DESIGN OF MEMORY ENCRYPTION AND AUTHENTICATION FOR SECURE IOT EDGE DEVICES

by

Recep Günay

B.S., Electrical and Electronics Engineering, Middle East Technical University, 2019

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Electronics Engineering
Boğaziçi University

2023

ACKNOWLEDGEMENTS

I would like to present my gratitude to my advisor, Assistant Professor Faik Başkaya, for his motivation and guidance along the duration of this thesis.

I want to give my thanks to my colleagues from TUTEL. Their encouragement and experience helped me gain the necessary skills to finish this thesis.

I also want to thank my undergraduate professors and friends from Autonymous group at ODTU where I acquired the skill to always continue learning in life.

I would also like to thank my family and especially my wife for their continuous support throughout this period.

ABSTRACT

DESIGN OF MEMORY ENCRYPTION AND AUTHENTICATION FOR SECURE IOT EDGE DEVICES

The security of computer systems has become very important as the Internet of Things (IoT) technology has improved and the number of electronic devices in our daily lives has increased dramatically. One particular weakness of these devices is the off-chip memory interface since they are easily accessible. They have been subject to various attacks focusing on this weakness such as cold-boot attack and replay attack. Most of the solutions in the literature try to solve this issue by memory encryption and memory authentication with high performance and high hardware cost using cryptography algorithms like AES and SHA.

A secure memory solution with memory encryption and authentication with low area and power consumption cost is designed in this thesis. ASCON, a finalist in the NIST lightweight cryptography standardization contest, is used for encryption and hash function. Using a single hardware block for both functions reduces the hardware cost with respect to the literature. A system on chip (SoC) is designed consisting of a secure memory controller with ASCON and metadata cache, key generation block with a built-in true random number generator (TRNG), and secure on-chip storage slots around the open-source RISC-V processor PICORV32. The performance and power costs of encryption and authentication are reduced by applying cache snoops during re-encryption and tree traversal. The SoC is designed in Verilog and implemented in FPGA for hardware verification. It has low area and power consumption overhead with reasonable storage overhead and acceptable performance reduction for IoT applications.

ÖZET

GÜVENLİ IOT UÇ CİHAZLARI İÇİN BELLEK ŞİFRELEME VE DOĞRULAMA TASARIMI

Neslenelerin interneti teknolojisinin gelişmesi ve günlük hayatımızda kullanılan elektronik cihazların artmasıyla birlikte bilgisayar sistemlerinin güvenliği büyük önem arz etmektedir. Bu cihazların önemli bir zayıf noktası kolay erişimi bulunan çip dışındaki bellekleridir. Soğuk önyükleme ve tekrar saldırıları gibi farklı saldırılar işlemci sisemlerinin bu zayıflığına odaklanmıştır. Literatürdeki çalışmaların büyük bir kısmı bu sorunu yüksek performanslı ve yüksek güçlü bir şekilde AES ve SHA gibi algoritmalar kullanarak bellek şifreleme ve doğrulama ile çözmeye uğraşıyor.

Bu tezin amacı, düşük alan ve güç kullanan bir güvenli bellek çözümü geliştirmektir. Bu tasarımda bellek şifreleme ve doğrulama bloklarının ikisi de kullanılmaktadır. Şifreleme ve doğrulama fonksiyonları için bir hafif kriptografi algoritması olan ve NIST yarışmasının finalisti olan ASCON kulanılmaktadır. İki ayrı fonksiyon için tek bir donanım bloğu kullanmak alan ve güç kullanımını düşürmektedir. Açık kaynaklı RISC-V işlemcisi PICORV32 etrafında güvenli bellek denetleyicisi, gerçek rastgele sayı üreteci bulunan bir anahtar üreticisi ve çip üzerinde güvenli saklama alanlarıyla beraber çip üzeri sistem tasarlanmıştır. Tekrar şifreleme ve doğrulama ağacı kontrolü sırasında önbellek okumaları yapılarak performans ve güç tüketimi iyileştirilmiştir. Sistem tasarımı Verilog ile yapılmış olup donanım doğrulaması FPGA üzerinde yapılmıştır. Makul bir performans düşüşü ve saklama alanı artışı ile düşük alanlı ve güç tüketimli bir sistem tasarlanmıştır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iii
ABSTRACT . . . iv
ÖZET . . . v
LIST OF FIGURES . . . viii
LIST OF TABLES . . . xi
LIST OF SYMBOLS . . . xii
LIST OF ACRONYMS/ABBREVIATIONS . . . xiii
1. INTRODUCTION . . . 1
2. BACKGROUND . . . 4
2.1. Attacks on Memory . . . 4
2.1.1. Cold Boot Attack . . . 4
2.1.2. Memory Bus Attacks . . . 6
2.1.2.1. Bus Snooping Attack . . . 6
2.1.2.2. Memory Tampering and Replay Attack . . . 7
2.2. Protection Schemes . . . 8
2.2.1. Protection against Data Theft . . . 9
2.2.1.1. Memory Scramblers . . . 9
2.2.1.2. Direct Memory Encryption . . . 10
2.2.1.3. Counter Mode Encryption . . . 11
2.2.2. Protection against Tampering . . . 13
2.2.2.1. Integrity Check by Hash Generation . . . 13
2.2.2.2. Integrity Check by Hash Tree . . . 15
2.2.3. Literature Review on Memory Encryption and Integrity . . . 16
2.3. Cryptography Algorithms . . . 18
2.3.1. Symmetric Cipher Algorithms . . . 19
2.3.2. Hash Algorithms . . . 21
2.3.3. Lightweight Algorithms . . . 26
3. DESIGN METHODOLOGY . . . 29

3.1. Secure Memory Controller 30
3.1.1. Encryption Controller 32
3.1.2. Integrity Check Controller 35
3.1.3. Metadata Cache and ASCON 38
3.2. Key Management 40
3.2.1. True Random Number Generator 40
3.3. On-chip Secure Data Storage 43
3.4. RTL Design and Verification 44
4. HARDWARE IMPLEMENTATION 47
4.1. FPGA Implementation 47
4.2. ASIC Implementation 51
5. EXPERIMENTS AND RESULTS 55
5.1. Experiment Setup 55
5.1.1. Software Environment 56
5.1.2. FPGA Setup 57
5.1.3. Security Tests 57
5.2. Evaluation Criteria 59
5.3. Experimental Results 61
6. CONCLUSION 68
REFERENCES 70

LIST OF FIGURES

Figure 2.1. DRAM cell schematic. 5

Figure 2.2. Cold-boot attack flowchart. 6

Figure 2.3. Bus snooping attack diagram. 7

Figure 2.4. Active attacks on the memory bus. 8

Figure 2.5. Memory scrambler block diagram. 10

Figure 2.6. Direct memory encryption method block diagram. 11

Figure 2.7. Direct memory encryption waveform. 11

Figure 2.8. Counter mode memory encryption block diagram. 12

Figure 2.9. Counter mode memory encryption waveform. 12

Figure 2.10. Integrity check by hash generation. 14

Figure 2.11. Integrity tree diagram. 16

Figure 2.12. Flowchart of the AES algorithm. 22

Figure 2.13. Digital signature with hash function. 23

Figure 2.14. Security requirements of hash functions. 24

Figure 2.15. ASCON authenticated encryption block diagram. 28

Figure 2.16. ASCON hashing block diagram. 28

Figure 3.1. Proposed secure SoC block diagram. 29

Figure 3.2. Secure memory controller block diagram. 31

Figure 3.3. The encryption controller block diagram. 33

Figure 3.4. The seed for split counter mode encryption. 34

Figure 3.5. Message block for data and counter/hash block. 36

Figure 3.6. Integrity check controller block diagram. 36

Figure 3.7. Bonsai merkle tree chart. 38

Figure 3.8. Modified tetrahedral oscillator schematic. 41

Figure 3.9. Proposed random number generator schematic. 42

Figure 3.10. Random number generator signal waveforms. 42

Figure 3.11. Secure on-chip storage block diagram. 43

Figure 4.1. The layout of the FPGA implementation. 50

Figure 4.2. FPGA implementation power consumption analysis. 51

Figure 4.3. Power consumption analysis after the synthesis in TSMC 65 nm. . 53

Figure 5.1. Full zero data values. 58

Figure 5.2. Encrypted data values. 59

Figure 5.3. Hardware cost overhead vs. performance slowdown graph. 63

Figure 5.4. Storage overhead vs. hash size. 64

Figure 5.5. Coremark results for different versions. 64

Figure 5.6. Execution time for different workloads. 65

Figure 5.7. Metadata cache size vs. performance and bus traffic graph. 66

LIST OF TABLES

Table 4.1. FPGA implementation hardware resources vs. security schemes. . 49

Table 4.2. FPGA implementation power analysis. 51

Table 4.3. Synthesis results in TSMC 65 nm technology. 52

Table 5.1. The storage size of each data type 60

Table 5.2. Comparison of the proposed work with literature. 61

Table 5.3. NIST 800-22 test suite results. 66

LIST OF SYMBOLS

$\mathrm{GF}(p^k)$	Galois Field with order p^k
V_T	Threshold voltage
$\oplus$	XOR operation

LIST OF ACRONYMS/ABBREVIATIONS

3DES	Triple Data Encryption Standard
AES	Advanced Encryption Standard
AEAD	Authenticated Encryption with Associated Data
AMD	Advanced Mico Devices
ASIC	Application Specific Integrated Circuit
B	Byte
BRAM	Block RAM
CBC	Cipher Block Chaining Mode
CTR	Counter Mode
DDR	Double Data Rate
DES	Data Encryption Standard
DRAM	Dynamic Random Access Memory
ECC	Error Correction Code
EFF	Electronic Frontier Foundation
FIPS	Federal Information Processing Standards
FIFO	First-in First-out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GB	Giga-Byte
GDSII	Graphic Design System
GF	Galois Field
GE	Gate Element
HDL	Hardware Description Language
HVT	High Threshold
IBM	International Business Machines
ILA	Integrated Logic Analyzer
IoT	Internet of Things
IP	Intellectual Property

ISA	Instruction Set Architecture
IV	Initialization Vector
KB	Kilo-Byte
L2	Level Two
LEF	Library Exchange Format
LFSR	Linear Feedback Shift Register
LIB	Liberty Timing File
LWC	Lightweight Cryptography
LUT	Look-up Table
LVT	Low Threshold
MAC	Message Authentication Code
MB	Mega-Byte
MD4	Message Digest Algorithm Four
MD5	Message Digest Algorithm Five
MMU	Memory Management Unit
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
NIST	National Institute of Standards and Technology
NSA	National Security Agency
PC	Program Counter
PHY	Physical Layer
PMP	Physical Memory Protection
PRNG	Pseudo Random Number Generator
RISC-V	Reduced Instruction Set Computer Five
RAM	Random Access Memory
ROM	Read Only Memory
RSA	Rivest-Shamir-Adleman
RTL	Register Transfer Level
SAIF	Switching Activity Interchange Format
S-box	Substitution Box
SCA	Side Channel Attack
SGX	Secure Guard Extension

SHA	Secure Hash Algorithm
SME	Secure Memory Encryption
SoC	System on Chip
SPICE	Simulation Program with Integrated Circuit Emphasis
SPI	Serial Peripheral Interface
SPN	Substitution–Permutation Network
SRAM	Static Random Access Memory
STA	Static Timing Analysis
TCF	Toggle Count Format
TRNG	True Random Number Generator
TSMC	Taiwan Semiconductor Manufacturing Company
UART	Universal Asynchronous Receiver Transmitter
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
WNS	Worst Negative Slack
XOM	Execute Only Memory

1. INTRODUCTION

Electronic devices have become really widespread in the last two decades with cheaper and more sophisticated chip manufacturing technology. A new information age started because these devices have become integral to everyday life, from robot vacuum cleaners to cars under the Internet of Things (IoT) tree. Most of these devices have a connection to the Internet and process and store personal data. Therefore, the security of these devices is of paramount importance since they are usually accessible by many people.

The security of computer systems was traditionally threatened on the software side since the hardware was more difficult to attack. Therefore, designers selected a performance and power-focused design approach, which led to hardware vulnerabilities as many hardware-based attacks on processor systems emerged in the last decade. For example, attacks like Spectre and Meltdown can obtain the data stored in the memory that are not available to the program by exploiting the speculative execution property of the processors. Some attacks even tried to probe inside the chip to obtain the information stored in on-chip memories.

A more vulnerable target of the attacks is the main memory of the computers since the interface between the processor, and the memory is usually accessible on the board. Hence, malicious users can exploit this interface to steal user data or break the execution of the system. Examples of the attacks on the off-chip memory are cold-boot attacks, eavesdropping attacks, tampering attacks and replay attacks. The user data is stolen with a cold-boot attack by inserting the DRAM into another system after being cooled down and with an eavesdropping attack by observing the accessible address and data bus. The processor is fed false data with tampering and replay attacks to break the execution.

Different methods have been proposed to protect the off-chip memory of processor systems against these attacks. Memory scramblers were designed to obfuscate the data, but they proved ineffective against dedicated attackers. The only way to prevent data theft is to encrypt the data with a strong cipher like Advanced Encryption Standard (AES). Direct encryption and counter mode encryption methods have been proposed where the data is directly encrypted with the cipher for the former. In contrast, a stored counter is encrypted with the cipher to XOR it with the data for the latter. In order to prevent tampering and replay attacks, memory authentication methods have been proposed where hash values are generated for data blocks and stored in the memory. An integrity tree is constructed for stopping replay attacks where multiple levels of hashes protect the lower levels.

Most of the literature work focuses on improving these methods' performance for high-performance applications. They employ AES for encryption and the SHA family for hash functions, both of which have a high hardware cost. Therefore, there is a need for a lightweight solution to protect the off-chip memory of computer systems. Lightweight cryptography algorithms have been designed to reduce the cost of these algorithms while remaining secure. The National Institute of Standards and Technology (NIST) started a competition to standardize the new authenticated encryption algorithm in 2013.

In this thesis, we design a lightweight secure memory solution that protects the off-chip memory against data theft attacks, tampering attacks, and replay attacks with low area and power consumption overhead while having a reasonable performance slowdown. We use a split-counter mode encryption method to reduce the latency and storage overhead of encrypting the data. The memory is authenticated by generating small but sufficiently secure length hash values for data blocks while generating an integrity tree design that constructs the tree over the encryption counters to reduce the storage and latency overheads. We design a complete system-on-chip (SoC) based on a RISC-V processor, which has a built-in key generator with a true random number generator (TRNG) and secure on-chip storage. We implement the design on an FPGA to verify

the design in the hardware and run performance measurements. We also perform the ASIC synthesis in TSMC 65 nm technology for area and power consumption metrics while finishing the IP design flow. In summary, the contributions of this thesis are as follows:

- A single cryptography hardware, NIST finalist ASCON, is used to implement memory encryption and authentication to reduce the area overhead of the method. Using ASCON for encryption and hash function resulted in a significant decrease in the hardware cost compared to the literature.
- The data cache is checked during the re-encryption period to see if the corresponding block is already on the chip to reduce the number of memory accesses. If a data block is in the data cache, it is not fetched again and re-encrypted with the cipher since inside the chip is considered secure. Lowering the number of memory accesses during re-encryption decreased power consumption due to the decrease in memory bus activity.
- Integrity tree traversal checks the metadata cache for on-chip hash nodes. If a hash node is in the cache, the tree traversal stops since the on-chip hash proves the integrity of the data. Lowering the latency of integrity tree traversal resulted in a decrease in power consumption while increasing the performance.
- A complete system-on-chip is designed, which combines off-chip security with secure on-chip storage and built-in key generation. The secure region on the chip is used for the most critical user data, which is stored by being encrypted and signed with ASCON. The key generator also has a novel TRNG with low hardware cost.

The construction of the thesis starts by first explaining the fundamental background for the topics in this thesis in Chapter 2. Chapter 3 presents the design methodology with details of the proposed work. The hardware implementation details are given in Chapter 4. Chapter 5 explains the experimental setup for the proposed design and presents the experimental results by comparing them with the literature. The thesis is concluded by summarizing the contributions in Chapter 6.

2. BACKGROUND

The security of a computer system can only be guaranteed by the protection of both hardware and software. The system's integrity is compromised if either of the two is not secure. Therefore, modern systems employ multi-layered security measures at both hardware and software levels. However, until the last decade, manufacturers focused on the software side of the system in terms of security since many attacks using the software vulnerabilities resulted in incidents like online theft, personal information leaks, political crisis, and catastrophic accidents. Attackers preferred the software-based attacks since it was easier to implement without physical access to the system. Therefore, computer system designers improved vulnerabilities against software attacks and focused on performance and power during the hardware design process because they assumed the hardware was secure.

The last decade proved that hardware is also far from being secure because there have been plenty of attacks on the hardware, most of them using a side channel in the system to retrieve information. The hardware attacks focused on the vulnerabilities in the memory since all data are kept in memory, and the memory is usually outside the processor chip. This section presents some of the most well-known attacks on the memory of computer systems.

2.1. Attacks on Memory

2.1.1. Cold Boot Attack

A widely known attack on the memory is the cold boot attack, where attackers remove the memory chip from the socket at cold temperatures and immediately connect to another system to steal the user data by dumping the contents. This attack is

performed by exploiting the retention of information in the DRAM cells. The circuit diagram of a DRAM cell is given in Figure 2.1. DRAM cells are refreshed to retain their data because DRAMs lose their content in milliseconds by static leaks when the power is on and immediately after it is turned off. However, this is valid for only a fraction of the DRAM cells because a majority of the cells actually retain their content for seconds with either power on or off. In [1–3], it was shown that this retention behaviour of DRAMs could be exploited to steal sensitive information. Furthermore, information such as passwords and cryptography keys are stored in DRAM in systems with disk encryption, assuming immediate data loss after DRAM disconnection or system reboot.

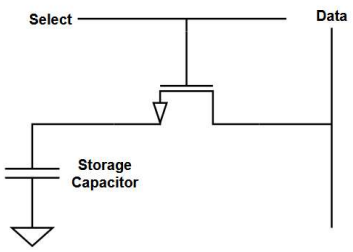

Figure 2.1. DRAM cell schematic.

The cold boot attack is performed in the following steps. First, the victim software is run on a victim system with a DDR2/3/4 DRAM module connected until the sensitive information is stored in the DRAM. Then, the DRAM chips are cooled down to increase the retention time of DRAM cells since capacitors lose their charge much more slowly at low temperatures. After cooling the chips down to a sufficient temperature, the DRAM is disconnected from the victim system and connected to the attacker's system. Finally, the DRAM's remaining contents are dumped with the help of software. These steps are also illustrated in Figure 2.2.

Halderman et al. proved the practicability of this attack by stealing 99.9% of the contents from DDR and DDR2 RAMs by cooling them with an air-compressor [4]. Different researchers later reproduced it on many other platforms with DDR2 and

DDR3 RAMs. Implementing the attack became more difficult with the advancement of memory scrambling methods. However, Yitbarek et al. performed the attack to steal DDR3 and DDR4 contents from systems that employ memory scrambling [3].

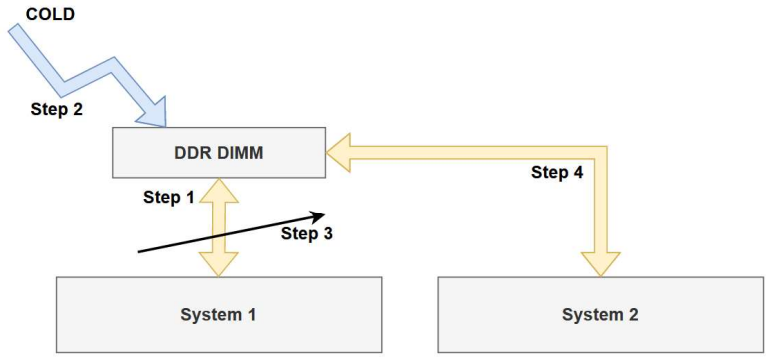

Figure 2.2. Cold-boot attack flowchart.

2.1.2. Memory Bus Attacks

Another type of attack happens when the bus between the processor and the memory chip is physically available. There could be two possible attacks; one where the attacker can only observe the bus passively and one where the attacker can both observe and alter the travelling signals.

2.1.2.1. Bus Snooping Attack. The bus snooping attack, or eavesdropping attack, is an attack with a passive attacker where attackers can snoop the memory bus to observe user data transfers. Since the attacker is able to observe the values change over time, this attack is more dangerous than the cold boot attack. Furthermore, determining multiple versions of specific data makes it possible to crack certain encryption schemes. The bus snooping attack is demonstrated with the block diagram in Figure 2.3, where the attacker is depicted between the SoC and DRAM with a unidirectional arrow representing that it can only read the bus.

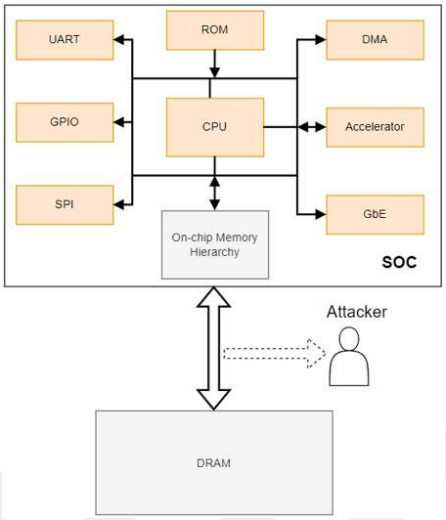

Figure 2.3. Bus snooping attack diagram.

Su and Ranasinghe demonstrated the danger this attack poses by acquiring all the content of the off-chip memory by snooping the bus interface [5]. Unlike Su and Ranasinghe, Lee et al. performed a bus snooping attack by observing the address bus and creating a side channel [6]. The bus snooping attack can also be used to steal the memory contents of a system with a weak encryption scheme by collecting different ciphertext samples and applying a known plaintext attack.

2.1.2.2. Memory Tampering and Replay Attack. An attacker with physical access to the system can also actively change the memory contents by tampering, or they can return wrong responses on the bus to break the execution of the processor. Passive attacks can be prevented by encrypting the memory contents, whereas memory authentication is also required to prevent active attacks. The need for memory authentication is why tampering with memory is a hazardous attack, and the cost of protection is usually too high. The memory contents can be changed by either physically tampering with the chip or software attacks such as rowhammer [7]. Figure 2.4 illustrates the mechanism of active attacks like tampering and replay attacks by showing that the attacker can change the contents of the RAM or the response from the RAM.

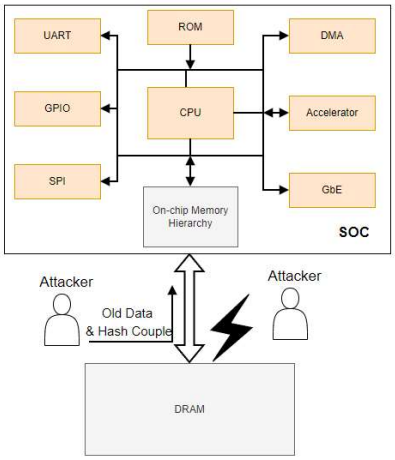

Figure 2.4. Active attacks on the memory bus.

Data integrity can be protected by an authenticated encryption scheme writing a message authentication code (MAC) together with the data to the memory. However, the replay attack can even disrupt the execution of a system with an integrity check mechanism by returning old data and MAC couples since the MAC algorithm will generate the same MAC for the same data, and the old values will go unnoticed. Gassend et al. performed a replay attack exploiting the integrity check mechanism of the XOM architecture [8]. Integrity protection for MACs is also needed to prevent a replay attack, resulting in the highest cost in terms of area and performance.

2.2. Protection Schemes

Many studies have been published that try to protect the off-chip memory of computer systems against various attacks mentioned previously. The protection methods can be divided into two categories. The first type of security scheme protects the memory contents against data theft. In contrast, the second type prevents the system from receiving incorrect memory content, which may make the system vulnerable.

2.2.1. Protection against Data Theft

Most of the attacks focus on stealing the contents of the memory. Therefore, a secure memory solution should protect against data theft. This section explains different methods of protecting the data from being stolen.

2.2.1.1. Memory Scramblers. The easiest method to protect data in off-chip memory from being stolen is to employ a memory scrambler. A memory scrambler obfuscates data to be written by XORing with a pseudo-random number generated according to the address of the data. When the same data is read back from memory, the same pseudo number is generated and XORed with the obfuscated data to obtain the original data. The working principle of memory scramblers is shown in Figure 2.5.

Older DDR and DDR2 systems did not use memory scramblers and stored the content in memory as plaintext, which made it easier to perform the cold-boot attacks. The memory scramblers were first introduced to reduce power supply noise and improve signal integrity of the high-speed memory bus [9]. By making the bit flips of the data bus uniform, current fluctuations are reduced, and the signal integrity is improved [10]. Furthermore, it was later discovered that these scramblers also provided obfuscation for off-chip data, and they prevented simple cold-boot attacks.

Earlier scrambler designs used LFSRs, but their designs have become more complex for later generations of DDR systems. However, even with more complex and secret scrambler designs, the system memory was not fully protected against cold-boot attacks. Indeed, Yitbarek et al. demonstrated that cold-boot attacks were still a threat by attacking DDR3 and DDR4 systems with complex scrambler designs and stealing the contents of the memory [3]. They perform a reverse cold-boot attack to obtain the scrambler keys by first writing all zeros to the memory in a system without a scrambler and then reading the contents of the memory in a system with a scrambler, resulting in reading the scrambler keys directly.

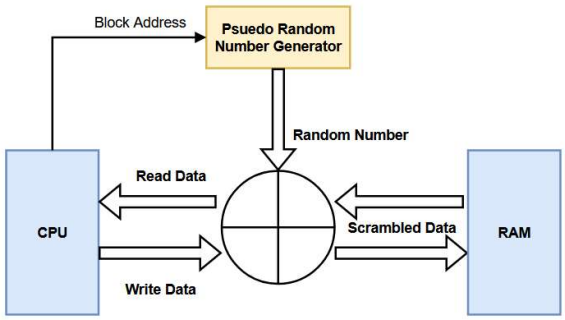

Figure 2.5. Memory scrambler block diagram.

2.2.1.2. Direct Memory Encryption. The straightforward way of providing data confidentiality is by encrypting the memory blocks directly when writing to memory. The data needs to be decrypted using the same algorithm and key while reading the ciphered data from memory. Memory modules can be protected from data theft with a strong cipher algorithm like AES since it is secure against analytical attacks. The block diagram of the direct memory encryption method is shown in Figure 2.6 where the same key is used for encryption and decryption.

AMD uses the direct memory encryption method to provide security for off-chip memory [11] in its Secure Memory Encryption (SME) architecture. SME uses AES with a 128-bit key for encryption and decryption of selected secure pages. The secure pages are determined by an encryption bit in the page table entries. The key is generated upon booting and stored in the MMU.

Using a strong cipher to encrypt the memory contents proved secure against data theft attacks like cold-boot and eavesdropping attacks. However, adding the latency of encryption and decryption algorithm to the path of DRAM access exacerbates the memory bottleneck of computer systems in direct encryption methods. The latency of decryption in an example data transfer is illustrated in Figure 2.7 as a waveform. The latencies of DRAM access and encryption algorithm, both of which are high, are added

to each other, reducing the processor's performance significantly. Multiple cipher blocks can be used in parallel and can be designed as unrolled to reduce the latency of encryption as implemented by Rambus in their total disk encryption designs [12]. However, this results in a considerable hardware cost and power consumption, which is unsuitable for low-cost applications.

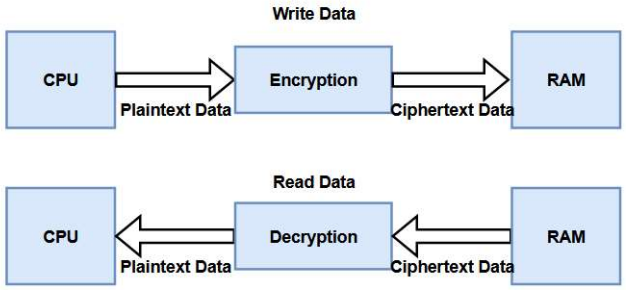

Figure 2.6. Direct memory encryption method block diagram.

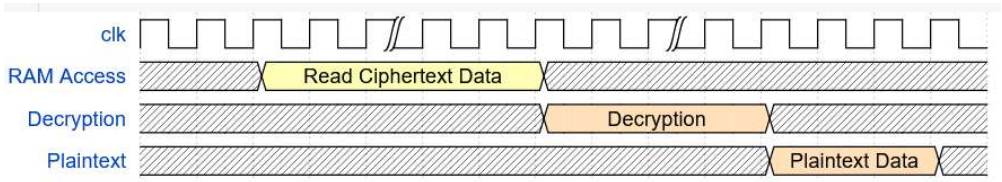

Figure 2.7. Direct memory encryption waveform.

2.2.1.3. Counter Mode Encryption. Another method of encrypting the memory contents is counter-mode encryption, where a counter is stored in the memory for every data block. A seed is generated using the corresponding data block's address and counter. The seed is encrypted using a strong cipher algorithm to produce a one-time pad when there is a transfer request. XORing the pad with the plaintext and ciphertext applies encryption and decryption, respectively, as shown in Figure 2.8.

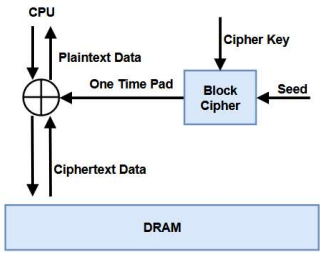

Figure 2.8. Counter mode memory encryption block diagram.

The security of this method comes from unique counters as they are incremented for every write operation [13, 14]. The counter of the corresponding block is concatenated with the address and an initialization vector (IV) to create the seed. This seed is encrypted with the cipher algorithm until the DRAM response arrives, as shown in Figure 2.9. By hiding the latency of the encryption algorithm behind the latency of the DRAM response during the read requests, this method performs better than the direct encryption method. Although this method performs better than the direct encryption method, it creates other problems.

One of the problems concerning counter mode encryption is the memory storage overhead since a counter is stored for every data block. Intel keeps a 56-bit counter for every 512-bit data block incurring 10.9% storage overhead [15]. The other problem is the re-encryption of the memory in case of counter overflow since the pads need to be unique in the system's lifetime, which requires an entire memory re-encryption with a new key. Therefore, the counter size should be large enough to prevent frequent overflow of counters and small enough to reduce the storage overhead.

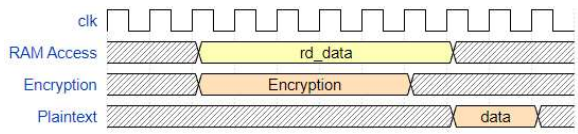

Figure 2.9. Counter mode memory encryption waveform.

There have been many works to solve the problems of counter-mode encryption of memory in the literature. One of the solutions to reduce the latency of counter access was to store frequently accessed counters in a cache on the chip [15,16]. Finding the counters in the cache eliminates the access latency of the counters. Shi et al. proposed a different solution to eliminate the counter access latency by predicting and pre-computing the counters on data access [14]. If the prediction is wrong, the latency is the same, but it is eliminated if the prediction is correct. Yitbarek and Austin modified the representation of the counters to store a base counter for a 4KB page and only store the delta differences of individual blocks from the base [17]. They were able to reduce the storage overhead of the counters in this way.

2.2.2. Protection against Tampering

Some attacks try to break the execution of the system by tampering with the memory contents. The computer system should know for sure that the data it receives is correct to be completely secure against these attacks. Different methods of protecting the system against tampering attacks are listed in this section. The first method is to create hashes of data, store them in the memory and compare the stored hash value with the newly generated one to determine if the data has changed. The other method is to create a hash tree where the root of the tree is secure on the chip to protect against replay attacks.

2.2.2.1. Integrity Check by Hash Generation. One method of checking the integrity of the memory contents is to create hashes for each data block and store them in the memory. Therefore, when an attacker returns a different value on a read request or tampers with the data, the processor can check the integrity of the data by reading the hash from memory and comparing it with the newly generated hash value from the data. If the two hash values do not match, the integrity check for the data fails. The system should take the necessary action upon a failure. The procedure for checking the integrity of memory contents is illustrated in Figure 2.10.

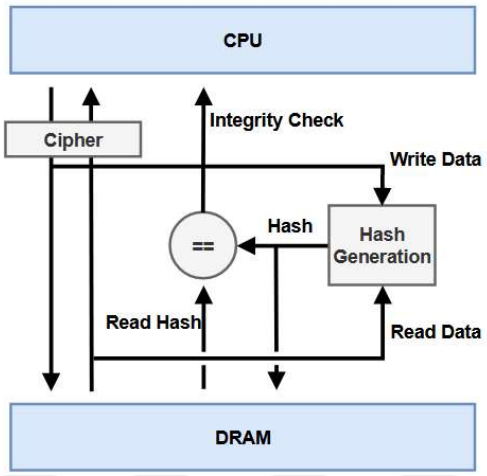

Figure 2.10. Integrity check by hash generation.

No unauthenticated data is sent to the processor using this method. The hash is computed using the ciphertext data; hence, it is generated after encryption during write and before decryption during read, as seen in Figure 2.10. This method has three overheads on the system. The first is the storage overhead since we store a hash for every data block. The latency overhead is the second since the hash is also brought from memory on a data read request. The third is the area overhead since the hardware implementation of secure hash algorithms occupies large areas.

The most preferred hash length is 128 bits for each 512-bit data block resulting in a storage overhead of 25%. Therefore, most of the works in the literature focus on reducing storage overhead. Intel uses 56-bit hashes for 512-bit data blocks decreasing the storage overhead down to 10.9%, and they proved that this length is sufficiently secure for the memory integrity applications [10]. Taassori et al. improve the Intel SGX architecture by compressing the cache blocks and storing the 56-bit hashes along with the cache blocks [18]. Furthermore, they implemented a hash-sharing algorithm to allocate one hash for 4 or 8 cache blocks. They were able to reduce the storage overhead from 10.9% to 3.1%. Some works use the ECC memory chips for hash storage to reduce the storage overhead and bring hashes simultaneously with data [17, 19]. Köylü et al.

designed an embedded memory security scheme for IoT devices, and they used 64-bit hashes for 512-bit data blocks and implemented a lightweight hash algorithm to reduce the area overhead [20].

2.2.2.2. Integrity Check by Hash Tree. Hash computation for each data block provides an integrity check against basic tampering attacks where each hash protects the integrity of each data block. However, an attacker can store the previously sent ciphertext data and hash pairs and return them on a read request. The system will generate the same hash value for that data, and the integrity check will not fail, but the processor will get the wrong data. In order to protect against this attack, integrity tree designs were proposed where multiple levels of hashes form a tree, and the tree's root stays on the chip. The root hash protects all the data and hashes while each level protects the next level of hash and data [8]. The algorithm updates all levels of the tree up to the root on each write. All levels are traversed to verify data and hash integrity. The system tracks all the changes to the data and achieves complete integrity protection in multiple steps with this method.

A Merkle tree authenticates data blocks against replay attacks [21]. In a Merkle tree, a hash, which is stored in a memory block, protects leaf-level data, and a hash protects this block on another level. If n hash values fit in a memory block, it results in an n-ary tree where the root is stored in on-chip storage secure against tampering, as shown in Figure 2.11. The storage overhead gets even worse than the basic integrity check mechanism by hash generation since now upper levels of the tree protect the hash blocks. Furthermore, the performance is significantly affected since all the levels of the tree should be traversed for each memory access.

Some implementations tried to reduce the latency overhead by caching the hash blocks in the tree on-chip since the spatial locality of hashes is high [8, 15, 22]. Taassori et al. tried to reduce the storage and traversal latency overhead of integrity trees by making the "arity" of the tree variable [18]. Saileshwar et al. also reduced the storage

overhead of the tree by increasing the "arity" of the integrity tree, which results in a smaller number of levels [23]. Guo et al. made two contributions to reduce the performance overhead of integrity trees; i)making the memory accesses along a tree path parallel, ii)creating a prefetcher algorithm aware of the integrity tree [24].

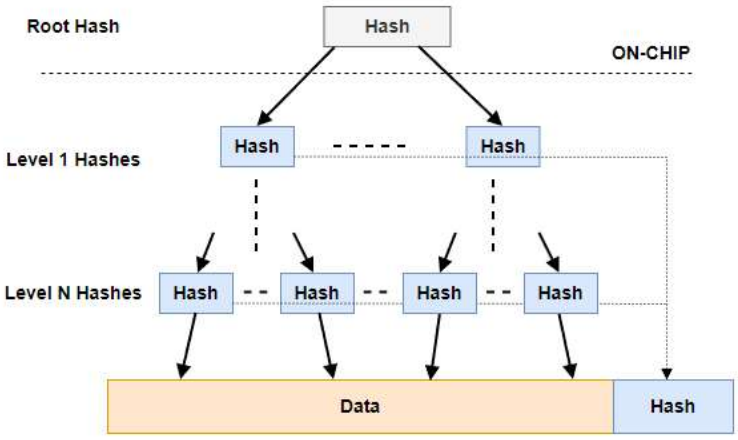

Figure 2.11. Integrity tree diagram.

2.2.3. Literature Review on Memory Encryption and Integrity

The total security of the memory can only be achieved by combining the previously mentioned protection schemes. Therefore, many researchers have implemented processor systems with memory encryption and integrity. The research's main focus is to provide absolute security against known and other possible attacks. However, since integrating these security solutions into a system incurs overheads in terms of performance, area, memory storage and power, another primary focus is on reducing either one of these overheads or multiple without compromising security.

Gassend et al. proposed a secure memory scheme to protect the memory against tampering attacks without memory encryption. They implemented a Merkle hash tree with the root hash on the chip against replay attacks and stored the hashes in the L2

cache to reduce the gap between verification and performance. Furthermore, they returned the read data to the core before its verification was finished to increase the performance [8]. However, storing the hashes in the L2 cache results in a loss of performance in the replacement policy of the cache, and speculatively using non-verified data by the core may result in vulnerabilities.

Intel implements counter-mode memory encryption and authentication by a hash tree for certain enclaves in the main memory in their SGX architecture. The encryption is performed using 128-bit AES, and they keep 56-bit counters and 56-bit MACs for 512-bit data blocks [15]. They also implement an integrity tree upon the encryption counters. Taassori et al. proposed an optimization on the SGX by implementing a variable arity integrity tree to reduce the storage overhead and increase performance. Additionally, they compressed the cache blocks to store the hash values in one 512-bit space, and some shared a hash value to reduce the storage overhead [18]. Zhang et al. proposed a memory protection scheme for multiprocessor systems where the encryption and authentication are performed on the system bus, and they used AES-CBC for encryption [25].

When they introduced split-counter mode encryption, a significant improvement in memory encryption was made by [22]. In this method, a 7-bit minor counter is kept for each 512-bit block while a 64-bit major counter is kept for a 4-KB page which contains 64 blocks; thus, 64 minor counters and one major counter create a cache block. They were able to reduce the storage overhead and re-encryption overhead significantly. They also implemented an integrity tree to prevent replay attacks. A similar encryption method was implemented by [17], which kept 64-bit base counters for 4-KB pages and only the 7-bit differences for each 512-bit block. While [22] formed the seed by concatenating the major and minor counters, [17] formed it by adding the base counter and the delta counter. [17] also reduced the storage and latency overhead of hashes by storing them in ECC chips of DRAMs. [19] used a similar method and stored the hashes in ECC chips by combining integrity and error correction algorithms

to have both reliability and integrity. The morphable counter method adjusted the size of the minor counters according to their usage and stored 128 minor counters in one cache block to reduce the storage overhead and the size of the integrity tree [23].

Replay attack can be countered by constructing an integrity tree [8, 22]. A new method called Bonsai Merkle Tree was proposed by [26], which reduced the size of the integrity tree significantly by constructing it not over all of the data but only on the counters of the encryption algorithm. This method indirectly protects the integrity of the data by including the corresponding counter while generating the first level of hash in the integrity tree constructed over the counters. Thus, they could reduce the integrity tree's storage and latency overhead without any security loss, which caused the use of this method by most of the later works [15, 18, 23].

2.3. Cryptography Algorithms

Cryptography is the science of keeping the meaning of a message secret. It can be divided into two main algorithms; ciphers and hash functions. Ciphers encrypt and decrypt the message while the hash functions protect the integrity of the message [27]. Ciphers are divided into symmetric and asymmetric ciphers where symmetric ciphers use the same key for both encryption and decryption, and asymmetric ciphers use key pairs for encryption and decryption. They can also be divided into stream and block ciphers where stream ciphers encrypt a stream of data while block ciphers encrypt a block of data. In this work, we will focus on symmetric block ciphers since data reside in memory as blocks, and there is only one master agent, the processor. The rest of this section explains different block cipher and hash algorithms.

2.3.1. Symmetric Cipher Algorithms

Symmetric cipher algorithms protect the channel between two users by sending the messages encrypted using a particular key and decrypting the message using the same key on the receiving end, where some malicious users can access the channel. The original message is called the plaintext, and the encrypted message is called the ciphertext. The decryption process is the inverse of encryption, and the key should always be kept secret. Therefore, the key should be shared on a secure channel.

The substitution cipher has been used in history for various reasons, where each letter is replaced with a different letter. It served its purpose for its times, but it was not a very secure algorithm since it could easily break under simple brute force or letter frequency attacks. Until 1972, a cryptography system's security depended on the algorithm's secrecy. In 1972, the US National Bureau of Standards (NBS) requested proposals for a standard cipher algorithm that could be used for different applications. IBM proposed the most promising candidate algorithm, and it was based on the cipher Lucifer designed by Horst Feistel. After some modifications by the National Security Agency (NSA), the Data Encryption Standard (DES) algorithm was created, which encrypts 64 bit-block of data by a 56-bit key [27].

The DES algorithm consists of 16 rounds performing the same operation on each 64-bit block of data. Different subkeys, derived from the input key, are used for each round. Each round's operations provide confusion and diffusion. The structure of the rounds is based on Feistel networks, where encryption and decryption are identical. The reverse key schedule is the only difference between encryption and decryption in the DES algorithm. Therefore, the software and hardware implementations are easy to design. The DES algorithm consists of an initial and a final permutation, DES rounds, an f-function and a key schedule. Only the f-function was kept secret when the DES algorithm was made public, which consists of an expansion block, an XOR with a key, 8 S-boxes and a permutation [28].

Analytical attacks or exhaustive key search attacks can attack ciphers. It has been proven that DES is sufficiently secure against analytical attacks such as differential analysis. However, a well-designed key-search attack can practically break the DES algorithm due to the small size of the key space of DES. Indeed, the Electronic Frontier Foundation (EFF) built the Deep Crack, which performed a successful brute-force attack on DES in 15 days and costed $250.000, in 1998 [27]. It was the official proof that the DES algorithm was not secure against well-equipped attackers and should not be used for applications with high lifetimes. Therefore, NIST started a new competition to create a new encryption standard resulting in the Advanced Encryption Standard (AES). Some variations of DES are still used nowadays, such as 3DES, where the DES algorithm is applied three times.

NIST opened the proposals for AES in 1997, which was open to the public and supervised by NIST, unlike DES. The international scientific community evaluated proposed algorithms in terms of advantages and disadvantages and announced that the block cipher Rijndael was the new AES. The block size of AES is 128 bits, and the key has different sizes, 128, 192 and 256 bits, as they were the constraints by NIST. AES consists of 10, 12, and 14 rounds for key lengths 128, 192 and 256 bits, respectively, and each round consists of different layers that provide confusion and diffusion, as shown in Figure 2.12. AES is not a Feistel cipher; each round encrypts the 128-bit block. Three layers in a round of AES are the key addition layer, the byte substitution layer and the diffusion layer, which consists of 2 sublayers; ShiftRows and MixColumns [29].

The byte substitution layer is made up of 16 identical parallel S-boxes, which map an input byte to a different byte, and it is the only nonlinear element of AES that provides confusion. The ShiftRows sublayer shifts each row of the block by different amounts where the second row is cyclically shifted to the left by 1 byte, the third by 2 bytes, and the fourth by 1 byte, while the first row is not shifted. The main diffusion element of AES is the MixColumns sublayer, where each column is mixed with all the other columns. The key addition is a simple XOR operation with the current key which

the key schedule block generates. The decryption is the exact inverse of the encryption, where all the layers and the key schedule are inversed. The initial key addition layer is moved to the end of the rounds for decryption [29].

Currently, no analytical or brute force attacks are practically dangerous for AES. It requires more hardware resources than DES. Through parallelization and improving ASIC technology, hardware implementations can reach up to 10Gb/s of throughputs [27]. However, some implementations are vulnerable to side-channel attacks, such as differential power analysis. Furthermore, AES requires very high costs for hardware implementations, which resulted in a need for lightweight cipher algorithms.

Various proposals for cipher algorithms for lightweight applications require low hardware cost and power consumption. PRESENT and Prince, permutation-based ciphers, are two of these algorithms. PRESENT consists of 32 rounds, has a block size of 64 bits and supports key lengths 80 and 128 bits [30].

2.3.2. Hash Algorithms

Hash functions are an essential part of cryptography and are widely used in many applications, such as digital signatures and message authentication codes. They produce a hash value with a fixed-length of a message which can be seen as the fingerprint of the message. The most important feature of the hash functions is that the output does not depend on a specific key.

One of the most crucial roles of hash functions in cryptography is signing long messages since producing the signature of a long message block by block introduces three problems; high computational overhead, message length overhead and security limitations [27]. Therefore, hash functions are suitable for producing a short fixed-length signature for an arbitrary-length message, as seen in Figure 2.13.

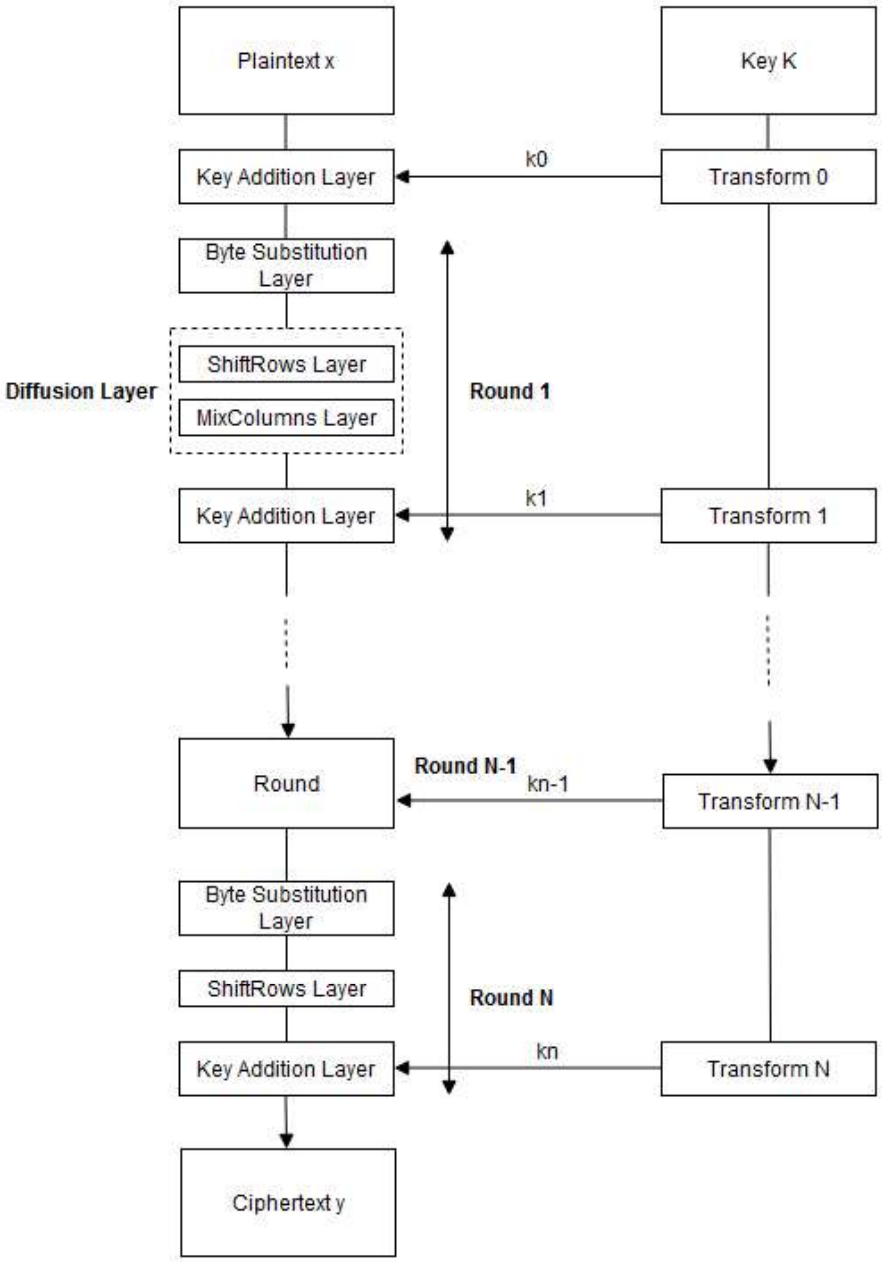

Figure 2.12. Flowchart of the AES algorithm.

Hash functions need to meet three requirements to be considered secure; i)pre-image resistance, ii)second pre-image resistance, and iii)collision resistance. The first requirement states that a hash function should be one-way, where it must be impossible

to compute the input message from the hash function's output. The second one requires that the hash function output must be different for any two input message pairs. Only one message is free to modify for trying to find a message with the same hash value. The third requirement dictates that it should be computationally impossible to find two different messages producing the same hash value while modifying both of the messages. The three requirements are illustrated in Figure 2.14.

Hash functions can process arbitrary lengths of data by dividing the input message into equal-length blocks and processing through a compression function. This type of architecture is called a Merkle-Damgard construction [27]. Hash functions can be divided into two groups: dedicated hash algorithms and block cipher-based hash algorithms. Dedicated hash functions are custom-designed and used specifically for hash generation, while block cipher-based hash algorithms are constructed using block cipher chaining.

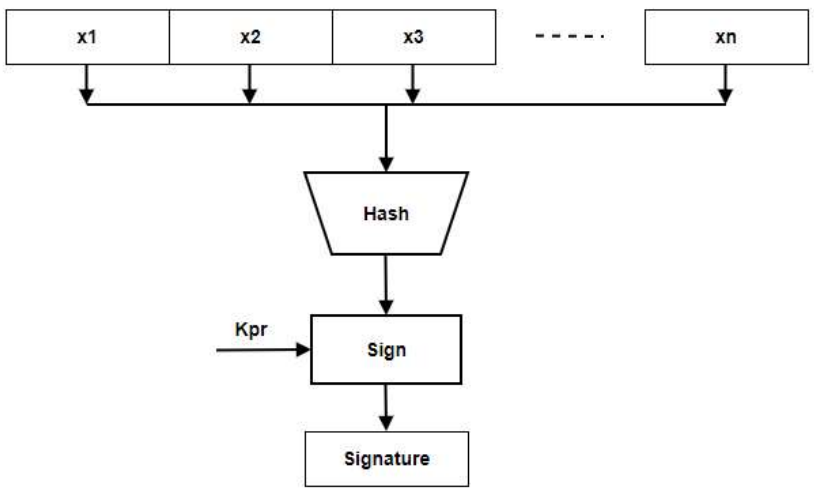

Figure 2.13. Digital signature with hash function.

Most of the dedicated hash functions are based on the MD4 family developed by Ronald Rivest. Only simple bitwise Boolean functions like AND, OR, and XOR are

used with 32-bit variables in this algorithm, making it very easy to implement in software. Later, MD5 was widely used after its proposition in 1991, and its hash output value was 128-bit with a collision resistance of 2^{64}. After some weaknesses were found in the MD5 algorithm, NIST standardized a new hash function called Secure Hash Algorithm (SHA) in 1993 and SHA-1 in 1995 with some modifications to SHA in terms of its compression function schedule, with both having 160-bit hash value. Then, NIST proposed three new standards known as SHA-2 with SHA-256, SHA-384 and SHA-512 to provide higher security levels for use with ciphers like AES, ECC and RSA since SHA-1 had a collision resistance of 2^{80}, which was not secure enough. Attacks like collision-finding attacks were applied to MD5 and SHA, and the same attack could be applied to SHA-1 with 2^{63} steps. However, being not resistant to collision attacks does not necessarily mean that the hash function is not secure for every application, such as password storage and key derivation [27].

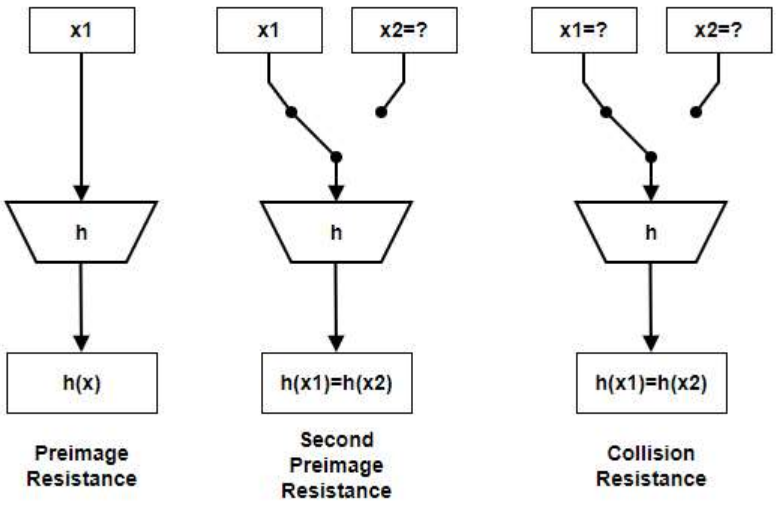

Figure 2.14. Security requirements of hash functions.

SHA-1 consists of 80 rounds, like block cipher rounds, and a preprocessing stage, where the input message is padded to a multiple of 512 bits. The computation of the function is constructed with four stages where each stage contains 20 rounds, and each

stage is different in terms of internal transformations and constants. Previous hash and message blocks are the inputs of the compression function. 32-bit words are computed from the message block for each round, as shown in

$$W_j = \begin{cases} x_i^{(j)} & 0 \leq j \leq 15 \\ (W_{j-16} \oplus W_{j-14} \oplus W_{j-8} \oplus W_{j-3})_{\lll 1}\, 1 & 16 \leq j \leq 79, \end{cases} \tag{2.1}$$

where W_j is the computed word and $x_i^{(j)}$ is the message block. Each round operates on the message with different functions and constants according to the stage and round number, as shown in

$$H_0, H_1, H_2, H_3, H_4 = \left(H_4 + f_t(H_1, H_2, H_3) + (H_0)_{\lll 5} + W_j + K_t\right), H_0, (H_1)_{\lll 30}, H_2, H_3, \tag{2.2}$$

where H_i are the hash word inputs to the round, W_j is the word computed from the input message for round j, f_t is the round function and K_t is the constant for stage t. [27] The round functions are made up of bitwise Boolean operations such as AND, OR and NOT. Therefore, implementing the SHA family of hash functions is suitable for software and hardware. The largeness of the number of rounds and the registers to store the intermediate hash outputs make the software implementations slower and result in hardware implementations with a large area, respectively.

The need for a new hash standard arose when successful attacks were implemented against SHA-1. NIST first proposed SHA-2 in 2001 and SHA-3 in 2015, even though there are currently no significant attacks on SHA-2. Thus, SHA-3 was not proposed to replace SHA-2 but to form a variety of hash algorithms. In total, 64 algorithms were submitted, and Keccak, a permutation-based hash function, won the competition and became the SHA-3 standard with a few modifications during the competition. SHA-3 has a sponge-like construction where the input message is first absorbed while being XORed with a part of the system state, transformed with a permutation function and then the result is squeezed out [31]. The fact that the system state contains additional information other than the output prevents the length extension attacks that are successful on other hash functions based on the Merkle-Damgard construction.

2.3.3. Lightweight Algorithms

Current standards for encryption and hash functions were proposed when the overheads of hardware implementations in terms of power and area were not considered as much as security. Thus, encryption standards like AES and secure hash standard SHA incur too much power and area overheads for recent lightweight applications, resulting in a need for new standards with low area and power requirements in hardware implementations. Therefore, many researchers worldwide have proposed and used such algorithms in many scenarios. PRESENT, Prince, Rectangle are three examples of lightweight encryption algorithms, while lightweight hash function examples are PHOTON, Spongent, and SipHash. Having too many options for these algorithms results in different algorithms in different products for a similar application, making it more challenging to analyze the security of each algorithm. Therefore, NIST started a new competition to make a new standard for lightweight authenticated encryption in 2015, with 56 candidate algorithms submitted in 2019.

The competition for the new lightweight cryptography standard was held in 2 rounds and a final stage. International researchers analyzed all candidates regarding security, performance and cost. At the time of writing this thesis, the competition is in the final stage with ten finalists. Some algorithms were designed for the competition, while others were modified to meet the NIST requirements. Since all the finalists have passed extensive tests by various researchers and some will become the new standard, one of them is chosen for this work, which is called ASCON. ASCON was selected for this work because it was the primary algorithm in CAESAR, a competition held from 2014 to 2019. It is also a hash function needed for memory integrity.

ASCON is a family of authenticated encryption and hash function algorithms designed for lightweight applications and can be easily implemented in software or hardware. It also has built-in side-channel attack protections. It is a permutation-based algorithm with a sponge-like construction like SHA-3. There are multiple levels

of security with multiple key and tag size options where it can be scaled for conservative security or higher throughput. All versions use the same permutation in which an SPN-based round transformation is applied iteratively. Its state is 320 bits long and divided into five words of 64 bits. The round transformation consists of three steps; i)addition of round constants where a 1-byte constant different for each round is XORed to the third word of the state, ii)nonlinear substitution layer where a 5-bit S-box is applied 64 times in parallel to the state words, and iii)linear diffusion layer where different rotated words are XORed [32].

The authenticated encryption mode of ASCON uses a duplex-sponge-like construction with a recommended key, tag and nonce size of 128 bits. It is divided into 4 phases; i) initialization, where the state is initialized with the key and the nonce, ii)associated data processing, where the state is updated with the associated data, iii)plaintext processing, where the plaintext blocks are inserted into the state while the ciphertext blocks are taken out, and iv) finalization where the key is inserted again and the tag is taken out for authentication. The block diagram of these stages is shown in Figure 2.15, and the decryption process is almost identical to the encryption. The permutation in the initialization and finalization stages are stronger with more rounds where the number of rounds a and b depend on the variant of the algorithm. In this work, the variant ASCON-128a is chosen where a is 12 and b is 8, key, nonce, tag and rate sizes are 128 bits, and the sponge capacity is 192 bits [32].

The hash function of the ASCON uses the same sponge-like 320-bit permutation as the authenticated encryption with two variants as, ASCON-HASH and ASCON-HASHA, both of them providing a security level of 128 bits with a minimum hash size of 256 bits. The hashing is divided into three stages; i) initialization, where the state is initialized with the initialization vector, ii)absorbing message, where the message blocks are inserted into the state, and iii)squeezing hash, where the hash blocks are taken out after permutation rounds [32], as illustrated in Figure 2.16.

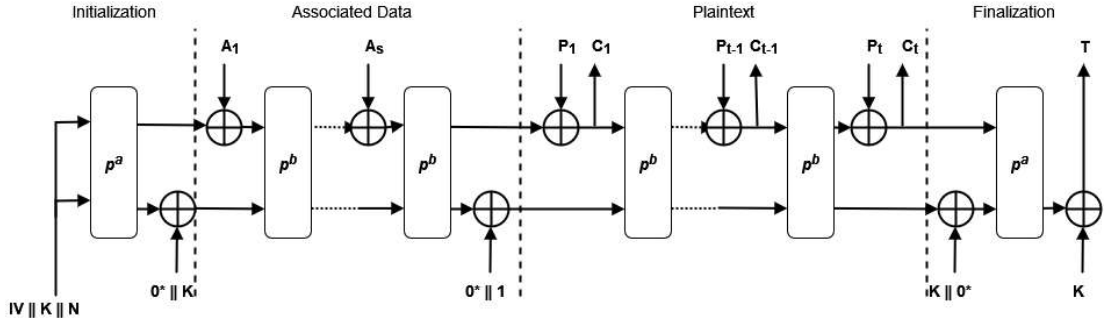

Figure 2.15. ASCON authenticated encryption block diagram.

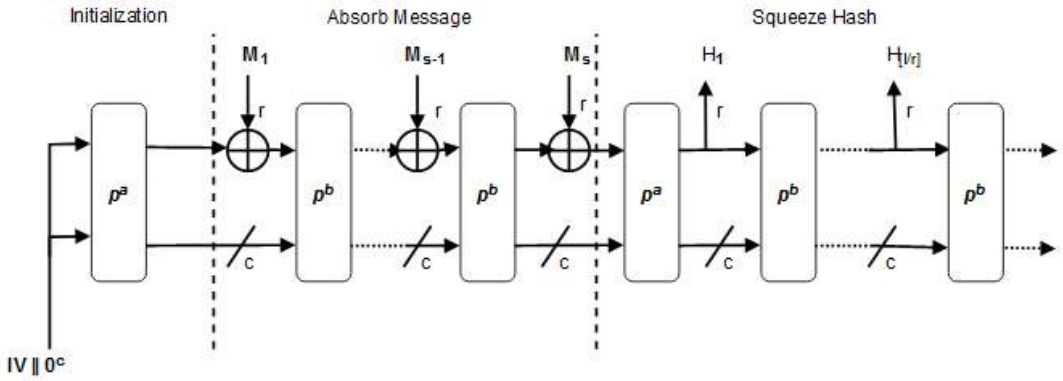

Figure 2.16. ASCON hashing block diagram.

3. DESIGN METHODOLOGY

The security of the memory interface of a processor is provided with encryption and integrity check that has a low area and power consumption overhead. The design in this work also includes a built-in true random number generator (TRNG) for generating the key of the symmetric cryptography algorithm. The system also supports on-chip secure storage for sensitive user data by using the authenticated memory encryption block. The block diagram of the system on chip is illustrated in Figure 3.1.

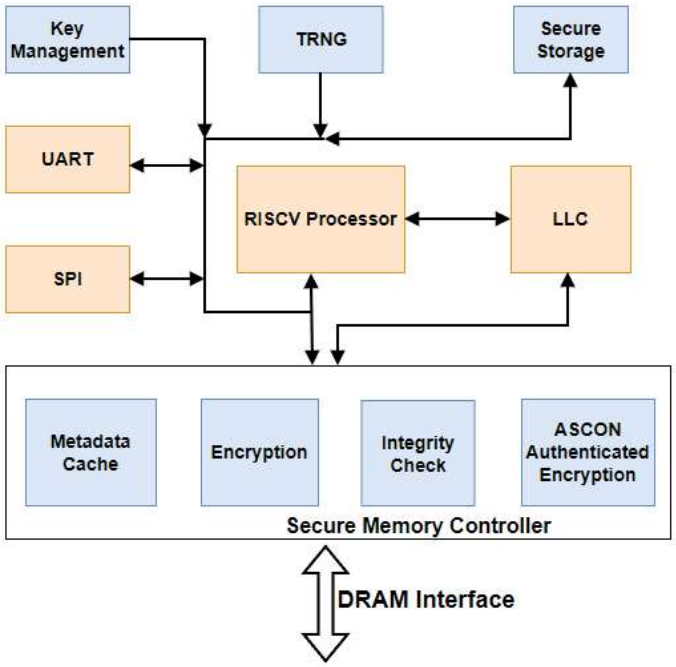

Figure 3.1. Proposed secure SoC block diagram.

The complete SoC includes an open-source RV32IMC processor, PICORV32 [33], UART and SPI peripherals, a key management block, a TRNG, 128-byte of secure storage, and a 32 KB unified data and instruction cache. Also, it includes the secure memory controller with the encryption and integrity check block, the ASCON module

with authenticated encryption and hashing block, and a metadata cache to store the counters and hashes. All blocks are connected with a 32-bit system bus, and the secure memory controller is connected between the last level cache and the DRAM interface. The SoC is designed using Verilog HDL, implemented on FPGA using Vivado, and the ASIC synthesis is performed using Cadence Genus in TSMC 65 nm technology. The simulation is performed using Vivado simulator and Cadence Xcelium, and the linting of the RTL is performed using Synopsys Spyglass.

This chapter will present the proposed design in detail by first explaining the secure memory controller and how each block connects within the controller containing the encryption block, integrity check block, ASCON and metadata cache. Then, the on-chip secure storage module will be explained with details of the TRNG while stating the RTL design flow of each part. Finally, the FPGA design flow and the ASIC synthesis flow will be explained by showing design layouts.

3.1. Secure Memory Controller

The encryption, decryption and integrity check of the off-chip memory data are performed between the last level cache and the memory interface with the secure memory controller block. The module includes the encryption and integrity check controllers, both of which interface with the metadata cache where encryption counters and hash values are stored, ASCON and the DRAM. Different arbiters are placed throughout the module where multiple agents are trying to access the same bus, such as the DRAM interface, which has four master agents. The ASCON block is also directly connected to the processor system bus for use by the processor and the secure on-chip storage. The block diagram of the secure memory controller is shown in Figure 3.2.

The controller is accessed when there is a request from the last level cache, and it can be either a write request or a read request. The address of the request is

checked first upon arrival from the cache to see if it belongs to the secure memory region. Memory-mapped registers control these regions with physical memory protection (PMP) as described by the RISC-V privileged Instruction Set Architecture (ISA) [34]. If the address is not in the secure range, the request is directly forwarded to the DRAM, and the response is to the cache. Otherwise, the encryption or decryption process is initiated depending on the request being a write or a read.

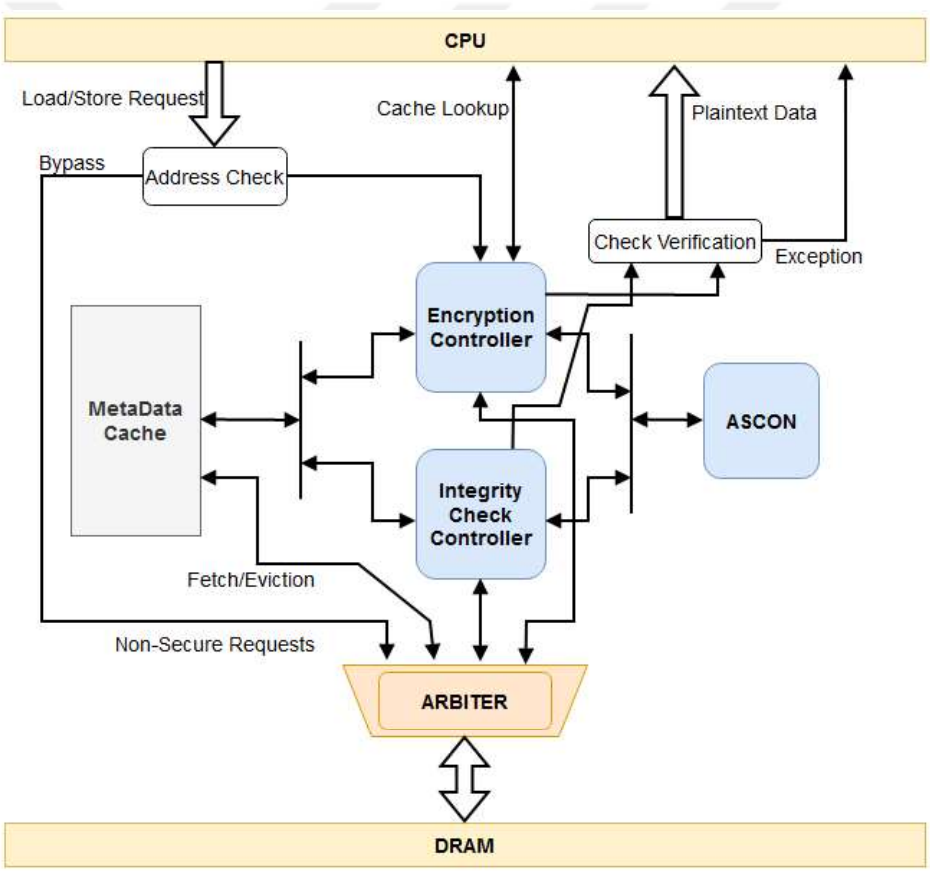

Figure 3.2. Secure memory controller block diagram.

The encryption controller immediately fetches the corresponding counter value from the metadata cache and encrypts it using ASCON to obtain the seed. The seed is XORed with the ciphertext block taken from the DRAM to obtain the plaintext block if the request is a read or with the plaintext block from the last level cache to encrypt

it for storage in the DRAM. The integrity check procedure for read requests and hash generation and update procedures are also carried out while the signals are travelling or encryption is being performed. For write operations, the ciphertext block is written to the DRAM once the encryption process is completed, and the hash update is performed in a pipelined fashion. For read operations, the plaintext block is only sent to the processor once the integrity check is completed and there are no errors. In case of integrity check faults, the processor is informed with exceptions which include the physical address of the fault and the data type, and the processor acts according to the exception handling software.

3.1.1. Encryption Controller

A strong encryption algorithm in a secure memory scheme can prevent data theft attacks. As mentioned previously, direct and counter-mode encryption are two main ways to encrypt memory data. Direct encryption is easier to implement and has lower hardware cost with high latency. In contrast, counter mode encryption solves the latency problem by encrypting counters with the strong cipher and XORing the data with the seed at the cost of higher storage overhead. The split counter mode encryption in [22] is used in this work so that a 64-bit counter is stored for a 4 KB page while 7-bit minor counters are stored for each 512-bit block. This method reduces the storage overhead and the re-encryption latency overhead since re-encrypting a 4 KB page takes a much shorter time than re-encrypting the whole memory.

The encryption controller includes three primary finite state machines (FSM) that control the metadata cache interface, re-encryption sequence, and the ASCON interface. It has an interface with the metadata cache to read and update the counters. The memory requests and the cache snoop responses are received from the last level cache, where the plaintext data is returned. The ASCON module is used to encrypt the seed to obtain the one-time pad. It reads and writes the ciphertext data through the interface with the DRAM. It also reads the key for encrypting the memory from a

special register that is updated either upon boot or when a major counter overflows, which is practically impossible for an IoT system since they have a size of 64 bits. The block diagram of the encryption controller is illustrated in Figure 3.3.

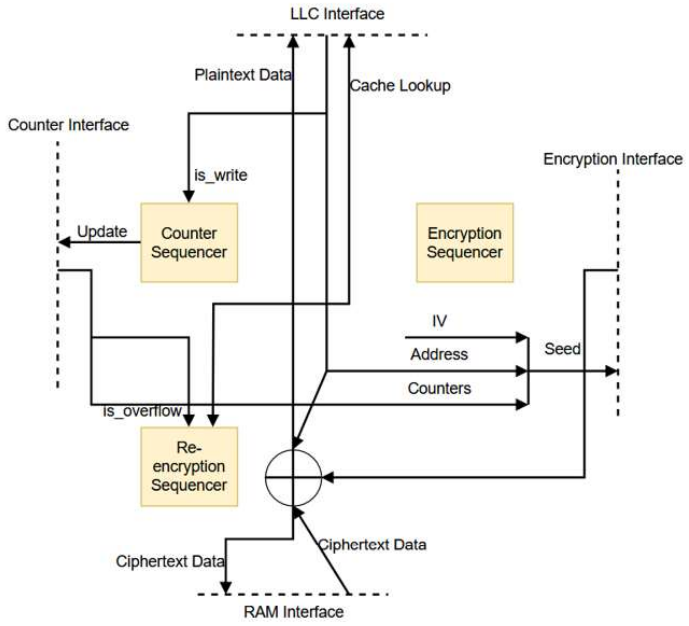

Figure 3.3. The encryption controller block diagram.

The operation starts when a request arrives from the last level cache with an address in the secure region. A counter read request is immediately sent to the metadata cache, which sends the counter in one cycle if present or after reading from the DRAM, decreasing the performance. When the counter arrives from the metadata cache, the major and minor counters are concatenated with the address of the data block and the initialization vector to form the seed, as shown in Figure 3.4. Then, the seed is encrypted with the encryption algorithm under the control of the encryption sequence FSM. If it is a read request, it is sent to the DRAM first so that the data can arrive while the encryption algorithm generates the pad. When the pad generation is complete and the ciphertext arrives, they are XORed to obtain the plaintext data block, which is then sent to the last level cache.

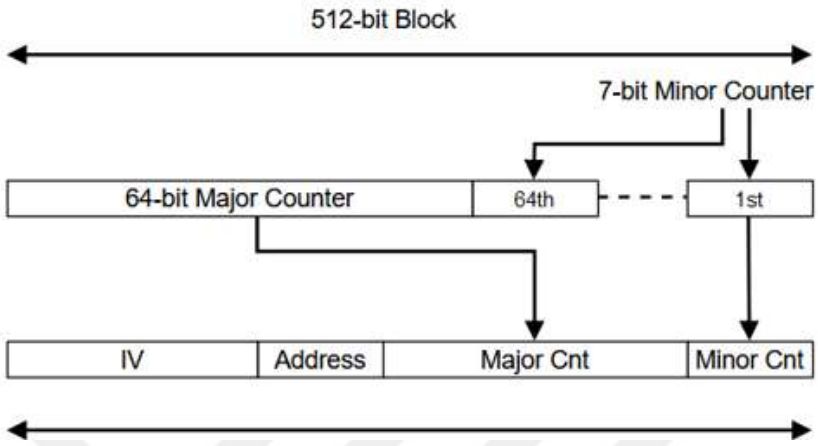

Figure 3.4. The seed for split counter mode encryption.

The operation of the controller is different for a write request in that the minor counter for seed generation should be incremented. For example, the minor counter, read from the metadata cache, increments before being used for seed generation for a write request. The encryption and the decryption occur by XORing with the incremented pad version. The counter is also updated in the metadata cache if the minor counter does not overflow. In case of an overflow, a re-encryption sequence is initiated where the corresponding 4 KB page is re-encrypted with new counter values. Also, all of the minor counters for that page are reset, and the major counter is incremented. During a re-encryption sequence, all the blocks in the page are read from memory, decrypted using the old counter values and written to the DRAM by encrypting them with the new ones. A cache look-up which adds little complexity to the design is added, which sends a read request to the last level cache for the current block in the page if it is present. If the block is present in the cache, that block is not read from the DRAM since on-chip caches are considered secure in this system. The probability of that page's blocks being present in the cache is high because the counter's overflow correlates with that page's high usage. Therefore, this method significantly reduces the performance overhead of the re-encryption while reducing the power consumption since the amount of DRAM bus traffic decreases.

The encryption sequencer handles the interface with the authenticated encryption algorithm. When an encryption request is received from either the counter sequencer or the re-encryption sequencer, the authentication encryption instruction is sent over the bus after sending the key activate instruction and the 128-bit key to the encryption block. Then, the 128-bit nonce is sent, followed by the associated data, which is used with authenticated encryption mode and skipped in this module since only encryption is used. After that, the plaintext, the seed, is sent over the 32-bit bus, after which the ciphertext, the pad, is received. Finally, a 32-bit status signal is sent from the encryption block to check if the operation is successfully completed.

3.1.2. Integrity Check Controller

The tampering attacks can be prevented by checking the integrity of the data with a strong hash function. An integrity tree should be constructed to prevent replay attacks. 64-bit hashes generated by the ASCON hash function are used in this design. Also, an 8-ary bonsai Merkle tree as in [26] is designed over the encryption counters to reduce the storage and performance overhead of the integrity tree. The hash tree is constructed over the encryption counters to protect them against replay attacks using this method compared to the traditional Merkle trees. Generating the hash with the address and the counters implicitly protects the integrity of data blocks. Figure 3.5 shows the hash messages for data blocks and the counter/hash blocks.

The integrity check controller block has three main FSMs that control the meta-data cache, the hash function, and the integrity tree sequence as shown in Figure 3.6. It has an interface with the top module where it receives the integrity check request and sends cache look-up requests. It reads and writes the hash values from the meta-data cache through which it makes the interface the DRAM. The ASCON module is connected to the integrity controller for sending message blocks and reading newly generated hash values. The module uses 64-bit hashes since it has been proven that this length is secure enough for memory authentication applications in [26].

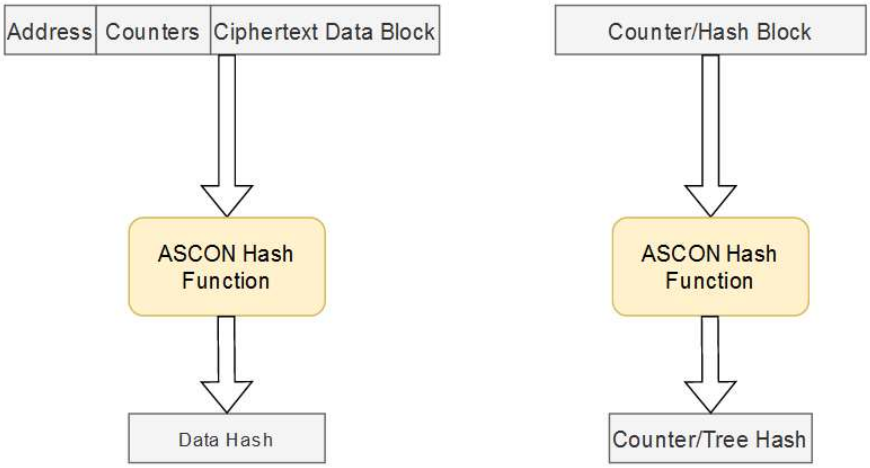

Figure 3.5. Message block for data and counter/hash block.

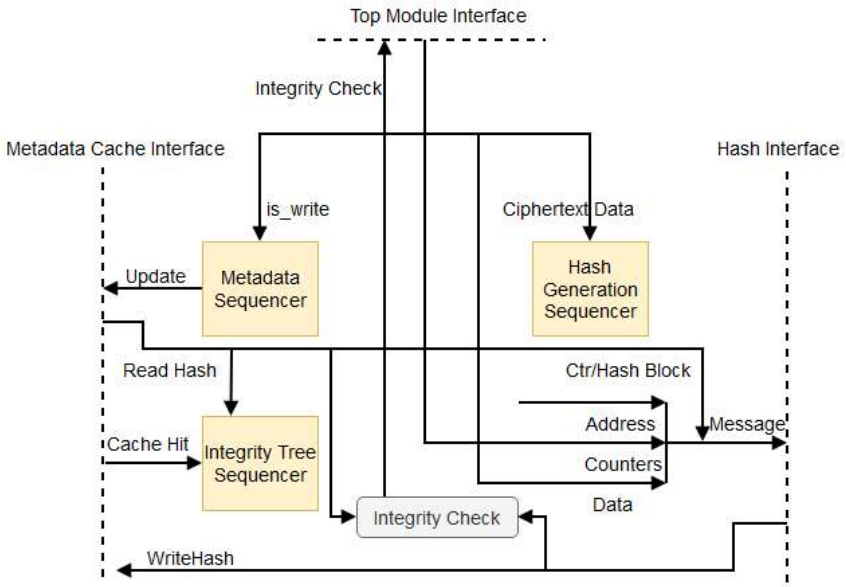

Figure 3.6. Integrity check controller block diagram.

The integrity check starts when a request arrives from the top level control for a write or a read operation. The controller requests the corresponding hash values from

metadata cache while the hash generation sequence starts immediately for the data. The ciphertext data and the counters arrive from the top level when they are ready, while the hash value is fetched from the metadata cache. When the ciphertext data and the counters arrive, the hash generation request is sent to the hash function over the 32-bit data bus. Once the hash generation is complete and the hash stored in memory arrives for a read operation, they are compared with each other to determine whether to move on with the integrity tree traversal or to produce a fault signal used by the top module to generate exceptions. Once the hash generation is complete for a write operation, the hash is sent to the metadata cache and updates the hash tree using the integrity tree sequencer.

A hit signal is generated from the metadata cache to use in the integrity tree sequencer, informing the controller that the requested hash is present in the cache. This signal reduces the latency overhead of tree traversal by stopping the integrity check when a hash node is found on the chip. This design utilizes a Bonsai Merkle tree with six levels, including the root hash, which is stored in a secure register to which only this module has access using the PMP. Each cache block contains eight hash values, each of which protects eight counter or hash blocks, and the tree is illustrated in Figure 3.7. Normally, this results in 6 additional memory accesses for each integrity tree check or update. However, using the cache hit signal to stop the integrity tree traversal or update when a certain level is found on the chip, the latency is significantly reduced since the spatial locality of the nodes will increase with going up to the root as they protect a larger memory region. Furthermore, the power consumption decreases with this method since additional memory accesses and hash generations are eliminated. When the integrity tree checking is completed, the integrity check signal is sent to the upper-level module for a read signal while the system indicates that it is ready when updating the tree finishes.

The hash generation sequencer handles the interface with the hash function. When a hash generation request is received from the top module or the integrity tree

sequencer, the hash generation instruction is sent, followed by the message block, either the counter/hash block or the concatenation of the ciphertext data, the address and the corresponding counter. Then, the sequencer waits for the hash value. The generated hash is written to the metadata cache for a write operation and checked with the hash from the metadata cache for a read operation. Finally, a status signal is received from the hash interface, indicating whether or not the hash generation was successful.

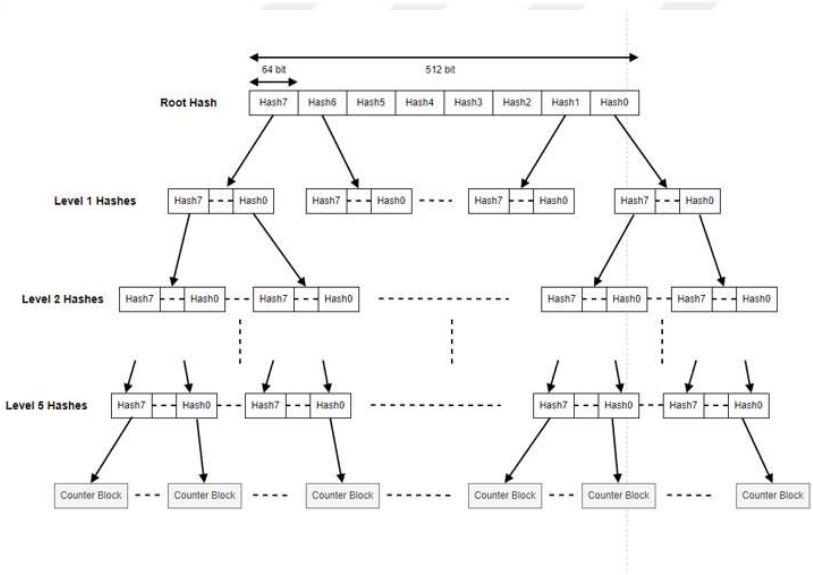

Figure 3.7. Bonsai merkle tree chart.

3.1.3. Metadata Cache and ASCON

The performance of memory encryption and authentication schemes can be increased by using different methods such as predicting the counters [14], storing the counters in a cache [22, 23, 26], increasing the arity of integrity trees [18, 23]. A metadata cache is designed in this work to store the encryption counters and the hashes. The cache is a 4-way set-associative cache with a least recently used replacement policy, write-back policy and a size of 32 KB for the base mode. The set associativity is chosen to balance low miss rate with a low tag comparison latency and hardware cost,

while the write-back policy is chosen to reduce the traffic between the cache and the DRAM by writing the updated value in the cache to the DRAM only when it is evicted from the cache. When a new request arrives from an agent, and it is not in the cache, this results in a cache miss, in which case the requested block is fetched from the DRAM. The received block is placed in one of the four ways selected by the replacement policy, and if the block in the selected way is changed while it was in the cache, it is written back to the DRAM. The cache is implemented with block RAMs for FGPA implementation, while SRAMs are used for ASIC implementation.

The metadata cache stores counters for the encryption and stores hashes for the integrity check controller. Reception and eviction of the metadata occur through the interface with the DRAM. The major and minor counters are sent to the encryption controller on a separate channel. Also, all minor counters are sent for use in the re-encryption sequencer as mentioned in Section 3.1.1. It receives the new major and the minor counters from the encryption block for a write operation, reads the current block from the cache and merges with them to write back to the cache. The interface with the integrity check controller is more straightforward as it only receives hash values for write. It sends hash values for read while sending a cache hit signal for use in tree traversal sequencer as mentioned in Section 3.1.2.

A block cipher and a hash function are needed for memory encryption and memory authentication, respectively. As mentioned in Section 2.3.1, the current standard for encryption algorithm is AES, and it has a high hardware cost, as do the standards for hash functions, SHA family, mentioned in Section 2.3.2. For this reason, multiple lightweight cryptography algorithms have been proposed and used in the literature [27], and NIST also started a competition for a lightweight algorithm. A finalist of this competition which also won a previous competition called CAESAR, is chosen for this work called ASCON, a lightweight authenticated encryption and hashing algorithm as mentioned in Section 2.3.3. There have been many implementations of ASCON in literature, and an open-source implementation written in VHDL is used for this work

as the implementation of the cryptography algorithm is not the main focus of this work [35]. This implementation is chosen for being lightweight and with many choices of latency and hardware cost trade-offs.

3.2. Key Management

The authenticated encryption algorithm in the design needs a 128-bit key input to operate. It can be either received from a key source outside of the chip or generated on the chip using a built-in manager. The module is connected to the main system bus and receives key requests from the processor or the secure memory controller. The secure on-chip storage unit also uses the keys. In total, the key manager stores three keys: the current and the previous secure memory controller keys and the secure on-chip storage key. When a new key generation request arrives, the manager generates a random 128-bit number using the true random number generator (TRNG) and sends it to the corresponding agent.

3.2.1. True Random Number Generator

Random numbers are quite useful in many applications, such as cryptography, where they are mainly used to generate keys. Although they can be generated using a deterministic function, as in pseudo-random number generators (PRNG), they are usually obtained from a random physical property with high entropy using special circuits called true random generators. One of the widely used types of these circuits is based on ring oscillators, where multiple ring oscillators are sampled to generate a random bit. This type of RNG has a high area overhead since many oscillators are needed for high entropy. A tetrahedral oscillator was used in [36] to reduce the hardware cost while generating higher entropy by nesting multiple oscillators with different number of inverters within each other. The race condition increases the randomness, which helps reduce the number of oscillators used for reliable random numbers.

A modified version of this tetrahedral oscillator is used in this work to reduce the area overhead, as shown in Figure 3.8. Three inverters are inserted into specific places in the oscillator that can be turned on and off with a multiplexer and a select signal. The regular tetrahedral oscillator has three stable loops and four unstable loops complementing each other. By switching the states of the adjustable inverters, the stability of these loops changes causing a different characteristic of oscillation. When the inverters are on, all the stable loops become unstable while all the stable loops become unstable [37].

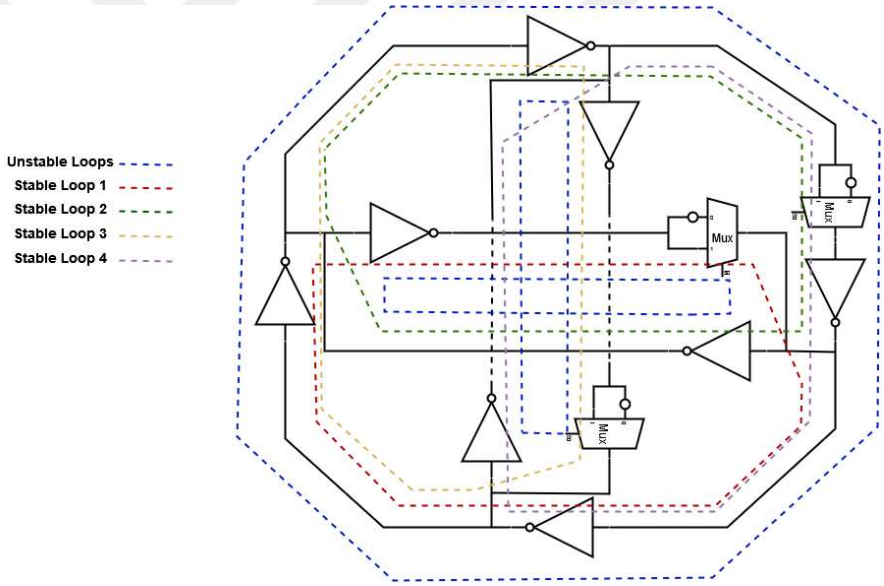

Figure 3.8. Modified tetrahedral oscillator schematic.

Four modified tetrahedral oscillators are used to generate a random bit sampled by a periodic clock signal. The structure of the RNG circuit is illustrated in Figure 3.9. The oscillators' outputs are sampled before they are XORed with each other because it increases the randomness by capturing the behaviour of the oscillators at exactly the sampling times. The select signal that switches the states of the inverters is generated in the oscillators. The oscillators switch the inverters on and off and then sample the output to combine the metastability of two events. The oscillation behaviour changes

at these events as shown in Figure 3.10. The yellow-coloured signal is the select signal that switches the state of the inverters. The blue one is the output of one of the oscillators, the red one is the sampling signal, and the green one is the sampled random output bit. The change of behaviour in the oscillator's output can be observed after the inverters' states are turned on and off in Figure 3.10. With this method, the hardware cost of the RNG is reduced by 60% without sacrificing randomness [37]. The RNG circuit is also connected to the system bus so the processor can obtain random numbers.

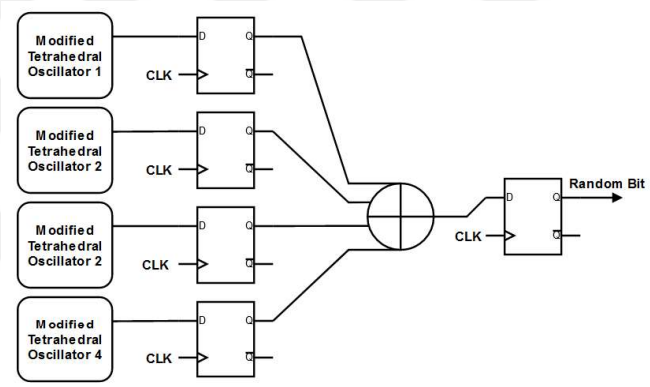

Figure 3.9. Proposed random number generator schematic.

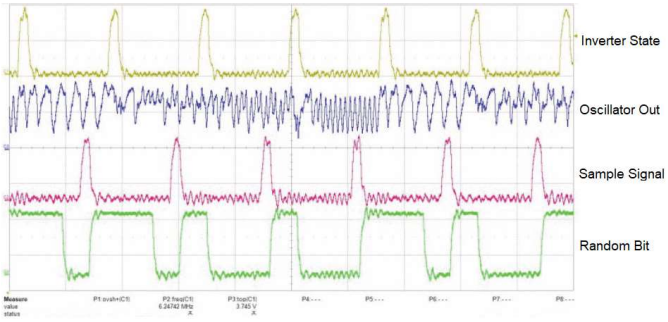

Figure 3.10. Random number generator signal waveforms.

3.3. On-chip Secure Data Storage

Some attacks can steal or tamper with the data on the chip by probing inside the chip. Considerable research has been done on the security of on-chip data. Even though this work focuses on protecting the confidentiality and integrity of off-chip memory, and inside the chip is assumed to be secure, a special storage unit is provided to the user for more sensitive data. The data stored in this space is protected using the ASCON authenticated encryption algorithm. The block diagram of the on-chip secure storage module is shown in Figure 3.11.

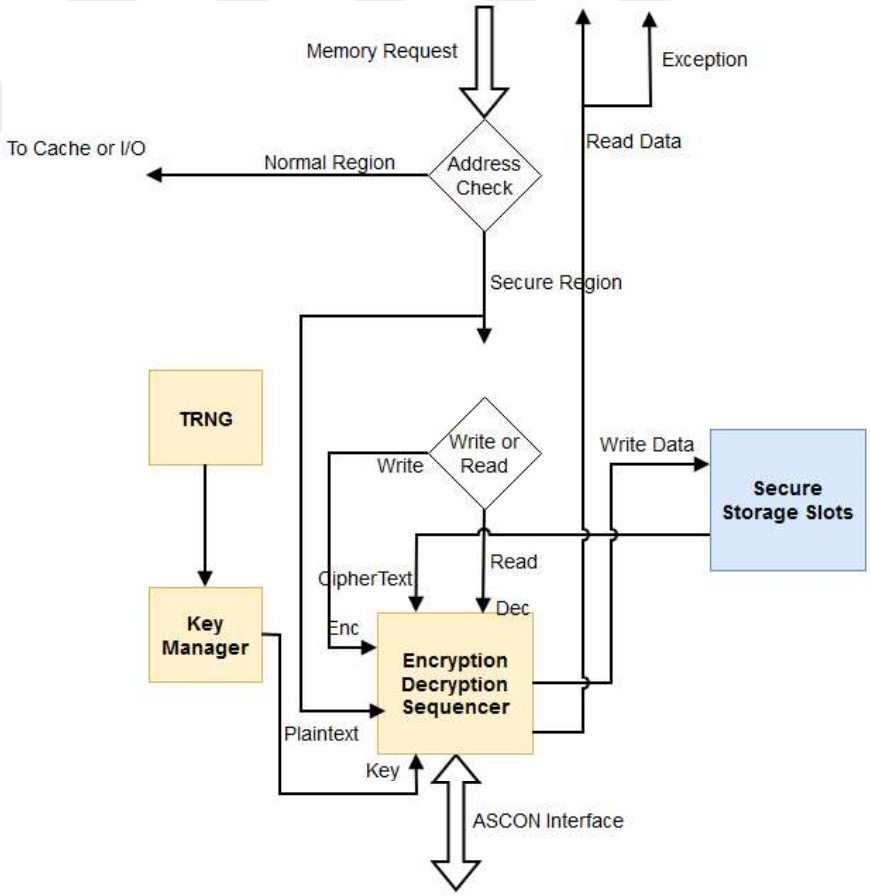

Figure 3.11. Secure on-chip storage block diagram.

The module first checks the address to determine if it resides in the secure region when a request from the processor is received. The encrypted data in the secure region and its authentication tag are stored in these special slots. The read operations initiate an authenticated decryption request to ASCON after reading the encrypted data and its tag from the slots. If the authentication is successful by the indication of a signal at the end of the operation, the deciphered data is returned to the processor through the system bus. The keys for the cipher are received from the key manager, and the encryption-decryption sequencer controls the authenticated encryption of the data.

3.4. RTL Design and Verification

The hardware is designed using a hardware description language (HDL) such as Verilog and VHDL, and this stage of hardware design is called the Register Transfer Level (RTL). The RTL design is mapped to either technology-specific gates in ASIC implementation or Look-up Tables (LUTs) in FPGA implementation. The proposed circuit is designed using Verilog except for the open-source ASCON implementation, which is designed using VHDL.

The circuit was designed to be easily reusable for different designs, and every part of the design can be removed or connected from the top level. Also, parameters are used throughout the design to modify the design by changing them from a single parameter file. All modules in the design are instantiated with ”ifdef” structures to add or remove from the top-level easily with ”define” statements.

The secure memory controller instantiates the metadata cache, the encryption controller, the integrity check controller and the ASCON with all the necessary parameters. The metadata cache can be modified in terms of cache size, block size and set associativity. Direct mode or counter mode encryption can be selected for the encryption controller, and the hash size can be chosen for the integrity check controller.

Since all pieces can be removed or connected easily, each security measure was added one by one into the design and implemented to observe their effects. The base address and the data region size are given from the top level, and they are calculated for each metadata used in the design. A 1GB of data region with a base address of 0xC0000000 were used for the system.

The RTL code was designed such that the implementation has a low power consumption with a low area. Unnecessary signals and gates were avoided, along with parallel structures that would result in high hardware usage. Switching data signals was minimized by applying an enable signal to each register so that the data only changed when the module was in use. Also, clock gating circuits were added to the design to minimize the power consumption even further by eliminating the switching power of the flip flops. Latch-based clock gating is used to prevent glitches in the clock signal. The gated clock signal is obtained by ANDing the original clock signal with the falling edge sampled enable signal.

There are also differences in the RTL code for FPGA and ASIC implementation. One of these differences is the on-chip memory used in caches, block RAMs in FPGA implementation and SRAM IPs in ASIC implementation. The block RAMs are inferred in the design by defining a "reg" memory with specific properties with a single port. The foundry or third-party firms generate the SRAMs for a specific technology. The foundry-generated SRAM IPs for TSMC 65 nm are used for this work and are instantiated in the corresponding modules. The memory usage is switched between SRAM to block RAM using another "ifdef" structure from the top level. Another difference in the RTL is the RNG circuit, where multiple sources drive the same node in the oscillator. Current FGPA designs do not allow multiply driven nets in their SoCs since it can lead to high power consumption or break-down of the chip. The input connection is switched from one driver to the next with a high-speed clock signal in the FPGA implementation to simulate the actual connection to multiple drivers. However, this solution is unnecessary for the ASIC implementation.

The verification of the design was performed by using a combination of methods and tools. The RTL linting was done using Synopsys Spyglass to remove all the bad practices in the design and improve it in terms of power and area. The simulation of the design was performed in the Vivado simulator and the Cadence Xcelium. The top module of the testbench was designed using SystemVerilog to make use of the assertions. Multiple assertions are placed in the testbench probing the design's hierarchy to check the circuit's correct behaviour. Random software was generated for the processor using RISCV-DV for stress tests to measure performance along with custom software such as a loop loading and storing the entire memory and the coremark code. The simulation was run multiple times with different seeds to debug all the corner conditions. Since some of the test software are too long to run on a simulation tool, they are run on the FPGA to speed up the test process. Debug signals were placed in the design to probe the internal signals using the Xilinx Integrated Logic Analyzer (ILA).

4. HARDWARE IMPLEMENTATION

The hardware implementation of the proposed design was made up of two parts; FPGA implementation and ASIC implementation. The FPGA implementation is performed for hardware verification, while the ASIC synthesis is performed to complete the IP design flow and obtain more accurate timing and power results for TSMC 65 nm technology. Four different versions are designed in this thesis;

- Base: This version is the base SoC with no security measures implemented. It is used for understanding the overhead of each security measure.
- Enc.: This version is the one where only memory encryption is implemented. Split counter mode encryption is used.
- Auth.: This version is the one where only memory authentication is implemented. Bonsai Merkle Tree is used for security against replay attacks.
- Comb.: This version is the proposed SoC design with both memory encryption and integrity that is secure against memory tampering and data theft attacks.

First, the FPGA implementation is explained with hardware resource and power results. Then, the ASIC synthesis flow is presented with timing, power and area reports.

4.1. FPGA Implementation

The design was implemented on the Xilinx Kintex Ultrascale Development Board with an XCKU105 FPGA chip. Xilinx Vivado Design Suite version 2022.1 was used for the implementation and the behavioural simulation. The necessary clocks are generated using the clocking wizard IP. The memories are generated with BRAM IPs, and the DDR4 controller and PHY IPs are used to interface with the DDR4 memory on the board. No other IP was used in the design, which is made up of LUTs, BRAM and

registers in the FPGA implementation. The FPGA flow starts during the RTL design with the elaboration of the Verilog code to see if the design can be mapped to logical gates. The RTL code is modified by checking the warnings during the elaboration stage. After this stage, the behavioural simulation is started to test if the circuit functions as expected.

A testbench is designed for the FPGA implementation, which includes an instruction memory to store the generated software code. The processor boots at the address of this instruction memory which is not in the protected region. The test software is loaded from the instruction memory and copied to the main memory by encrypting first. After observing no faults in the functional simulation, the synthesis stage is started, which maps the logical design after elaboration into LUTs, BRAM and registers. If the design meets the constraints and there are no errors, the implementation stage begins, where the synthesized netlist is placed and routed in the FPGA chip according to the design constraints. After the design is successfully routed, a bitstream file is generated containing all the information to program the FPGA.

The DDR4 controller IP of Xilinx has a native interface with 512-bit data width. The DRAM interface of the proposed design has a data width of 32 bits. Therefore, an adapter circuit was designed to use the DDR4 IP, accumulating the 32-bit data and sending a 512-bit package to the IP, returning the 512-bit block received from the IP in 32-bit chunks to the processor. The DDR4 controller IP works at 300 MHz, while our design works at a lower frequency of 100 MHz due to the routing delay of the FPGA. Therefore, an asynchronous FIFO was used at the processor-IP junction for clock domain crossing to transfer data reliably.

The RNG was specially designed for the FPGA implementation to make the multiple-driver connection. The clocking wizard IP generates a high-speed signal, which is used to switch the connection of the inverter input from one inverter output to the other, as mentioned in Section 3.2.1, which emulates the real connection among

three nodes in the hardware. Also, the oscillators are implemented using the "don't touch" constraint, which prevents the synthesizer from deleting the inverters during optimization. The switching signal has a frequency of 465 MHz, while the sampling frequency of the TRNG is 10 MHz. The RNG was tested on the FPGA by collecting the random data from a UART interface.

The proposed design was implemented with all the protection schemes, including split-counter mode encryption, integrity check and integrity tree, key manager, TRNG, ASCON authenticated encryption and hashing function, and the metadata cache with a size of 32 KB. The layout of the implemented design is shown in Figure 4.1. The red part is the DDR4 controller and PHY IP, the dark blue part is the encryption controller, the green part is the integrity check controller, the purple part is the ASCON, the yellow part is the metadata cache, and the light blue part is the rest of the design.

The resource utilization of the implemented design is shown in Table 4.1. LUTs and registers are used for a hardware cost comparison with the literature. When only the encryption is included in the design, the LUT usage increases by 83% and the register usage increases by 39%. The authentication has an area overhead of 72% in LUTs and 33% in registers. When they are combined in the proposed design, it has a hardware usage overhead of 99% in LUTs and 46% in registers, with a 46% increase in BRAMs.

Table 4.1. FPGA implementation hardware resources vs. security schemes.

Resource	**Base**	**Enc.**	**Auth.**	**Comb.**
CLB LUTs	4034	7416	6978	8049
CLB Registers	1692	2368	2259	2486
CARRY8	80	88	86	92
BRAM Tiles	8	16	16	16

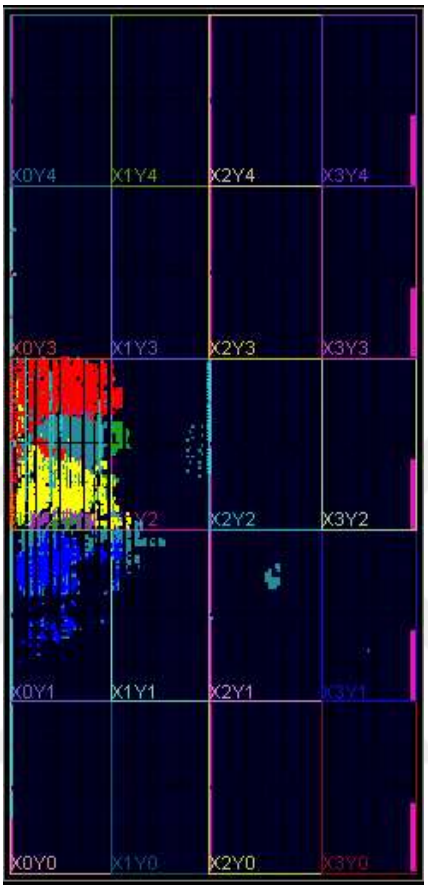

Figure 4.1. The layout of the FPGA implementation.

The power analysis of the FPGA implementation is shown in Table 4.2. The power results are calculated at the typical process and 25 °C of chip temperature. A Switching Activity Interchange Format (SAIF) file was generated by running a post-implementation timing simulation to see how much every node in the design is switching for a more accurate power consumption calculation. The total on-chip power is increased by 5.6% when the encryption is added. It increases by 5.4% when the integrity check is added. The total power consumption overhead of the design is equal to 7% when both of these security measures are implemented. The device static power remains mostly the same, constituting a large portion of the total on-chip power as seen in Figure 4.2.

The timing results were also clean, with no negative slacks and timing warnings regarding clock domain crossing or input and output constraints. The clean state of the timing report indicates that all of the signals in the design can successfully propagate from input to register, from register to register, and from register to output.

Table 4.2. FPGA implementation power analysis.

Power Analysis	**Base**	**Enc.**	**Auth.**	**Comb.**
Device Static Power(mW)	95,005	95,208	95,156	95,548
Device Dynamic Power(mW)	44,086	52,152	51,456	54,212
Total On-Chip Power(mW)	139,091	147,36	146,612	149,76

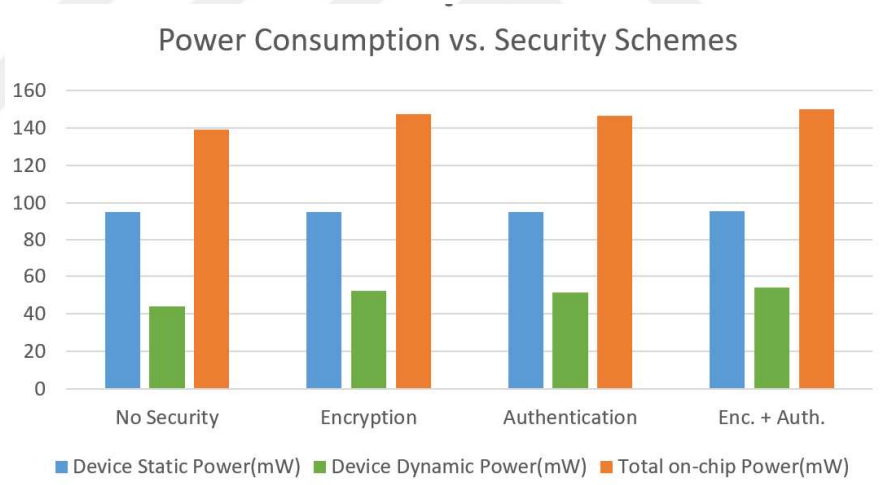

Figure 4.2. FPGA implementation power consumption analysis.

4.2. ASIC Implementation

The design was implemented on the TSMC 65 nm technology in the application specific integrated circuit (ASIC) flow. The ASIC flow starts with the synthesis stage,

where the RTL design is mapped to the technology-specific standard cells, and a netlist is created. After this stage, the standard cells are placed and routed on the chip according to the design constraints. Then, sign-off checks are applied to ensure that the design meets the foundry rules. Finally, the Graphic Design System (GDSII) file is created, which contains all of the information needed by the foundry for tape-out. The IP design flow also contains the ASIC synthesis stage to ensure that the design meets the timing and power requirements alongside the functional requirements.

The design has a few differences in the ASIC implementation from the FPGA implementation. The caches are constructed from SRAMs instead of BRAMs, the DDR4 IP in the Kintex FPGA is not used, and the ring oscillators are normally designed by forcing the synthesis tool to allow multi-driven nets. The DRAM interface controller in the FPGA implementation is removed, and the 32-bit data bus is left. The synthesis is performed using Cadence Genus with the RTL, standard cell library (lib), standard cell physical cell library (lef), design constraints (sdc), and RC extraction library (QRC) files as inputs. The frequency was specified as 200 MHz in the constraints file. The RNG was running at a different frequency, so handshake-based clock domain crossing circuits were used between the system bus and the RNG. The clock domain crossing checks were performed using Synopsys Spyglass. The synthesis results are shown in Table 4.3.

Table 4.3. Synthesis results in TSMC 65 nm technology.

Synthesis Results	**Base**	**Enc.**	**Auth.**	**Comb.**
Logical Instances	10089	28581	27868	31024
Area(mm^2)	0,096	0,285	0,299	0,313
Frequency(MHz)	200	200	200	200
WNS(ns)	0,051	0,046	0,036	0,032
LVT(%)	4,3	5,6	5,2	5,9
SVT(%)	2,2	1,7	1,9	1,5
HVT(%)	93,4	92,7	92,9	92,6

The number of logical instances increases as each security measure is added to the design. The encryption-only design has a gate count increase of 18,5 kGE, the authentication-only design has a gate count increase of 17,8 kGE, while the design with both measures combined has a gate count increase of 21 kGE. The gate counts do not account for the SRAMs in the design, for they are also not taken into account in the compared works. The area results also increase aligned with the logical instances. The slacks are positive, so the setup timing checks are fulfilled. The last three elements in Table 4.3 are the percentage of the gates used in the synthesis. Low threshold (LVT), standard threshold (SVT) and high threshold (HVT) gates are used throughout the design for high performance, trade-off, and low-power consumption, respectively. The percentage of LVT cells increases as the design gets more complex to meet the timing constraints. If a critical path violates the constraints, the tool tries to meet them by placing SVT or LVT cells instead of HVT cells. The power consumption results are illustrated in Figure 4.3 for the different versions of the design.

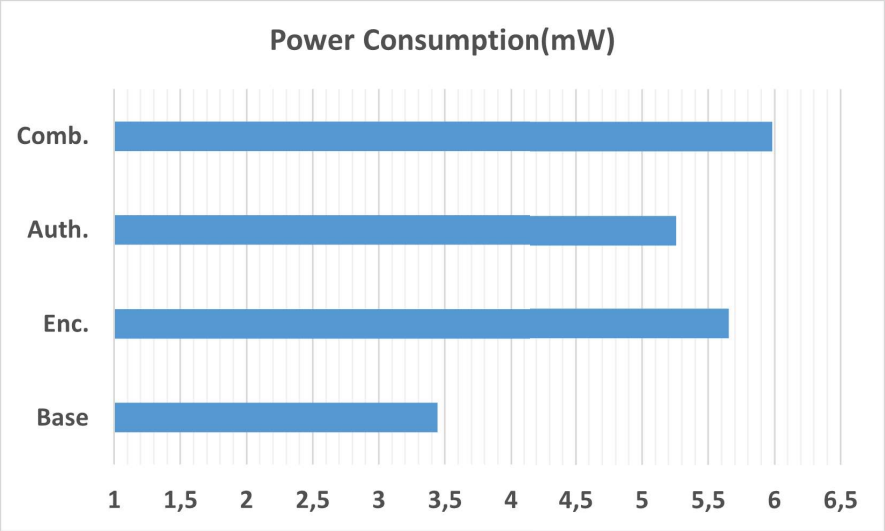

Figure 4.3. Power consumption analysis after the synthesis in TSMC 65 nm.

Power consumption results are calculated by creating an activity file in the Toggle Count Format (tcf) and inserting it into the synthesis tool. This file keeps how many

times each net switched during the simulation. The designs are run at 200 MHz, and the gate types are shown in Table 4.3. The power consumption overheads are consistent with the FPGA power consumption results. The encryption adds 2,2 mW, the authentication adds 1.8 mW and both combined add 2,55 mW power consumption overhead.

5. EXPERIMENTS AND RESULTS

This chapter explains the experiment setups and the evaluation criteria and compares the proposed design with the literature regarding these criteria. The ASIC synthesis and the FPGA implementation results are combined when the comparison is made. In other words, longer programs such as benchmarks are run on the FPGA to obtain the performance metrics while verifying the design in hardware. On the other hand, the hardware parameters like power consumption, frequency, and area results are obtained from the ASIC synthesis. The performance metrics from the compared works are obtained by getting the average of different benchmark scores. Furthermore, some works did not implement their designs in hardware. Thus, their power and area overhead results are obtained by using an average result from other hardware implementations depending on the algorithms used for encryption and hashing.

The experiment setup is explained first, including the software environment, the FPGA setup, and the security tests. Then, the evaluation criteria used to compare the proposed work with the literature are explained. Finally, the experimental results and the system's status regarding security are presented, and the system is compared with the literature.

5.1. Experiment Setup

The proposed system is tested using functional simulation, post-synthesis simulation, static timing analysis (STA), linting, clock-domain-crossing checks, and hardware verification on FPGA. The design is functionally verified from bottom to top by simulating each block with random stimuli and observing internal signals along with assertion results. The top level is tested by generating software for the system using the RISC-V toolchain and running the software in simulation and FPGA implementation.

The designed hardware is verified by checking the timing reports of ASIC synthesis and FPGA implementation to see if there are any violations. Also, clock domain checks are performed to guarantee no metastability in the design other than the RNG with Vivado for FPGA implementation and with Synopsys Spyglass, also used for RTL linting for ASIC synthesis. Gate level simulation is performed after the ASIC synthesis to check if there is any X propagation or any discrepancy between functional simulation and gate level simulation.

5.1.1. Software Environment

The software is generated using RISC-V GNU compiler toolchain and creating the necessary files [38]. The linker file is created to make the compiler know the address space of our processor. The main memory is placed at 0xC0000000, and the instruction memory is placed at 0x10000000. The program counter starts at the instruction memory base address, which starts the boot procedure. The load address of all the executable instructions and data generated after compilation is specified as the instruction memory, while the active address is set as the main memory. A bootloader is placed at the beginning of the executable code that loads the payload software from its start address and stores it in the main memory. Then, it jumps to the start address of the main memory after clearing all the register files.

The compilation generates hex, bin and assembly files. The hex file is stored in the instruction memory, while the assembly file is used to check the running of the code. The bin file is used to run the code on Spike, a RISC-V ISA Simulator [39] to compare the outputs with the processor system. A UART programmer module is placed in the instruction memory for the FPGA implementation to load a new program into the system without implementing the SoC again. RISCV-DV is used to generate random instruction streams [40] specifically for our system to verify correct behaviour by comparing with Spike outputs.

5.1.2. FPGA Setup

Kintex Ultrascale Development Board was selected for FPGA implementation because it contained a DDR4 controller IP and a DDR4 memory chip and was available in our lab. The native interface of the DDR4 controller IP is used as the interface, which is converted to the custom bus protocol used in the system. The RNG circuit was implemented with a special technique as mentioned in Section 4.1, and it was tested separately from the rest of the system. A UART interface with an asynchronous FIFO is used to collect the generated random data which is not placed on the system.

The instruction memory is generated from BRAMs, which can be programmed using a UART interface to program the processor without another bitstream generation. The programming code first sends a secret code to insert the instruction memory into programming mode. Then, the number of instructions is sent to inform the system when to stop receiving instructions. When the programming is over, the instruction memory sends a reset signal to the rest of the processor system to start again. The program is sent over the UART with a TTL cable connected to a computer that serially sends the instructions from a file using a Python program.

5.1.3. Security Tests

The system's security against malicious attacks such as bus snooping and replay attacks is tested on the FPGA implementation. For this purpose, an agent with the capabilities described in Figure 2.3 and Figure 2.4 is designed and placed between the secure memory controller and the DRAM. This agent can read the contents of the DRAM, can snoop the memory bus, return false data, and return old data and hash couples. Therefore, it covers all the attacks against which the system is made secure.

All zero values are written to a memory location for the data theft attack, as shown in Figure 5.1. The secure memory controller writes these values to the DRAM

by encrypting with ASCON. After the writing of these values is over, the agent is started by using a switch on the board, which begins reading that specific address and sending the results through a UART interface. The resultant values are shown in Figure 5.2. The data values are turned into a grayscale image in Excel. The encrypted values are random, and they are mostly different from each other, even though their plaintext values are equal. Even though the counter values are also equal for these blocks, the encrypted values are different because their address is different and used in creating the seed, and the strong encryption algorithm has diffusion and confusion properties. It is impossible to obtain the plaintext from these values without knowing the key.

Figure 5.1. Full zero data values.

The agent is used to write a different value to a memory location which the processor reads for the tampering attack. The integrity check controller detects the integrity failure in the first level check and enters the processor into exception. The exception handling software for this application was to jump the PC to 0x00000000 address where the instruction is a jump to itself, so the processor waits at this address

on a memory exception. For the replay attack, the agent stores the data and hash couple for a specific address at one time. Later, when it detects another write operation to that data value, the agent changes the data and hash couple in the memory with the old one. Therefore, when the processor sends a read request to that data, and the secure memory controller reads the hash to check the integrity, the old hash value returns with the old data value. The first level of the integrity check passes, but the next level of traversal fails the integrity check, and the system is put into exception mode. All these tests combined show that the system is secure against data theft, tampering and replay attacks.

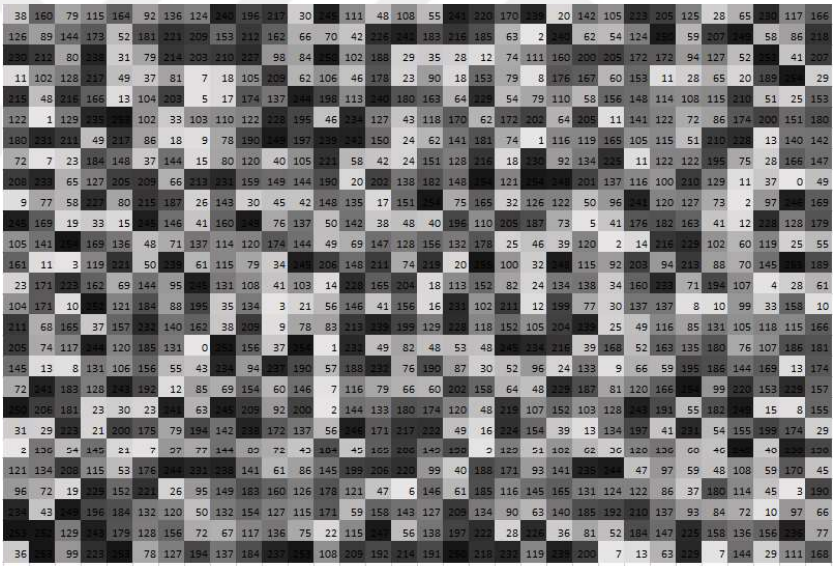

Figure 5.2. Encrypted data values.

5.2. Evaluation Criteria

The system is compared against the literature regarding area overhead, power consumption, memory storage overhead, and performance slowdown. Further, different

versions of the proposed system are implemented to evaluate the effect of each parameter on the whole system. For example, the metadata cache size is changed from zero to 32 KB to see the performance impact of the cache mechanism, the hash size is changed to see the storage overhead, and each method is implemented separately to see the impact on all the evaluation metrics.

The area results are obtained from the ASIC synthesis reports of Genus and the FPGA implementation reports of Vivado. The power consumption results are obtained from the Genus synthesis reports by inserting a simulation activity file that contains the number of switching times for all nets. The storage overhead is calculated by assuming a data space of 1 GB which contains 262144 4-KB pages, and each page contains 64 512-bit data blocks. The number of first-level hashes and counters is calculated using these values. The storage of the hashes in the integrity tree is calculated from the number of counter blocks. The table for these calculations for 1 GB data memory is shown in Table 5.1.

Table 5.1. The storage size of each data type

Data Type	**Size**
Total Storage(MB)	1170,3
Data(MB)	1024
Data Hash(MB)	128
Counter Blocks(MB)	16
Counter 1st Level Hash(MB)	2
Counter 2nd Level Hash(KB)	256
Counter 3rd Level Hash(KB)	32
Counter 4th Level Hash(KB)	4
Counter 5th Level Hash(B)	512
Counter 6th Level Hash(B)	64
Total Metadata(MB)	146,3

The performance results are obtained from the FPGA implementation by running benchmarks and random stress tests. In order to measure the execution time, a 64-bit timer is placed on the SoC with an address of 0x20000000, which is running at the same frequency as the processor. All the benchmarks use this timer with the help of a function in the software driver. Coremark [41] and Dhyrstone [42] benchmarks for RISC-V are used for performance evaluations. Also, using RISCV-DV, a memory-intensive stress test and a jump stress test programs are generated for performance evaluations.

5.3. Experimental Results

In this section, the proposed secure memory controller is compared with the literature regarding the previously mentioned metrics. Furthermore, different design versions are analyzed to obtain the best result. Table 5.2 compares the proposed design with the most relevant works in the literature in terms of hardware cost, storage overhead, power consumption and performance overhead. The SMC stands for the proposed secure memory controller.

Table 5.2. Comparison of the proposed work with literature.

Comparison Metric	**[15]**	**[22]**	**[26]**	**[18]**	**[17]**	**[23]**	**[20]**	**SMC**
Storage Overhead(MB)	128	425	215	30	20	220	512	137
Area Overhead(kGE)	350	350	500	360	350	380	24	21
Performance Dec.(%)	10	5	2	5	2	2	15,5	17
Power Cons. Inc.(mW)	>16	>18	>18	>16	>16	>17,3	N.A	2,6
Bus Traffic Inc.(%)	30	35	32	32	29	29	27	29

The storage overhead is calculated by the amount of extra security metadata for a 1 GB data space. The area overhead is the gate element representation for the area

of the ASIC implementations. Some results are obtained roughly by the algorithms used in the designs since results were reported with no hardware implementation. Performance decrease, power consumption increase, and memory bus traffic increase are calculated by how much they change compared to the baseline system when the security measures are added. They are calculated in terms of the ratio since each work's baseline and technology are different. The baseline in the proposed work is the PICORV processor with a 32 KB last-level cache which is connected to the DRAM with a latency of 30 cycles.

The proposed design has a higher storage overhead than only [15, 17, 18]. [17, 18] store their hash values in the ECC chip of the DRAM module to reduce the storage overhead, which makes the method only suitable for systems with ECC support. The hash size of [15] is 56 bits reducing the storage overhead at the cost of lower security and higher design complexity since efficiently storing 56-bit hashes into 512-bit memory blocks increases the area overhead. The proposed design is only comparable with [20] in terms of area overhead because it was also an IoT secure memory implementation. They used lightweight ciphers and hash functions with low hardware cost, but the combined cipher and hash function in the proposed design reduces the hardware cost. Furthermore, the design in [20] is vulnerable to replay attacks.

Most of the designs in Table 5.2 use parallel encryption and hash functions to reduce the latency at the cost of area overhead, which improves the performance. The proposed design has a close performance metric with [20], which uses a direct encryption method with low hardware cost, which reduces the performance significantly, but no replay attack protection is provided. The performance decrease in this thesis seems acceptable for a huge area overhead since IoT applications do not require very high performance. The performance decrease with respect to area overhead is also illustrated in Figure 5.3 where the best is the lower left corner of the graph. The power consumption also decreased due to the reduced memory bus traffic compared to other works with complex cryptography algorithms, where some works do not include power

analysis. The bus traffic is lowered by stopping the re-encryption sequence and the integrity tree traversal by snooping caches for necessary data.

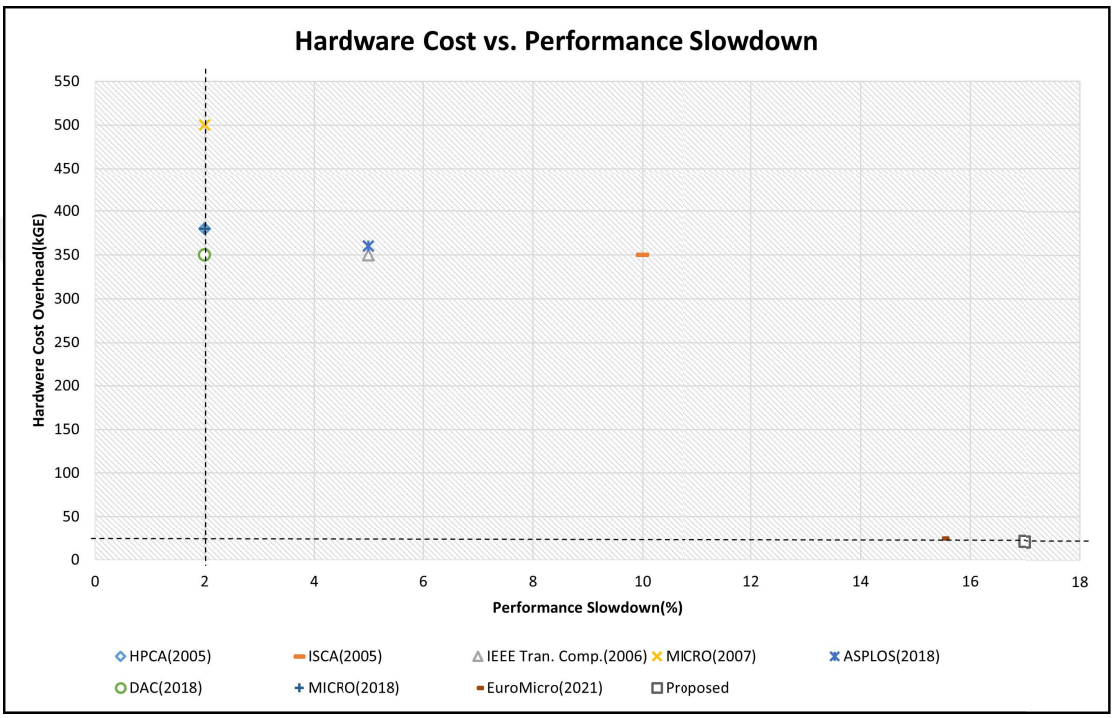

Figure 5.3. Hardware cost overhead vs. performance slowdown graph.

Different designs employ different hash sizes, which affect the security and the storage overhead for which hash size is the most significant contributor. The storage overhead with respect to the hash size is shown in Figure 5.4 for this work. Hash sizes smaller than 56 bits are not considered secure for memory authentication applications. A length of 64 bits is chosen for this design, with sufficient security and reasonable storage overhead.

The design's performance is measured using four different software programs, including coremark and dhyrstone benchmarks, memory intensive and jump stress tests generated from RISCV-DV. Four different versions of the design are implemented, consisting of the baseline, with only encryption, authentication, and all security measures.

The coremark results for all these versions are shown in Figure 5.5. All benchmark results are shown in terms of normalized execution time in Figure 5.6.

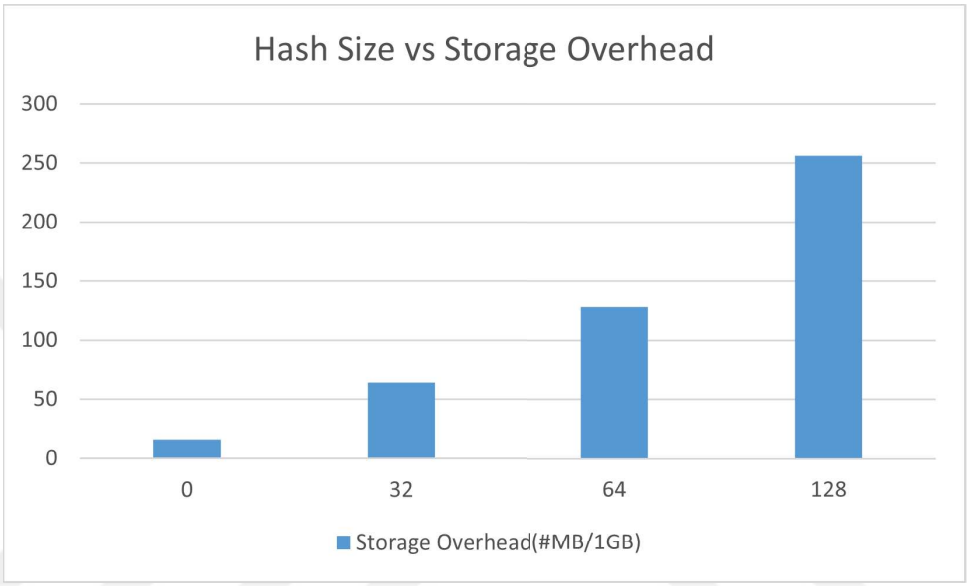

Figure 5.4. Storage overhead vs. hash size.

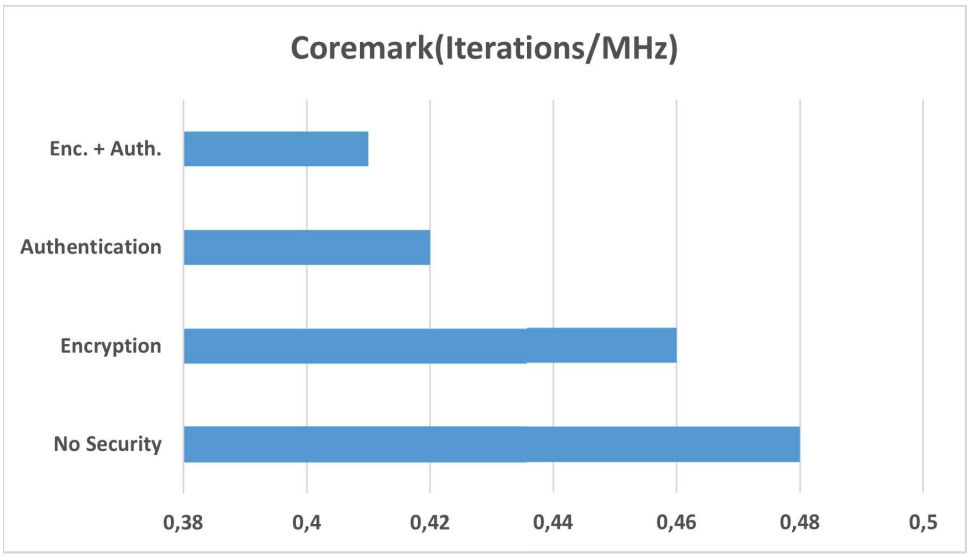

Figure 5.5. Coremark results for different versions.

The memory-intensive workload has the largest execution time overhead, as expected, since it accesses the memory in a random manner and much more than other workloads. Memory authentication has a larger performance overhead than memory encryption since the counter mode encryption hides the latency of the cipher while the hashing function can only start once data has arrived. Also, the integrity tree forms a large portion of the performance overhead and bus traffic since it results in more than one extra memory access. The average results of the performance measurements are also shown in the same graph, which are taken as the comparison metrics in Table 5.2.

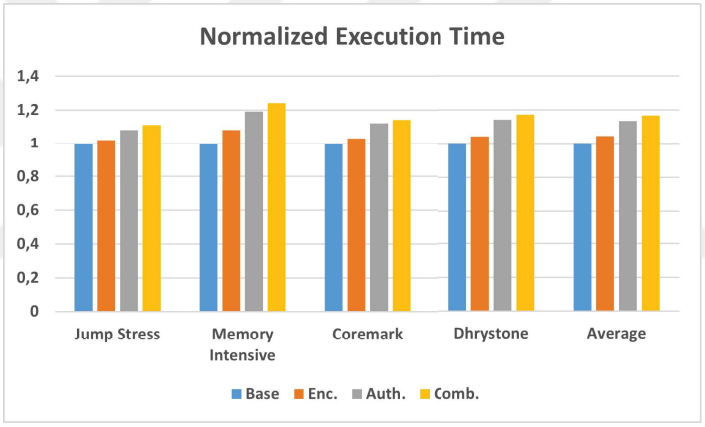

Figure 5.6. Execution time for different workloads.

Metadata cache size also affects the system performance. Different-sized versions are implemented, and the impact on the performance and the bus traffic is measured and shown in Figure 5.7. Adding an 8-KB metadata cache to the design improves the performance significantly since it reduces the bus traffic. When a counter or a hash is found in the cache, the latency of fetching from the DRAM is completely eliminated. The spatial locality increase as the integrity tree level increases also helps the cached design performance increase. The performance metrics are also the average of the workloads from Figure 5.6. The performance slowdown improves as the cache size increases, so a 32-KB metadata cache is used in the design as the performance improvement of larger cache sizes is not high compared to the area overhead [18].

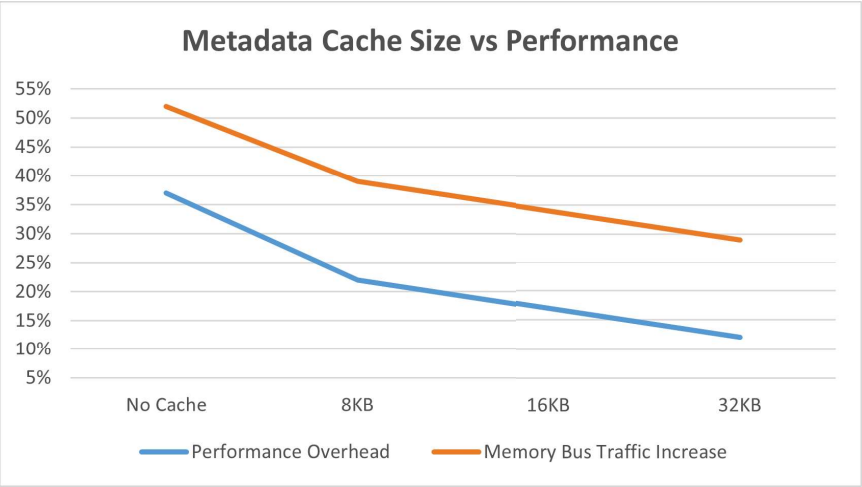

Figure 5.7. Metadata cache size vs. performance and bus traffic graph.

Table 5.3. NIST 800-22 test suite results.

NIST Statistical Tests	P-Value	Pass Rate
Frequency	0.656634	318/320
Block Frequency	0.082702	317/320
Cumulative Sums	0.617605	319/320
Runs	0.911413	318/320
Longest Run	0.068198	315/320
Rank	0.075138	318/320
FFT	0.235285	317/320
Non-Overlapping Template	0.944403	318/320
Overlapping Template	0.907251	319/320
Universal	0.911413	319/320
Approximate Entropy	0.855534	317/320
Random Excursions	0.451234	203/205
Random Excursions Variant	0.663130	204/205
Serial	0.448892	318/320
Linear Complexity	0.927083	316/320

The design is secure against data theft attacks, data tampering attacks and replay attacks, as explained in Section 5.1.3. The statistical properties of the RNG are tested by collecting 320 bitstreams with 1 million bits. These bitstreams are tested with the NIST 800-22 test suite [43]. More than 311 successful bitstreams out of 320 fulfil all of the tests in the suite means that the statistical properties of the number sequence are similar to random numbers. The entropy source's randomness and successful statistical properties prove that the RNG is secure. The RNG also has a 60% lower area overhead than the traditional tetrahedral oscillator-based RNG [37]. The NIST test suite results are shown in Table 5.3.

The RNG fulfils all the statistical tests for more than enough bitstreams. In other words, the statistical quality of the number sequence produced with this RNG is sufficient for 40-MB long data. The average p-values are also in the acceptable range, ensuring that the test results for all the bitstreams are evenly distributed.

6. CONCLUSION

The security of the user data in processor systems has become very important as the devices have become more easily accessible with IoT technology. Many attacks have been performed on the off-chip memory to steal user data or manipulate the processor with the wrong data. Different methods have been implemented to prevent these attacks, such as encryption and authentication. Most of these methods employ high-cost cryptography algorithms like AES and SHA, while NIST is creating a new standard for lightweight cryptography algorithms.

This thesis proposes a memory encryption and authentication solution with a low hardware cost and power consumption. The design uses the NIST LWC competition finalist ASCON for both encryption and authentication to reduce the area overhead and power consumption. Split counter mode encryption reduces the storage overhead and improves performance by decreasing the re-encryption cost. 64-bit major counters are combined with 7-bit minor counters to protect 4-KB pages and 512-bit blocks in these pages, respectively. The data cache is snooped during the re-encryption sequence to reduce the memory traffic. Bonsai Merkle tree design is used to protect the memory against replay attacks where the tree is constructed over the encryption counters to reduce the storage overhead and improve performance. The metadata cache is snooped during tree traversal to reduce the memory traffic. A key generation module is also designed with a built-in TRNG with a novel ring oscillator circuit with high metastability and low hardware cost. An SoC is constructed by combining these modules and a secure on-chip storage block and connecting them with the system bus.

The SoC is designed using Verilog and implemented on the Kintex Ultrascale Development Board for hardware verification. The ASIC synthesis is performed using Cadence Genus on the TSMC 65 nm technology. The proposed design is compared with the literature regarding area, storage, power consumption and performance overhead. The storage overhead is calculated theoretically from the data memory size, page size,

counter size and hash size. The area and power consumption results are taken from the ASIC synthesis, where the power analysis is performed using a simulation activity file. The performance analysis is performed on the FPGA implementation by running Coremark and Dhyrstone benchmarks, memory intensive and jump stress test codes generated by RISCV-DV. Different versions of the design are implemented, where each security measure is implemented separately to analyze the individual effects. Different metadata cache sizes have been implemented to analyze their performance impact.

The proposed design has a significantly lower area overhead than the previous works in the literature at the cost of higher performance reduction, which is acceptable for IoT applications. The average performance reduction of the proposed design is 17% while the area overhead is 21 kGE. The storage overhead is 137 MB for 1 GB of data memory. The power consumption increases by 2,6 mW when all the security measures are used. The synthesis was run at 200 MHz, and the area of the design is 0.313 mm^2. The TRNG has a 60% decrease in the area compared to the classical version of the tetrahedral oscillator. The security of the design has been verified against data theft, tampering and replay attacks on the FPGA implementation. The design was functionally verified using simulation, assertions and the FPGA implementation by running test software on the processor. The RTL code was verified by linting on Synopsys Spyglass, and the timing analysis was performed in Genus.

The complete ASIC implementation on this technology or another can be among the future works. After the ASIC implementation, the chip can be manufactured, and the design can be tested at the chip level. Also, the split counter mode encryption algorithm can be improved by representing the counters differently to reduce storage overhead and improve performance without increasing the area too much. The secure memory controller IP can also be combined with other processor systems to observe its effect in terms of performance and its re-usability. Another idea could be combining this method with a compression algorithm to open space for the hash values along the data values to read the data and hash in parallel while decreasing the storage overhead significantly, as hashes occupy a considerable portion of the memory.

REFERENCES

1. Lindenlauf, S., H. Höfken and M. Schuba, "Cold Boot Attacks on DDR2 and DDR3 SDRAM", *2015 10th International Conference on Availability, Reliability and Security*, pp. 287–292, Toulouse, France, 2015.

2. Bauer, J., M. Gruhn and F. C. Freiling, "Lest we forget: Cold-boot attacks on scrambled DDR3 memory", *Digital Investigation*, Vol. 16, pp. S65–S74, 2016.

3. Yitbarek, S. F., M. T. Aga, R. Das and T. Austin, "Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors", *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 313–324, Austin, TX, USA, 2017.

4. Halderman, J. A., S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum and E. W. Felten, "Lest We Remember: Cold-Boot Attacks on Encryption Keys", *Communications of the ACM*, Vol. 52, p. 91–98, New York, NY, USA, 2009.

5. Su, Y. and D. C. Ranasinghe, "Leaving Your Things Unattended is No Joke! Memory Bus Snooping and Open Debug Interface Exploits", *IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pp. 643–648, IEEE Computer Society, Los Alamitos, CA, USA, 2022.

6. Lee, D., D. Jung, I. T. Fang, C.-C. Tsai and R. A. Popa, "An Off-Chip Attack on Hardware Enclaves via the Memory Bus", *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*, pp. 487–504, USENIX Association, Austin, TX, USA, 2020.

7. Xiao, Y., X. Zhang, Y. Zhang and R. Teodorescu, "One Bit Flips, One Cloud

Flops: Cross-VM Row Hammer Attacks and Privilege Escalation", *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*, p. 19–35, USENIX Association, Austin, TX, USA, 2016.

8. Gassend, B., G. E. Suh, D. Clarke, M. van Dijk and S. Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification", *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA '03)*, pp. 295–306, Anaheim, CA, USA, 2003.

9. Mosalikanti, P., C. Mozak and N. Kurd, "High performance DDR architecture in Intel® Core™ processors using 32nm CMOS high-K metal-gate process", *Proceedings of 2011 International Symposium on VLSI Design, Automation and Test*, pp. 1–4, Hsinchu, Taiwan, 2011.

10. Intel Corporation, *Desktop 4th Generation Intel® Core™ Processor Family, Desktop Intel® Pentium® Processor Family, and Desktop Intel Celeron® Processor Family*, 2015, https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/4th-gen-core-family-desktop-specification-update.pdf, accessed on 27 October 2022.

11. Kaplan, D., J. Powell and T. Woller, *AMD MEMORY ENCRYPTION*, 2016, https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, accessed on 27 October 2022.

12. Rambus, *Full Disk Encryption of Solid State Drives and Root of Trust White Paper*, 2019, https://go.rambus.com/thank-you-full-disk-encryption-of-solid-state-drives-and-root-of-trust, accessed on 27 October 2022.

13. Shi, W., H.-H. S. Lee, M. Ghosh and C. Lu, "Architectural Support for High Speed Protection of Memory Integrity and Confidentiality in Multiprocessor Systems", *Proceedings of the 13th International Conference on Parallel Architecture*

and Compilation Techniques, 2004. PACT 2004., pp. 123–134, Antibes, Juan-les-Pins, France, 2004.

14. Shi, W., H. Lee, M. Ghosh, C. Lu and A. Boldyreva, "High Efficiency Counter Mode Security Architecture via Prediction and Precomputation", *32nd International Symposium on Computer Architecture (ISCA'05)*, pp. 14–24, Madison, WI, USA, 2005.

15. Gueron, S., "A Memory Encryption Engine Suitable for General Purpose Processors", *IACR Cryptology ePrint Archive*, pp. 204–217, 2016.

16. Shi, W. and H.-H. S. Lee, "Authentication Control Point and Its Implications For Secure Processor Design", *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pp. 103–112, Orlando, FL, USA, 2006.

17. Yitbarek, S. F. and T. Austin, "Reducing the Overhead of Authenticated Memory Encryption Using Delta Encoding and ECC Memory", *55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, San Francisco, CA, USA, 2018.

18. Taassori, M., A. Shafiee and R. Balasubramonian, "VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures", *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*, p. 665–678, Association for Computing Machinery, New York, NY, USA, 2018.

19. Fakhrzadehgan, A., Y. N. Patt, P. J. Nair and M. K. Qureshi, "SafeGuard: Reducing the Security Risk from Row-Hammer via Low-Cost Integrity Protection", *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 373–386, Seoul, Republic of Korea, 2022.

20. Köylü, T. C., H. Okkerman, C. R. W. Reinbrecht, S. Hamdioui and M. Taouil, "Protecting IoT Devices through a Hardware-driven Memory Verification", *24th*

Euromicro Conference on Digital System Design (DSD), pp. 115–122, Palermo, Italy, 2021.

21. Merkle, R. C., *Secrecy, Authentication, and Public Key Systems.*, Ph.D. Thesis, Stanford University, 1979.

22. Yan, C., D. Englender, M. Prvulovic, B. Rogers and Y. Solihin, "Improving Cost, Performance, and Security of Memory Encryption and Authentication", *33rd International Symposium on Computer Architecture (ISCA'06)*, pp. 179–190, Boston, MA, USA, 2006.

23. Saileshwar, G., P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao and M. K. Qureshi, "Morphable Counters: Enabling Compact Integrity Trees For Low-Overhead Secure Memories", *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 416–427, Fukuoka, Japan, 2018.

24. Guo, Y., A. Zigerelli, Y. Cheng, Y. Zhang and J. Yang, "Performance-Enhanced Integrity Verification for Large Memories", *International Symposium on Secure and Private Execution Environment Design (SEED)*, pp. 50–62, Washington, DC, USA, 2021.

25. Zhang, Y., L. Gao, J. Yang, X. Zhang and R. Gupta, "SENSS: security enhancement to symmetric shared memory multiprocessors", *11th International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 352–362, San Francisco, CA, USA, 2005.

26. Rogers, B., S. Chhabra, M. Prvulovic and Y. Solihin, "Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly", *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pp. 183–196, Chicago, IL, USA, 2007.

27. Paar, C. and J. Pelzl, *Understanding Cryptography: A Textbook for Students and*

Practitioners, Springer, Berlin Heidelberg, 2010.

28. National Institute of Standards and Technology (NIST), *FIPS 46-3 Data Encryption Standard (DES)*, NIST, 1977.

29. National Institute of Standards and Technology (NIST), *FIPS 197 Advanced Encryption Standard (AES)*, NIST, 2001.

30. Bogdanov, A., L. Knudsen, G. Leander, C. Paar, A. Poschmann, M. Robshaw, Y. Seurin and C. Vikkelsoe, "PRESENT: an Ultra-Lightweight Block Cipher", *Cryptographic Hardware and Embedded Systems - CHES*, Vol. 4727, pp. 450–466, Berlin, Heidelberg, 2007.

31. National Institute of Standards and Technology (NIST), *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, NIST, 2015.

32. Dobraunig, C., M. Eichlseder, F. Mendel and M. Schläffer, "ASCON v1.2", *Submission to the NIST Lightweight Cryptography Standardization Process*, 2021, https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf, accessed on 19 October 2022.

33. Clifford, W., *PicoRV32- A Size-Optimized RISC-V CPU*, 2017, https://github.com/YosysHQ/picorv32, accessed in August 2022.

34. Waterman, A., K. Asanovic and J. Hauser, "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203", *RISC-V International*, December 2021.

35. Primas, R., *NIST LWC Hardware Reference Implementation of Ascon v1.2*, 2020, https://github.com/ascon/ascon-hardware, accessed on 21 October 2022.

36. Liu, D., Z. Liu, L. Li and X. Zou, "A Low-Cost Low-Power Ring Oscillator-Based

Truly Random Number Generator for Encryption on Smart Cards", *IEEE Transactions on Circuits and Systems II: Express Briefs*, Vol. 63, No. 6, pp. 608–612, 2016.

37. Günay, R. and S. Ergün, "IC Random Number Generator Exploiting Two Simultaneous Metastable Events of Tetrahedral Oscillators", *IEEE Transactions on Circuits and Systems II: Express Briefs*, Vol. 67, No. 9, pp. 1634–1638, 2020.

38. Dabbelt, P., *RISC-V GNU Compiler Toolchain*, 2016, https://github.com/riscv-collab/riscv-gnu-toolchain, accessed on 25 September 2022.

39. Waterman, A., *Spike RISC-V ISA Simulator*, 2017, https://github.com/riscv-software-src/riscv-isa-sim, accessed on 25 September 2022.

40. *RISCV-DV*, 2018, https://github.com/google/riscv-dv, accessed on 28 September 2022.

41. Torelli, P., *Coremark*, 2018, https://github.com/eembc/coremark, accessed on 3 November 2022.

42. Weicker, R. P., *Dhyrstone Benchmark Program*, 2018, https://github.com/sifive/benchmark-dhrystone, accessed on 4 November 2022.

43. Rukhin, A., J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray and S. Vo, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications", *National Institute of Standards and Technology*, 2010.