcal{E}, \mathcal{D}_{128,64,12,6}$	128	128	128	64	12	6
Ascon-128a	$\mathcal{E}, \mathcal{D}_{128,128,12,8}$	128	128	128	128	12	8

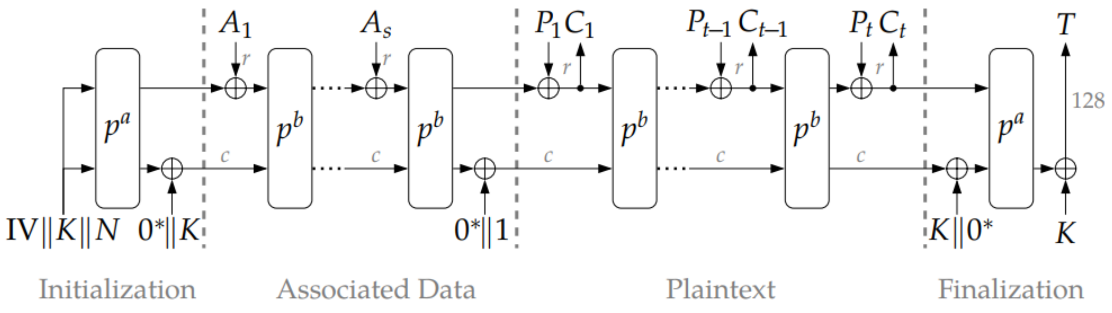
Figure 2.3: Parameters for recommended authenticated encryption schemes

initialization where we enter the initialization vector IV concatenated with the key K and the nonce N to form the state S of 320 bits, then is performed the permutation process a times over the state S and XORed with the key. In the second stage, the associated data is divided into blocks of r bits (r is the data block size), XORed with the state S, and then permuted b times between each associated data block. The third stage is where the plain text (in the case of encryption) or the cipher text (in case of decryption) is also processed in blocks of r bits and XORed like the previous stage, in each XOR operation, we obtain as an output a block of r bits of the cipher text or the plain text. The last stage is the finalization stage, where the state S is XORed with the key K, and after a permutations (like in the initialization stage) and one last XORed with the key, we obtain the tag T. We can find the detailed specification in [19].

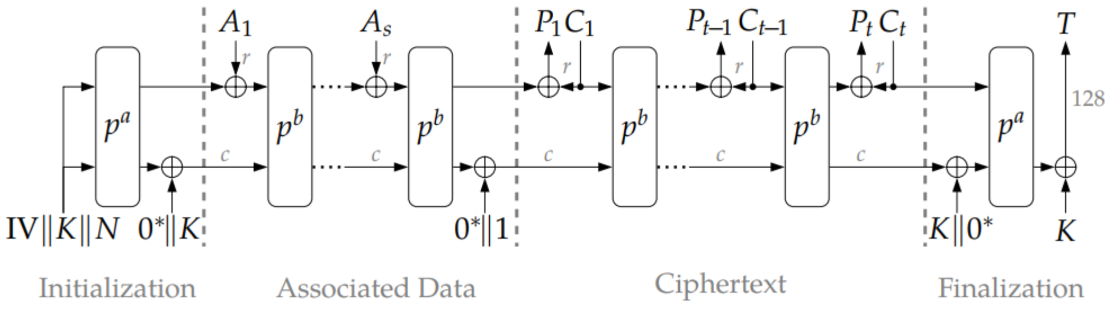
The most interesting part of the algorithm occurs in the boxes p^a and p^b that we see in Figure 2.4 that is the permutation. The permutations, in an iterative way, apply an SPN-based round transformation p over the state S that, in turn, consists of three steps: a constant addition p_C, substitution layer p_S, and diffusion layer p_L. Those steps are described in Figures 2.5 and 2.6.

2.6 FPGAs and HDL

The following Sections have been written to introduce the reader to the platform used to develop the proposal's implementation.



(a) Encryption $\mathcal{E}_{k,r,a,b}$

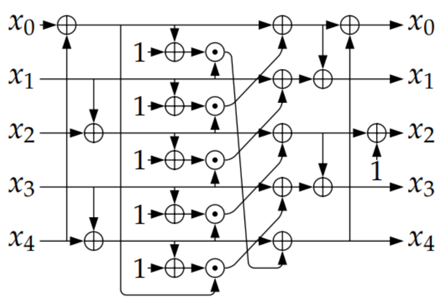


(b) Decryption $\mathcal{D}_{k,r,a,b}$

Figure 2.4: Ascon's mode of operation

p^{12}	p^8	p^6	Constant c_r	p^{12}	p^8	p^6	Constant c_r
0			00000000000000f0	6	2	0	0000000000000096
1			00000000000000e1	7	3	1	0000000000000087
2			00000000000000d2	8	4	2	0000000000000078
3			00000000000000c3	9	5	3	0000000000000069
4	0		00000000000000b4	10	6	4	000000000000005a
5	1		00000000000000a5	11	7	5	000000000000004b

Figure 2.5: The round constants c_r used in each round i of p^a and p^b.



$$
\begin{aligned}
x_0 &\leftarrow \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28)\\
x_1 &\leftarrow \Sigma_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39)\\
x_2 &\leftarrow \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6)\\
x_3 &\leftarrow \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17)\\
x_4 &\leftarrow \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)
\end{aligned}
$$

Figure 2.6: Ascon's substitution layer and linear diffusion layer.

FPGAs (Field Programmable Gate Arrays) are integrated circuits that can be programmed to perform custom logic functions, which means they can be used to implement hardware designs tailored to specific applications. FPGAs are used in various applications, including digital signal processing, image processing, and machine learning.

Hardware description languages (HDLs) are programming languages used to describe hardware designs. Several HDLs available, including VHDL and Verilog, two of the most commonly used languages for hardware design.

The process of designing hardware using FPGAs and HDLs typically involves the following steps:

- Design specification: The hardware design specification is defined, which includes the required inputs, outputs, and processing requirements.
- HDL coding: The hardware design is described using an HDL, such as VHDL or Verilog.
- Simulation: The HDL code is synthesized into a hardware design that can be implemented on an FPGA.
- Synthesis: The hardware design is implemented on an FPGA using tools like Quartus or Vivado.

- Implementation: The hardware design is implemented on an FPGA using tools like Quartus or Vivado.
- Testing: The hardware design is tested to ensure it meets the design specifications.

Hardware design using FPGAs and HDLs can be complex, but it provides excellent flexibility and customization. It allows designers to create custom hardware designs tailored to specific applications, resulting in improved performance and reduced costs. Additionally, FPGAs can be reprogrammed, which means that the same hardware can be used for multiple applications, further reducing costs and increasing flexibility.

2.6.1 SoC

SoC design, or System-on-a-Chip design, integrates complex electronic systems onto a single chip. SoC design integrates components, such as processors, memory, peripherals, and communication interfaces into a single chip. SoC design aims to create a highly integrated system that can perform complex functions with minimal power consumption and cost.

The SoC design process typically involves several stages: system design, specification, architecture design, microarchitecture design, verification, and implementation. System design involves defining the requirements and specifications of the system, while architecture design involves selecting the components and designing the interconnects between them. Microarchitecture design involves the detailed design of the individual components, such as the processor and memory subsystems. Verification ensures that the design meets the functional and performance requirements, and implementation involves physically designing and manufacturing the chip.

SoC design is a complex and challenging process requiring expertise in various disciplines, including digital and analog design, software development, and verification. SoC designers must also consider various factors, including power consumption, performance, cost, and design constraints such as size and packaging.

SoCs are used in various applications, including mobile devices, embedded systems, automotive electronics, and consumer electronics. Using SoCs has enabled the development of highly integrated and efficient systems that can perform complex functions with minimal power consumption and cost.

2.7 Platform used for implementation

A platform for designing an SoC that includes an FPGA would typically involve a combination of software and hardware tools.

The software tools would include a hardware description language (HDL), such as Verilog or VHDL, which describes the functionality of the SoC and the FPGA. The HDL is typically used with a simulation tool, allowing designers to simulate the system's behavior and test its functionality before it is implemented.

The hardware tools would include an FPGA development board, which provides a physical platform for designing and testing the FPGA component of the SoC. The development board typically includes a range of interfaces and connectors that allow designers to interface with the FPGA and the rest of the system.

In addition to these tools, designers would need access to various design resources, including IP blocks, reference designs, and design methodologies. These resources can help accelerate the design process and ensure that the resulting SoC is efficient and reliable.

The design platform for an SoC that includes an FPGA would need to provide a range of tools and resources to support the design process and a flexible and scalable architecture that can accommodate a wide range of applications and requirements.

Chapter 3

State of the Art

3.1 Memory Encryption Engines

In this Section, we mention the different implementations of hardware-based memory encryption that we can find in the state of the art. All the implementations were classified into two big groups: those that are open source and those that are closed source.

In the closed-source hardware memory encryption group, we can find proposals like the MEE of Intel SGX [5] and AMD Secure Memory Encryption [21]. Furthermore, in the group of open source implementation, we have the MEMSEC [15], which is a framework that consists of a pipeline designed to test different cryptographic primitives, the MEAS [22] that in the pipeline of MEMSEC enhanced with some features to protect against side-channel attacks, the ELM [23] a hardware architecture designed to provide memory encryption improving the latency in large amounts of memory and finally MemEnc [13] that is a memory encryption engine designed to provide memory encryption to constrained devices.

However, in this work, we are going to focus on the following works: the MEE of Intel SGX [5], MEMSEC [15], and MemEnc [13] that are the implementations on which the proposal presented is based:

3.1.1 MEE Intel

Gueron is the first to introduce the term Memory Encryption Engine (MEE) in his work [5]. He describes it as a hardware-based encryption engine implemented in Intel SGX, which provides transparent encryption and decryption of data in DRAM memory, which helps protect the data's confidentiality and integrity against physical memory attacks. The MEE works by encrypting the contents of DRAM memory as they are written to memory and decrypting them when they are read from memory. The encryption and decryption are performed by the MEE, which is a dedicated hardware module that is integrated into the memory controller of the processor.

When Intel SGX and MEE are enabled in a processor, it works as follows: During regular operation, memory transactions are continuously issued by the processor's Core, and transactions that miss the cache are handled by the Memory Controller (MC). The MEE works like an extension of the MC, taking over the cache-DRAM traffic that points to the "Protected" data region. An additional portion of the memory, the "seized" region, accommodates the MEE's integrity tree. The union of these regions is called the "MEE region." It forms a range of physical addresses fixed to some size at boot time (the default size is 128MB) in a trustworthy way. Read/write requests to the protected region are routed by the MC to the MEE that encrypts (decrypts) the data before sending (fetching) it to (from) the DRAM [5]. The MEE autonomously initiates additional transactions to verify (update) the integrity tree based on a construction of counters and MAC tags. The self-initiated transactions access the seized region on the DRAM and some on-die SRAM array that serves as the tree's root [5]. The operation is shown in Figure 3.1.

The MEE uses the Advanced Encryption Standard (AES) encryption algorithm to encrypt and decrypt data. The AES algorithm is a widely used symmetric-key encryption algorithm that is considered to be secure and efficient. The MEE also supports multiple encryption keys, which allow different parts of memory to be encrypted with different

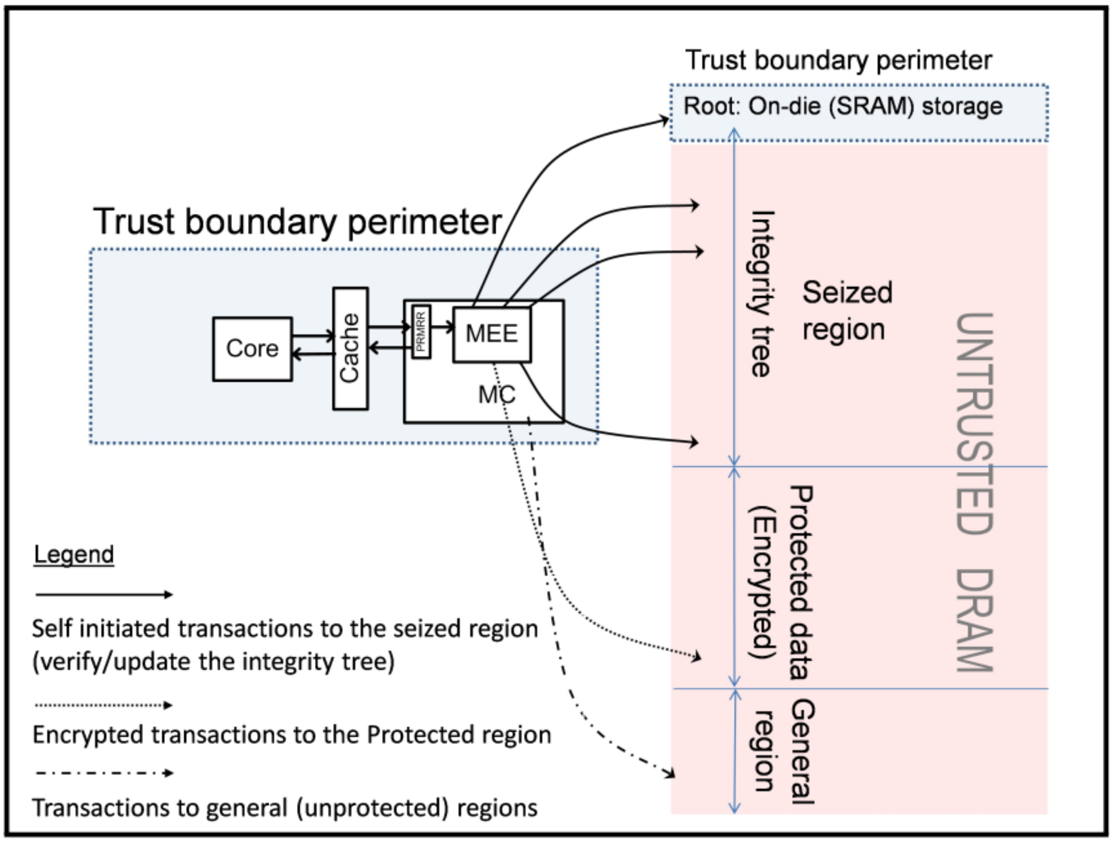

Trust boundary perimeter
Root: On-die (SRAM) storage
Trust boundary perimeter
Core
Cache
PRMRR
MEE
MC
Integrity tree
Seized region
UNTRUSTED DRAM
Protected data (Encrypted)
General region
Legend
Self initiated transactions to the seized region (verify/update the integrity tree)
Encrypted transactions to the Protected region
Transactions to general (unprotected) regions

Figure 3.1: MEE in Intel SGX [5]

keys, providing an additional layer of security by preventing an attacker from accessing all the data in memory, even if they can obtain one encryption key.

The MEE is a crucial component of Intel SGX, a hardware and software technology set that provides secure enclaves for executing sensitive code and protecting data. The MEE helps to protect the confidentiality and integrity of data in memory within the secure enclave.

3.1.2 MEMSEC

MEMSEC is an open-source framework of modular building blocks to implement RAM encryption solutions. A simple, fully synchronized stream interface connects the individual blocks and allows to replace specific components as needed quickly [15]. As a result, realizing arbitrary encryption pipelines is as simple as connecting the needed blocks according to the data flow graph of the design (shown in Figure 3.2).

The evaluation of the framework was performed in a ZYNQ board with benchmarks like Tinymembench (for memory bandwidth) and LMBENCH (for latency) running over an embedded Linux operative system, inferring an implementation of full memory encryption 2.1, using various cipher primitives like ASCON or AES and PRINCE in modes like ECB, CBC and XTS showed that this framework is very flexible and can easily support differing block sizes and memory alignment constraints.

Furthermore, the results demonstrate that retrofitting memory encryption to Zynq SoCs is feasible and that Ascon (with and without tree) is a decent choice for memory encryption when authenticity and confidentiality are desired.

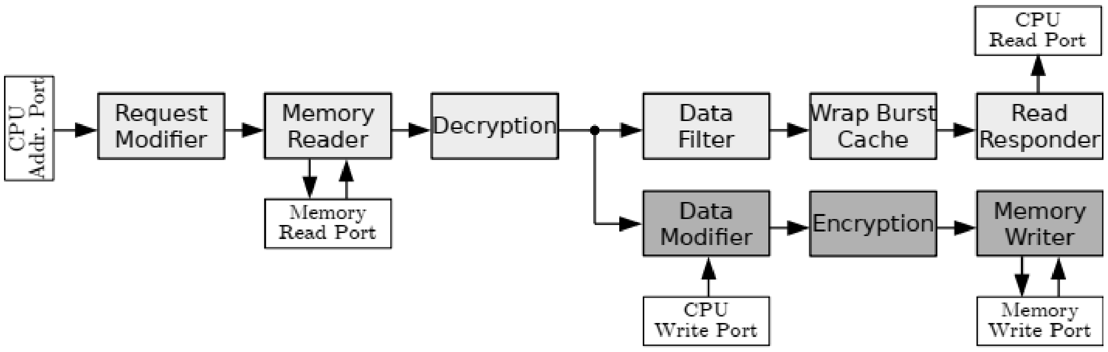


Figure 3.2: MEMSEC Pipeline [15]

3.1.3 MemEnc

MemEnc [13] is a lightweight memory encryption scheme based on AES and tested within a ZYNQ board with an ARM. However, it mentions that the scheme could be implemented with other processors like RISCV or MIPS.

The MEE is implemented in FPGA and communicates with the processor through AXI interfaces like MEMSEC, as shown in Figure 3.3; the engine then encrypts each work using AES in CBC mode and saves in the memory the encrypted word with the nonce used to encrypt it as it has shown in Figure 3.5. The scheme consists of only a few blocks of hardware 3.4, operates at 175 MHz, and requires 1648 LUTs and 885 registers on an FPGA. Compared with MEMSEC, its results show that MemEnc performs better in some scenarios. Furthermore, using comprehensive benchmarking, it was demonstrated that even though memory encryption has overheads, it is suitable for real-time systems with strict timing requirements and edge computing applications. Moreover, partial encryption of critical data can be performed for specific applications with minimal overhead.

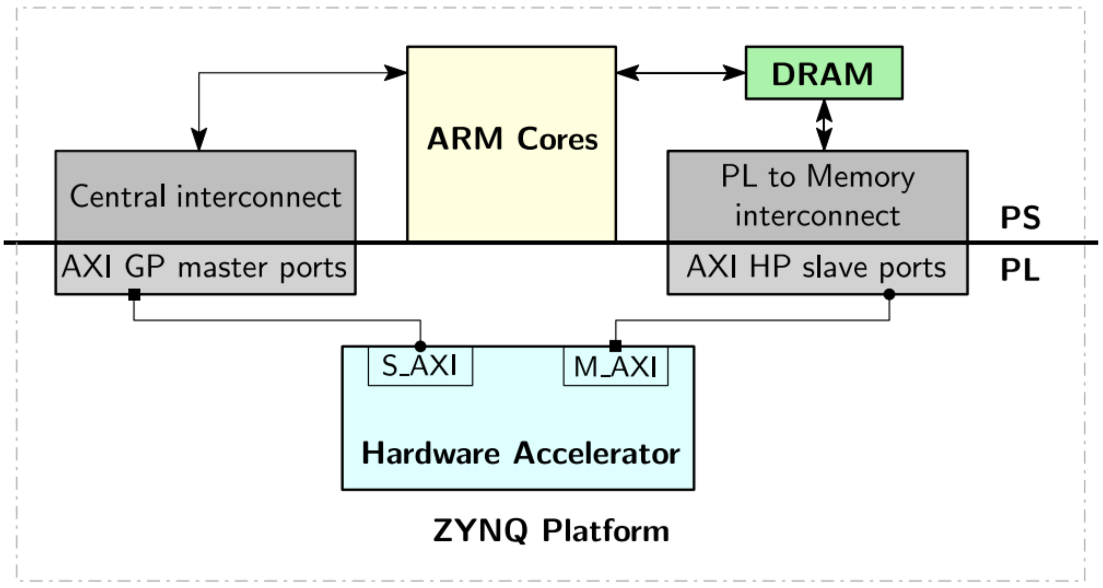


Figure 3.3: Zynq platform [13]

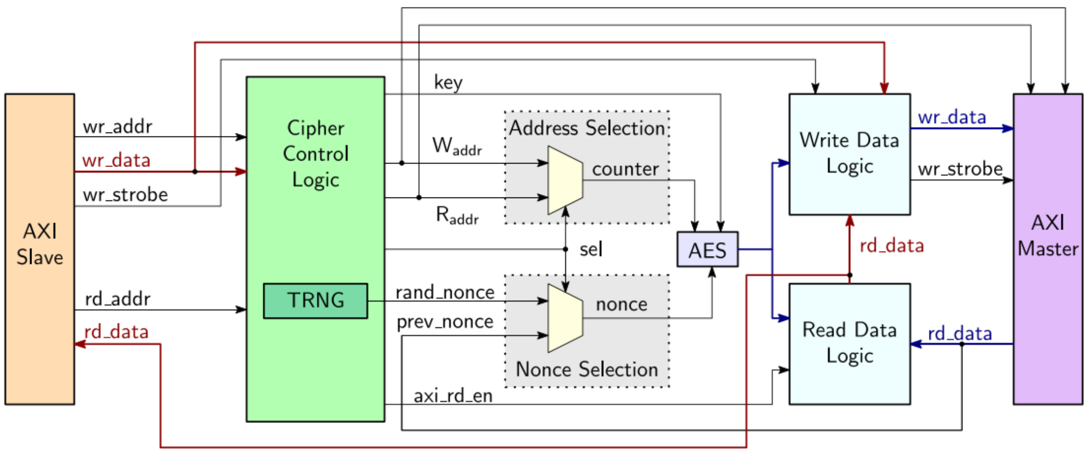


Figure 3.4: MemEnc Architecture [13]

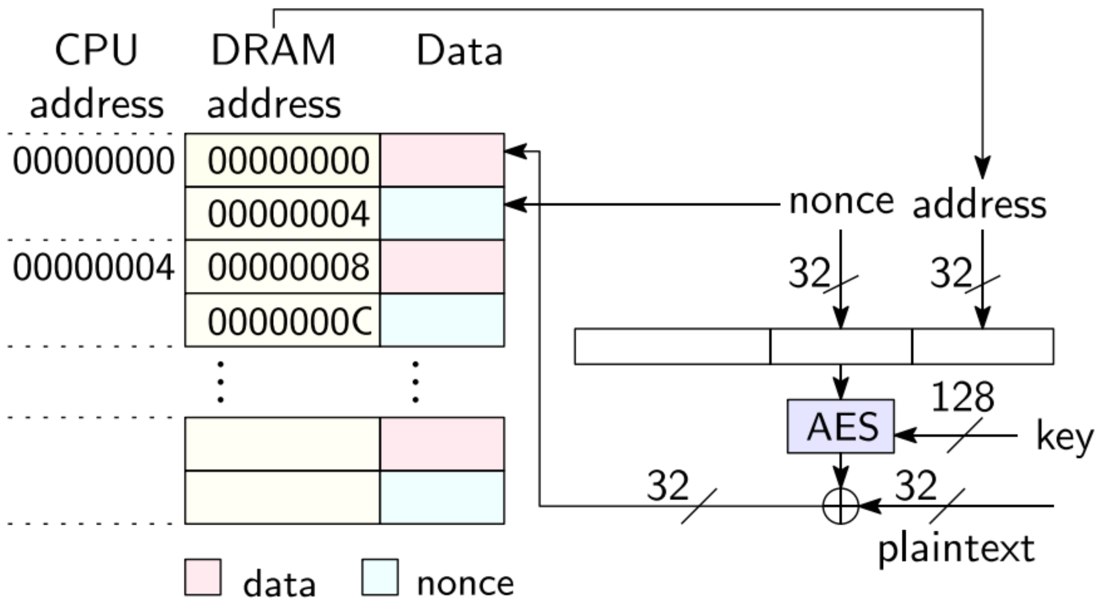


Figure 3.5: MemEnc encryption process [13]

3.1.4 State of the art conclusions

As we observed in this Section, there is not a wide range of options we can take as a reference for developing our proposal, which shows us that there is a large area of opportunity for research and development of memory encryption engines. However, as mentioned in the project introduction, the main goal is to propose a memory encryption engine option that can be used on devices with limited resources. Our primary reference will be the proposal mentioned in Gupta's work [13]. On the other hand, thanks to the work of Werner [15], we are sure that ASCON is an excellent option to use as an encryption core.

The design of our proposal will then start from the structure proposed with the MemSec but replace its encryption core with ASCON instead of AES. We must also pay attention to the modifications that must be made to both the structure of the memory encryption engine and the algorithm so that it can fit and function according to the needs that arise in the objectives of the work.

Chapter 4

Proposal Design

This Chapter describes the design process of our memory encryption engine, which starts from the selection of the platform on which the design was built, in this case, a Xilinx SoC Zynq, whose architecture gives us the possibility to work with an ARM processor connected to an FPGA by AXI buses and a controlled memory, which will allow us to simulate a system in which the processor does its write and read transactions through the memory encryption engine that we will design in the FPGA. Having an ARM processor on the same FPGA platform removes the work of having to design a processor ourselves, allowing us to focus on the design of the memory encryption engine. Further details of the design impact of choosing this platform are mentioned in Section 4.1 of the Chapter.

In order to comply with the objectives set out in this project, several particular considerations must be made during the design of the memory encryption engine (MEE). Since the MEE must be suitable for limited resource devices such as IoT devices, it was decided to use the ASCON lightweight cryptography algorithm as core encryption. As mentioned in Chapter 2, lightweight cryptography was introduced to protect the information created and transmitted by IoT devices. In particular, the ASCON algorithm was chosen because it was selected as the winner of the light cryptography contest organized by the NIST and will undergo a process of standardization; this means that, unlike other algorithm options within the lightweight cryptography catalog, ASCON will have more interest in research, in addition to allowing implementations using this algorithm to become commercialized.

Logically, the design would not end up with the selection of a lightweight algorithm and fit it into a structure of some existing encryption engine. Other aspects had to be taken into account, such as that the system must also be transparent to the operating system and the application; this means that it should not be necessary to modify either the code of an application or the operating system to enable the memory encryption functionalities, everything must be done in the hardware of the MEE. Last but not least, we looked for the MEE to have the lowest possible latency. To do this, we had to carefully analyze the ASCON algorithm to validate which steps were unnecessary for our application and decide how read and write transactions would be processed. All these challenges and how they were addressed are described in more detail in Section 4.2 of this Chapter.

Finally, Section 4.3 presents as the result of the design stage a block diagram of the memory encryption engine, named LLMEE (Low-latency Lightweight Memory Encryption Engine), and describes the function of each of its blocks.

4.1 Design Platfrom: SoC Zynq

Since the implementation will be carried out using a Xilinx Zynq, specifically the development card Zybo z7, which is a platform that provides us with an ARM Cortex A9 CPU of 2 cores and a part of FPGA, in addition to communication with a DDR3 memory of 1GB. The design must take into account the characteristics of this development card.

We must consider that our memory encryption engine (MEE) will be implemented in the programmable logic part. Communication with the processing system will be carried out through the AXI bus using the AXI4 protocol. The AXI bus allows us to communicate between the processing system and different peripheral devices that could be implemented in the programmable logic part, as shown in Figure 3.3. The AXI4 protocol consists of communication between 2 interfaces, one known as a master and the other as a slave, through 5 channels, three commissioned for write operations and 2 for read transactions. The writing channels are AW, in charge of transmitting the writing address; W, in charge of transmitting the data to write; and B, in charge of notifying that the writing was done correctly. On the other hand, AXI channels for reading are AR which transmits the reading direction, and R, which transmits the reading data. The AXI protocol also has other features that allow faster and more efficient communication between the CPU and peripheral devices; the documentation [24] describes all the technical specifications in detail of the AXI protocol.

After defining the considerations specific to the platform to be used for implementation, we will proceed to define the decisions and optimizations that were followed to meet the project's objectives.

4.2 Design Challenges and Optimizations

4.2.1 Cipher, Encryption Mode and Block Size

In order to make our implementation of MEE lightweight and at the same time with very low latency was sought within the different hardware implementations of ASCON. It was decided to use the fast core of ASCON shown by Groß et al. in [25], which, as we can see in Figure 4.1, consists of an unrolled round of ASCON permutation with different control signals to indicate the stage of the algorithm (initialization, associated data, encryption, decryption, finalization). In this way, performing the ASCON algorithm is enough to iterate the necessary round number on the fast core. This variant of ASCON aims to have maximum data performance with minimal processing delay [25].

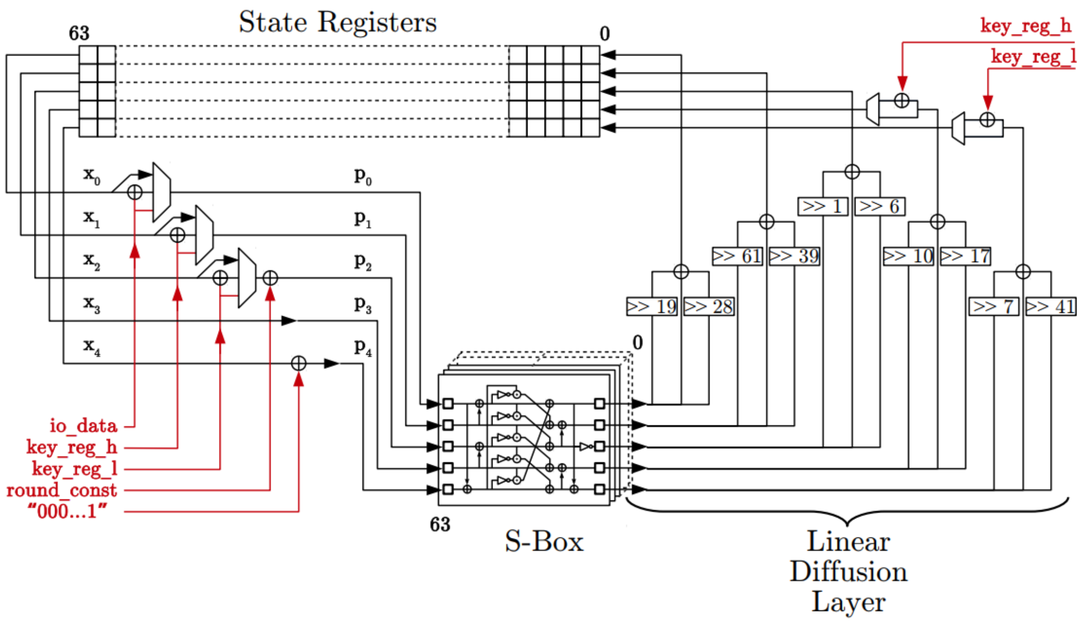


Figure 4.1: Ascon Fast core [25]

Now, talking about how the encryption process works in our MEE, it is essential to consider how the CPU transactions will be performed to memory to consider how the data encryption will be performed. CPU-to-memory transactions can be performed in 32-bit block bursts; however, making an implementation that allows encrypting in bursts of more than one block involves the addition of hardware that processes burst operations and accommodates that data to power the cipher and hardware to accommodate the data to the output of the cipher to then feed to memory or the CPU. Adding such hardware implies, in addition to the increased resources that would be required, that our MEE would have more stages in the pipeline, as can be seen in [15], which could increase transaction latency. While burst-mode transactions allow us higher data throughput between memory and CPU for large amounts of data, according to [13], IoT devices do not take advantage of this mechanism since transactions on this class of devices are usually from smaller memory blocks, even just a few bytes. That is why for our MEE, only 32-bit transactions will be considered.

The next issue to solve is related to the data that feed the cipher, according to [19]. As shown in Figure 2.4, ASCON requires a 128-bit key, a 128-bit nonce (fast core [25] also allows 64-bit nonce), and 64-bit data blocks, suggesting that if we want to use different keys and nonces, we would have to store them in memory to be able to recover the encrypted data; this would make us occupy 320 bits of memory for every 32 bits transactions. On the other hand, if we decide to use a fixed key and nonce, we would need only 64 bits of memory for each 32-bit transaction. However, we will compromise the algorithm's security and make it vulnerable to side-channel attacks, for example.

To solve this issue and reduce the memory increase as much as possible while not losing security robustness, we decided to use the ASCON cipher as a tweakable block cipher in a counter (CTR) like mode. To do this, we will have a fixed key, and the nonce will be formed by the 32 bits of the memory address concatenated with a 32 bits random nonce (this will be the tweak); in this way, in a write-in-memory transaction, the ASCON's S state will be formed by the write address, a generated random nonce and the constant key. Once the 320-bit S state is formed, the 12-round initialization will be done; a xor operation will be performed between the 32 bits of plain text and the first 32 bits of the S state; this way, we will get the 32 encrypted bits that will be saved in memory followed by the 32 bits of nonce random used to encrypt. The process will be the same for read transactions, but now we must first read the memory data to get the saved nonce and the encrypted text. The 12 initialization rounds are done, and finally, the xor operation between the encrypted text and the first 32 bits of the S state; in this way, we will get the 32 bits of the plain text, and our read transaction can be completed. This process is shown in Figure 4.2.

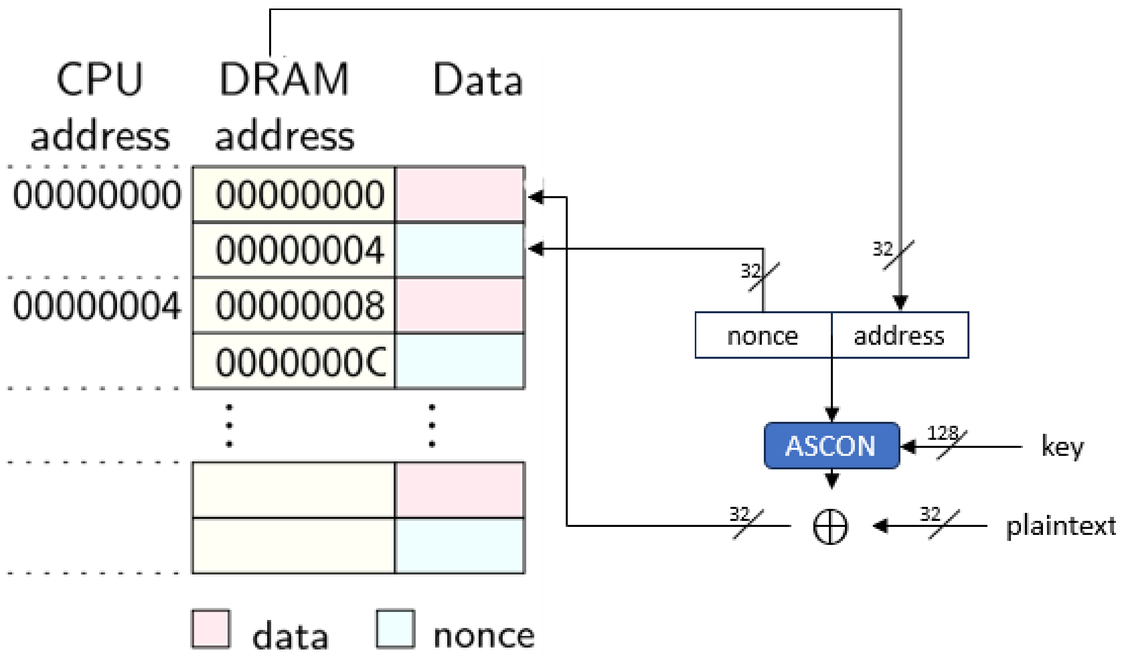


Figure 4.2: Encryption Process of LLMEE

4.2.2 Low Latency and Transparency

Another objective of the project is to design our LLMEE so that it is transparent. This means that the hardware must carry out the entire encryption process and that it will not be necessary to make any modification to the application software that will run the device or the operating system with which it counts. It is necessary to identify the control signals from the CPU to the memory to perform the writing and reading processes and design finite state machines that control such transactions and the encryption process.

Write transaction FSM

As shown in Figure 4.3 based on the description of the AXI protocol in [24] and similar to the state machine showed in [13], when receiving the AXI signals of writing (invalid or

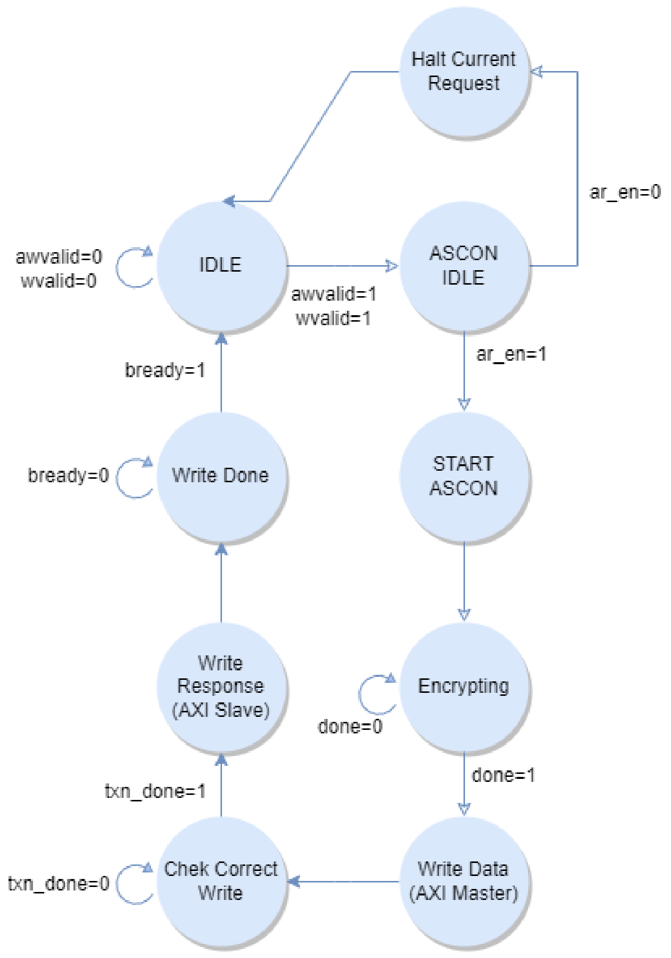


Figure 4.3: Finite State Machine of the write transaction in the LLMEE

valid), the machine proceeds to verify if the cipher is not processed another transaction; if so, the current request is stopped; in case the cipher is accessible, the data to write is encrypted, once the encryption is finished, it sends a termination signal, and the master interface sends the appropriate signals to write in memory, it is expected to receive the memory confirmation that the writing was done correctly. Finally, the slave interface of the MEE sends the necessary signals through channel B to the CPU that the writing was already done correctly.

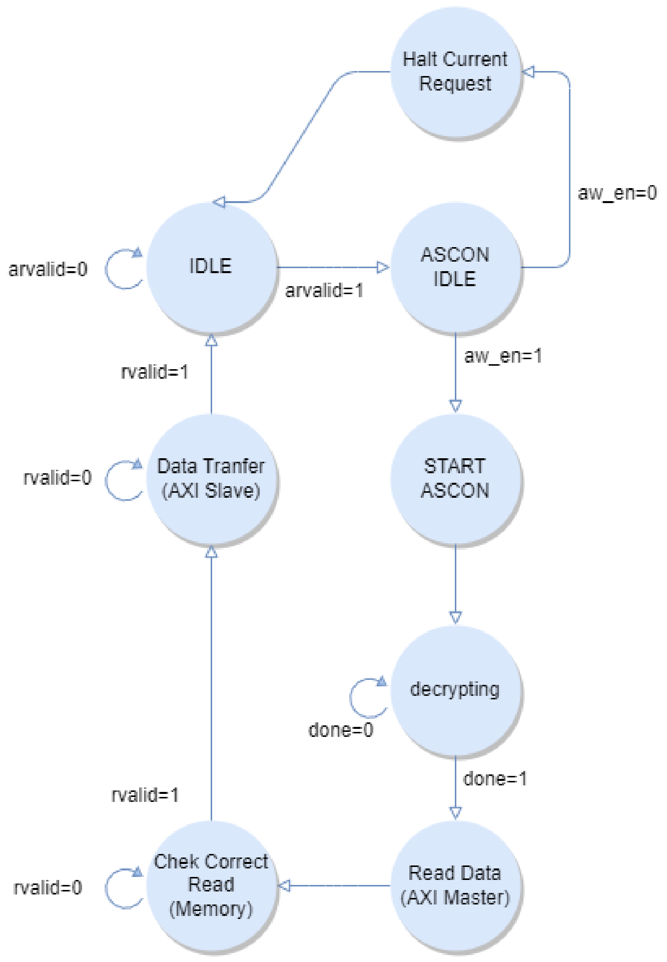


Figure 4.4: Finite State Machine of the read transaction in the LLMEE

Read transaction FSM

The reading process is very similar to the writing process, as seen in Figure 4.4; the difference is the AXI signals that are received to confirm that it is a reading transaction [24] (arvalid instead of awvalid or wvalid). Likewise, the confirmation by the slave interface with the CPU that the transaction was performed correctly is triggered, and the data read is transmitted to the CPU.

These state machines give us an overview of the processes to be performed and the signals that could intervene to change between states; in the implementation, we will see that these processes involve more signals and processes.

4.3 Design Architecture

Figure 4.5 shows a diagram of the complete LLMEE architecture, represented by blocks connected by signals and data buses.

- **AXI Slave:** This block will handle the data transmission between the CPU and the MEE; the CPU will act as the master interface and, for each transaction, will activate the required signals to the slave. This block will be responsible for sending the signals to confirm if a write was performed correctly and sending the data read to the CPU in case of a read transaction.

- **Control Logic:** When we want to do the secure transaction through the LLMEE, the CPU must write and read from the LLMEE's address. The LLMEE need to translate this address to the one needed to write and read in the DDR, as we can see in Figure 2.1 in the part of partial memory encryption PME or full memory encryption FME, so the control logic block is responsible for receiving the addresses of the AXI interface and converting them to the address that will be read or written in memory DDR *waddr* to *Waddr* and *raddr* to **Raddr**, it also generates the encryption key and random nonce for each transaction. Finally, this block is also responsible for identifying whether the transaction is a read or write to control the selector controls address and nonce multiplexers.

- **Cipher control:** This block controls the ASCON Fast Core instance; as mentioned above, ASCON Fast Core is only an unrolled ASCON round. Therefore, it needs control signals to indicate which stage is running and that the necessary data must be fed at specific times. Likewise, the cipher control is a state machine that calls the cipher and sends signals to the other blocks to indicate if the cipher is busy with a transaction or has already finished encrypting, and the data at its output can be used.

- **AXI Master:** The master interface handles transactions between MEE and DDR memory [26]. The function of the state machines in this block is essential because

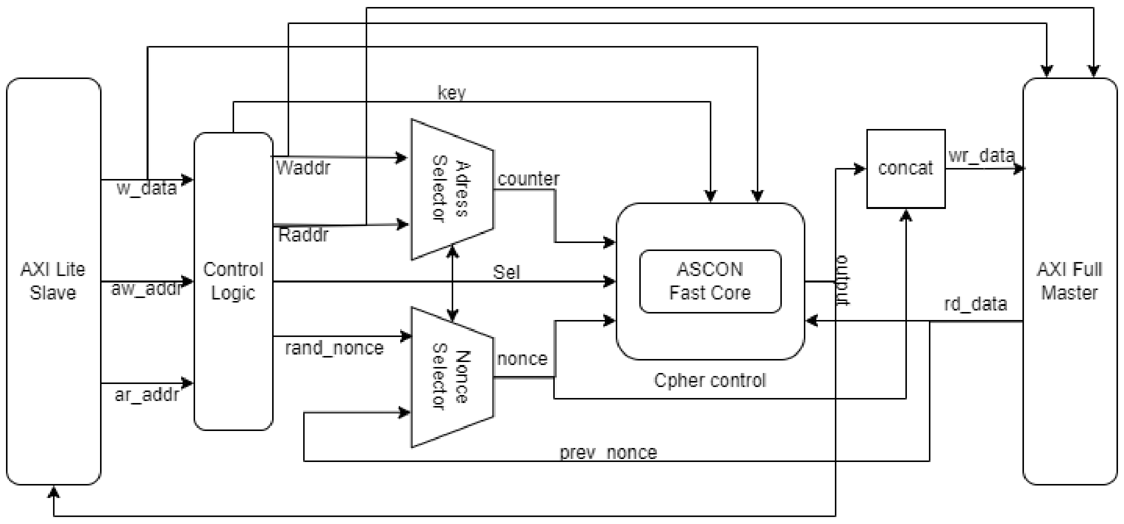


Figure 4.5: LLMEE architecture

they must work in synchrony with the data and signals received from the other blocks to write the correct data in the right direction. It is also responsible for transmitting signals from DDR memory to confirm if data was written correctly and transmit the data read from the memory to the rest of the MEE.

Synchrony between signals and data is essential throughout the architecture pipeline, so it will be necessary to add several latches, flip-flops, and other strategies to achieve this synchrony. For this point, it will be necessary to review the data obtained in the first simulations of the MEE.

Chapter 5

Implementation and Results

5.1 Platform description and EDA Tools

5.1.1 Zybo Z7 Zynq 7000

The Zybo Z7 (Figure 5.1) is an advanced, out-of-the-box development board designed for embedded software and digital circuit creation. It revolves around the Xilinx Zynq-7000 family, based on the Xilinx All Programmable System-on-Chip (AP SoC) architecture. This architecture integrates a dual-core ARM Cortex-A9 processor with Xilinx 7-series Field Programmable Gate Array (FPGA) logic. The Zybo Z7 incorporates a wide range of multimedia and connectivity peripherals, augmenting its capabilities as a robust single-board computer. The FPGA further enhances its flexibility and processing power. With its video-enabled features like the MIPI CSI-2 compatible Pcam connector, HDMI input, HDMI output, and high DDR3L bandwidth, the Zybo Z7 serves as an economical solution for high-end embedded vision applications, which are renowned for utilizing Xilinx FPGAs. Thanks to the Zybo Z7's Pmod connectors, expanding its functionality is effortless. These connectors allow easy integration of Digilent's extensive collection of over 70 Pmod peripheral boards, including motor controllers, sensors, displays, and more [26].

Features important for this project:

ZYNQ Processor

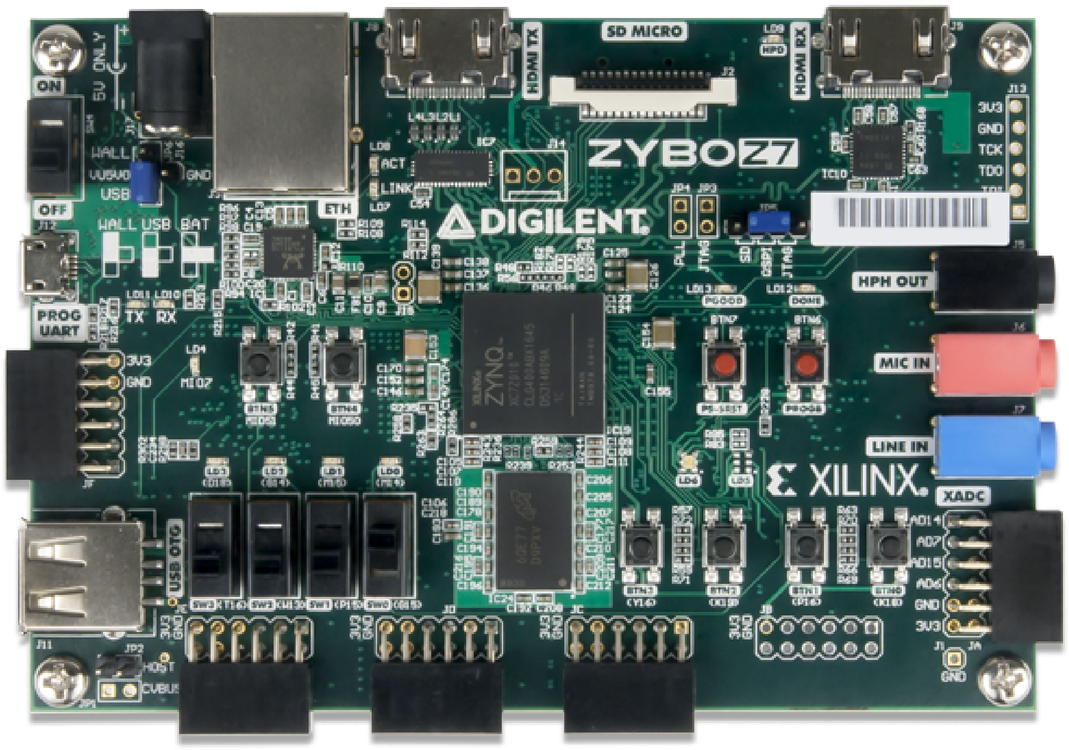


Figure 5.1: Zybo z7 10 development board

- 667 MHz dual-core Cortex-A9 processor
- DDR3L memory controller with 8 DMA channels and 4 High Performance AXI3 Slave ports
- High-bandwidth peripheral controllers: 1G Ethernet, USB 2.0, SDIO
- Low-bandwidth peripheral controllers: SPI, UART, CAN, I2C
- Programmable from JTAG, Quad-SPI flash, and microSD card
- Programmable logic equivalent to Artix-7 FPGA

Memory

- 1 GB DDR3L with 32-bit bus @ 1066 MHz

5.1.2 Xilinx Suite

Vivado 2022.2

Vivado is an advanced software development environment created by Xilinx. It is specifically designed to develop and implement digital systems using Xilinx FPGA and SoC (System-on-Chip) devices. Vivado provides a comprehensive suite of tools and features that enable designers to perform tasks such as synthesis, simulation, implementation, and programming of Xilinx devices. With Vivado, engineers can design complex digital systems by creating and connecting various hardware components, configuring the FPGA or SoC device, and optimizing the design for performance, power consumption, and area utilization. The software supports various programming languages and design methodologies, including VHDL and Verilog. Vivado offers an intuitive graphical user interface (GUI) and powerful automation capabilities, making managing and streamlining the design process easier. It includes features like IP (Intellectual Property) integration, where designers can utilize pre-designed IP cores to accelerate development and reduce time-to-market. Vivado also supports advanced debugging and verification techniques to ensure the functionality and correctness of the designed systems. In summary, Vivado is a versatile software toolset crucial in developing digital systems using Xilinx FPGAs and SoCs. It simplifies the design process, improves productivity, and empowers engineers to create innovative and efficient solutions for various applications [27].

Vistis 2022.2

Vitis is an integrated software development platform developed by Xilinx. It is designed to facilitate the development of applications targeting Xilinx FPGAs, SoCs, and ACAPs (Adaptive Compute Acceleration Platforms). Vitis enables software engineers to leverage the power and flexibility of Xilinx devices and accelerate their applications through hardware acceleration. The Vitis platform provides a unified development environment that allows developers to write and optimize their applications using familiar software programming languages such as C, C++, and OpenCL. It supports various development flows, including both host-centric and kernel-centric approaches. With Vitis, developers can design heterogeneous systems that combine traditional software running on CPUs with custom hardware accelerators implemented on FPGAs. The platform provides software profiling and optimization tools, allowing developers to identify computationally intensive parts of their code and offload them to the FPGA for significant performance gains [28].

Petalinux 2022.2

Xilinx PetaLinux is a software development toolchain designed explicitly for creating and customizing Linux-based systems for Xilinx embedded platforms. It provides a streamlined workflow for building, configuring, and deploying Linux distributions tailored to Xilinx's FPGA and Zynq-based System-on-Chip (SoC) devices. PetaLinux offers tools and utilities that simplify creating a Linux operating system for Xilinx platforms. It utilizes the Yocto Project, an open-source collaboration framework, to build customized Linux distributions. PetaLinux extends the Yocto Project by providing additional Xilinx-specific components, drivers, and features. Using PetaLinux, developers can customize the Linux kernel, bootloader, device tree, and root file system to match the specific requirements of their embedded system. They can add device drivers, turn features on or off, integrate custom applications, and configure the system to optimize performance, power consumption, and resource utilization. PetaLinux provides a command-line interface (CLI) and a graphical user interface (GUI) for managing the development workflow. It includes tools for configuring hardware interfaces, managing software packages, generating bootable images, and debugging the system [29].

5.2 Hardware Implementation

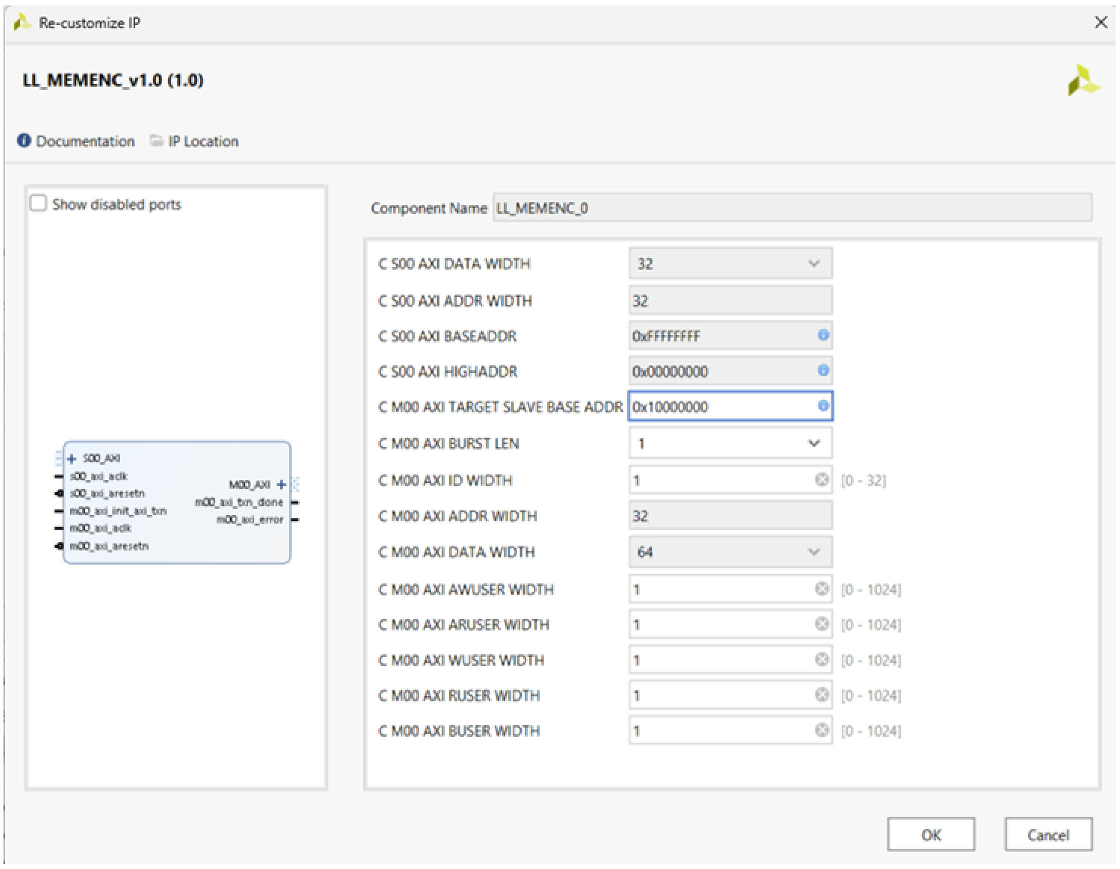
For the hardware implementation, we use the tool Vivado ”create and package new IP” Using the option of ”create new peripheral AXI4”, we can assign the AXI interfaces we need. For our LLMEE, we use an AXI lite slave interface with a 32-bit data bus since, as mentioned in the previous Chapter, we will only consider 32-bit transactions, and since we will not use the burst mode, the AXI lite interface is sufficient, this interface will be used for communication with the CPU. On the other hand, we will use a full master AXI interface for transactions between LLMEE with DDR memory; in this case, we will use the full AXI interface because it allows us to have a 64-bit data bus, as we will need to read and write the encrypted 32-bit data concatenated with the 32-bit nonce for each transaction.

After defining the interfaces, the tool allowed us to edit the IP as if it were a standard Vivado project, so we added more entities. It is at this point that we proceed to create the other blocks defined in the architecture using Verilog as hardware description language:

- Control Logic Entity: We connect the control signals of the AW, W, B, AR, and R channels from the slave interface, as well as the data and address buses for writing and reading. Since the LLMEE module is mapped in a memory address, we must subtract this value from the addresses from the slave interface so that when we do the transactions with the DDR memory, these transactions are made in the correct addresses. It will also have the register containing the value of the 128-bit key and a 32-bit random number generator (for project tests, the random number generator was disabled to verify that a default nonce was written correctly). Finally, we will do some combinational logic so that the entity can define if it is a read or write transaction using the AXI slave interface control signals and enable the selector.

- Control Cipher Entity: For the cipher control entity, we defined a 3-state machine to indicate that the cipher is accessible (IDLE), encrypting, or had finished, so we could indicate if it was busy with another transaction or to send the ”done” signal when it finished encrypting. The ASCON Fast Core (written on VHDL) was directly instantiated as an entity and was not modified. Since the cipher will only process 32-bit blocks, one at a time, it will only be necessary to activate the initialization stage (12 rounds), and it will be the same procedure for encryption and decryption (writing and reading).
- AXI Master entity: The most complicated part was the entity of the master interface, since here, a state machine was implemented to control whether a write or read transaction would be performed. Care was also taken to send the control signals correctly to memory and ensure that we received the confirmation responses from memory to continue processing more information.

In the latter part, great care had to be taken to synchronize the control signals well with the data since the control signals came directly from the CPU while the data had delays caused by the encryption stage. Several flip-flops were swapped for latches to ensure the synchrony of signals in order to solve this synchronization issue. Although there are better practices for encoding sequential logic, it was the fastest way to ensure that control signals are considered with the corresponding transaction data. The elaborated design was packed in an IP as shown in Figure 5.2.



LL_MEMENC_v1.0 (1.0)

Component Name LL_MEMENC_0

C S00 AXI DATA WIDTH	32	
C S00 AXI ADDR WIDTH	32	
C S00 AXI BASEADDR	0xFFFFFFFF	
C S00 AXI HIGHADDR	0x00000000	
C M00 AXI TARGET SLAVE BASE ADDR	0x10000000	
C M00 AXI BURST LEN	1	
C M00 AXI ID WIDTH	1	[0 - 32]
C M00 AXI ADDR WIDTH	32	
C M00 AXI DATA WIDTH	64	
C M00 AXI AWUSER WIDTH	1	[0 - 1024]
C M00 AXI ARUSER WIDTH	1	[0 - 1024]
C M00 AXI WUSER WIDTH	1	[0 - 1024]
C M00 AXI RUSER WIDTH	1	[0 - 1024]
C M00 AXI BUSER WIDTH	1	[0 - 1024]

Figure 5.2: LLMEE IP Package

5.3 Simulation

5.3.1 RTL Simulation

For the simulation at the RTL level, the AXI_VIP IP (Verification IP) was used, which allows us to send AXI transactions easily from a testbench encoded in SystemVerilog; in this way, we only connect the axi_vip with our LLMEE and a BRAM block that will simulate the system memory as it has shown in Figure 5.3. With this small system, we verified the correct functioning of our LLMEE in the waveform after running the live behavior simulation. The purpose of this simulation was to observe that the axis control signals were triggered correctly and that the encryption worked correctly, in addition to verifying that it was written and read correctly from memory.

5.3.2 Hardware Debug

In addition to the simulation in RTL, it was necessary that our LLMEE also worked already implemented in our development board Zynq z7; for this, we used debug tools in Vivado hardware, which allowed us in the synthesis stage to put test points (as if it were a real oscilloscope) in the signals that we are interested in observing, later the implementation is run, the bitstream is generated, and the FPGA is programmed. Once programmed, the FPGA opens the tool XSCT (Software Command-line Tool) of Vitis, which allows us to interact with the CPU of our card; in this way, we can directly send reading and writing instructions with read (mrd) or write (mwr) commands to the addresses corresponding to our module. In the waveform of Vivado, we can observe the behavior of the signals where we place the probes. Since we cannot put too many test points as this increases the hardware considerably, we tested only the aw_ready, w_ready, ar_ready, and r_ready AXI control signals to verify the proper functioning of our system on hardware. Seeing that the signals were activated when sending mwr and mrd commands from XSCT, we checked that our system was working correctly.

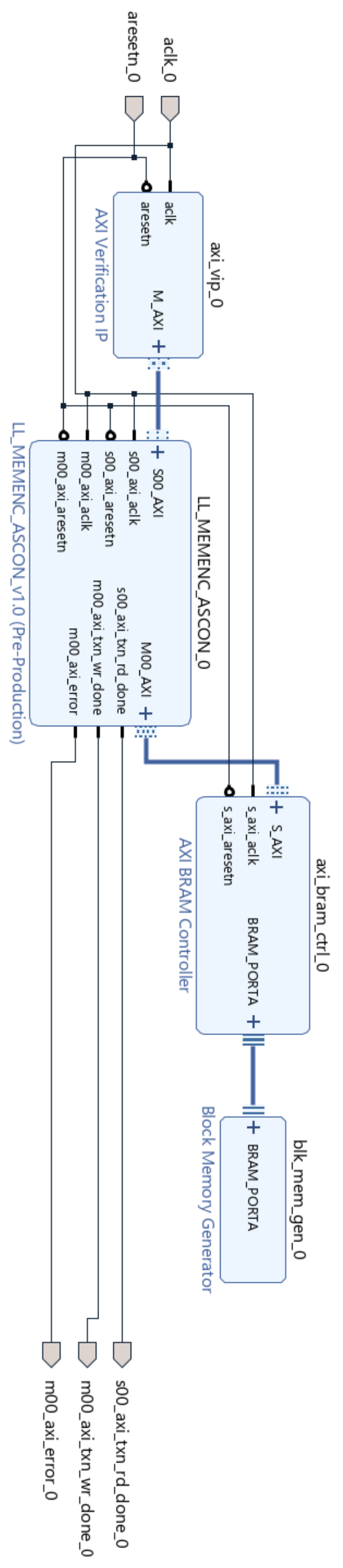


Figure 5.3: Blok Diagram for RTL Behavioral Simulation using AXI_VIP

5.4 Resource Utilization

After checking that our LLMEE worked correctly in the behavior simulations and hardware, we built the complete system already with the processor, our module, and the connections to memory with the help of the block design of Vivado. We use the ZYNQ7 IP, which is the block that is required to enable the CPU part of our system, we enable the general purpose AXI master GP AXI master interface, which is where the CPU communicates with our LLMEE module, and we also enable the high-performance interface HP AXI slave interface of 64 bits that allows us to communicate directly with the DDR memory, more details about these ZYNQ system interfaces can be found at [26]. Later we ran the option of automatic connection that adds us other IP AXI interconnect that serves so that any connection of some peripheral module adapts to the protocol that uses the processor; maybe an IP processor system reset is added, which synchronizes the AXI reset signals of all modules. Finally, we assign the direction 0x40000000 to our LLMEE and a space 0f 1GB or 0x40000000 as shown in 5.4. In the end, the Block design is observed as shown in Figure 5.5.

The synthesis, implementation, and bitstream were executed once the complete system was generated with the block design. The implementation generates a report that gives various data about the system, such as resources used, time analysis, and energy consumption. For this work, the data that we take from this point were those of occupied resources measured in LUTs and Registries, as well as the time analysis that gives us the delay of the critical path of the system and with which we can deduce the maximum frequency at which our system could operate without running any risk of errors.

Table 5.1 shows the data obtained compared with the works of [13] and [15], where we can observe that our MEE is lighter than the other two but is the second best in terms of frequency that can reach.

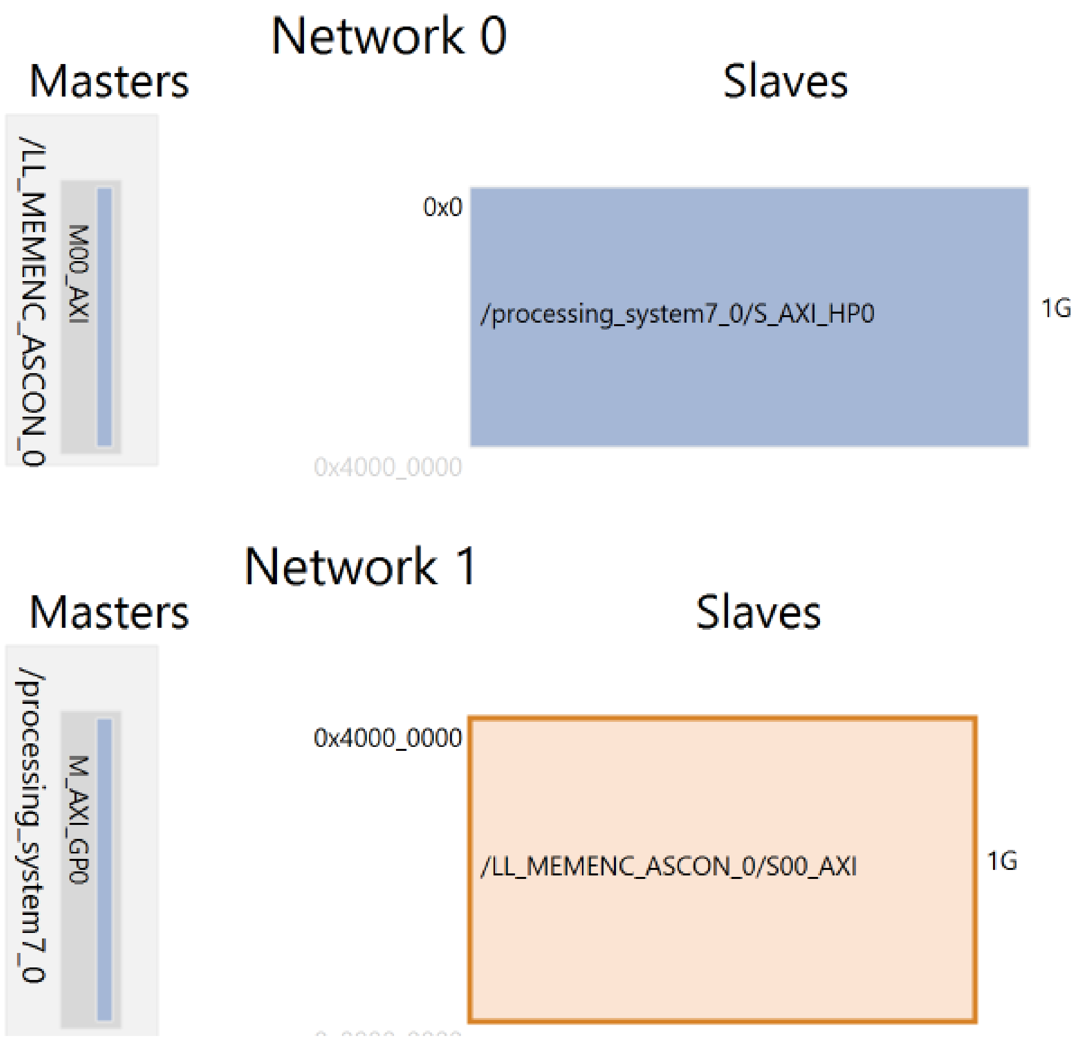


Figure 5.4: Addres MAP of LLMEE

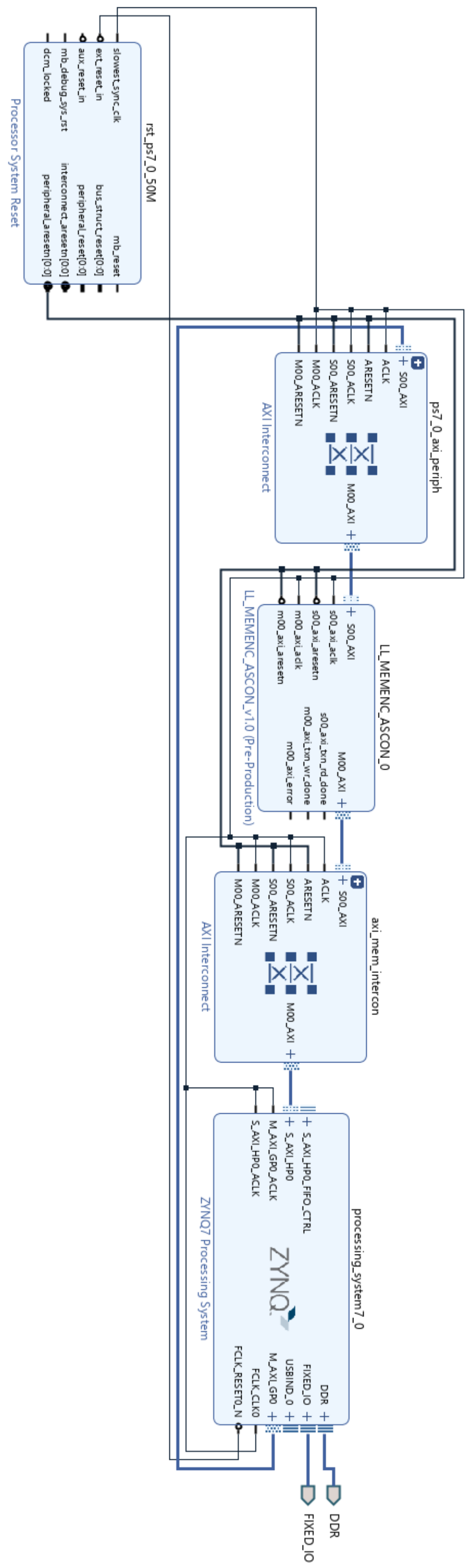


Figure 5.5: Block Design of System Hardware in Vivado

MEE	LUTs	Registers	Max Freq (MHz)
MemEnc [13]	1648	885	175
MEMSEC [15]	4701	2332	56
LLMEE [This work]	792	766	100

Table 5.1: FPGA resources utilization

5.5 Baremetal Tests

From then on, the project started the tests already with the system implemented in the development card. Using Vitis, we create a platform from the XSA files; Vivado generates this file and includes all hardware information, including bitstream, and then we create applications on that platform.

5.5.1 Memory Test

For memory test was developed a Memory Test Application that consists of performing write-in-memory transactions through the module and then reading all the written data; the application averages the clock cycles it took to do all the writes in memory and then all the readings; these data were taken as comparison data.

Two tests were made with the Memory Test Application, one with 1000 transactions and another with 16000, equivalent to 4KB and 64KB of data written and read (since each transaction is 32 bits). The tests were done on four systems to compare, the first without any encryption engine, the second using the MEMSEC of [15] in plain mode, that is, without cipher, the third using the MEMSEC but with the cipher prince in ECB mode and the fourth using our LLMEE. The reason to use these modes in MEMSEC is that they are the ones that present the best performance in terms of latency and bandwidth in [15].

The results obtained are shown in Table 5.2 and in the graph 5.6, where we can see that even when our LLMEE module takes 2.4x clock cycles in write operation and 3.7x clock cycles in read operation compared with the unprotected system, is 1.2x faster in write operation and 2x read operations of 32-bit word readings compared to the faster versions of MEMSEC (MEMSEC Prince ECB).

	4kB		64kB	
MEE	Write (Clock Cycles)	Read (Clock Cycles)	Write (Clock Cycles)	Read (Clock Cycles)
Unprotected System	35136	25088	560282	443168
MEMSEC Prince ECB [15]	177132	124588	2832134	2016088
MEMSEC Plain [15]	177132	124588	2832136	1992988
LLMEE [This work]	87136	102700	1392132	1642192

Table 5.2: Table with the results in clock cycles of the memory test performed in bare metal with different MEE

5.6 Memory Bandwidth

To test the bandwidth that can support our module, we use a test bench called Tinymembench, the same used by [13] and [15] for their tests. In this way, we could once again compare ourselves with these works. However, to be able to run this test, it was required that our system also run an operating system since the test uses operating system-specific libraries to measure times and optimize dynamic memory allocation functions.

So we had to create a first boot loader, a bootloader, and a Linux kernel for our system from the XSA generated by Vivado, using the Petalinux 2022.2 tool. However, to make our operating system also run in safe memory space (on the LLMEE address), it was necessary to modify the system's device tree to add the LLMEE as a memory device, indicating its base address and size according to 5.4, in this way, the bootloader can identify that a device exists in that region of memory in which it can write. We must also configure the bootloader (U-boot) to copy the kernel image to that secure memory space, unpack the kernel and run it from there.

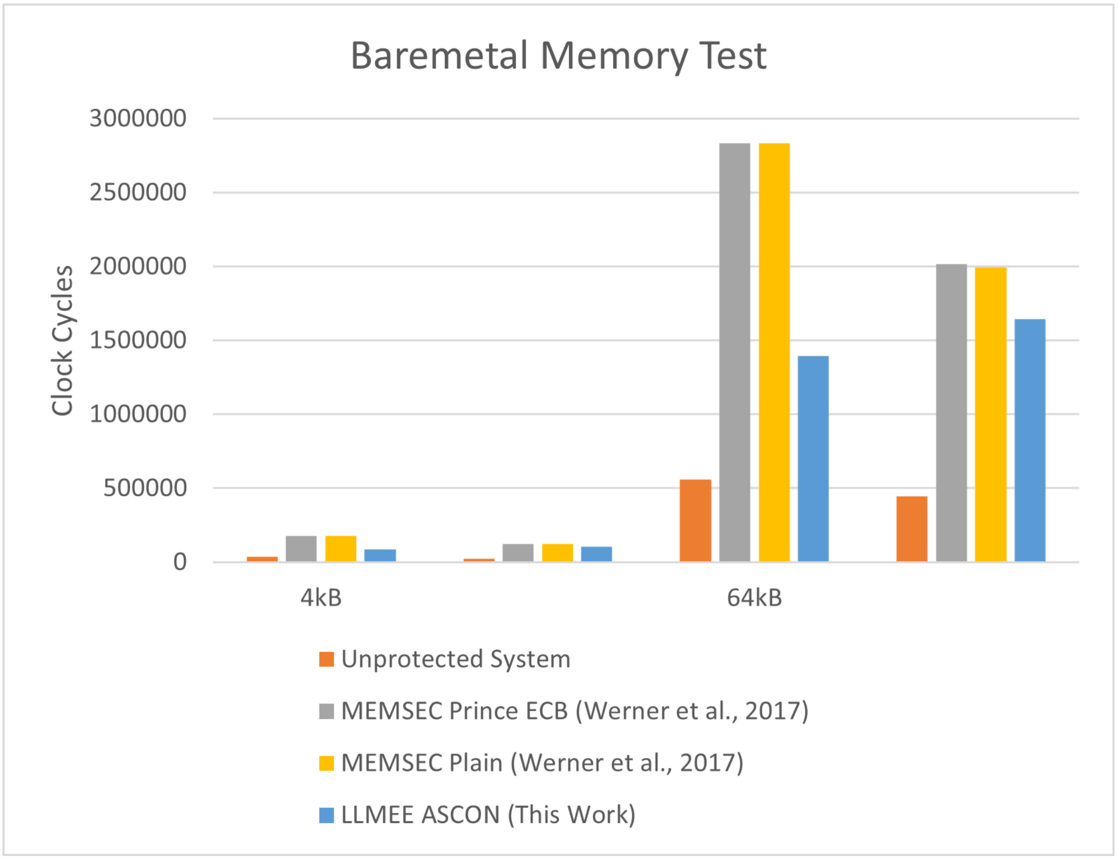


Figure 5.6: Graph comparing the results of different MEE in the memory test.

Once we had our zynq_fsbl.elf (first bootloader), u-boot.elf (bootloader), system.bit (bitstream to program the FPGA), system.dtb (device tree), and image.ub (kernel image) files, we used jtag to boot. JTAG facilitates the booting process by saving the part of flashing some external memory. Booting from the JTAG requires using the Xilinx System Debugger (XSDB) tool included in Vitis, which allows us to connect to our development card and upload programs and files. Once we connect to the card from XSDB via the JTAG, we load the system.bit file to program the system FPGA with our LLMEE; then, we load the zynq_fsbl.elf file, which takes care of initializing the hardware of the CPU system, memory, clocks, and more initial features. Then we load the system.dtb since u-boot.elf (which is the file we load later) will read it to know the system's devices. Finally, we load the image of the kernel image.ub at the secure address (starting at address 0x40000000), and from the bootloader's CLI, tell you to start the boot from the address where we load the kernel image. In the case of our system, it was also necessary to disable the cache memory since there was a coherence problem.

Since we were sure that our operating system was appropriately booted and logged in, we assigned an IP address, then connected the card with an Ethernet cable to our computer and ensured a connection; this step is necessary to run applications on the system from Vitis. Once the system is ready, we open Vitis, compile the Tinymembench application, and run it on our card.

The tests were done with an unprotected system, an unprotected system with the cache disabled, and, of course, with our LLMEE and the cache disabled. The data obtained were compared with those obtained in [13] in Table 5.3. We have to consider that there is a bottleneck in the communication between CPU and PL when we use protected system with MEE, due we have to use CPU Programmed I/O for data movement, this allow us $< 25MB/s$ throughput according [26]. However, even with this limitation, we can see in the graph in Figure 5.7, our system obtained 55% higher bandwidth in memset and 84% higher bandwidth in memcpy than MemEnc, those tests consist of writing data in memory and copying data from one memory space to another (read and write) respectively. However, in random fill, out LLMEE had a 16% lower performance than MemEnc, and in Neon read a 28% lower performance than MEMSEC; the latter can be attributed to MEMSEC's ability to use burst mode.

Platform	standard memset (Mb/s)	standard memcpy (Mb/s)	Random fill (Mb/s)	Neon read prefetched (MB/s)	Max frecuency (MHz)	Board
MemEnc [13]	10.5	5.9	12.3	12.4	175	MiniZed
MEMSEC-AES [15]	3	3.7	5.9	27.9	50	ZedBoard
Unprotected	2121.7	342.3	732.3	978.8	667	Zybo z7
Unprotected-Cache Disabled	665.9	255.2	83.8	395.3	667	Zybo z7
$LLMEE_{50hz}$	9.1	6.3	5.9	11.6	100	Zybo z7
$LLMEE_{100hz}$	16.3	10.9	10.3	20.1	100	Zybo z7

Table 5.3: Table with the data obtained with the test and the data from [13].

5.7 Results Analysis

The first result obtained was that of the use of resources, which evaluates if our MEE is light, and as we can see in Table 5.1, our LLMEE needs only 46% of LUNs and 86% of registers needed by MemEnc and only 17% of LUNs and 32% of registers that MEMSEC needs from an FPGA; this is due firstly because we occupy fewer pipeline stages than the MEMSEC framework, and secondly, because ASCON uses much fewer resources than AES as can be seen in [30].

The next test performed with the bare-metal system evaluated the latency of the

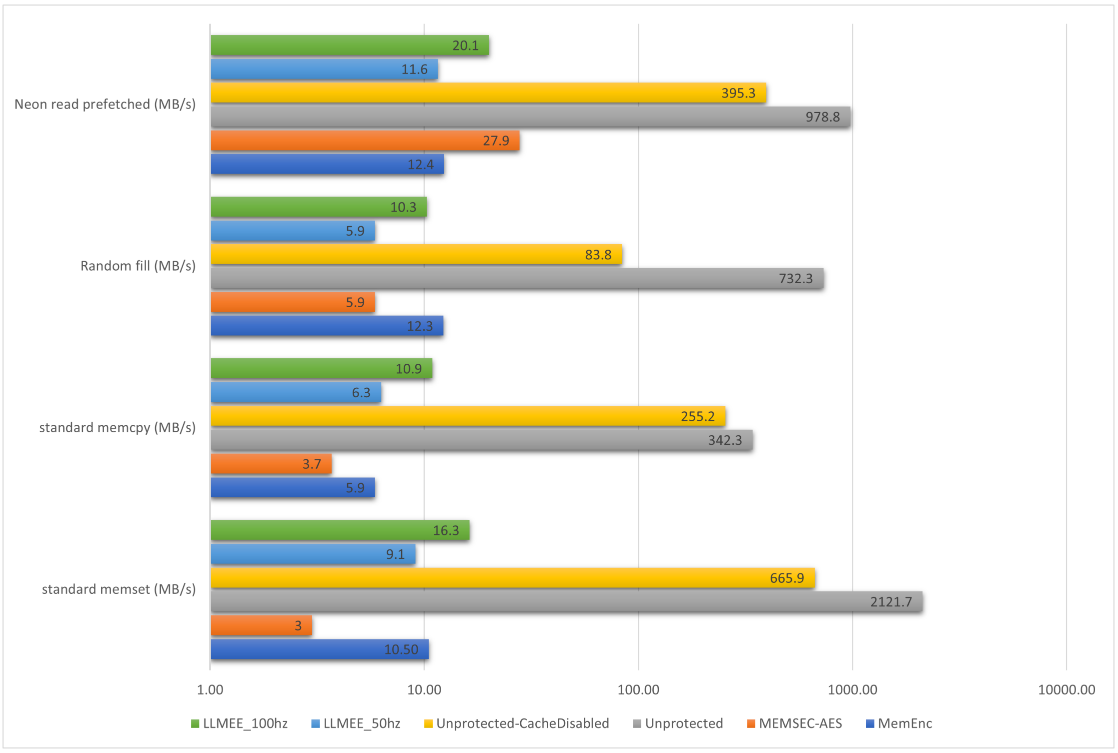


Figure 5.7: Memory Bandwidth Measured with Tinymembench

MEE: how long it took the CPU to write and read to memory through the MEE. We can observe that the clock cycle overhead of our LLMEE 148% in write operation and 270% with respect to the results of the unprotected system is minimal compared to the overhead of MEMSEC Plain framework (405% for write and 350%) for read and its cipher with lower latency MEMSEC Prince ECB (405% for wirte and 354%).

Regarding the bandwidth test, we could first evaluate that the LLMEE could be transparent since we managed to run both the operating system and the application without making functional modifications in both codes to make our system work. In other words, all transactions were encrypted and decrypted only by the system hardware, our LLMEE. As for the bandwidth results, we can conclude that, although tests like Neon read prefetched (where the fact of being able to enable AXI burst mode, which we cannot enable with our LLMEE) had results 28% lower MEMSEC, in other tests (standard memset, standard memcpy, and random fill) is the MEE with the best bandwidth. We can also see that all MEEs have a significant overhead compared to the unprotected system. We also could observe that our LLMEE reached about 65% the maximum achievable bandwidth ($< 25MB/s$ [26]) meanwhile MemEnc and MEMSEC only reached 42% and 12% respectably.

Chapter 6

Conclusions

This work focused on the performance improvement that would give us to include lightweight cryptography as the encryption core of an MEE, and, in this way, we could design an MEE that can be implemented on devices with limited resources, such as IoT devices.

One topic not addressed in this work is the security of our implementation because it requires testing the system against side-channel attacks like in [22]. However, such analysis would require more time and effort, which might be material for another research project.

Generally, good results were obtained, and it could be said that the objectives set at the beginning were fulfilled, since, regarding resources used in FPGA, our proposal uses about 2x less resources than the lighter option in the state-of-the-art that is MemEnc [13].

Regarding the latency tests, we could not compare it with MemEnc because its source code was unavailable to test it with our developed application, even though we obtained 2.7x less clock cycles overload in write operations and 1.3x less clock cycles overhead in read operations compared with the fastest and most lightweight version of MEMSEC (MEMSEC Prince ECB). We also managed to run an operating system without modifying the source codes of this application and running an application without running the source code of that application, demonstrating that our solution works transparently.

Compared with the other works, our design could improve the performance in the bandwidth test with the Tinymembench; this issue could be solved by addressing the cache coherency problem. Also, we can modify our design, adding the option to support burst mode to obtain better results in the testbench.

6.1 Reviewing Objectives

As for the overall goal, we managed to come up with a proposal for a hardware memory encryption engine based on ASCON, which is the most relevant algorithm currently within lightweight cryptography, demonstrating that our proposal uses only 48% of LUTs and 86% of registers of the lighter option that existed in the state of the art. We also showed that in terms of memory transactions, it requires up to 51% less clock cycle for writing and 18% fewer clock cycles for reading than the fastest option we could test, and although it does not have burst mode enabled or the cache coherency problem solved (cache must be disabled so that an operating system can be run) the bandwidth you got exceeded the state of the art options in the memset and memcpy tests.

The specific objectives set initially were to review the options of available lightweight cryptography algorithms and choose those that would provide us with better latency when implemented in a memory encryption engine; however, the fact that ASCON has been chosen as the winner of the NIST contest and is in the process of being standardized, forced us to focus our efforts on verifying the performance of this algorithm since it would not make as much sense to use other algorithms if, in the end, they would lose relevance both in the world of research and in that of the technological industry. However, even though we did not experiment much with other lightweight algorithms if we could observe which algorithms, like PRINCE or Skinny, could have a better performance than ASCON.

We were able to explore the design space for a memory encryption engine; we understood the different possibilities and features that these can have, as well as the different options available in both private industry and research, noting that there is still an area of opportunity in the design and implementation. There were very few solutions with which we could compare directly, and unfortunately, in many cases, it is not possible to have the source codes to test existing works, as is the case of [13].

Despite the few implementation references we could review, we were able to design a memory encryption engine based on a lightweight cryptography algorithm and implement it into a SoC. It was beneficial to work with all the tools that Xilinx provides us for its design and implementation since it covers all the habits, from the design in hardware with Vivado, applications to test our design in Vitis, and even a way to facilitate the design of a custom operating system for our solution with Petalinux.

6.2 Future work

The option to modify the module to support the cache and burst mode of AXI is open; in the same way, the proposal can be improved by adding an integrity tree, which would provide an additional security service to the design using the identical ASCON kernel to do the hashes.

Likewise, multiple tests against physical attacks are pending to see the system's security robustness and to design more test scenarios that demonstrate the security of our module and the feasibility of its implementation in an IoT scenario.

Since there are not many options to compare with, several instances of the module could be made using more standardized algorithms, even if they are not necessarily lightweight, to have more benchmarks.

Appendix A

Appendix

A.1 List of Acronyms

- IoT - Internet of Things
- MEE - Memory Encryption Engine
- NIST - National Institute of Standards and Technology
- LWC - Lightweight Cryptography
- ASCON - Authenticated Encryption and Associated Data
- SoC - System on Chip
- HDL - Hardware Description Language
- FPGA - Field-Programmable Gate Array
- RTL - Register-Transfer Level
- FME - Full Memory Encryption
- DRAM - Dynamic Random-Access Memory
- CPU - Central Processing Unit
- SME - Secure Memory Encryption
- DPA - Differential Power Analysis
- Intel SGX - Intel Software Guard Extensions

Bibliography

[1] Asif Ali Laghari, Kaishan Wu, Rashid Ali Laghari, Mureed Ali, and Abdullah Ayub Khan. A review and state of art of internet of things (iot). *Archives of Computational Methods in Engineering*, pages 1–19, 2021.

[2] Kinza Shafique, Bilal A Khawaja, Farah Sabir, Sameer Qazi, and Muhammad Mustaqim. Internet of things (iot) for next-generation smart systems: A review of current challenges, future trends and prospects for emerging 5g-iot scenarios. *Ieee Access*, 8:23022–23040, 2020.

[3] K Virgil English, Islam Obaidat, and Meera Sridhar. Exploiting memory corruption vulnerabilities in connman for iot devices. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 247–255. IEEE, 2019.

[4] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, may 2009. ISSN 0001-0782. doi: 10.1145/1506409.1506429. URL https://doi.org/10.1145/1506409.1506429.

[5] Shay Gueron. Memory encryption for general-purpose processors. *IEEE Security Privacy*, 14(6):54–62, 2016. doi: 10.1109/MSP.2016.124.

[6] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia che Tsai, and Raluca Ada Popa. An Off-Chip attack on hardware enclaves via the memory bus. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 487–504. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/lee-dayeol.

[7] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. Improving cost, performance, and security of memory encryption and authentication. *SIGARCH Comput. Archit. News*, 34(2):179–190, may 2006. ISSN 0163-5964. doi: 10.1145/1150019.1136502. URL https://doi.org/10.1145/1150019.1136502.

[8] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. Improving cost, performance, and security of memory encryption and authentication. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, page 179–190, USA, 2006. IEEE Computer Society. ISBN 076952608X. doi: 10.1109/ISCA.2006.22. URL https://doi.org/10.1109/ISCA.2006.22.

[9] Filippo Gandino, Bartolomeo Montrucchio, and Maurizio Rebaudengo. Tampering in rfid: A survey on risks and defenses. *Mobile Networks and Applications*, 15: 502–516, 2010.

[10] François-Xavier Standaert. Introduction to side-channel attacks. *Secure integrated circuits and systems*, pages 27–42, 2010.

[11] Yang Su and Damith C. Ranasinghe. Leaving your things unattended is no joke! memory bus snooping and open debug interface exploits. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 643–648, 2022. doi: 10.1109/PerComWorkshops53856.2022.9767390.

[12] Vishal A Thakor, Mohammad Abdur Razzaque, and Muhammad RA Khandaker. Lightweight cryptography algorithms for resource-constrained iot devices: A review, comparison and research opportunities. *IEEE Access*, 9:28177–28193, 2021.

[13] Naina Gupta, Arpan Jati, and Anupam Chattopadhyay. Memenc: A lightweight, low-power, and transparent memory encryption engine for iot. *IEEE Internet of Things Journal*, 8(9):7182–7191, 2021. doi: 10.1109/JIOT.2020.3040846.

[14] Michael Henson and Stephen Taylor. Memory encryption: A survey of existing techniques. *ACM Comput. Surv.*, 46(4), mar 2014. ISSN 0360-0300. doi: 10.1145/2566673. URL https://doi.org/10.1145/2566673.

[15] Mario Werner, Thomas Unterluggauer, Robert Schilling, David Schaffenrath, and Stefan Mangard. Transparent memory encryption and authentication. Cryptology ePrint Archive, Paper 2017/674, 2017. URL https://eprint.iacr.org/2017/674. https://eprint.iacr.org/2017/674.

[16] Sumit Singh Dhanda, Brahmjit Singh, and Poonam Jindal. Lightweight cryptography: a solution to secure iot. *Wireless Personal Communications*, 112:1947–1980, 2020.

[17] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. Present: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems-CHES 2007: 9th International Workshop, Vienna, Austria, September 10-13, 2007. Proceedings 9*, pages 450–466. Springer, 2007.

[18] Morris J Dworkin, Elaine B Barker, James R Nechvatal, James Foti, Lawrence E Bassham, E Roback, and James F Dray Jr. Advanced encryption standard (aes). 2001.

[19] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1. 2: Lightweight authenticated encryption and hashing. *Journal of Cryptology*, 34:1–42, 2021.

[20] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, et al. Prince–a low-latency block cipher for pervasive computing applications. In *Advances in Cryptology–ASIACRYPT 2012: 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings 18*, pages 208–225. Springer, 2012.

[21] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016.

[22] Thomas Unterluggauer, Mario Werner, and Stefan Mangard. Meas: Memory encryption and authentication secure against side-channel attacks. *Journal of cryptographic engineering*, 9:137–158, 2019.

[23] Akiko Inoue, Kazuhiko Minematsu, Maya Oda, Rei Ueno, and Naofumi Homma. Elm: A low-latency and scalable memory encryption scheme. *IEEE Transactions on Information Forensics and Security*, 17:2628–2643, 2022.

[24] ARM IHI 0022. *AMBA AXI Protocol Specification.*

[25] Hannes Gross, Erich Wenger, Christoph Dobraunig, and Christoph Ehrenhöfer. Suit up!–made-to-measure hardware implementations of ascon. In *2015 Euromicro Conference on Digital System Design*, pages 645–652. IEEE, 2015.

[26] UG585. *Zynq-7000 SoC Technical Reference Manual.*

[27] UG949. *Vivado Design Suite User Guide.*

[28] UG910. *Vivado Design Suite User Guide: Getting Started.*

[29] UG1144. *UltraFast Design Methodology Guide for FPGAs and SoCs.*

[30] Yalcin Tolga and Ghandali Samaneh. Need for low-latency ciphers: A comparative study of nist lwc finalists. NIST Lightweight Cryptography Workshop 2022, 2022.

List of Figures

List of Tables