UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Self-healing and secure low-power memory systems

Mădălin Neagu

SELF-HEALING AND SECURE LOW-POWER MEMORY SYSTEMS

Author:
Mădălin NEAGU

Supervisors:
Joan FIGUERAS
Salvador MANICH

September 2017

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Enginyeria Electrònica

SELF-HEALING AND SECURE LOW-POWER MEMORY SYSTEMS

Tesi doctoral presentada per a l'obtenció del títol de Doctor per la Universitat Politècnica de Catalunya, dins el Programa de Doctorat en Enginyeria Electrònic

Author:
Mădălin NEAGU

Supervisors:
Joan FIGUERAS
Salvador MANICH

Barcelona, September 2017

Abstract

Memory systems store critical information in any digital system, thus they are susceptible to transient errors and are the focus of various types of attacks. It is crucial for a memory system to keep the information as accurate as possible. There is a need to design, implement and test systems capable of handling errors by themselves, thus, to run autonomously. Self-healing capabilities for memory systems translates into error detecting and correcting codes and replacing/replicating methods of memory elements. Security and data privacy is difficult to implement in memory systems, due to the overwhelming variety of attacks. This thesis proposes strategies against specific attacks that can occur in memory systems. The self-healing methodology and the security solutions are evaluated from varied perspectives: performance, area and delay overhead, and power consumption.

Keywords— error detection, error correction, memory systems, data scrambling, cache memories, side-channel attack, simple and differential power analysis

Prologue

The main objective of this thesis is to bring new contributions to the self-healing and secure systems domain. In particular, to develop a self-healing technique for memory systems and to increase security of memory systems, techniques which favor low-power consumption. In order to achieve the main objective, three major research objectives were proposed: design of an error detection and correction scheme for errors that occur in memory systems and integrate them in a memory system, design techniques to increase the security and data privacy of memory systems against different types of attacks and to combine the previous two into a single solution, in order to achieve a self-healing and secure low-power memory system. The low-power aspect of the proposed solutions and techniques is evaluated during design stage and afterwards through simulation. Also, the architectures are evaluated from several other points of view, such as error detecting and correcting performance, area and delay overhead, and security efficiency.

I want to thank all the people who have helped and supported this doctoral thesis. I would like to express my sincere gratitude to prof. Salvador Manich for introducing me to the security domain, for the continuous support of my Ph.D study and related research, for his patience, motivation and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I also want to thank prof. Joan Figueras for indicating a research direction and for stimulating me during the doctoral program and, last but not least, I want to thank prof. Liviu Miclea for the support and encouragement.

I would like to thank my family, especially my wife and parents for supporting me spiritually throughout writing this thesis and my life in general.

This PhD thesis has been partially sponsored by the Spanish government project TEC2013-41209-P.

Contents

List of Figures

List of Tables

List of Acronyms

BCH . . . . . Bose Ray-Chaudhuri
CPU . . . . . Central Processing Unit
DCM . . . . . Duty Cycle Modulator
DRAM . . . . . Dynamic Random Access Memory
ECC . . . . . Error Correcting Code
EDC . . . . . Error Detecting Code
EDLC . . . . . Error Detecting, Localization and Correcting
FA . . . . . Full Adder
IC . . . . . Integrated Circuit
IST . . . . . Interleaved Scrambling Technique
KB . . . . . Kilo Bytes
LFSR . . . . . Linear Feedback Shift Register
LUT . . . . . Look-Up Table
MECU . . . . . Memory Encryption Control Unit
OSV . . . . . Old Scrambling Vector
PRNG . . . . . Pseudo-Random Number Generator
R-S . . . . . Reed-Solomon
SAS . . . . . Set Address Synchronized
SAU . . . . . Set Address Unsynchronized
SEC-DED . . . . . Single Error Correction, Double Error Detection
5EC-6ED . . . . . 5 Errors Correction, 6 Errors Detection
SER . . . . . Soft Error Rate
SRAM . . . . . Static Random Access Memory
ST . . . . . Scrambling Table
YSV . . . . . Young Scrambling Vector

Chapter 1

Introduction

1.1 Context

In nowadays technology, there are more and more huge and complex digital systems. Because of the continuous evolution of the technology node, the components and circuits are getting smaller, they occupy less area, have higher speeds and favor low-power consumption. Thus, the systems are more complex and can handle huge amounts of data at the same time. As the complexity increases, devices become resource and power hungry, hence it must be kept under balance. For any given domain, a system must be able to perform complex computations, to output high speed and to keep the information as accurate as possible.

Modularity plays an important role in any system, each one is composed of several subsystems, which perform specific tasks (e.g. monitoring, planning, optimization, data acquisition, etc.). This characteristic is better understood when considering the reliability and maintainability points of view, because if one subsystem fails, it can be easily replaced by a spare one. Also, the system can be easily extended because the architecture of such a system is highly scalable.

Autonomous systems are also a current tendency. Such systems are able to monitor themselves, they can repair themselves if a subsystem is faulty, and they can adapt to environment changes (e.g. new components, errors occurrence, etc.). The main characteristic of autonomous systems is the capability to run without human intervention. To achieve such features, there are several "self-*" type properties, like: self-configuration, self-management, self-healing, self-optimization, self-protection, etc. Each one of these properties is a priority for autonomous systems.

Although the technology node decreases almost every 2 years and offers higher density and higher speeds for integrated circuits, the occurrence of errors increases

and needs to be mitigated. The errors severely affect any component of a system, but can be mitigated through several methods, techniques and algorithms.

Another relevant trend is the security of devices and systems. There are a large variety of attacks which target memory systems and try to retrieve important information such as encryption and public keys, digital signatures, biometric information, etc. However, developing a system that is secure against every type of attack is nearly impossible due to the diversity and nature of attacks and infeasible because of performance, power and size considerations.

There is a need to design, implement and test systems that can handle errors, to be tolerant to faults. Memory systems are a special and critical category, because they store data which is afterwards used by the system. When errors occur, the system must be able to perform self-healing, to detect and correct the errors, which is achieved by implementing several self-healing methods and techniques. On the other hand, when attacks eventuate, the system must either repel them or disclose useless data. However, the cost for implementing a self-healing memory system must be as low as possible and a trade-off between performance and cost is necessary.

1.2 State of the art

1.2.1 Motivation

Most modern systems present novelty through the self-healing concept, which can be modeled on any given type of system and can be built through several methods, functions and techniques that must accomplish the self-healing objective. This tendency can be extended to high availability and stability (measured by uptime), increased reliability, high maintainability and scalability. Any modern system is desired to have no errors (if some occur, they should be corrected) and to keep the above tendencies as high as possible. So far, the self-healing concept is widely used in distributed network systems, along with other "self-*" features. Nevertheless, it can also be employed on any system, even on memory systems which encounter high error rates and are accustomed to any type of attacks.

Memory systems are critical in any large and complex digital design. This is because memory systems retain the information which is then used in processing elements of the digital design. The concerns regarding memory systems are because the technology minimum feature size keeps decreasing, therefore, the memory elements are shrinking, have higher density and speed. In addition, the percentage of memory area in System on a Chip (SOC) integrating circuits is expected to grow significantly during the next decade. According to the ITRS Roadmap 2013 [1], the size of the area occupied by memory increases faster than the area dedi-

cated to logic. There is a need to keep the information as accurate as possible by using mitigation techniques. Because the size of the circuitry is getting smaller and smaller, high-energy neutrons and alpha-particle strikes (background radiation) can generate soft (i.e. transient) errors in a memory system and can affect more than one location. Therefore, multiple-bits errors may occur in any memory system, whether it is SRAM or DRAM. There is a need to keep them error free, since critical data is stored at any given time.

Self-healing methods, techniques and algorithms have been employed in order to mitigate these types of errors. For memory systems, self-healing is perceived as error detecting and correcting codes and replacing/replicating methods for the elements of a memory system. Self-healing is mainly used for autonomous systems (i.e. systems that can adapt themselves to the given circumstances and perform repair of erroneous elements), but the concept is widely used for any system that can perform self-monitoring, planning and repair operations without the human assistance.

On the other hand, a low-power trend has emancipated and targets all systems to use as less energy as possible by using intelligent architectures, designs and techniques. This trend is mainly for limited energy source devices, because maximizing the life of the device is the main concern for these types of devices. The main idea of the low-power concept is to use as less power as possible when a device is operating and use no power when it's idle (i.e. no processing is done). However, this approach may limit the computation speed of a device. Therefore, a balance between power consumption and performance is needed and it depends very much on the design and implementation of the device. Any modern self-healing technique must have low area overhead and low power consumption, while the performance degradation must be kept to a minimum.

Data security in memory systems is another growing trend in the last decade. Modern day ICs contain significant information that must be highly secured, so that it stays hidden. In the last 5 years, there have been reported a large variety of attacks against ICs and memories, mostly targeting the vulnerabilities of cache and main memories, smartcards, etc.

In a recent report [2], the author pointed out that 62% of companies worldwide were subject to payment fraud in 2014 and that credit/debit cards are the second most frequent target of payment fraud. Mobile payments are a relatively new payment method, but this trend is increasing among large companies and organizations. However, there are several uncertainties about it such as disclosure of sensitive information or secure transfer of information.

Ensuring the confidentiality of sensitive information is becoming more and more crucial because targets such as credit cards and other kind of legal supports store biometric and personal information [3]. Very often general purpose devices

like desktops, laptops or smartphones are used for private transactions with financial entities or health-care issues, among others. In the case of devices without specialized hardware, all cryptographic operations are executed in software, resulting in an intensive use of memory [4]. This poses sensitive data at risk, including that stored in the cache memory [5].

Solving the challenges imposed by the trends explained above is the main purpose of this work, which are important in any modern device or system. Achieving data security, low power consumption, high performance, small area overhead and new self-healing techniques for memory systems is a priority in nowadays electronic engineering.

1.2.2 Introduction

In modern technology, the concept of a self-healing system is becoming increasingly common in memory systems. This trend revealed itself due to some factors and requirements: autonomous systems, employing "self- features", the evolution of the technology node, multiple random errors, etc. Nowadays, the technology node used is 45nm, 32nm and even 22nm, but it's still decreasing [1]. By scaling down the components of any system, a gateway for errors and faults has been created. This scaling affects the dimensions of circuits, the operating voltage and current, and the power consumption [1]. Decreasing the operating voltage favors low power consumption when the circuit is active, but implies lower limits and thresholds for detecting the operating state (e.g. voltage levels for 1 and 0). Also, when a circuit is idle or deactivated, the leakage current must be negligible. Due to the reduced dimensions, the packaging is very dense, in order to achieve a considerable size. This high density favors, in a negative way, the occurrence of soft and hard errors. The balance is achieved somewhere in the middle, because the new technologies allow higher performance and higher switching speed, but have the disadvantage to be susceptible to any kind of errors.

In order to increase the reliability of systems against soft and hard errors, the self-healing concept emerged. This concept is part of the auto-management features that appeared in autonomous systems. The idea was developed by IBM in 2001 [6] with the purpose of creating a distributed system capable of self-management, which adapts to environmental conditions and defects that may occur, based on certain high-level decisions and policies. In such a self-manageable system, the human intervention is minimal and has the role to define the general policies and rules that govern the system. Thus, IBM defined four self-* properties:

- Self-configuration: automatic configuration of components and devices
- Self-healing: automatic discovery and correction of errors and faults

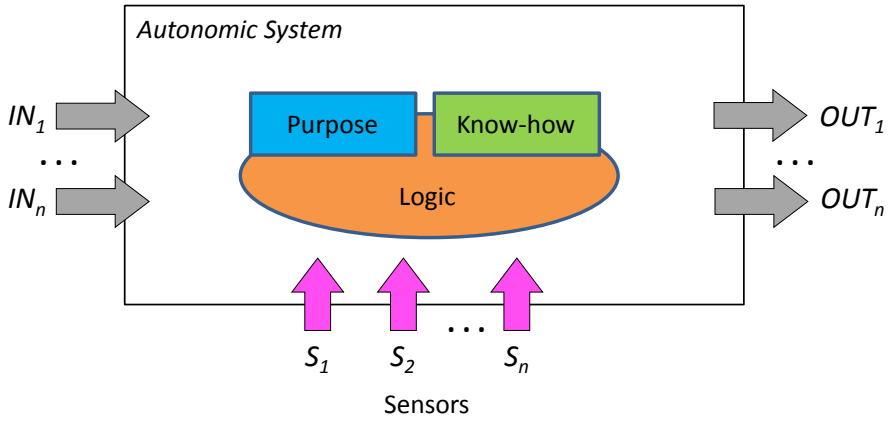


Figure 1.1: Conceptual model of the autonomic system [9]

- Self-optimization: automatic self-monitoring and resource control to ensure optimal functioning with respect to the requirements defined
- Self-protection: proactive identification and protection against random malicious attacks

Depending on the evolutionary level, IBM defined five levels of implementation [7]: the first level is the basic one which shows the current situation for manually managed systems, levels two to four introduce new automated management functions, and level five is the ultimate goal of autonomous systems, the self-management. Moreover, the complexity of an autonomous system can be simplified by using various design patterns, like Model-View-Controller (MVC) to improve the separation between system functions [8].

A basic concept of the autonomous systems is *closed control loops*. These loops monitor specific resources (hardware and software) and try to maintain autonomously specific parameters in a desired range. Depending on the size and complexity of the system, the control loops can be hundreds, even thousands.

The conceptual model of an autonomous system contains several vital elements: sensors, knowledge and logic (Figure 1.1). The sensors allow the system to observe the running operations, as well as the external environment. The knowledge is basically the intention and the ”know-how” to self-management, without external intervention. Current operations are dictated by the logic system, which is responsible for making the correct decisions, based on the internal knowledge and the information from sensors.

The model highlights the fact that the operation of an autonomous system is driven by purpose, by intention. This includes the mission (e.g. the service that it must provide), system policies (e.g. what defines the behavior) and "survival instinct". If this system would be a control system, it would be coded as a function of errors with feedback or as an algorithm combined with a set of heuristics, in an heuristic assisted system.

Although autonomous systems may have different objectives and behaviors, each autonomous system must exhibit a minimum set of properties to achieve its own goals. Thus, an autonomous system must be automatic, i.e. to be able to self-control internal functions and operations. Also, it must be independent and can start automatically, without external intervention. Another important feature is adaptability. It should be able to change the current operations (e.g. configuration, status, etc.), so allows the system to cope with spatial and temporal changes in the context of operations, either on long term (e.g. optimization, modifying specific parameters), or short term (e.g. malicious attacks, defects, faults, errors, etc.). The last important property is defined as the awareness of the autonomous system, meaning it is able to monitor (via sensors) the operational context, as well as the internal state, in order to achieve its computational objectives.

A memory system with self-healing features is a relatively new concept in the digital era and requires the existence of algorithms, methods and techniques implemented in the memory system, that try to keep the information as accurate as possible.

As technology evolves, memory systems have developed from small, slow and simple architectures to huge, fast and complex designs. This evolving trend was governed by the technology node which kept shrinking continuously. Therefore, the density and speed of integrated circuits have increased exponentially and the trend was described by Moore's law in 1965. However, the sources which generate errors and faults were not completely removed and still exist in several forms. This leads to developing techniques, methods and algorithms needed for the mitigation of errors in any given system.

In the ITRS Roadmap 2013 [1], it is shown that the area occupied by memory in SOC integrated circuits increases faster than the area dedicated to logic (Figure 1.2).

Low-power consumption is another trend for most systems, they favor low power consuming operations, conservation of energy and maximizing the life of limited energy sources. Figure 1.3 shows the power consumption trends for SOC integrated circuits, based on the ITRS Technology Roadmap 2013 [1].

Because there are several and different types of self-healing techniques and low-power architectures, the scope of this chapter is to summarize the existing and proposed methods of self-healing and models for low-power capabilities for

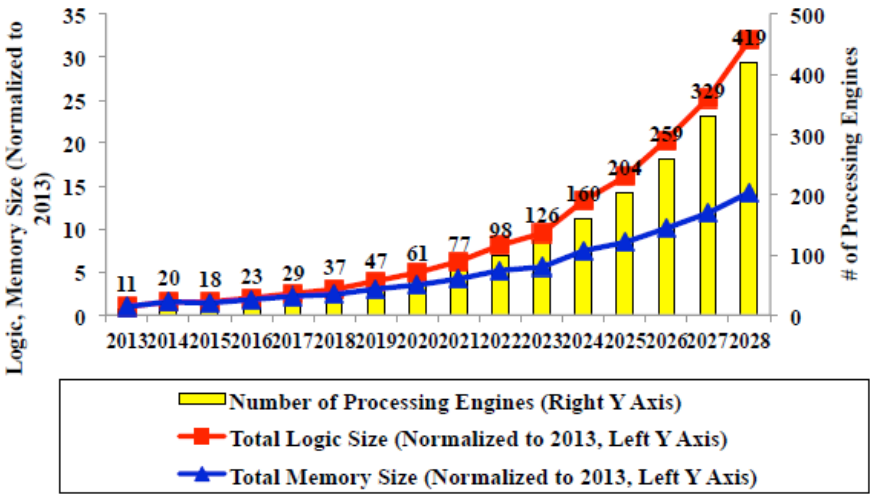


Figure 1.2: SOC Consumer Portable Design Complexity Trends [1]

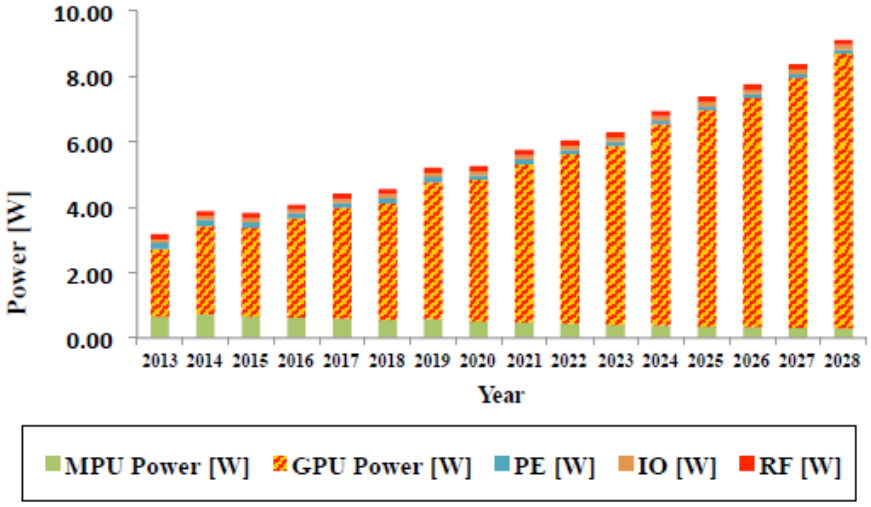


Figure 1.3: SOC Consumer Portable Power Consumption Trends [1]

memory systems.

This work is focused on current self-healing techniques for systems in general, but more in detail for memory systems. Also, because there are several types of memories that are used nowadays, the most important ones are taken into consideration: SRAM and DRAM. Also, data security in memory systems is tackled from different perspectives and security models and methodologies are proposed and analyzed.

1.2.3 The Self-Healing concept

Self-healing can be described in numerous ways, but when speaking about a self-healing system, the best definition is: "System automatically detects diagnoses and repairs localized software and hardware errors" [10]. Self-healing is just a type of the more general self-* properties. For example, self-configuration is defined as automated configuration of components and systems follows high-level policies. Self-optimization is known as automated tuning of the system parameters in order to improve performance and efficiency. Self-protection is well-known in security systems where they have the ability to defend against malicious attacks or cascading failures. Usually, an autonomic system must support these four features. An autonomic system is composed of several autonomic elements, each containing internal resources and delivers services to other autonomic elements.

Given the autonomic manager (which controls and represents an element), the managed element could be a hardware resource (e.g. storage), a CPU, hardware device or a software resource (e.g. database, directory service, etc.) Each element is responsible for managing its internal state and behavior and for managing its interactions with an environment that consists of messages and signals from other elements.

Self-healing techniques have been in continuous development, beginning with 1970s when fault tolerant computing was achieved through fault model specifications, fault avoidance, detection and masking. In 1980s, the methods have advanced to dependable computing where they focused reliability, availability, integrity, degradation and maintainability of systems. From 1990 until now, the term of self-healing was used more often when describing systems with capabilities of dependability, modeling, monitoring, diagnosis, planning and adaptation for repair.

Nowadays, a self-healing system can modify its behavior in response to the changes occurring in the environment. In [11], the author states that a self-healing system has in its lifecycle four major activities: monitoring the system at runtime, planning the changes, deploying the change descriptions and adopting the changes. Based on these activities, the architecture of such a system should include: monitoring the system's performance and recognition of faults or anomalies, adapt-

ability to structural, dynamic and run-time aspects, communication integrity and internal state consistency and ability to address the anomalies when discovered (planning, deploying and enacting the necessary changes).

In [12], the authors define, in a formal way, several "self" properties from the perspective of distributed systems. From the self-healing definition, we can infer that this technique is focused on maintaining or restoring a system's safety properties. When these are violated, healing may take an arbitrary but finite amount of time and guarantees the recovery from that subset of actions that perturbs its state. Systems with self-healing will often exhibit a degraded level of performance when the external action causes a crash failure. Most of the self-healing solutions are non-masking or reactive, but there are several proactive versions, known as predictive self-healing. These systems anticipate failures from symptoms of erratic behavior, but protect the system from catastrophic service disruption by internal restarting certain modules (also known as micro-rebooting) and masking the failure from the user.

Self-healing can be employed in a system by using different measures, most of them rely on the problems/errors/faults that can occur in the given system and, of course, on the architecture and design of that system. Thus, self-healing techniques implementations and proposals are addressed in Chapters 2, 3 and 4.

1.2.4 Memory systems

The memory system is the repository of information (data) in any computer system. A CPU will read the data, perform operations and writes it back to the memory. The memory system is composed of a collection of storage locations, organized in memory words and has a numerical address. A key performance parameter of a memory system is the effective speed of the memory. Capacity is important as well, but these properties are always in conflict. Usually, technology tradeoffs are employed to optimize between the two factors (note that cost also influences the tradeoffs). So, there are large but slow memories and fast but small ones.

Because latency (i.e. the delay from when the processor first requests a word from the memory until that word arrives and it's available for use) plays an important role when designing a memory system, it is important to engineer a memory system with the lowest latency. Therefore, a simple solution for a memory hierarchy is to use a small but very fast memory in front of the large, slow memory. This idea led to the hierarchy in Figure 1.4, which nowadays is very common.

At the top of the hierarchy are the CPU registers (very small and very fast), then the next level is a special, high-speed semiconductor memory known as *cache memory* and the last level is the main memory, semiconductor as well, but slower and denser than a cache. The cache memory is usually divided in several distinct

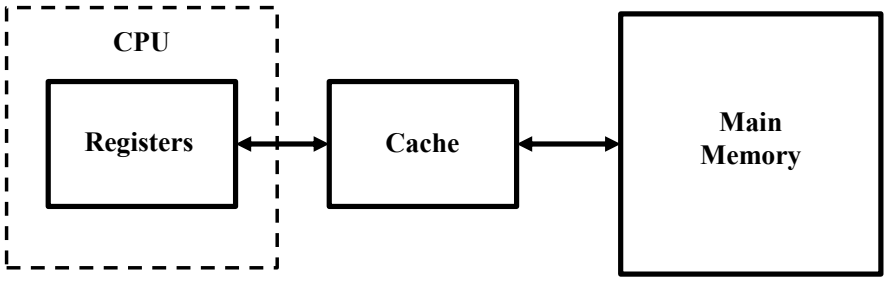


Figure 1.4: Common construction of a memory hierarchy [13]

levels (nowadays, at least 2 or 3 levels) and is situated on the CPU chip itself. This is explained below by how the cache memory works. Figure 1.5 exhibits a common cache memory design.

Because the cache memory acts like an intermediate between the main memory and CPU, it contains copies of data from the main memory. But because it has a small capacity, only some data is copied, depending on which technique is used. Common techniques are based on temporal and spatial locality. Spatial locality is the property that an access to a given memory location greatly increases the probability that adjacent locations will be accessed immediately. Temporal locality is based on the fact that an access to a given memory location increases the probability that the same location will be accessed again. These techniques are used for cache management strategy, when the cache memory is full and free blocks are necessary. Also, common cache memories can be associative, or content-addressable. An associative memory, the address of a memory location is stored along with its content.

When reading from the cache memory, a hit occurs if the needed word is found in a level of the hierarchy, however if a *miss* occurs the request must be sent to the next lower level. So, for multi-level caches, the processor proceeds by checking the smallest *level 1* cache first and if it hits, the processor runs at high speed. If level 1 cache misses though, the next larger *level 2* cache is checked and so on, until the main memory is finally checked.

1.2.5 Low-power systems

This concept is currently employed in most modern chips in order to reduce the *dynamic power* (when the circuit or device is executing instructions – normal operations) and the *static power* (when the device or circuit is in idle mode). Although

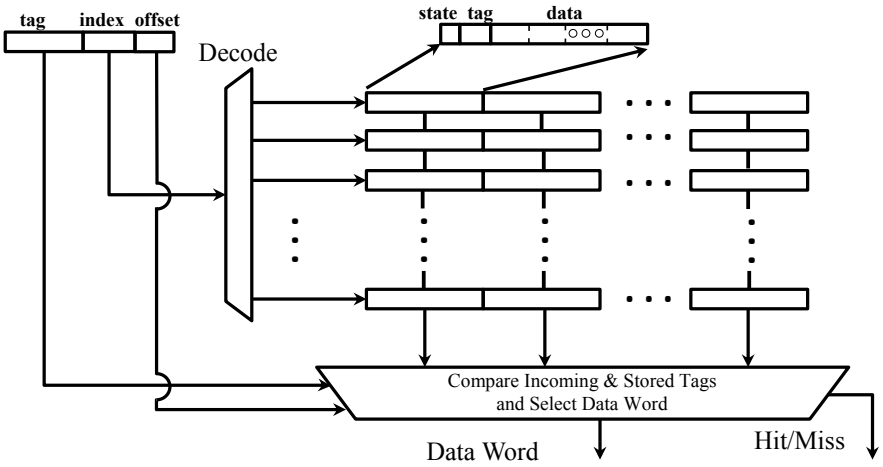


Figure 1.5: Components of a cache memory [14]

the density and the speed of integrated circuits computing elements have increased almost exponentially for several decades (trend described by Moore's Law), these are limited primarily by power dissipation concerns [15].

There are several methods and techniques to reduce the power consumption low enough so that the device or circuit still performs in normal parameters. Dynamic power is mainly reduced to a minimum by using a "full static logic" which can stop the clock and hold their state indefinitely, thus using no dynamic power, but they still have a small static power consumption caused by leakage current [16]. The static power appears when there is no switching (idle mode) and it's the result of sub-threshold leakage, which is important if the circuits keep on shrinking. A simple technique to reduce this loss is to raise the threshold voltage and decrease supply voltage, but this will reduce the performance. Some modern circuit use dual supply voltages to provide speed for critical parts of the circuit, and lower power for non-critical paths [17]. Another method is to use sleep transistors to disable entire blocks when not in use.

Another method to reduce the dynamic power when changing states is to reduce the operating voltage of the circuit or to reduce the voltage change involved in a state change (use only a fraction of the supply voltage). For clocked logic circuit, the technique of clock gating [18] is used, to avoid of functional blocks that are not required for a given operation when changing the state.

A power management technique in computer architecture is dynamic voltage scaling [19], where the voltage of a component is increased or decreased, depend-

ing on the circumstances. Undervolting is used to conserve power, especially in devices with a limited energy source. Most modern components and devices allow voltage regulation to be controlled through software (e.g. BIOS).

To summarize, the main advantages of low-power systems is that they have a reduced power consumption, but at the expense of computation power. Several methods, techniques and implementations have been proposed so far, each having its own advantages and disadvantages. However, because the low-power designs depend very much on the architecture of the memory system or on the techniques proposed, examples of low-power implementations are included in each chapter. The low-power aspect is considered in each method and technique from each chapter and mainly focuses on designs that have low power consumption, hence the impact is minimal.

1.2.6 Security in memory systems

Modern day integrated circuits (ICs) contain significant information that must be highly secured, so that it stays hidden. In the last 5 years, there have been reported a large variety of attacks against smartcards and ICs dedicated to security mostly targeting the vulnerabilities of cache and main memories. This problem broadens when the attacks focus on devices used by general public like credit cards and other kind of legal supports containing biometric information [3]. Another report [2] points out that 62% of companies worldwide were subject to payment fraud in 2014 and that credit/debit cards are the second most frequent target of payment fraud. Mobile payments are a relatively new payment method, but this trend is increasing among large companies and organizations. However, there are several uncertainties about it such as disclosure of sensitive information or secure transfer of information.

Ensuring the confidentiality of sensitive information is becoming more and more crucial [3]. Very often general purpose devices like desktops, laptops or smartphones are used for private transactions with financial entities or health-care issues, among others. In the case of devices without specialized hardware, all cryptographic operations are executed in software, resulting in an intensive use of memory [4]. This poses sensitive data at risk, including that stored in the cache memory [20].

Computer systems have drawn a lot of attention, because of the fact that sensitive information is stored in the memory during runtime. Some attacks on the main memory have been investigated because the stored information is still visible a small period of time after the system loses power (known as memory remanence). These attacks, called cold-boot attacks [21, 22], target the recovery of the last information which was stored in the main memory. Thus, relevant information such

as encryption keys could be retrieved. Moreover, by keeping the memory module at low temperatures, the information can remain recoverable for as long as 5 min [21].

In memory systems, the main memory has usually a low level of protection because it is more accessible than other levels of memory and thus it can be removed by the attacker by simpler technical means. In cold-boot attacks, the information is frozen for a long enough period of time such that the memory modules can be removed and the attacker can download the content into a backup system which stores data in plain text. To avoid this, security is improved by encrypting data while the memory transaction is undertaken [22, 23, 24, 25]. The decrease of bus throughput can be compensated by an increase of L2 cache size.

Unlike the main memory nowadays, for performance reasons, the cache memory is placed in the same central processing unit (CPU) package, either stacked on or embedded within it. This configuration provides a higher protection degree at a free cost. Because of this and also to avoid strong penalty in the throughput between cache and CPU, the information is stored in plain text. Even more, some encryption algorithms are designed to run all the rounds using only cache memory and avoiding main memory transfers [26].

Attacks that target the cache memory and retrieve sensitive data have been analyzed and investigated in several works [24, 25, 26, 27, 28]. Most of these are side-channel attacks, which are based on the information leaked by a cryptographic device, such as power consumption, timings and so on. Attacks against Advanced Encryption Standard (AES) algorithms have been reported that target the stored private keys [24, 25, 26]. It cannot be ruled out the possibility of direct reading of the cache content [27]. If the cache is conveniently frozen, a power-up sequence can derive to the recovery of the same content existing before the power-up [29].

To sum things up, ensuring data security and privacy in digital devices is challenging and, because of the large variety of attacks, there are several methods and techniques that can be implemented. In Chapter 3 and 4, existing security measures are analyzed and discussed, as well as known attacks against memory systems are explained in detail.

1.3 Objectives

The main objective of this thesis is to bring new contributions to the self-healing systems domain. In particular, to develop a self-healing technique for memory systems, technique that favors low-power consumption and provides security of data against different types of attacks.

A self-healing low-power memory system is usually achieved by using error

detection and correction codes and/or replacement techniques using spare elements. Data security and privacy on the other hand, can be achieved through several methods and means. Thus, there were defined three major research objectives.

The first one is the design of error detection and correction codes for memory systems. This was motivated by the fact that the memory systems are susceptible to soft and hard errors, in particular, DRAMs present unidirectional errors. This led to the Berger code, an error detection code for unidirectional errors, with a simple and straightforward design. The new codes must also be evaluated from different points of view, such as error detection and correction rate and performance and area overhead. A specific task here is to integrate the proposed codes in a memory system, in order to achieve a self-healing methodology for memory systems. Also, other mitigation techniques can be used, in order to improve the capability of the self-healing technique (especially when dealing with uncorrectable errors). The overall system design must also be evaluated, especially from the perspective of speed, area overhead of additional circuitry and power consumption. The low-power feature is achieved through the design of the proposed codes.

The second research direction is related to data security and privacy in memory systems. Considering the performance peculiarities of cache memories and the existence of a large variety of attacks against them, data scrambling can be used and integrated due to the minimal penalty they impose in performance. Hence, this research direction targets the design and implementation of a novel methodology that can obscure the critical data in the cache memory, rendering it useless for the attacker.

The third and last research direction is to integrate the results of the first two research directions in order to obtain a secure self-healing low-power memory system. The motivation behind is related to power analysis side-channel attacks, in particular simple and differential power or electromagnetic attacks. These types of attacks are widely used in memory systems because they are very powerful and resourceful in breaking systems protected by keys. By combining error detection and correction codes with data scrambling techniques, memory systems can be designed to defend themselves against a large variety of attacks. The overall design must be evaluated in terms of performance, delay penalty, area overhead and security.

In order to better exhibit and demonstrate the objectives, below are some questions (related to the objectives) which are answered in the final chapter, 5.

- Which was the main objective of this thesis?
- Is the main objective accomplished?
- How is the self-healing memory system achieved?

- Is the low-power perspective fulfilled?
- How is the security in memories dealt with?
- From the evaluation point of view, can the proposed methodology be considered a good solution?
- Can the self-healing technique be further researched and used in other systems?

1.4 Structure

The thesis is organized in five chapters. The first chapter contains a short introduction of the domain and subject of the thesis, motivation, objectives and this sub-chapter Also, the biggest sub-chapter here contains survey of previous works in this domain. It begins with a motivation of the thesis and continues explaining the systems focused in this work. Then, the self-healing concept is presented and afterwards memory systems are described. Note that the review of memory systems contains both SRAM and DRAM, old and new designs. Also, the unidirectional nature of errors is explained and referenced. The security in cache memories is described from several points of view, data scrambling is introduced and power analysis attacks are explained. The rest of the chapter contains recent proposals and implementations of self-healing techniques for memory systems, as well as low-power strategies.

Chapter 2 contains the design of the proposed error detection and corrections codes, which are used for the self-healing technique. The codes are explained and illustrated through several figures and methods. Then, they are analyzed from the following points of view: coding scheme, error localization, error correction and error escapes. For the latter three, metrics are defined, in order to evaluate the code effectiveness. The rest of this chapter contains the integration of the proposed codes into memory systems, presented as DRAM repair strategies.

In Chapter 3, the security in cache memories is illustrated and explained. Data scrambling is presented as a method to obscure the stored data in the cache and the design methodology is exhibited. The implementation is explained, illustrated and evaluated from several points of view. Finally, the experimental results are presented and discussed accordingly.

Chapter 4 consists of the merge between error detection and correction codes from Chapter 2 and the data scrambling technique presented in Chapter 3. The power analysis side-channel attacks are introduced and explained in a sub-chapter and then the methodology proposed against these types of attacks is presented. The

evaluation process contains metrics such as performance, area overhead, power consumption and, in the end, the experimental results are unveiled.

The final Chapter, 5, contains the conclusions of the work, scientific contributions and future research directions.

Chapter 2

Unidirectional eDLC

2.1 Introduction

The continuous shrinking of memory cells size as the technology evolves has made them more susceptible to errors. The sources that cause the single or multi cell upsets, and therefore generates multi-bit errors, have been observed and studied intensively. As reported in [30], [31], soft errors due to memory cell upset can be induced from terrestrial cosmic rays (i.e. high-energy neutrons), nowadays more important than alpha-particle induced errors. Because of the high packing density of the DRAM cells, the upset can be on multiple adjacent cells and creating burst errors in the word line [32].

The memory cell upsets in DRAM are unidirectional, which means that the errors occur for all memory cells from 1 to 0 or from 0 to 1, but not in both ways. This is because a charge in the capacitor of the memory cell represents either logic 1 or 0, depending on the implementation [33]. A particle strike can upset the cell, discharging the capacitor, thus changing its original value.

Error detection and correction codes (EDC & ECC) have been used in memories in order to deal with errors of low multiplicity. These codes are based on computing the check bits for the original information data and appending them to the data bits. The simplest form is the parity bit which is an extra bit added to the data that represents the XOR (or its complement) of all the information bits. If an odd number of errors occur, the parity bit will have a different value, hence the error(s) are detected. For an even number, the errors will not be detectable.

In order to detect and correct errors, Hamming codes have been widely used. The most commonly known is SEC-DED (Single Error Correct – Double Error Detect) such as Hamming codes [34], [35], upon which various improvements have been made in order to be able to detect and correct multiple errors [36], [37], [38].

Although these codes are powerful, they require large hardware overhead, thus increasing chip area and requiring higher power consumption and higher latency when computing the check bits. Power also increases due to detecting and correcting the errors because of the complex computations. Obviously these classes of codes are not adequate for errors of high multiplicity concentrated in one word.

BCH codes are a class of parameterized error-correcting codes. The advantage of these codes is the simplicity of the electronic hardware to perform the syndrome decoding. Based on these codes, a good example of error correcting code which has the ability to correct 5 and detect 6 errors (called 5EC6ED) is proposed in [39]. The authors use this ECC architecture to correct errors in a memory which has more than 2 errors on a line. Also, because it is a high latency ECC, they use a mechanism of disabling the specific erroneous line called bit-fix [40]. Single errors are handled by a quick and simple ECC, thus reducing the power and improving speed. The downside is when multiple errors occur, because the erroneous word line is disabled and cannot be used until corrected, if possible.

Even more powerful codes such as the Reed-Solomon (RS) [34] have been proposed. The processing is done in symbols; a symbol consists of several bits. The code is efficient when detecting and correcting random multiple-bit burst errors. For t check symbols, the RS code can detect up to t erroneous symbols and correct up to $t/2$ symbols. However, the computations for detecting and correcting are complex and, thus, it has a low speed and a high area overhead. Because of the complex algorithm, several proposals have been made in order to improve the speed and minimize area and power [41], [42]. An interesting concatenation of RS and Hamming codes has been proposed in [43].

None of the codes above consider the fact that DRAM cell upsets produce unidirectional errors. A well known EDC is the Berger code [44]. The check bits are created by expressing in binary the sum of all the 1s (called B_1 encoding scheme) or 0s B_0 encoding scheme) in the information bits. Berger codes are able to detect any number of errors appearing in the information bits. However, localization and correction of errors is not possible. Other issues are the speed and area overhead, especially for large word sizes. Several proposals have been made in order to improve the speed and area. In [33], the authors propose a tree-shaped parallel counter using partial sums to compute the Berger code check bits.

2.2 Motivation

Developing and employing error detection and correction codes for memories is? challenging. There are several facts that need to be taken into consideration, such as:

1. Every EDC/ECC imposes a performance penalty on the memory operation. Hence, an adequate scheme/solution must be chosen, considering the memory type, usage and performance.

2. EDCs and ECCs generate redundant bits for specific number of data bits, hence hardware overhead estimation is needed.

3. What kind of errors (permanent or transient) does it need to detect and correct? And what system is it for?

While employing an existing code into a memory system is somehow common, most of the time these solution might overkill the functionality and performance of the memory system. Hence, error detection and correction codes tend to become more flexible, more usable in almost every scenario. This trend has motivated the development of a new technique, designed for a specific memory (DRAM), with resilient characteristics due to its architecture and capable of detecting and correcting multiple unidirectional errors.

Unidirectional soft errors are common in DRAM cells due to the capacitor architecture. A charged capacitor in a cell usually indicates a binary value of 1 (it can be a zero, depending on the other components in the cell), which basically means that any change in the capacitor voltage results in an alternation to the binary value. Soft errors in DRAM cells are possible to be induced from high-energy neutrons or alpha-particle rays. But these can only alter a charged capacitor, thus to discharge it. The reverse action is nearly impossible in a normal environment, except research laboratories.

Flexible error detection and correction codes are desired in any memory system, because of the architectural nature or characteristics. It can be scaled to any size, type or access times, etc. No need to adapt every time new data enters the system, basically it self-adapts to the information size. Data redundancy can be high if correcting multiple errors, but a balance can be obtained, a balance between redundancy, power consumption and performance overhead. This balance is illustrated in Fig. 2.1 as a triangle. Good data redundancy and good power consumption might result in a bad error detecting and correcting performance, but good data redundancy and good error correcting performance ensues a bad/higher power consumption. On the other hand, having good error correcting performance with minimal power consumption generates a high/bad data redundancy.

2.3 Theoretical background

In this section, a short review of the current state-of-the-art is performed. The following subsections present several static and dynamic memory designs, sources

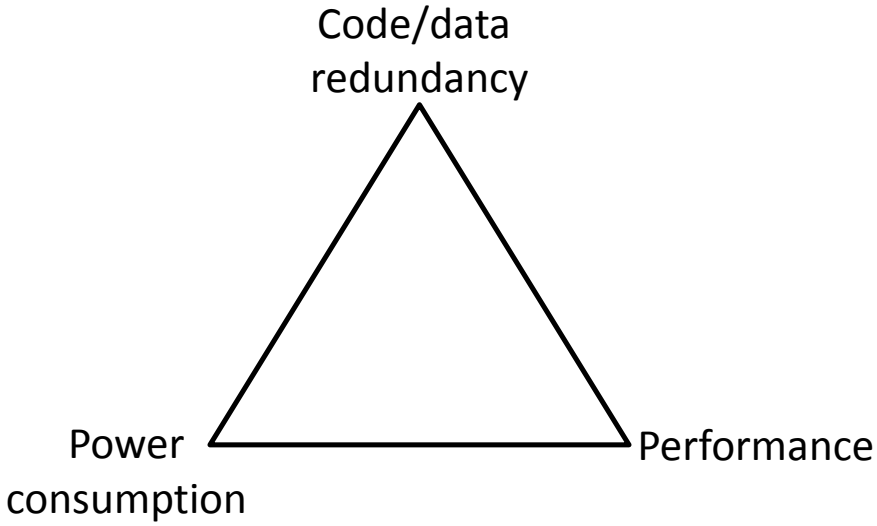


Figure 2.1: The triangle of balance for EDCs and ECCs.

of errors and error types in memories, error detecting and correcting codes and self-healing memory systems from the scientific literature.

2.3.1 Memory systems

The most used cell for cache memories is the SRAM. The most common implementation is 6T SRAM. As the name imposes, 6 MOSFET are used to store each memory bit. There are several implementations of SRAM, some that use more transistors than 6 in [45], [46] or even less in [47], [48].

The basic 6T SRAM is shown in Figure 2.2. Access to the cell is enabled by the word line (WL) which controls the two access transistors M_5 and M_6, which, in turn, control whether the cell is connected to the bit lines BL and $\overline{BL}$. They are used to transfer data for both *reading* and *writing* operations.

The SRAM operations are related to three different states: *standby* when the circuit is idle, *reading* when the data has been requested and *writing* when updating the contents. The *standby* state is when the word line is not asserted, therefore the access transistors M_5 and M_6 disconnect the cell from the bit lines. For *reading*, we assume that the content of the cell is a 1, stored at Q. The *reading* cycle starts with pre-charging both bit lines to 1, then asserting the word line, enabling both access transistors. Next, the values stored in Q and $\overline{Q}$ are transferred to the bit lines

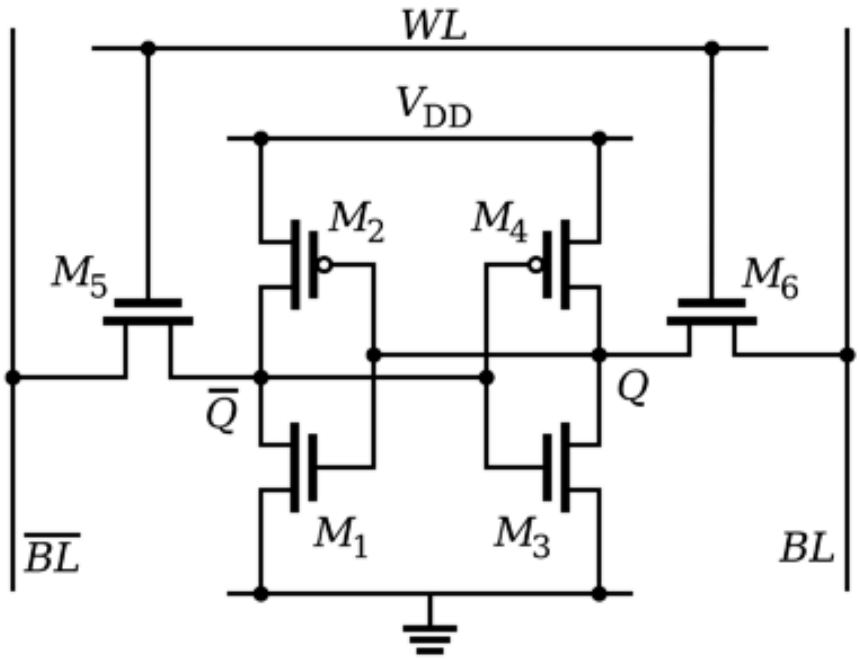


Figure 2.2: Basic 6T SRAM cell.

by leaving BL at its pre-charged value and discharging $\overline{BL}$ through M_1 and M_5 to a logical 0. On the BL side, the transistors M_4 and M_6 pull the bit line towards V_{DD}, a logical 1. The start of the *writing* cycle begins with applying the value to be written to the bit lines. For a logical 0, BL is set to 0 and $\overline{BL}$ to 1. Word line is then asserted and the value that is to be stored is latched in. This works because the bit line input drivers are much stronger than the relatively weak transistors in the cell itself, so that they can easily override the previous state of the cross-coupled inverters.

The main memory consists of cells of the same type, organized in arrays of rows and columns and may have several banks of these arrays. Nowadays, the most common basic cell can be a SRAM or DRAM. However, SRAM is more expensive, but faster and significantly less power hungry than DRAM. Also, they have a complex internal structure, therefore less dense than DRAM and is not used for high-capacity, low-cost applications such as the main memory.

A simplified design of an older 32MB DRAM chip is shown in Figure 2.3. A combination of the two control signals, the *Row Address Strobe (RAS)* and *Column Address Strobe (CAS)* selects a specific memory location, one byte in the figure

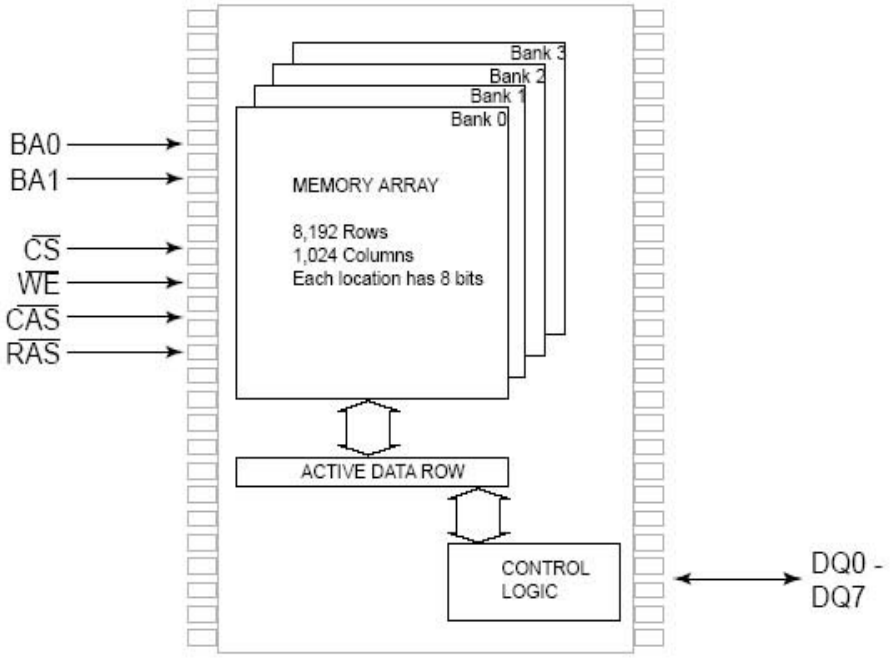


Figure 2.3: Older 32MB DRAM design.

below, for reading and writing. Note in the figure that the external pins no longer exist in modern DRAMs because the RAS and CAS signals are generated internally in response to external commands.

A newer design of the functional block of a DRAM is the one in Figure 2.4. The authors in [49] propose an efficient and flexible distributed interface for 3D-stacked DRAM. The term 3D-stacked DRAM is symbolized in Figure 2.4 by using more than one layer of active elements (they are integrated both vertically and horizontally into a single circuit): 4 banks of 128MB memory array, 4 row decoders and 4 column decoders. The improvement is the data bus (colored in red) which is bidirectional, and there are two independent buses for read and write operations, each of 32 bits. Note that in a conventional DRAM design, the data bus is bidirectional as well, but is only one bus that connects to the memory for both read and write operations and the bus width is 4 bits.

Another interesting design is the one in Figure 2.5. In [50], the authors propose a hybrid architecture for the main memory that uses 2 types of memories: DRAM and PRAM. Phase-change random access memory (PRAM) has lower read access time and standby power compared to DRAM, while having a comparable through-

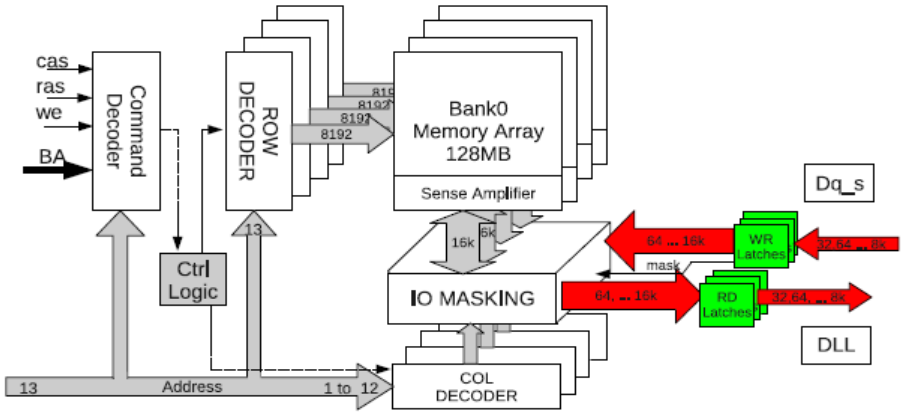


Figure 2.4: Modified DRAM design [49].

put. Also, the PRAM is designed to retain its data even after the power is turned off, unlike DRAMs which need constant refreshing power (i.e. to refresh the value stored in the capacitor, which wears out due to leakage and frequent accesses).

However, all these advantages come with a price. There is a higher power cost when writing a value to a PRAM cell and it has limited write endurance $(10^9 - 10^{12})$ cycles. In order to balance the advantages and disadvantages of both types of RAMs, the authors propose a hybrid memory architecture which consists of both DRAM and PRAM. Also, because the PRAM has limited write endurance, they propose a hybrid hardware-software based solution. The hardware is based in the memory controller which manages the access information to different PRAM pages based on an addition cache memory (Figure 2.5). The software portion is part of the operating system (OS) memory manager (in this case, page manager), which performs the wear leveling by page swapping/migration. The results based on several benchmarks demonstrate that the hybrid memory system proposed achieves up to 37% energy savings at low overhead, better overall energy and performance efficiency.

A DRAM stores each bit of data in a separate capacitor within an integrated circuit. The charged or discharged state of the capacitor represents the two values of a bit, a logic 1 for charged or 0 for discharged (or vice-versa, depending on the implementation). However, since capacitors leak charge, the information will eventually fade unless the capacitor is refreshed periodically. Only one transistor and one capacitor are needed for 1 bit, therefore a DRAM is much denser than SRAM and is mainly used in the main memory. Figure 2.6 exhibits a DRAM cell with one transistor and one capacitor.

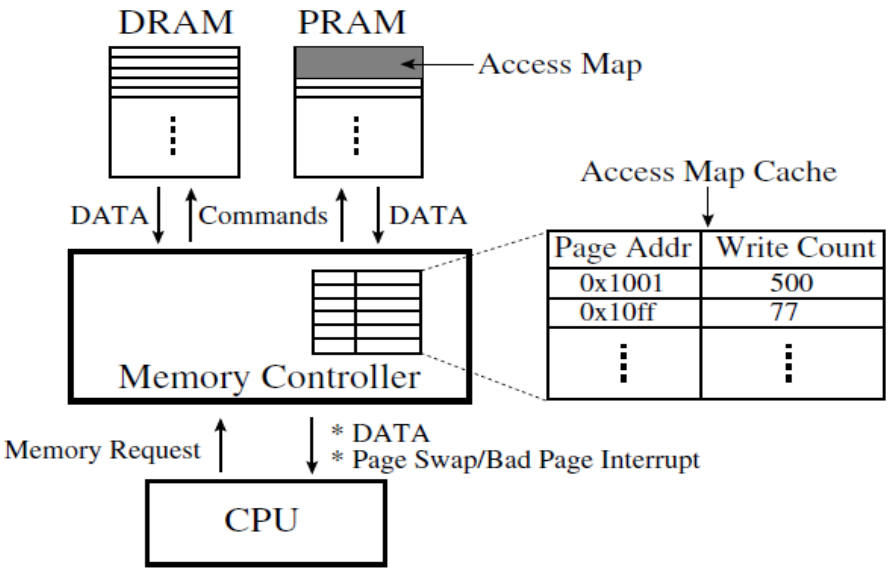


Figure 2.5: PDRAM memory controller [50].

Electrical and magnetic interference inside a system may cause a single bit DRAM to spontaneously flip to the opposite state. Research has shown that the majority of the one-off soft errors in DRAM chips occur as a result of background radiation, chiefly high-energy neutrons from cosmic rays secondaries or alpha-particle strikes, which may change the contents of one or more memory cells or interfere with the circuitry used to read/write them (in Figure 2.6, symbol ”n”). For SRAMs, neutron or alpha-particle strikes can upset the storage or access transistors, thus flipping its original value. But because the density of SRAMs is lower than DRAMs, fewer cells will be upset per unit strike. Figure 2.7 displays the ”insensitive state” of a DRAM cell [51] (i.e. the state of the storage capacitor which cannot be modified by radiation).

There is a concern that as DRAM density increases further, thus the components are smaller and operating voltages will decrease, more DRAM cells will be affected by such radiation, producing multiple-bit errors. However, a recent study in [30] shows that single event upsets due to cosmic radiation have been dropping dramatically with process geometry.

These problems can be mitigated by using DRAM modules that include extra memory bits and memory controllers that exploit these bits. These extra bits are used to record parity or to use an error-detecting or correcting code (EDC/ECC). These codes will be discussed in detail in this chapter.

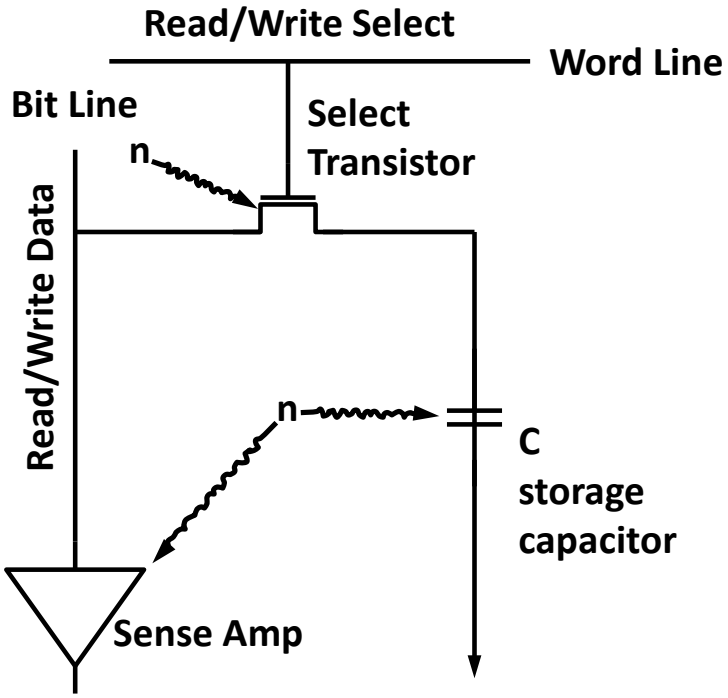


Figure 2.6: Basic DRAM cell [32].

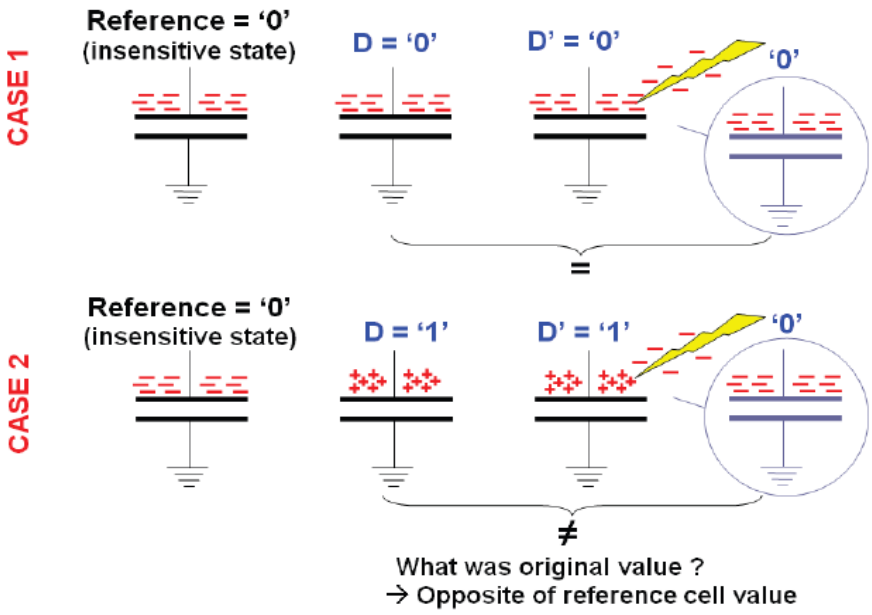


Figure 2.7: The "insensitive state" of a DRAM cell. One cell is used as reference to store the discharged state logic value (immune to radiations), useful data are stored in 2 other cells, D and D'. If the 2 values are different, the original data is sure to be the opposite of the reference cell value [51].

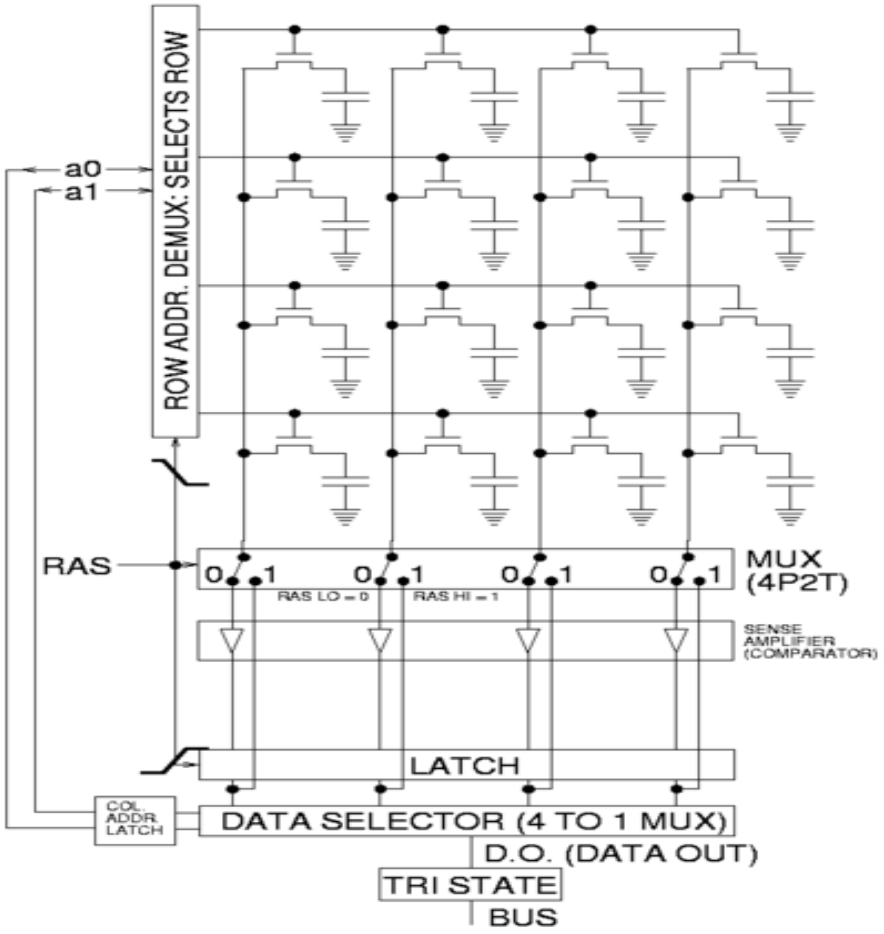


Figure 2.8: Principle of DRAM read operation

In order to read a bit from a column, there are several operations that take place and are shown in Figure 2.8 (4 by 4 matrix). First, the sense amplifier is disconnected, then the bit lines are pre-charged to exactly equal voltages that are in-between high and low logic levels. Afterwards the pre-charge circuit is switched off. Because the bit lines are relatively long, they have enough capacitance to maintain the pre-charged voltage for a brief time. The desired row's word line is driven high to connect a cell's storage capacitor to its bit line. This causes the transistor to conduct, transferring charge between the storage cell and the connected bit line. If the capacitor is discharged, it will greatly decrease the voltage on the bit line as the pre-charge is transferred to the storage capacitor. However, if the capacitor is charged the bit line voltage will increase slightly. Then, the sense amplifier is switched on and the positive feedback takes over and amplifies the small voltage difference between bit lines until one bit line is fully at the lowest voltage and the other one is at maximum high voltage. Once this has happened, the row is "open". All columns are sensed in simultaneously and the results sampled into the data latch. A provided Column Address then selects which latch bit is connected to the external port. Many reads can be performed quickly without extra delay sense for the open row since all data has been already sensed and latched. While the reading of all columns proceeds, current flows back up from the sense amplifiers to the bit lines of the storage cells, thus refreshing the charge in them by increasing the voltage in the capacitor if it was initially charged, or by decreasing it if it was initially discharged. When the reading of all the columns in the current row is done, the word line is switched off to disconnect the cell storage capacitors (i.e. row is "closed"), the sense amplifier is also switched off and the bit lines are pre-charged again.

To write a value to a cell, the row is opened again and a given column's sense amplifier is temporarily forced to the desired high or low voltage state, thus driving the bit line to charge or discharge the cell storage capacitor to the desired value. Due to the positive feedback, the amplifier will hold its state even after the forcing is removed. During a write to a particular cell, all the columns in a row are first sensed simultaneously just as in reading, then a single column's cell storage capacitor charge is changed, and finally the entire row is written back in.

The refresh logic is provided in the DRAM controller which automates the periodic refresh, so no other software or hardware has to perform it. Some systems even refresh every row in a burst of activity involving all rows in order to reduce the power consumption.

2.3.2 Sources of errors in SRAM and DRAM memories

As explained in the previous section, memory systems are susceptible to soft errors when facing high energy neutrons or alpha-particle strikes. These types of radiations can create upsets that generate transient errors in a memory. However, hard errors must not be neglected. Hard (i.e. permanent) errors are usually caused by faulty components or by being induced on the datapath. They are strongly correlated between them and are usually caused due to high system utilization, unlike soft errors which are random. Bianca Schroeder et. al. reported that the soft error rate (SER from now on) is much higher than previously stated [52]. Also, the uncorrectable error rate per machine is only 1.3%, thus, a first conclusion drawn is that error correcting codes are critical for reducing the large number of memory errors. Despite the high SER, the authors claim that hard errors still dominate soft errors. This is because each soft error will eventually be detected, either by the memory scrubber or by being accessed by an application. Also, events that cause soft errors happen randomly over time and are not in correlation with other errors, like hard errors. This work is focused on soft errors caused by radiation or leakage, which occur in DRAMs. Note that the self-healing strategies proposed can also work for hard errors.

Any memory cell, in SRAM or DRAM memories, stores a bit of information by accumulating a certain amount of charge in one or more parasitic capacitances. In SRAM cells the feedback maintains this amount of charge constant while in DRAM cells it decreases caused by the capacitor leakage and needs to be refreshed periodically. When a particle strike hits the cell it may change the bit state (produce a soft error) if it is able to inject an amount of charge over a critical amount Q_{crit} in the parasitic capacitances storing the bit. The Q_{crit} threshold directly depends of the size of this capacitance and is affected by the technology node.

The SER is defined as the rate at which a system encounters or is predicted to encounter soft errors and is expressed in Failures-In-Time (FIT=failures in 10^9 hours). In [52], the SER is reported as a FIT of 25,000 to 70,000 per Mbit and more than 8% of DIMMs affected per year. Both SRAMs and DRAMs are susceptible to soft errors, therefore it would be expected that the SER would increase as the technology node scales down because as the technology node decreases the threshold Q_{crit} for a soft error would be reduced. However, studies in [53], [5], [54] have shown that DRAMs will have a lower SER as the technology node decreases while SRAMs will have a constant SER (Figure 2.9). Note that the SER is calculated as FIT/bit and 1 FIT is 1 upset per 10^9 hours. The trend of SER for SRAMs is flat because the decrease in the cell critical charge Q_{crit} caused by the process shrink is compensated by the decrease in cell area. Notice that a smaller area reduces the probability that a particle strike fully hits a single cell, therefore limiting

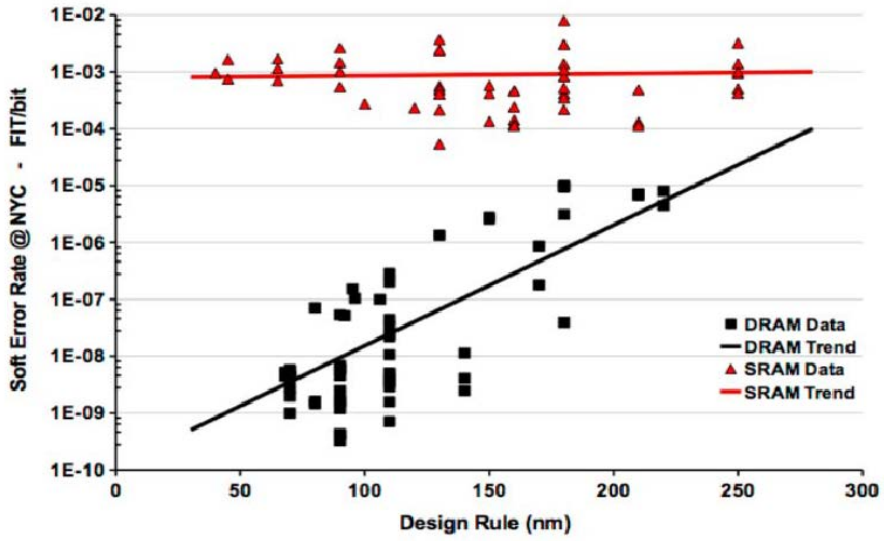


Figure 2.9: Soft Error Rate (SER) per bits in DRAM and SRAM [53].

the amount of charge injected. In DRAMs, the trend is downwards because the cell capacitance of C (Figure 2.6) is constant despite the cell area reduction which gives a constant Q_{crit} too. As a result, the area reduction dominates and the SER is expected to decrease in next generations of DRAM memories.

Although these recent studies show that DRAMs will have less SER as technology node shrinks, the problem is that because of this higher packaging density, a Single Event Upset (SEU from now on) may generate multi-bit errors. This is shown in Figure 2.10 and 2.11. Note that the SEU rate (R_{SEU}) is calculated as SER per 1 GB chip, and the Bit Error Rate (BER) contains both cell and address & control logic upsets. Also, the problem gets even worse when considering the two types of upsets that can be produced: cell and logic. The cell upsets are the ones that affect the DRAM cell itself, while the logic upsets occur after a particle strike in the memory address & control circuitry (Figure 2.12). As can be observed in Figure 2.10 logic upsets tend to generate a large number of multiple bit errors, over 1,000 in the figure.

The authors in the references above propose several ideas and the use of existing techniques to mitigate cell and logic upsets that may occur in the memory. The errors produced by cell upsets can be reduced or even removed by using ”memory cell interleaving”, parity bit or different types of Error Correcting Codes (ECC). The errors caused by upsets in logic are much more difficult to deal with and mitigation techniques can be employed on a limited scale (for example: Triple Modular

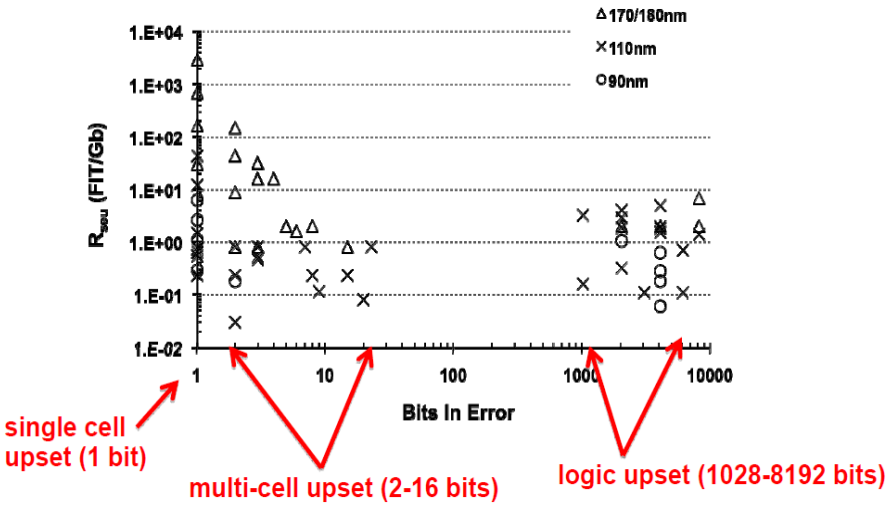


Figure 2.10: DRAM cell upset distribution [53].

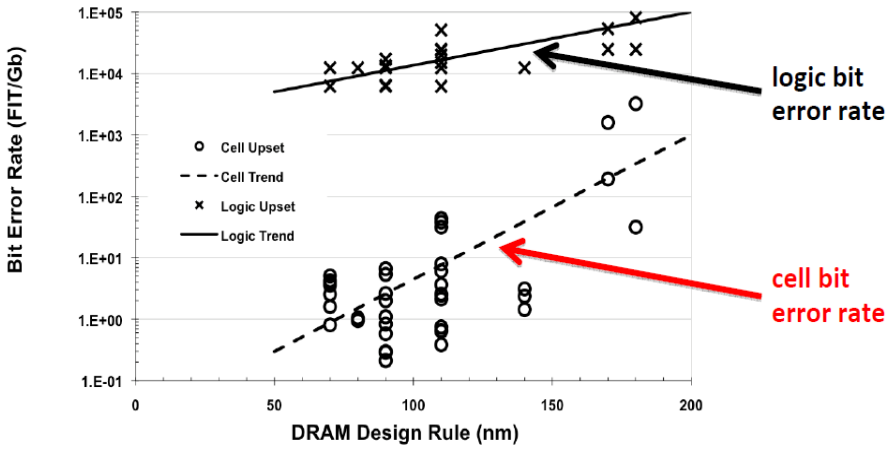


Figure 2.11: DRAM Bit Error Rate (BER) [53].

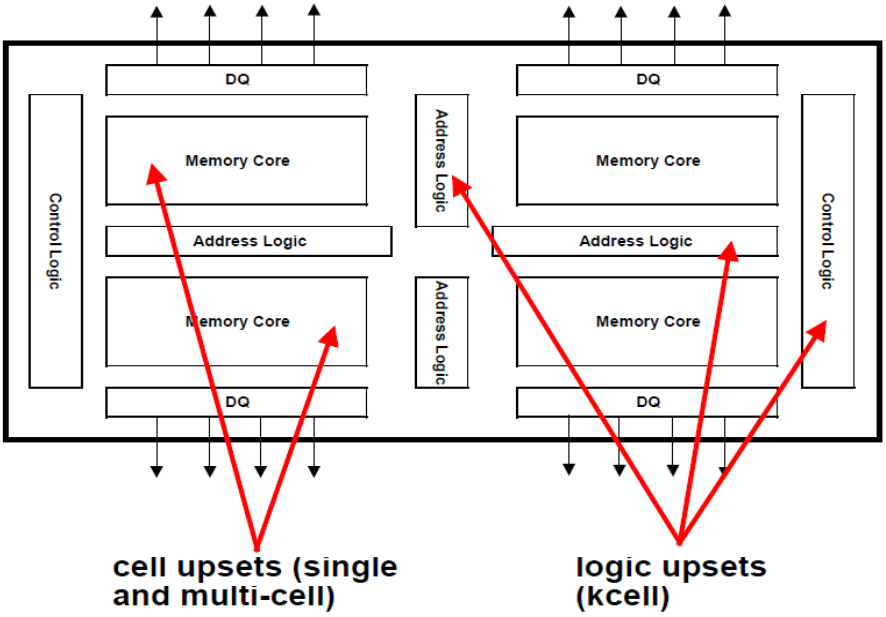


Figure 2.12: Neutron strikes on DRAM logic [30].

Redundancy - TMR).

Memory cell interleaving is efficient when multi-cell upsets can occur in a word line, therefore generating multi-bit errors. By increasing the bit interleave physical distance, it separates the cells in the same word line (Figure 2.13). Thus, multi-cell upsets will produce multiple single bit errors and can be handled separately by a Single Error Correction - Double Error Detection (SEC-DED) ECC which have low overhead and can correct 1 or detect 2 errors. But, if no interleaving is used between the cells of a word line, multi-bit error correction codes are necessary which require greater overhead and circuitry and have lower speed.

2.3.3 Self-healing memory systems through error detection and correction schemes

Parity bit is the simplest form of error detecting code and it needs only one bit to express if a word has an even or odd parity (by XOR-ing the ones or zeros in that word) and it's appended to the word. Is the simplest form of detecting if a word doesn't have errors [34].

Triple Modular Redundancy (TMR) is a concept in which each processing unit/circuit is multiplied and there is a voting element for each one [55], [56].

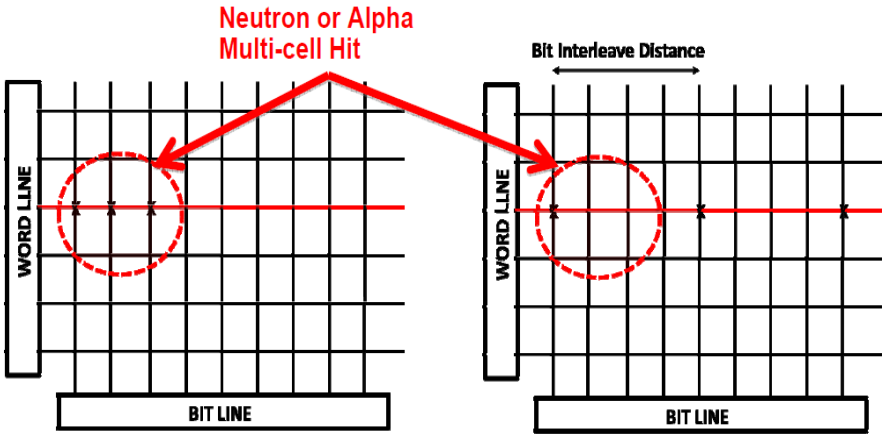


Figure 2.13: Memory cell interleaving [53].

The output is displayed after the comparison of the voting elements. It's based on the idea that errors cannot occur in all the processing units, but it has the disadvantage if the final voting element is corrupted and this design also implies a lot of redundancy and area overhead.

Soft errors are not permanent, they are transient and can be corrected by a simple re-write, re-compute or circuit reset operation. On the other hand, hard errors may also occur in chips which are the result of some permanent (or possibly temporary) physical change of the characteristics of a device.

Considering the above, the most efficient techniques to mitigate such errors include deploying error detection and correction codes (EDC, ECC) and mitigation techniques using spare elements (i.e. rows, columns, circuits, etc.) for replacing/replication of memory elements. These mitigation techniques are addressed in detail in this section.

Error detection and correction codes (EDC & ECC) have been used in memories in order to deal with error of low multiplicity. These codes are based on computing check bits for the original information data and appending them to the data bits. There are several types of EDC and ECC for memories, each having his own advantages and disadvantages. All of the EDC or ECC for memories are *forward error correction* which means a word in a memory contains the information (data) bits and the additional check bits and if errors occur, the original information can be reconstructed using the check bits.

An ECC widely used in memories are the Hamming codes [34], [35]. These codes are linear error-correcting codes that can detect up to 2 simultaneous bit

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	X		X		X		X		X		X		X		X		X		X		
	p2		X	X			X	X			X	X			X	X			X	X		...
	p4				X	X	X	X					X	X	X	X					X	
	p8								X	X	X	X	X	X	X	X						
	p16																X	X	X	X	X	

Figure 2.14: General algorithm for Hamming codes

errors and correct single-bit errors. However, there are several implementations and proposals that improve the correcting and detecting capability, by using more check bits and circuitry (increased code and area overhead).

The Hamming codes are based on a *parity-check matrix* constructed by listing all the columns of length m (number of parity or check bits) that are pair wise independent. To better understand the general algorithm of constructing the *parity-check matrix*, the table in Figure 2.14 displays how the general algorithm generates a single-error correcting code for any number of bits.

In the figure above, by using 5 parity bits (green boxes on the left), an information of 15 bits is encoded. Parity bit 1 (p_1) covers all bit positions which have the least significant bit set to 1 (i.e. 3, 5, 7, etc.), parity bit 2 (p_2) covers all bit positions which have the second least significant bit set to 1 (i.e. 3, 6, 7, 10, 11, etc) and so on until the last parity bit. The idea of the Hamming code is that any given bit is included in a unique set of parity bits. To detect for errors, the parity bits are checked. The patterns of errors are called *error syndromes* which identify the bit in error. The sum of the positions of the erroneous parity bits identifies the erroneous bit. For example, if the parity bits in positions 1, 2 and 8 indicate an error, then the bit 1+2+8=11 is in error. If only one parity bit indicates an error, the parity bit itself is in error.

A simple popular Hamming code in memory systems is the (7,4), also known as SEC-DED (Single Error Correction – Double Error Detection). (7,4) indicates that 4 data bits are encoded into 7 bits by adding 3 parity bits.

Hamming codes can be computed in linear algebra computation, using matrices. For the purposes of Hamming codes, there are 2 matrices that can be defined: *generator matrix* G and *parity-check matrix* H. The *generator matrix* G has 7 rows (the total number of bits) and 4 columns (the number of data bits). The *parity-check matrix* H has a dimension of 3 rows by 7 columns, which corresponds to the 3 parity bits and its 7 total encoded bits of a Hamming (7,4) code. Practically, the rows

of the H matrix are the rows of parity bits p_1, p_2 and p_4 in Figure 2.14. These rows are used to compute the *syndrome vector*, which is null when the word is error-free or non-zero otherwise (the value will also indicate the flipped bit position).

The encoding of the original data bits is done by multiplying the *generator matrix* with the value of the data bits (organized as a 1 column, 4 rows matrix and starting with the first bit value). This will result in a matrix of 7 rows and 1 column. Then the received encoded word is multiplied with the *parity-check matrix* and if the result matrix is null it means the word is error-free. Otherwise, the result matrix will indicate the bit in error (the position of that bit in the encoded word). Therefore, the erroneous bit it's flipped. Once the received word is error free, the decoding process involves multiplying the received error-free word with an R matrix which is similar to the inverted G matrix, with the data bits on rows (a value of 1 where the data bit is positioned in the encoded word) and the columns for the parity bits are null. The result of this multiplication is the original word. Note that all the operations are done in modulo 2.

Although Hamming codes are largely used in memory systems, they have limited capabilities for correcting and detecting errors. These capabilities can be extended to more bits at the expense of more parity bits [36]. Also, as described above, the Hamming codes impose lots of additional circuitry and several time-consuming computations (when writing and reading a word), thus increased power consumption, all of which can be critical when multi-bit errors occur in a word and the algorithm must be performed in run-time specifications.

In [37], the authors propose an implementation of the Hamming Code decoding circuit for high-speed semiconductor memory. The design is shown in Figure 2.15. The idea is to balance between serial and parallel stages. By using more parallel stages (i.e. more processing elements for decoding), the decoding time decreases from 20.48 ns to 1.28 ns for 512 bits. Although the number of processing elements for decoding increases, they don't have a significant area cost because the authors use simple XOR and AND gates.

The authors in [38] propose a modification of the Hamming Codes by minimizing the column weight, row weight and total weight of the H-matrix. In order to do this, they use several constraints when calculating the H-matrix. By reducing the weights of the H-matrix, less XOR2 gates can be used when computing the check bit generator, thus reducing the area overhead, power consumption and even the speed of the code (this is because less XOR2 gates imply fewer levels of gates). The savings in terms of area overhead and power consumption are of 14.4 - 17.7% and in terms of access time, about 12.5 – 16.6%, all compared for different sizes with several original Hamming Code implementations.

BCH (Bose-Chaudhuri-Hocquenghem) are a class of parameterized error correcting codes [34]. They can achieve multi-bit error correction by using the sim-

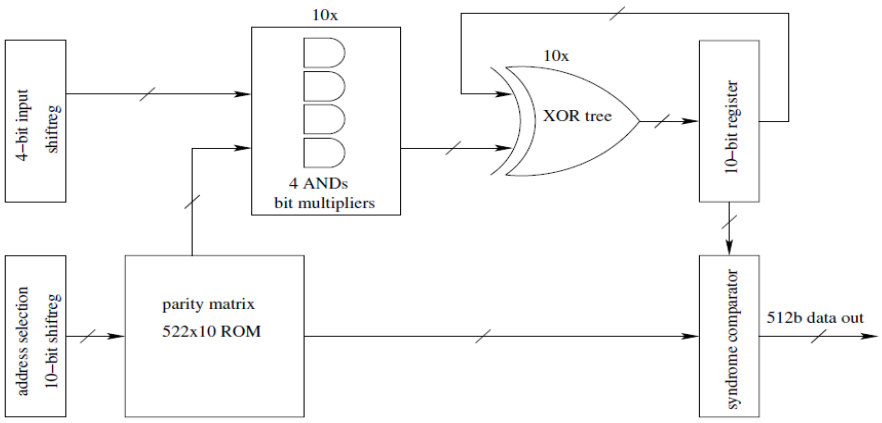


Figure 2.15: Hamming-decoder circuit [37].

plicity of electronic hardware to perform the syndrome decoding, but at the expense of computation time and area overhead. A BCH code is a polynomial code over a finite field with a particularly chosen generator polynomial. While the encoding of words can be done pretty easy, the decoding part is more complex and time-consuming. Therefore, there are many algorithms for decoding a BCH code. The most common ones follow a guideline of 4 operations: calculate the syndrome values for the received vector, calculate the error polynomials, calculate the roots of this polynomial to get the error locations positions and finally calculate the error values at these error locations. In [57], the authors propose a new method to implement an ultra-fast-efficient BCH decoder. It is based on the Berlekamp-Massey algorithm [34], but reformulated and inversion less, which can reduce the hardware complexity. Also, when determining the roots of the polynomials, the Chien Search algorithm [57] was extended and modified to compute in parallel reducing the computation complexity and area by 33% compared to the regular method.

In [39], the authors propose a BCH code which can correct 5 errors and detect 6 errors (called 5EC-6ED). The authors use this ECC architecture to correct errors in a memory which has more than 2 errors on a word line. This paper will be addressed in detail further in this section because it combines ECC with other error mitigation techniques and offers a low-power refresh rate.

Even more powerful ECC are the Reed-Solomon (RS) codes [34]. They use *symbols*, each one consists of several bits. The code is efficient for large amounts of data and can detect and correct random multiple-bit burst errors. For t check symbols, the RS code can detect up to t erroneous symbols and correct up to $t/2$ symbols. Reed-Solomon codes are a ”simplified” version of BCH codes, so the

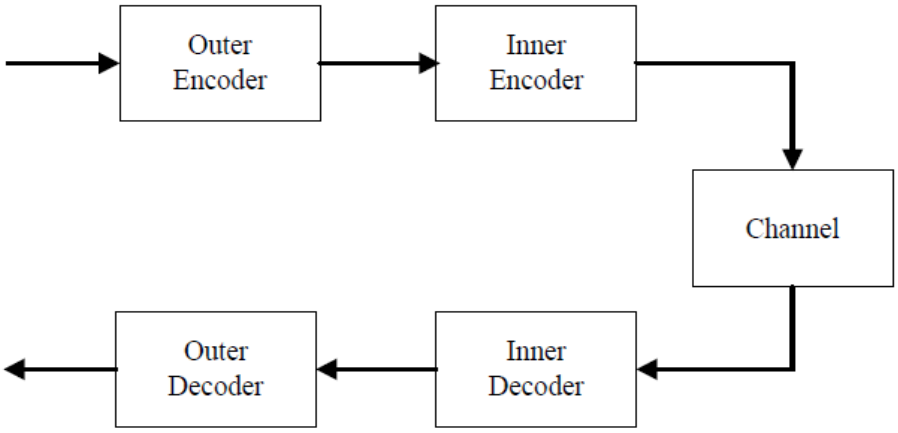


Figure 2.16: Block diagram of concatenated Hamming and R-S codes [43].

algorithms for encoding and decoding are mainly the same. This means it inherits the complex decoding computations, including the error detection and correction algorithms (there are several implementations, each with advantages and disadvantages), the large area overhead and low speed when it is needed.

Other proposals have been made in order to enhance the speed and to minimize area and power consumption. The authors in [41] compare 2 types of reconfigurable hardware architectures. They conclude that the bottleneck of Reed-Solomon codes is the syndrome calculation when decoding because the power consumption and computation times are high. The Chien Search Algorithm for finding the roots of the polynomials is overlooked because this block is less frequent in operation (only when errors occur).

In [43], the authors propose an interesting design, a concatenation between Reed-Solomon and Hamming codes for the DRAM controller. The block diagram of the concatenated code is displayed is Figure 2.16. The outer code is Reed-Solomon, while the inner code is a combination between two shortened RS and Hamming codes. This concatenation can prove to be successful because the shortened RS codes with an outer RS encoder improve the error correction capability for burst errors, whereas a Hamming code covers dispersed random errors. The results demonstrate the ability of concatenated ECCs, thus achieving low power dissipation, low area cost and a symbol error rate lower than the normal implementation of codes.

Error Detecting Codes (EDC) are mainly used just for detection of errors. While the simplest form is the *parity bit*, a more complex and comprehensive code is the Berger code [44]. Mostly used in telecommunications, this code is a uni-

directional EDC. Unidirectional errors are errors that only flip ones into zeros or zeros into ones, but not in both directions. The Berger code check bits are computed as the sum of all the ones (called B_1 encoding scheme) or zeros (called B_0 encoding scheme) in the information word. The sum is expressed in binary and appended to the information bits. If the information word consists of n bits, the Berger code check bits needs $k = \lceil \log_2 (n + 1) \rceil$ check bits, which gives a Berger code length of $k + n$. Berger codes can detect any number of one-to-zero bit-flip errors, as long as no zero-to-one errors occur in the same word. The detection process consists of comparing the check bits from the received word with the newly computed check bits from the information word. If the check bit values are equal, the word is error-free. If the values are not equal, depending on the implementation (B_1 or B_0 encoding scheme), the result of the comparison shows that there is at least one error in the information or check bits.

In [32], the author explains the causes of soft errors in DRAM cells when the sources are high-energy neutrons or an alpha-particle strikes and what causes a burst of charge in the cell area. If the charge collected is greater than a critical charge Q_{crit} required to preserve the data in the capacitor of the cell, the bit will be upset and lead to an error. This means that if the capacitor contains a charge, after a high-energy neutron or an alpha-particle strike, the capacitor will discharge, thus changing the value stored. Depending on the DRAM implementation, the presence or absence of a charge in the capacitor can symbolize a logical value of '1' or '0'. Therefore, if the presence of a charge in the capacitor means that a value of '1' is stored inside the cell, after the hit, this will be '0'. Otherwise, if the presence of a charge in the capacitor means that a value of '0' is stored inside the cell, after the hit the value will change to '1'. But in any case a high-energy neutron or an alpha-particle strike cannot charge the capacitor of a cell and therefore they only provoke unidirectional errors.

Considering the above, Berger codes can be used in DRAMs to detect all unidirectional errors that can occur after radiation hits. In this scope, the authors in [33] propose a tree-shaped parallel counter using partial sums to compute the Berger code check bits for DRAMs unidirectional errors. The tree-shaped parallel counter uses full adders (FA from now on) to count the number of 1s or 0s (depending on which encoding scheme is used). An example of 10:4 bits is in Figure 2.17.

The authors in [33] propose that the computation can stop before the last level of FA, in order to achieve greater computation speed (for the check bits) and less area overhead, at the expense of more Berger code check bits. For example, in Figure 2.17 they consider 10 information bits and by stopping at the first level ($\mathbf{C}_4$), there are 7 check bits, with a delay of 1 FA stage and only 3 adders are used. If the computation stops at the second level of FA ($\mathbf{C}_2$), there are 5 check bits generated with a delay of 2 FA stages and 5 adders are used. The last level ($\mathbf{C}_1$) generates the

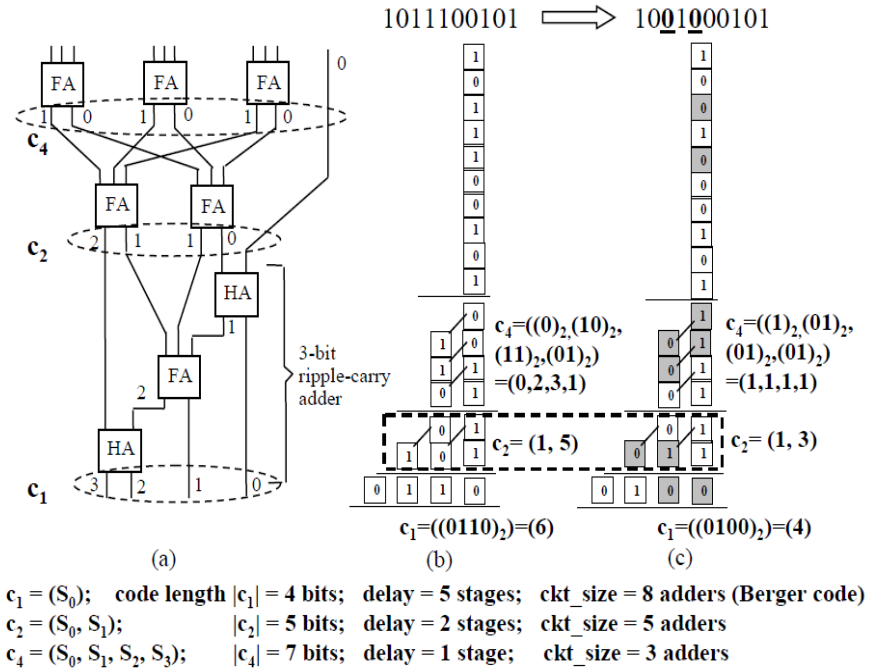


Figure 2.17: (a) 10:4 parallel counter tree and check bits generation for (b) an uncorrupted word and codes and (c) a corrupted data word (2 bits go from 1 to 0) and codes [33].

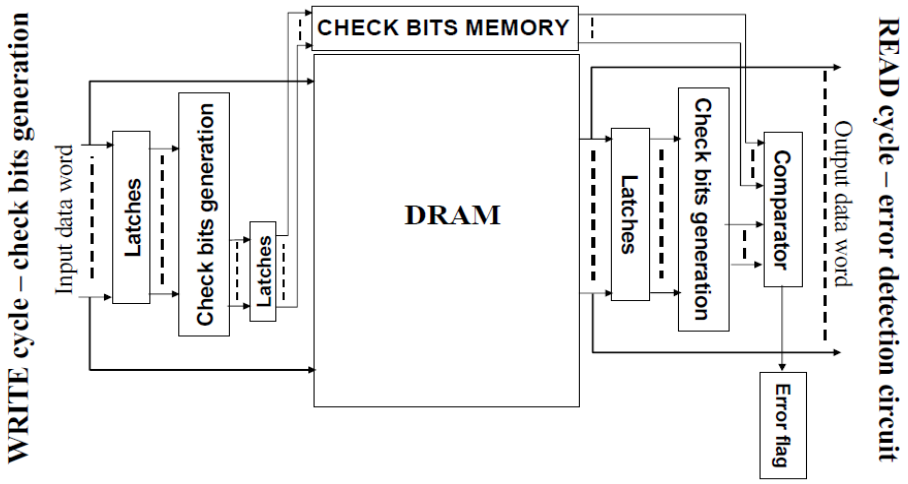


Figure 2.18: Proposed architecture in [33].

Berger code check bits (4 bits in this example) at the expense of 5 FA stages and 8 total adders. If the original information bits increase, there are more than 3 levels of FA that are generated using the same logic of parallel counter tree. However, there is a certain stage where the parallel counter tree has to use a ripple-carry adder design in order to generate the correct check bits. This is because a full adder has 3 inputs (one of which is a carry-in) and has 2 outputs: a carry-out which has a weight of 2^1 and a sum with the weight of 2^0. The main idea of this proposal is to find a balance between the number of check bits that are generated and the speed of generating those check bits (calculated in FA stages), so code redundancy (i.e. check bits divided by the total amount of bits) versus number of FA stages. The proposed architecture in [33] is displayed in Figure 2.18. Note that during the write cycle, the check bits are generated, appended to the original information and stored in the memory system. When reading, the check bits are computed again from the information bits and compared with the stored value. Any discrepancy between the 2 values means that at least one error occurred in the information bits. Because the check bits are stored together with the information bits, the authors suggest using enhanced memory cells (less leaky and radiation hardened) to reduce the number of false errors (i.e. errors in the check bits).

More complex codes similar to the Berger code are *t-symmetric error correcting / all unidirectional error detecting* (t-SyEC/AUED) codes [35], [58], [59], [60], [61]. The design consists of binary blocks which are capable of correcting up to t symmetric errors and detect all unidirectional errors. Note that if $t = 0$, these codes

become the Berger codes. However, these types of codes are suited for asymmetric channels, such as digital transmissions.

EDCs are a faster and economical alternative to ECC, without the correction part. So, the errors are detected and other mitigation techniques are used in order to eliminate the errors. This is a solution when considering the energy consumption, performance degradation and time. The energy consumption is greatly reduced because the detection process is fast and simple, rather than the complex decoding circuit of the BCH or Reed-Solomon codes. There is no performance degradation in the system during the error detection phase, but when an ECC reaches the correcting stage, the overall system performance is reduced due to the complex computations. Also, any ECC is time consuming especially when the errors are random, unrelated and affect more than one bit.

Besides ECC and EDC, there are other mitigation techniques for soft errors in memory systems, which use spare elements that replace or replicate, based on specific algorithms, various memory elements. Because soft errors are transient (temporary) and not permanent, the need of replacing a specific memory element is not that pressing or important. However, when considering the current trends for Soft Error Rate (SER), the fact that neutron or alpha-particle strikes can provoke multi-bit errors in the same word line and also affect the circuitry of a memory, these techniques of replacing and replicating are becoming more and more useful.

The authors in [39] propose a multi-bit error correction system (Hi-ECC) for embedded DRAM (eDRAM) which incorporates a strong BCH code with the ability to correct 5 and detect 6 errors (5EC6ED). Because most multi-bit ECCs have a low-latency decoder that has a large area overhead, the authors propose a simple ECC decoder optimized for 99.5% of the lines that require little or no correction. For more complex multi-bit correction, they propose using the BCH code 5EC6ED. But because for correcting multi-bit errors creates high latency processing, the Hi-ECC architecture disabled the lines with multi-bit errors (while the correction is being done). The block diagram of the proposed Hi-ECC is displayed in Figure 2.19.

The ”Quick ECC” block contains the syndrome generation logic and the error correction logic for lines with zero or one failure (SEC-DED code). If a line has multi-bit errors, it is transferred to the ”High latency ECC processing” block which contains the BCH error correcting code 5EC6ED. Also, in order to further reduce the power overhead, the author propose the use of a structure that keeps track of lines that have been referenced in the last 30us (retention time baseline for the considered eDRAM) called Recently Accessed Lines Table (RALT). They use this structure because the recently referenced lines implies a refresh, so retention errors will not occur for 30us after a line has been read.

To avoid incurring the high latency of the BCH ECC when multiple-bit errors

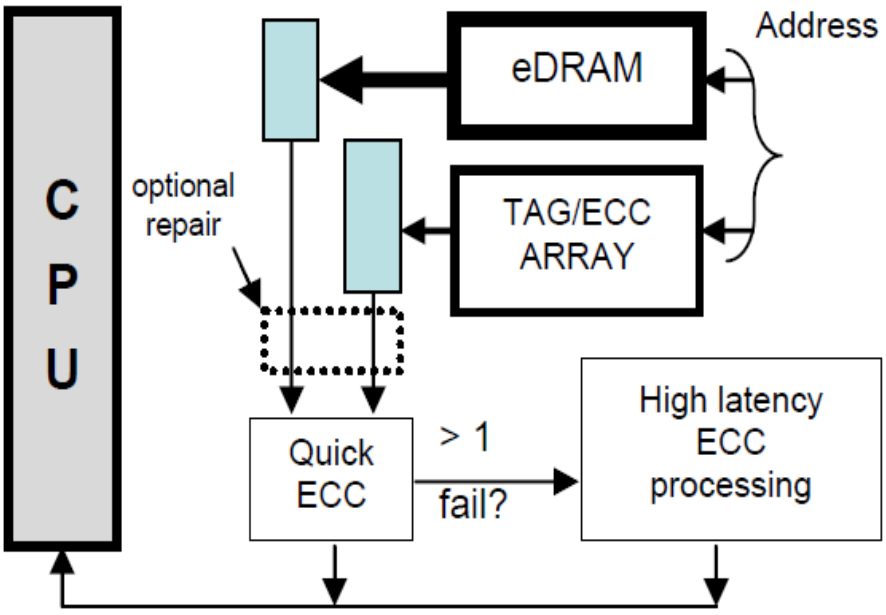


Figure 2.19: Block diagram of the proposed Hi-ECC architecture [39].

occurred in a line, the dotted block *optional repair* could be any type circuit that disables those cache lines until they are corrected. But if they are multiple lines that have errors, the cache capacity is reduced a lot. Therefore, they propose the use of a complex mechanism called ”bit-fix” [40] which is able to minimize the capacity lost to disabling. However, the result show that the Hi-ECC architecture reduces the cache refresh power by 93% compared to an eDRAM with no error correction, and by 66% compared to a eDRAM with single error-correcting code SEC-DED, with only 2% storage overhead and without the loss of any cache state.

The ”bit-fix” scheme [40] mentioned above is displayed in Figure 2.20. The idea is to use multiplexers to remove the defective pair of bits from a vector of bits and apply a patch by shifting-in, based on the repair pointer. The authors of [40] also use this scheme to disable cache words or lines (i.e. removing defective word or cache line). The defective part can be corrected by an ECC or it can be re-initialized, and then used again. This mechanism/scheme can prove to be effective for large amounts of information bits when multiple-bit errors occur in the same word and there is the possibility to localize the part or parts of the information bits that are defective (i.e. localize the errors).

Embedded DRAM follows the same principle of low power in conjunction with error detection capabilities. In [62], the authors propose an on-line consis-

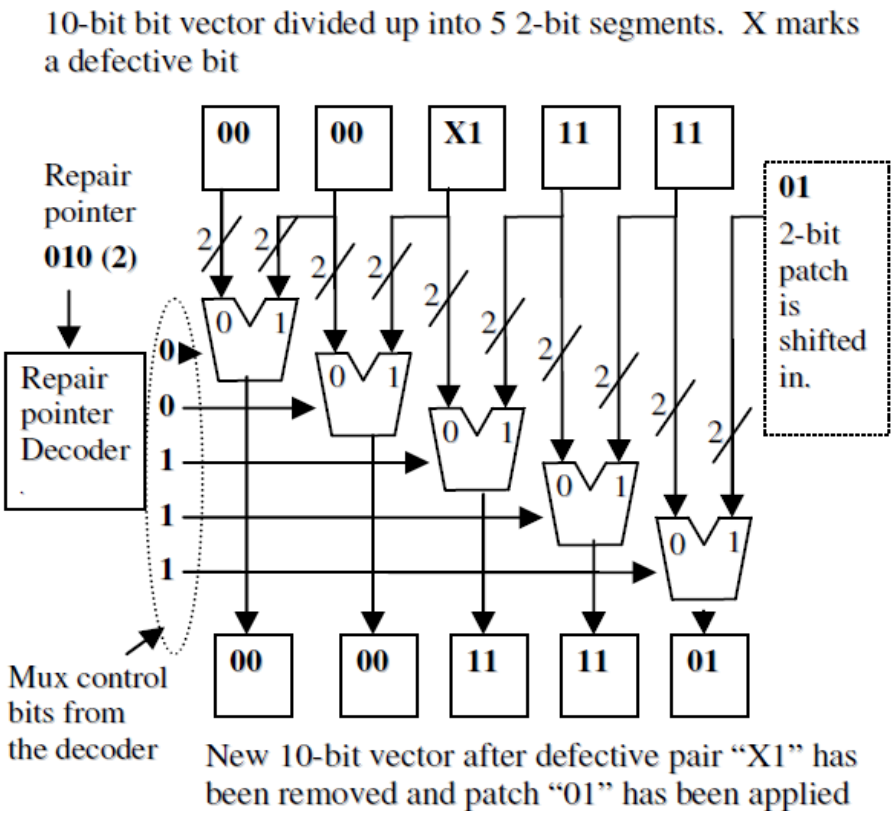


Figure 2.20: Design of "bit-fix" scheme for 10 bits [40].

tency checking based on the modulo-2 address characteristic, which can ensure low error detection latencies and reduced power consumption. With the use of extra logic, the proposed design checks memory locations for errors during a refresh cycle. A similar idea of using ECC to reduce refresh rates for DRAMs is explored in [63]. This has proven to be effective because the read operation in DRAM is destructive and all reads are immediately followed by restore operations. Hence, the information is refreshed more often if an ECC runs as a "background" operation (similar to a data scrubbing method [32], [64], [65], which means less power consumed when the normal refresh cycle performs. However, more hardware is needed to provide information about the latest refreshed memory locations (use of *timestamps* [66]). A trade-off between hardware overhead and power consumption is needed, as explored in [39] and [40].

In [67], the authors propose a simple method to improve the reliability of a SRAM. The method proposed prevents a SRAM from executing successive multiple read operations on the same position. Figure 2.21 depicts the proposed reliability enhanced scheme. The "successive read detector" circuit will detect if multiple successive reads are made to the same memory location by comparing the memory addresses. If this is the case, the SRAM will remain in idle mode (because CEN' is 1) and the data at the output of the SRAM is the same as the first read. Nowadays, embedded SRAM memories tend to have a Built-In-Self-Repair (BISR) circuit which consists of: a Built-In-Self-Test (BIST) component, a Built-In Redundancy-Analysis (BIRA) component and reconfiguration component. The BIST detects the targeted functional faults and BIRA allocates the redundancy (i.e. spare component) according to the fault pattern. Then the defective element is replaced by the corresponding reconfiguration spare element. In [67], a new BISR scheme is proposed, which includes, besides BIST and the reconfiguration mechanism, the reliability-enhancement circuit and a test collar. The test collar is used to switch the RAM between BISR mode and normal mode (done by multiplexers). The idea is to whether the faults are repaired or not. The proposal increases the repair rate with 6% to 10% compared with a SRAM with normal BISR, at the expense of 2% area cost for two spare rows and two spare columns.

Column replacement techniques have been widely used in memories to replace a defective column by a spare one. This does not actually remove or correct the error, it masks it. In [68], the authors propose the use of a column replacement mechanism with a special case of Single-Error Correcting (SEC) codes, called restricted SEC (RSEC). They are characterized by programmable parity-check matrices which allow the correction of different sets of errors. In order to generate these matrices, the authors use the built-in test results which will indicate the columns with defective storage cells. They propose the use of the redundant spare columns either to store RSEC check bits or to replace completely the defective columns.

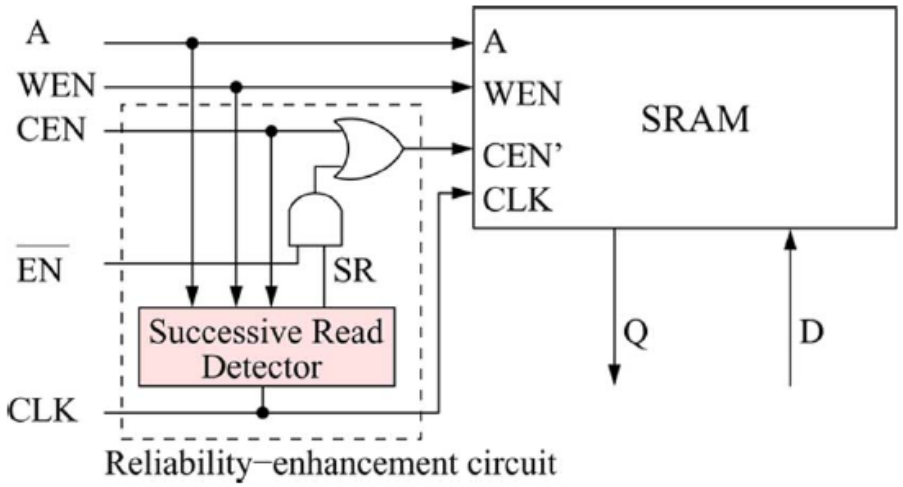


Figure 2.21: Reliability-enhancement circuit proposed in [67].

Figure 2.22 displays the combination of RSEC and column replacement in case of a segmented memory (for a memory read access).

Because the matrices of restricted SEC are constructed after the built-in test results, RSEC has limited capability and offers reduced protection against soft errors. Otherwise, for hard errors, the proposed scheme can improve the memory repair capability with limited performance overhead.

Because only one state of a DRAM cell is sensitive to radiations (unidirectional errors) the authors in [51] propose a new mitigation technique used for detecting errors. The idea is based on finding the insensitive state of a DRAM cell and then storing the opposite value (which corresponds to the charged state) as the reference value in a cell. Consider a memory system implemented with the same type of DRAM cells and the charged state corresponding to a logic value of '1'. If radiation upsets a cell, this will change the value to '0', therefore the insensitive state is the discharged state (which corresponds to a logical value of '0'). If radiation hits the cell when it's discharged, the value in the capacitor will not change (i.e. unidirectional error). So, by storing the insensitive state value in a cell, this can be used as a reference value when errors occur in 2 cells (Figure 2.23). However, by using this proposed principle, the additional resources can reach 49.9% because every useful data bit must be stored twice, and the insensitive state must be known before using this principle.

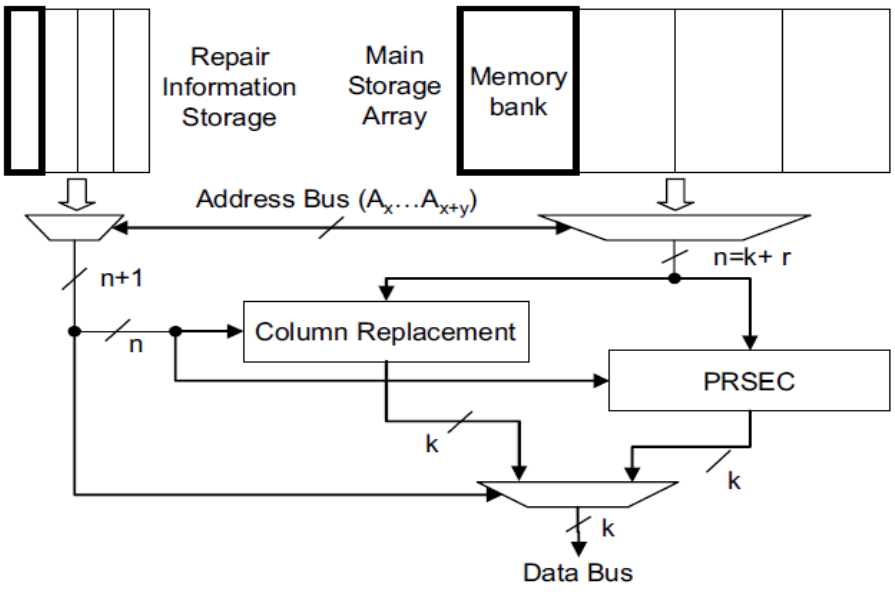


Figure 2.22: Combination of restricted single-error correction codes and column replacement proposed in [68].

2.4 Proposed solution

2.4.1 Modified Berger codes

The proposed error detection, localization and correction scheme is based on a modification of the original Berger code implementation. The information bits of each memory word will be divided into segments of a specific size, while the check bits will be generated by layers or levels of Full Adders. The highest level will generate the Berger code check bits. A segment represents a group of information bits, which have a specific size, established at design time. The information bits will therefore consist of several segments of equal length. By doing this splitting, when one or more errors occur, the segment(s) can be identified as erroneous. Moreover, if a specific error pattern applies, the segment can even be corrected.

Stopping before the Berger code check bits is advantageous when considering higher speeds and low latencies for fast DRAMs and, also, provides information regarding the erroneous segments. Hence, error localization and correction is possible.

In order to compute the check bits for the Berger code, a full adder tree-shaped parallel one's counter is used. The idea is to choose a carry-save strategy which do

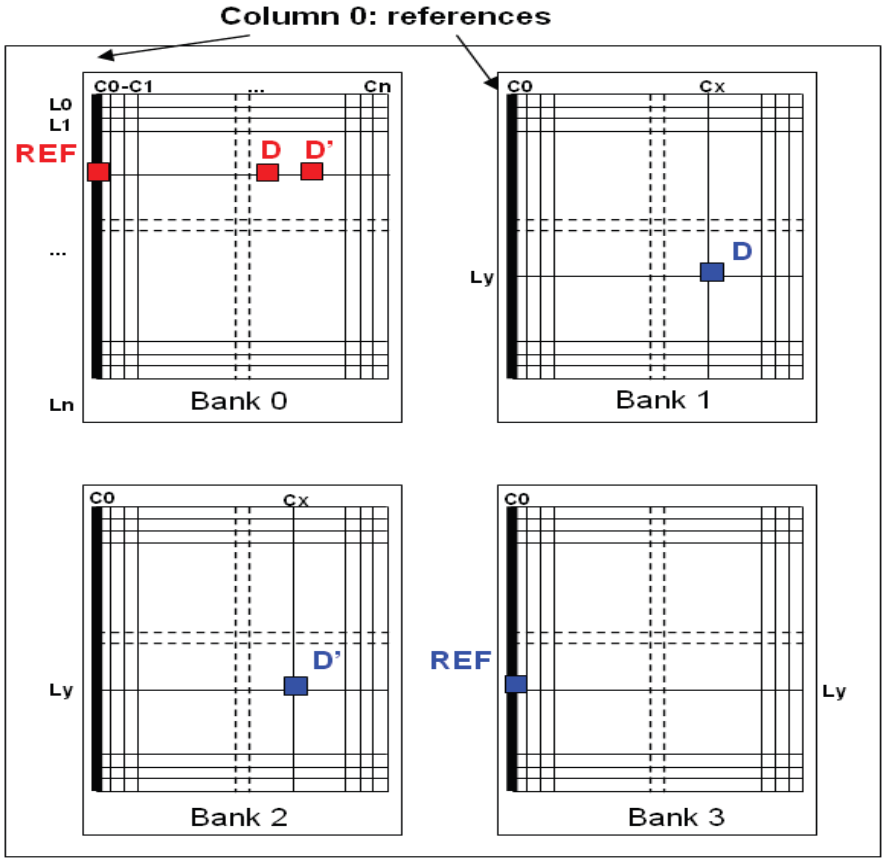


Figure 2.23: Error recovering principle proposed in [51]. One cell is used as reference (REF) to store the discharged state logic value (immune to radiations), useful data are stored in 2 other cells, D and D'

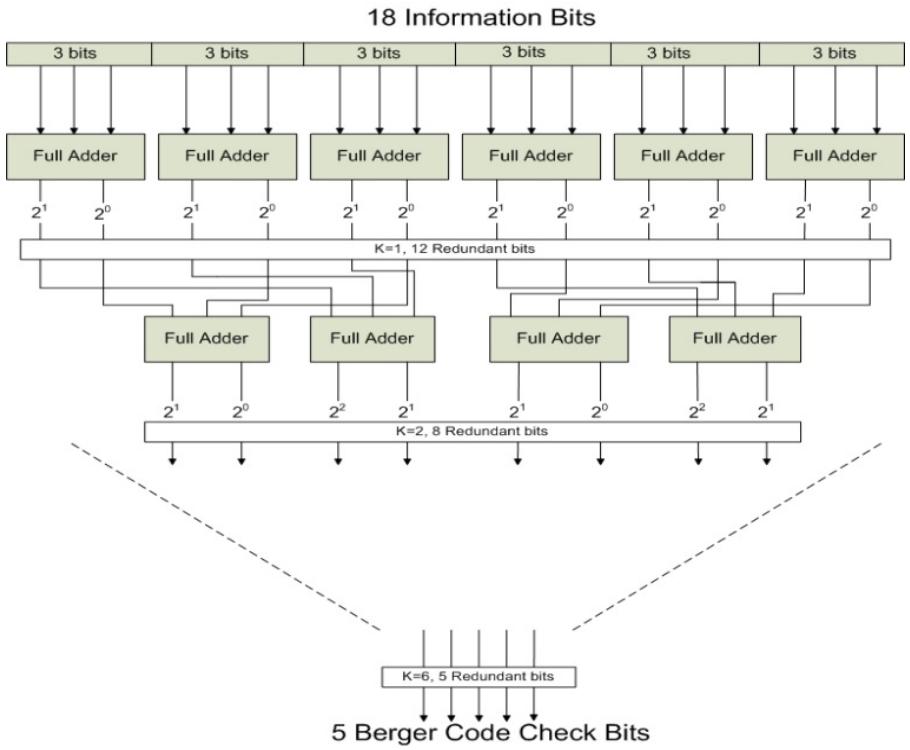


Figure 2.24: Generation of the check bits for CS-1 Berger, CS-2 Berger and plain Berger code $K = 6$. The number of information bits is $N = 18$. The plain Berger code requires six levels of FA.

not propagate the carry bit to the output. Therefore, at each stage or level of the tree-shaped parallel counter, the output will be a sequence of partial sum bits and another sequence of carry bits. We will name this scheme as Carry Save K Berger (CS-K Berger, from now on), where K denotes the level of FAs which is in use.

As explained in the previous chapter, if a radiation strike occurs in a DRAM cell, a burst of charge near the capacitor will discharge the capacitor, thus changing its initial state and value. However, if the capacitor is initially discharged, the state and value will remain the same, state which is also known as the insensitive state.

The tree-shaped parallel counter generates at each level partial check bits of the Berger code, in the Sum and Carry outputs from the FAs. Therefore, at each level of FAs, a unique code can be formed. Each of these codes has its own advantages and disadvantages: the lower the level, the faster it gets computed, but at the expense of more check bits; while at higher levels, check bits are computed more slowly than

previous levels, but with the benefit of having less check bits. In the end, the last level of FAs generate the Berger code check bits. Note that the computing tree of the Berger code needs a ripple-carry adder design because the overall partial check bits have to be added into a single binary outcome - the Berger code check bits. Because of this, the design for this tree-shaped parallel counter grows quickly as the word size increases as it is illustrated in the following figures for small sizes of the information bits. Figure 2.24 depicts the tree-shaped design for 18 information bits and exposes all the FA levels required by the Berger code check bits.

Consider N to be the number of information bits. The first level has $\lceil \frac{N}{3} \rceil$ FAs and $\lceil N \cdot (\frac{2}{3}) \rceil$ outputs; the second level has $\lceil \lceil N \cdot (\frac{2}{3}) \rceil \cdot \frac{1}{3} \rceil$ FAs and $\lceil \lceil N \cdot (\frac{2}{3}) \rceil \cdot (\frac{2}{3}) \rceil$ outputs, and so on. These expressions continue to expand until the ripple-carry adder design is reached. Note that for the first level, only 3 information bits are necessary to generate 2 check bits, while for the second level, at least 9 bits are necessary to generate 4 check bits. The minimum information bits size for a specific level will be called *segment* from now on and this value is multiplied when considering higher information bits sizes. The total number of levels of the tree-shaped parallel counter grows as $\mathcal{O}(\log_2(N))$ and the total number of FAs is $\mathcal{O}(N \log_2(N))$.

Figure 2.25 displays the same tree-shaped parallel counter, but for 9 bits of information. Note that the first level of FAs has 6 check bits, the second level has 4 check bits and the rest of levels implement the ripple-carry adder which produce the Berger code of 4 check bits defined by the expression: $k = \lceil \log_2(N) \rceil$, where k is the number of check bits. Each level of FAs adds an extra delay as can be seen in Figure 2.24 in which the computation of the Berger code check bits from 18 information bits takes 6 FA delays, while for 128 bits of information it would take about 12 adder delays which would increase the computation time significantly. The total delay can be reduced by stopping at the first levels of the FA tree at the expense of having more check bits, despite this does not compromise the detectability of errors. Actually, it provides localization and even correction performances given by the fact that there are more partial check bits than the Berger code and that each partial check bits are "responsible" of a sector of the information bits.

2.4.2 Coding schemes

The Berger code check bits are computed either by summing all the 1's (called B_1 coding scheme) in the information bits, either by summing all the 0's (called B_0 coding scheme). The B_1 scheme offers no protection if false errors appear in the coded word (i.e. errors in both information and check bits) because, in some cases, the error could escape, but, when considering the first levels of the FA tree,

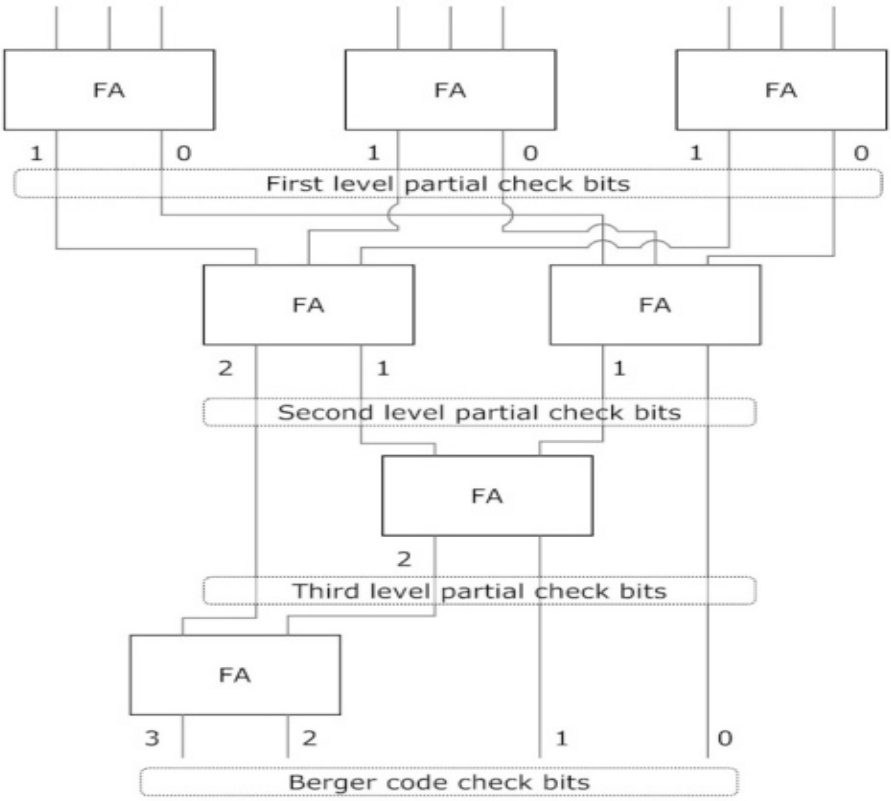


Figure 2.25: Tree-shaped design for 9 information bits and the corresponding FA levels.

Table 2.1: Example of B_0 coding scheme for 1 bit in error.

Original word	B_0 **Error Words**	**Check bits comparison**
110 01	1**0**0 01	$10 > 01$
110 01	**0**10 01	$10 > 01$
110 01	110 0**0**	$01 > 00$

the B_1 scheme can be used to localize and correct errors. On the other hand, the B_0 scheme does not have false errors, and cannot be used to localize, nor correct errors.

Each coding scheme has its own advantages and disadvantages which are explained below. Consider that the original value of the information bits is 110. Considering the B_0 coding scheme, the check bits of the 1st level of the FA tree have a value of 01. Table 2.1 displays the possible error words: if the error occurred in the information bits, the possible error words are 100 01 or 010 01; if the error occurred in the check bits, the possible error word would be 110 00. For the first two error words in the table below, comparing the check bits would be $10 > 01$ (2 zero bits in 100 versus the original value of 01 from the coded word). For the third error word, the comparison would be $01 > 00$. In all three cases, the newly computed value of the check bits is higher than the original stored value, no matter if the error occurred in the information or check bits. This gives the code inconsistency when trying to recover to the previous value. This is the main characteristic of the B_0 coding scheme: the newly computed check bits will have always a higher value than the original value of the check bits from the coded word. This means that any number of errors will be detected, no matter where they appear and no false errors can occur (i.e. errors in both parts will be detected as well). But, by using the B_0 coding scheme, the check bits don't provide any information regarding the positions of the erroneous bits and, as a consequence, nor can be used to correct errors.

Consider the same original value for the information bits 110, but the check bits being generated using the B_1 coding scheme. Table 2.2 displays the possible error words. Note that, in the first two cases, the comparison of the check bits will be $01 < 10$, which means that the error is in the information bits, with 2 possible locations (the zero value bits). For the last error word, the comparison will be $10 > 00$, which means that the error is in the check bits. Furthermore, in the last case, the error can be corrected because only 1 possible location could have been in error.

As stated before, the conventional Berger code can detect unidirectional errors all localized in the information bits or in the check bits. If a word has one or more

Table 2.2: Example of B_1 coding scheme for 1 bit in error.

Original word	B_1 **Error Words**	**Check bits comparison**
110 10	1**0**0 10	$01 < 10$
110 10	**0**10 10	$01 < 10$
110 10	110 **0**0	$10 > 00$

unidirectional errors in the information bits, the newly computed check bits will be less than the original computed value of the Berger code check bits. Or if the error is in the check bits, the value is greater. Errors in both information and check bits are unlikely to happen, but it is a possibility and may cause an error escape (i.e. undetectable error). The probability of error escapes is discussed in 2.4.5. The proposed codes have error detection, localization and in some cases correction capability as will be shown in the following paragraphs.

In order to better illustrate how errors can affect the information and check bits, Figure 2.26 shows the error states graph for the simple case of a single segment of 3 information bits. Each segment generates 2 check bits (using a full adder circuit - FA). On the left of Figure 2.26, the error states for a segment of 3 bits can be observed. If one error occurs, any bit with the value of 1 can change to a zero (illustrated by the arrows). If two errors occur, 2 bits of value 1 will change to 0. This can happen when we have at least two bits with the value of 1. Three errors can occur only when the initial state is 111 and transits to 000. On the right side of the figure, the check bit error states are illustrated, for both B_1 (counting 1s) and B_0 (counting 0s) coding scheme. The B_0 scheme offers no error escapes, a limited localization and no error correction. Note that from now on, only the B_1 coding scheme is used and only transitions from 1 to 0 are considered.

2.4.3 Error localization

The error localization is measured by the ambiguity of the possible error locations. It expresses the total possible error locations consistent with the received code. Given an error in a word, the possible error locations are the bits with values of 0, because they could have been modified from an original value of 1.

In order to generate the error localization, correction and escapes graph all the possible error words are considered. They are created through a recursive C++ combinatorial program which calculates for all possible information words, all possible error locations depending on the number of errors that can occur. The locations that have a value of 1 are set to 0, thus simulating an error. The error words are generated (maximum number of errors is 5) and in each word the algorithms for error localization, correction and escapes are executed to evaluate the

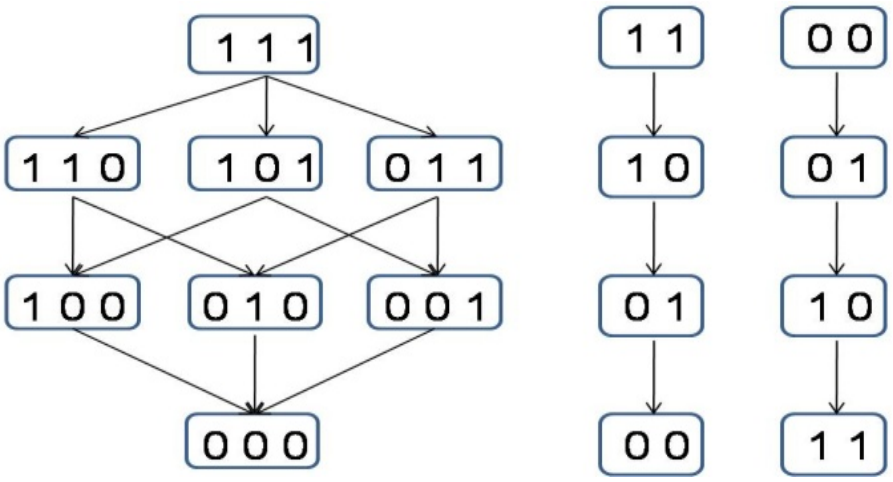


Figure 2.26: Unidirectional error states graph for $K = 1$. On the left, all possible single errors are indicated. On the right, the corresponding check bits for B_1 and B_0 scheme with single error transitions

codes.

Consider the following example for 3 information bits and 2 check bits (the first level of a FA tree). In Figures 2.27, 2.28 and 2.29 all possible information and error words are presented. Look at the 3rd coded word from Figure 2.27: the information bits are 010 and the check bits 01. With the coding scheme B_1 the error can only occur on the bits with value 1. In this case, two possible locations can happen: one in the information bits and one in the check bits. Therefore, there are two possible error words: 00001 or 01000.

If the number of errors increase to two, the same word 01001 can have a single error word: 00000 (Figure 2.28). This case is problematic because there is no information related to the original word. In comparison, the previous case (same original word, but only one error) has two possible error words which have error that can be localized or even corrected.

Increasing the number of errors to three the number of possible error words decreases, as visible in Figure 2.29. Because the locations for errors are the bits with the value of 1, the number of possible error words is actually computed as combinations between the number of bits with a value of 1 and the number of errors. Thus, for the original word of 11111, with three errors occurring there are ten possible error words.

The error localization is measured by the ambiguity of the possible error lo-

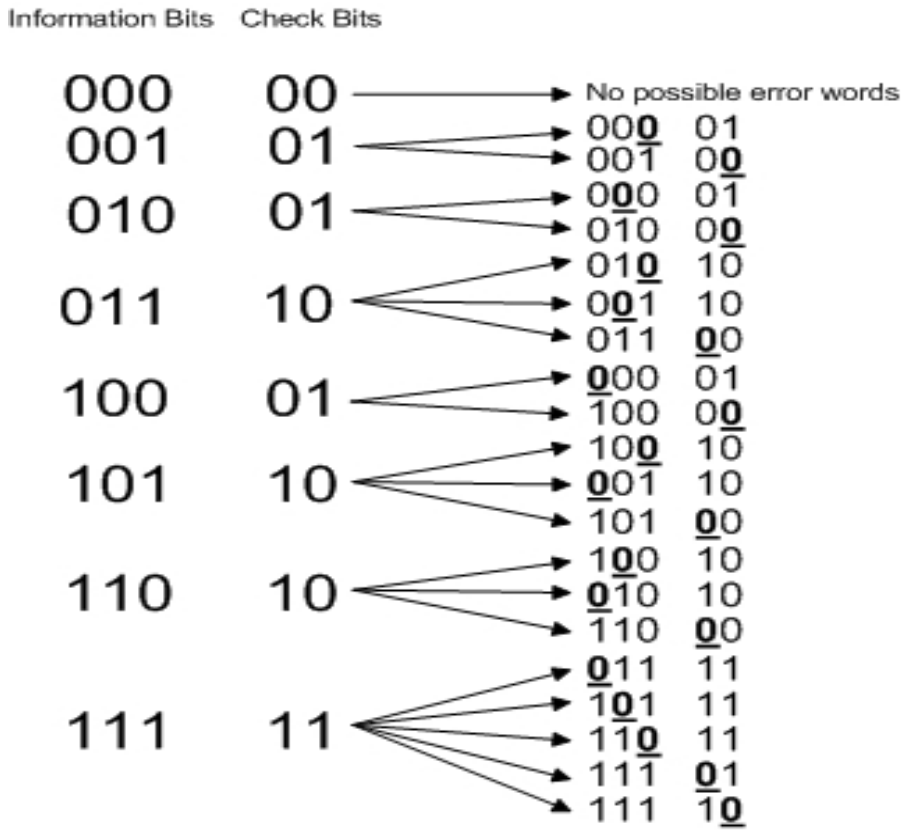


Figure 2.27: Example of possible error words for 3 information bits and 2 check bits, with one error occurring (the first level of FAs is considered $K = 1$).

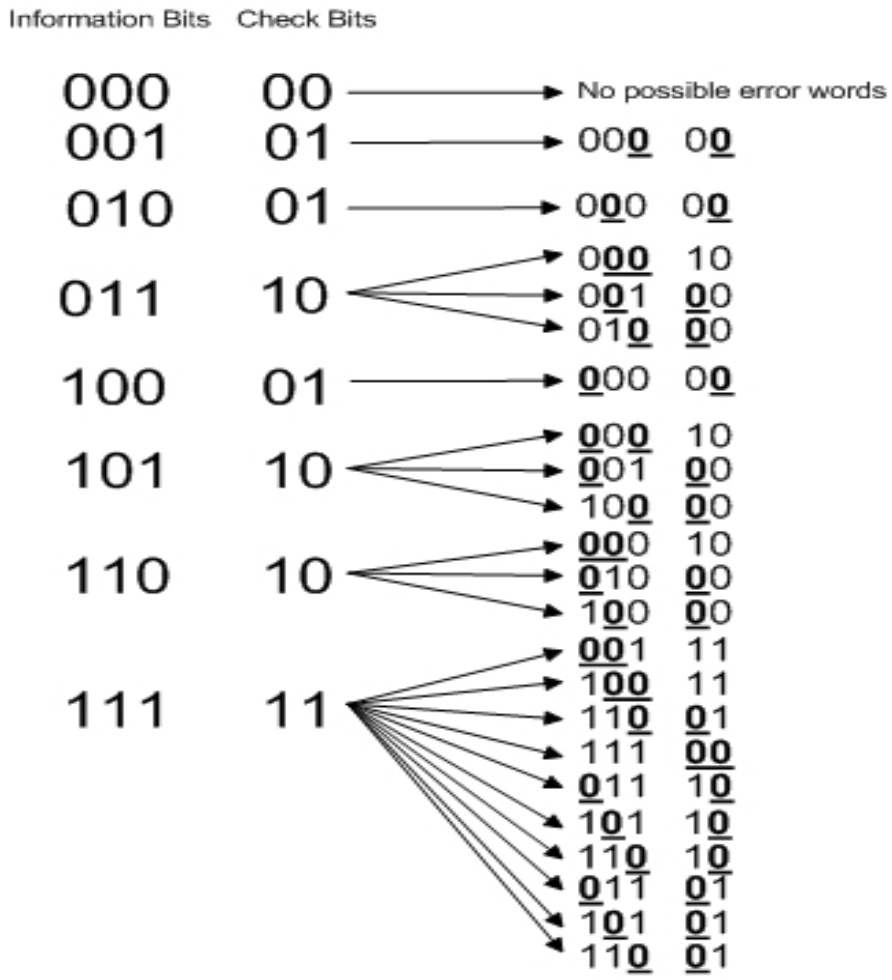


Figure 2.28: Example of possible error words for 3 information bits and 2 check bits, with 2 errors occurring (the first level of FAs is considered $K = 1$).

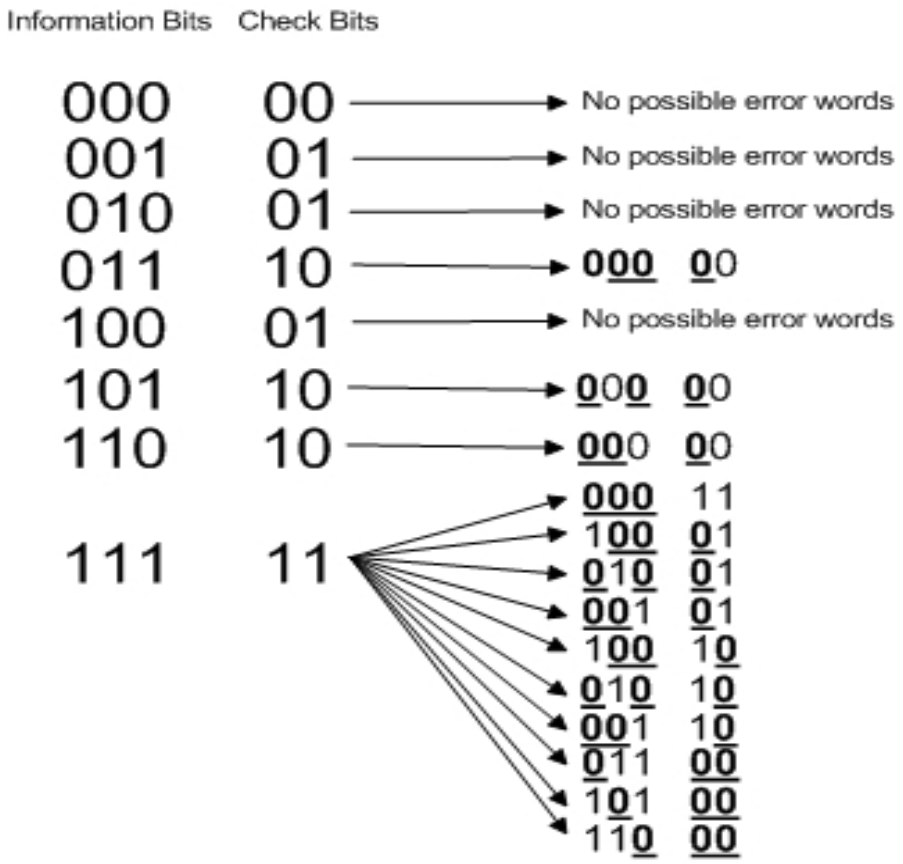


Figure 2.29: Example of possible error words for 3 information bits and 2 check bits, with 3 errors occurring (the first level of FAs is considered $K = 1$).

Table 2.3: Example of ambiguity for one bit in error.

Original word	Error word	Ambiguity
010 01	0**0**0 01	$3/5$
	010 0**0**	0
110 10	**0**10 10	$2/5$
	1**0**0 10	$2/5$
	110 **0**0	0

Table 2.4: Example of ambiguity for two bits in error.

Original word	Error word	Ambiguity
110 10	**00**0 10	$3/5$
	1**0**0 **0**0	$3/5$
	010 **0**0	$3/5$

cations. It expresses the total number error locations consistent with the received code. Given an error in a word, the possible error locations are the bits with values of 0, because they could have been modified from an original value of 1.

In Table 2.3, we consider a simple example of 5 bit words where there are 3 bits of information and the corresponding 2 check bits. The original information is 010 and the corresponding check bits are 01. If one error occurs, the possible erroneous words (given the position of the values of 1) will be $000\ 01$ and $010\ 00$. For the first case the error is in the information part ; we have 3 possible locations, therefore an ambiguity of $3/5$. In the second case we have 2 possible locations for the error, but because only 1 error occurred this is a correctable pattern and has an ambiguity of 0. There is only one cell that could be upset (given the information bits of 010), so there is no ambiguity regarding the possible error location.

For the same amount of information and check bits if the number of errors is increased we get slightly higher values of ambiguity. An example is in Table 2.4. The original word is 11010 and with two errors occurring, there are 3 possible error words and all have the same localization ambiguity of $3/5$.

The ambiguity for a word is expressed as:

$$Ambiguity_{word}(N_C(0), n_e) = \frac{N_C(0)}{n_w} \tag{2.1}$$

where:

1. $N_C(0)$ expresses the number of '0's which could have been changed from an original '1'.

2. n_e is the number of errors.

3. n_w is the size of the word in bits.

For a given size for the information bits denoted n_i, there are 2^{n_i} possible word values. Therefore, the average ambiguity for all the words in the code:

$$Ambiguity_{code} = \frac{\sum Ambiguity_{word}(N_C(0), n_e)}{2^{n_i}} \cdot 100 \tag{2.2}$$

Consider the first level of full adders. Each 3 bit *segment* in the information bits is related to a FA and generates 2 check bits. The advantage of this configuration is speed at expense of high redundancy. By stopping at this level, we can detect each *segment* that is modified if at least one error occurs in the corresponding *segment*. Each pair of check bits from a FA are appended at the end of information bits such that if an error occurs in the check bits the localization is done in these *segments* of 2 bits.

For the second level of FAs ($K = 2$), the minimum size of a segment is 9 bits that generate 4 check bits. The error localization ambiguity metric will be lower than for the case of $K = 1$. This is because the check bits correspond to 9 information bits, giving place to more possible error locations for the same amount of errors. The localization ambiguity is analyzed in sub-chapter 2.6, where the impact of the word size is obtained by exhaustive simulation.

2.4.4 Error correction

The error correction capability depends on the error pattern which occurs in the erroneous *segment*. There are several error patterns for which correction is possible. Correctable error patterns for the information and check bits are displayed in the following tables (Table 2.5 for the information bits and Table 2.6 for the check bits). For the rest of this section, consider words of 5 bits with a *segment* of 3 information bits and its corresponding 2 check bits.

For the first level of FAs the condition to guarantee the correction of single, double and triple errors is that they occur in the *segment* of the information bits: the initial value must be 11111. Therefore if 1, 2 or 3 errors occur the check bits (the last 2 bits) will be 11 and will lead to the correction of all the zeros that appear in the information bits (bit flipping the values of 0 to 1). For other information *segment* values, different from 111, correction is not possible. This is because the localization ambiguity is not null (i.e. there are more than one possible locations for errors). If the error is in the check bits there are 3 patterns that can be corrected. Because the information bits remain the same, by counting the number of 1s, the corrected value for the check bits can be generated.

Table 2.5: Correctable error patterns in the information bits ($K = 1$).

Original word	Error words	# of errors
111 11	11**0** 11 or 1**0**1 11 or **0**11 11	1
	1**00** 11 or **0**1**0** 11 or **00**1 11	2
	000 11	3

Table 2.6: Correctable error patterns in the check bits ($K = 1$).

Original word	Error words	# of errors
111 11	111 **0**1 or 111 1**0**	1
	111 **00**	2
110 10	110 **0**0	1
101 10	101 **0**0	1
011 10	011 **0**0	1
100 01	100 0**0**	1
010 01	010 0**0**	1
001 01	001 0**0**	1

The error correction efficiency (ECE) is calculated per *segments* and has the following expression:

$$ECE_{segment}(n_e) = \frac{N_{correctable\ error\ patterns}}{N_{error\ patterns}} \tag{2.3}$$

where: $N_{correctable\ error\ patterns}$ is the number of correctable error patterns that can appear in a *segment* (as explained above) and $N_{error\ patterns}$ is the total number of error patterns in a *segment*. The average error correction efficiency for all the possible values is:

$$ECE_{code}(n_e) = \frac{\sum ECE_{segment}(n_e)}{N_{segments}} \tag{2.4}$$

where: $N_{segments}$ is the total number of *segments* in the code. The error correction efficiency is analyzed further below.

2.4.5 Error escapes

A special case of errors are the ones that escape. This happens when some multiple error combinations occur simultaneously in the information bits and its corresponding check bits. They are not traceable because when comparing the original check bits with the newly computed ones, the values are equal, which is interpreted as a

Table 2.7: Examples of multiple error escapes.

Original word	Error words	# of errors
100 01	000 00	2
111 11	001 01	3

Table 2.8: Actions based on the check bit comparison. A_i are the check bits generated from the information *segment* of the code during the verification and validation. B_i are the check bits extracted from the code. See Figure 2.30.

Comparator output	Action/Description
$A_i = B_i$	Values are equal, word is error free
$A_i < B_i$	Error in the information bits
$A_i > B_i$	Error in the check bits

correct word. Table 2.7 shows a couple of possible error escapes. In sub-chapter 2.6.5 the probability of occurrence of these rare events is computed and its dependence on word size is illustrated in a graph. Note that for word sizes above 30 bits the percentage of error escapes $(ne = 2, 3, 5)$ remains below 0.01%.

Usually, the error detection process is performed when a memory word or line is read. But these codes can also be used in a pipelined architecture to further increase the detection of latent errors, in combination with a data scrubbing method. In the scrubbing method each location is periodically checked for errors when the memory is idle. The pipelined architecture consists of generating the check bits for the next word while the previous word is being checked for errors. As soon as the detection process ends for the current memory word, the check bits of the second word are pushed to the digital comparators and the check bits of the third word are generated.

2.5 Implementation

2.5.1 Cadence implementation

Depending on the number of levels each code can be implemented separately in hardware with verification and validation circuits. In the verification and validation process, the first step is to compare the check bits in the coded word with the newly ones computed from the information bits. If the values are equal the word is valid (i.e. no errors occurred) , but if the values are not equal, depending on the encoding scheme (B_1 or B_0), the error(s) can be in either parts, information or check bits as classified in Table 2.8 .

Table 2.9: Detection and localization circuit behaviour. A_i, B_i, C_i, D_i are outputs of circuit in Figure 2.34.

Output$_i$	**Description/Action**
A_i	OK (no errors)
B_i	REPLACE extended *segment*$_i$ (SE_i)
C_i	CORRECT information$_i$
D_i	CORRECT check$_i$

A magnitude comparator is used after the verification process. Figure 2.30 exposes the general structure of the codes and the architecture of the verification and validation circuits. Depending on which level of the FA tree is used and on the number of pairs of partial check bits (more information bits means more pairs), more digital comparator circuits are included to increase the detection speed. The check bits generated at time i (Cb^i) are compared to the check bits from moment $i-1$ (Cb^{i-1}) and the comparator shows if there are any discrepancies.

Figures 2.31 and 2.32 display the implementation in Cadence environment (the 65nm library) for the 1st and 2nd level of FAs. The implementations are for 9 information bits and were used to find the delay and power consumption. Note that the inputs are files which contain all the possible word values, which were found through an exhaustive simulation of all the possible cases.

The implementation of the magnitude comparator is shown in Figure 2.33. Note that a standard design is used for the implementation and evaluation of the codes and a faster and less-power hungry design is employed for the magnitude comparator [66].

2.5.2 Integrating the proposed self-healing technique in memory systems

Error detection and localization is performed by comparing the check bits for each *segment*. Therefore, for $K = 1$, each segment of 3 bits will generate 2 check bits which will be compared to the old values stored in the same code word. In Figure 2.34 the error detection, localization and correction controller hardware needed for a word of 15 bits is shown. Table below 2.9 summarizes the actions that need to perform for the B_1 encoding scheme according to the result of the comparison.

The outputs of the EDLC (Error Detection, Localization and Correction controller) combinational circuit indicates if the extended *segment* (i.e. *segment* of information plus its corresponding check bits) is correct, if we have at least one error in the information bits or check bits, or if it is uncorrectable. Since each *segment* of information and its corresponding check bits have a unique EDLC circuit,

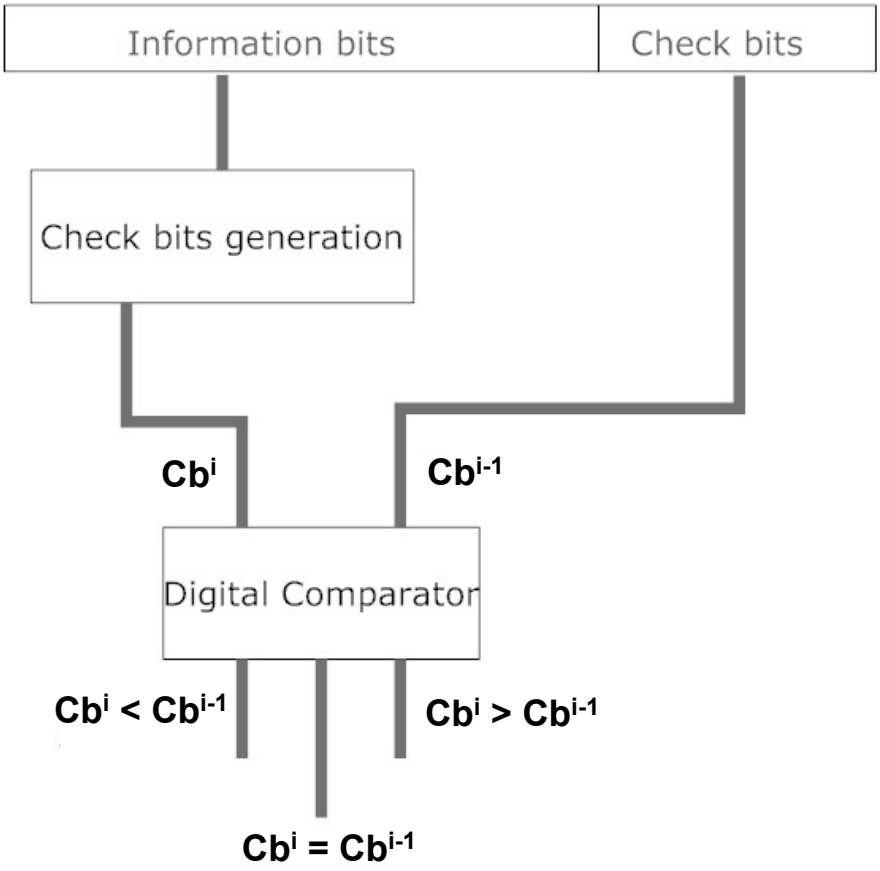


Figure 2.30: General architecture of the implementation.

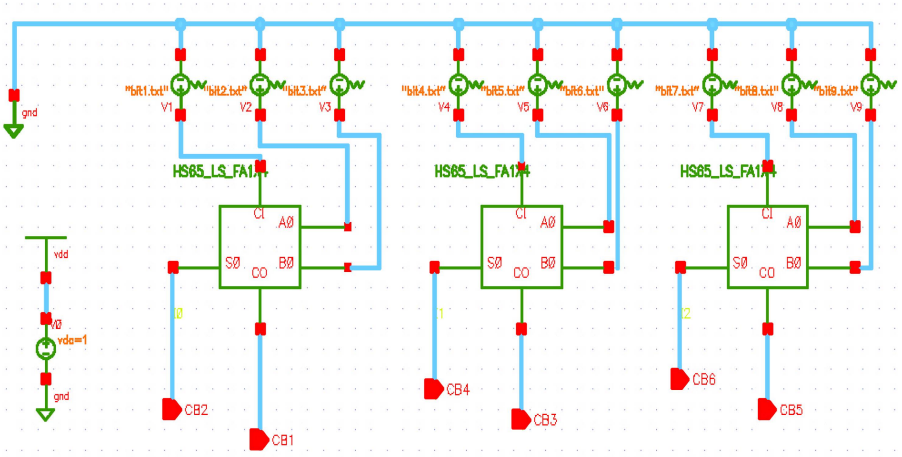


Figure 2.31: Implementation for the 1st level of the FA tree for 9 information bits.

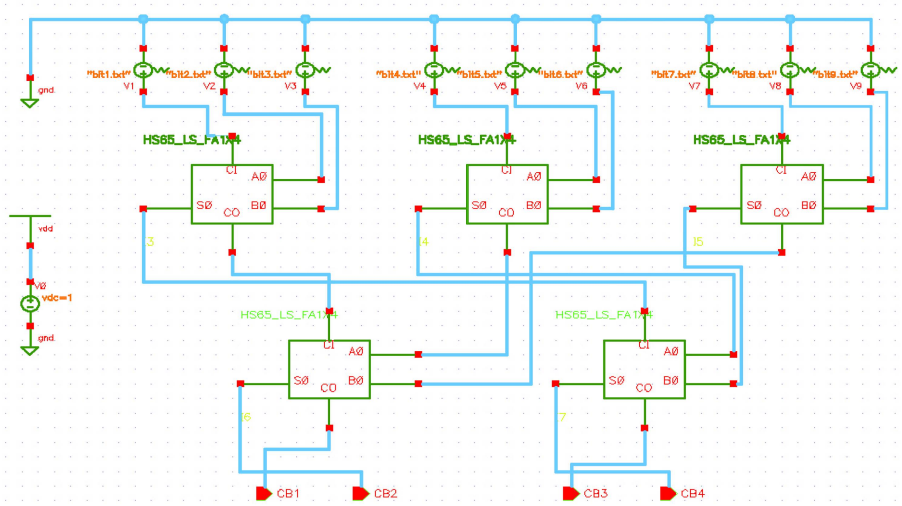


Figure 2.32: Implementation for the 2nd level of the FA tree for 9 information bits.

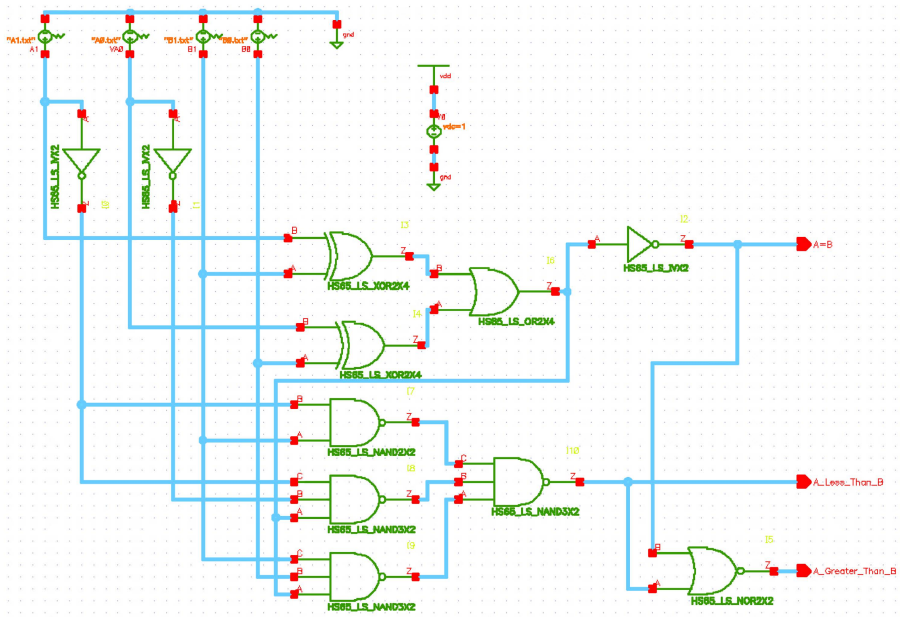


Figure 2.33: Implementation of digital comparator with input sizes of 2 bits.

the erroneous *segment* will be easily localized and the EDLC outputs will control the error correction circuit. The error correction is performed on the segments in error when this is possible which includes error escapes (sub-chapter 2.4.5), otherwise if the errors are uncorrectable a spare *segment* is used for replacement (Figure 2.35).

The B_i output from the EDLC circuit denotes the *segment* that needs replacement. If more than one *segment* needs replacement, a priority is defined at design time. In Figure 2.35 a single spare *segment* is presented where the most significant *segments* have higher priority in front of uncorrectable multiple erroneous *segments*.

The strategy used for uncorrectable errors is to shift-out the erroneous *segment* and to replace it with a spare one. The damaged segment is reset to an initial value and can be used again. This strategy can also be used for hard errors, but the damaged *segment* is replaced completely and not reset since the errors are permanent. Note that the number of spare *segments* is decided when implementing the code, taking into consideration the number of possible errors that can occur in the memory system during normal operation. The figure above contains only one spare segment to illustrate the replacement method. Likewise, the priority when replacing is decided at run time.

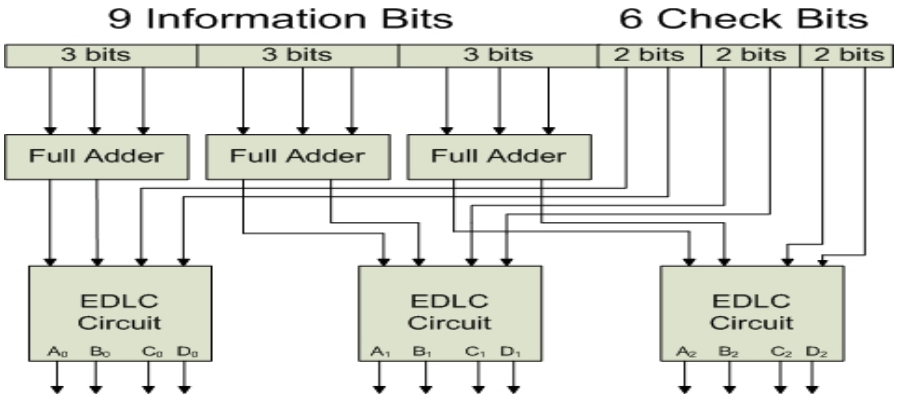


Figure 2.34: Error detection and localization circuit (example for 9 information bits, 6 check bits and $K = 1$).

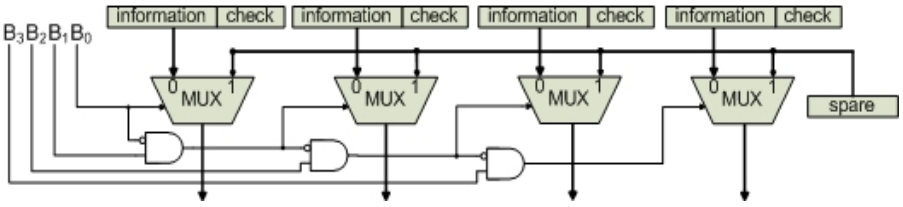


Figure 2.35: Error correction for uncorrectable segments using a spare segment.

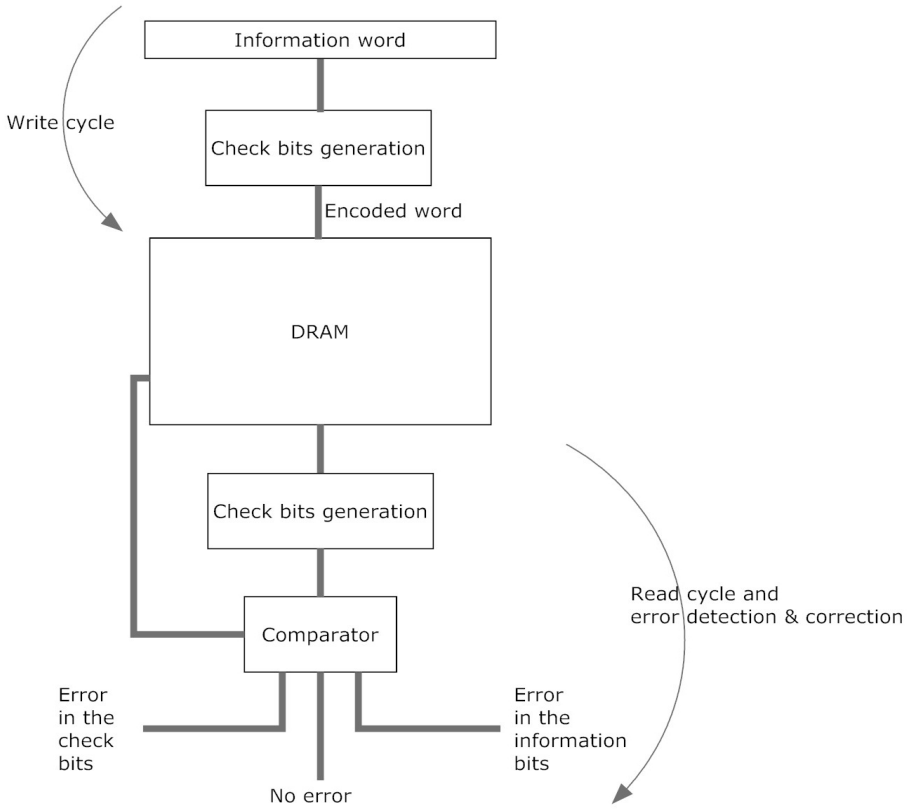


Figure 2.36: General architecture of the implementation.

The error correction capability depends entirely on the patterns explained before. Therefore, the implementation is able to detect these patterns and flip the bit(s) in error. A simple and efficient design in terms of speed and area is using XOR gates. By XOR-ing a bit with a value of 1, the output is 0 if the bit is 1 and 0 otherwise. Thus, the erroneous bits can be forced to a value of 1. The general design used during run-time is visible in Figure 2.36. The fast cache DRAM is in the center of the implementation. Consider that the memory is addressable by word line and that each one holds an encoded word (information plus check bits). During the write cycle, the check bits are generated from the information bits, then appended to the original information word and stored in the DRAM as a word line. The read cycle begins by generating the check bits from the information bits of a specific word line, followed by a comparison between the newly generated check bits and the check bits stored in the encoded word.

Given the fact that the proposed codes are based on the Berger code B_1, the detection process is made with a simple comparison between the check bits. The 3 outputs from Figure 2.36 are inputs to the localization and correction circuit. This is a straightforward implementation where each stage (error detection, localization and correction) is clearly defined. However, a more optimal solution is to design and implement a single circuit that does all three functions faster and with less area requirements. Both solutions are applicable and valid; it all depends on the memory systems' final requirements. The first solution offers modularity (i.e. each phase is done by a different circuit), therefore if one of them has a fault or is not functioning properly it can be easily replaced. The second alternative is based on an "all-in-one" circuit (called EDLC), which comprises all the necessary circuitry for all phases. The latter is presented in the previous paragraphs. Note that both proposed solutions for the error detection, localization and correction phases support the self-healing concept. Likewise, other techniques as memory scrubbing or operation pipelining can be used in combination with the proposed code, in order to create a self-healing memory system. These techniques are also discussed in the following paragraphs.

The two solutions explained above are usually enforced at run-time and detect and correct errors "on-the-fly". This can be problematic if we consider large amounts of errors (multiple random errors) and the fact that the performance of a system can be lower than in normal operation (i.e. degraded performance). Thus, it can be more efficient to run these operations when the memory system is idle. The memory scrubbing technique is employed in memory systems as a background operation, which runs when the memory is not active (i.e. idle state, when no read or write operations are running). Therefore, the proposed codes are used to check every memory location to see if errors have occurred. This can prove to be effective since radiation upsets are mostly random and unexpected. Employing memory scrubbing on DRAMs is also effective in terms of reducing the refresh power, because the checked memory locations are automatically refreshed (i.e. the value is re-written) and the refresh timer is reset.

Another integration solution is to pipeline the operations in order to save computing time. Before each phase (detection, localization and correction) the corresponding memory word(s) need to be ready for processing. Note that each phase is different and needs specific information as inputs, therefore pipelining is possible. By pipelining the operations, we prepare in advance the inputs for the circuits. Figure 2.37 describes in more detail the proposed pipeline operation . The idea is to generate the check bits for the next word while the previous word is being checked for errors. As soon as the detection process ends for the current memory word, the check bits of the second memory word are "pushed" in to the digital comparators and the check bits of the third word are generated. Please consider the

fact that after the detection process, if errors occur and are detected the pipeline architecture expands with two new branches: the correcting stage (if the errors can be corrected) and the replacement stage using spare segments (if the errors are uncorrectable). The replacement mechanism is displayed in Figure 2.35 and has the following steps:

1. The errors are detected and localized, but cannot be corrected.
2. The segment of the memory word containing the errors is replaced by a spare one.
3. The replaced segment from the memory word is re-initialized to a default value (default value is 0) and can be used again (by default is set as a new spare segment).

The timing analysis for the proposed pipeline architecture is illustrated in Figure 2.38. Note that at time T_{i-1} the previous code word check bits are compared while the check bits of the next code word are generated. This proposed pipeline method, used in conjunction with a memory scrubbing technique, can achieve up to 50% speed improvement and significant less refresh power consumption because the checked memory locations are automatically refreshed. However, in order to achieve lower refresh power consumption, the refreshed memory words by the scrubbing method need to have a time-stamp, which can be kept in a Lookup Table (LUT) or an array of recently refreshed memory lines/words.

2.6 Experimental results and evaluation

In this chapter, several metrics calculated for the proposed code are illustrated and applied to evaluate the metrics of the CS-K Berger codes when the word size increases. In order to demonstrate the advantages given by the proposed codes, all the evaluations are compared with the original Berger code. The delay of codes *code delay* is evaluated as the number of FA gate levels generating the partial-check bits. The proposed *code delay* is then compared to the delay of the Berger code implementation from [41].

The proposed CS-K Berger codes have a constant delay, depending on which FA level of the tree-shaped design is used. Thus, the first level has a delay of only 1 FA, the second level has a delay of 2 FAs and so on. Because the Berger code check bits are generated by the last level of the parallel tree-shaped design, including the ripple-carry adder, the slope of delay increases much faster when considering large word sizes.

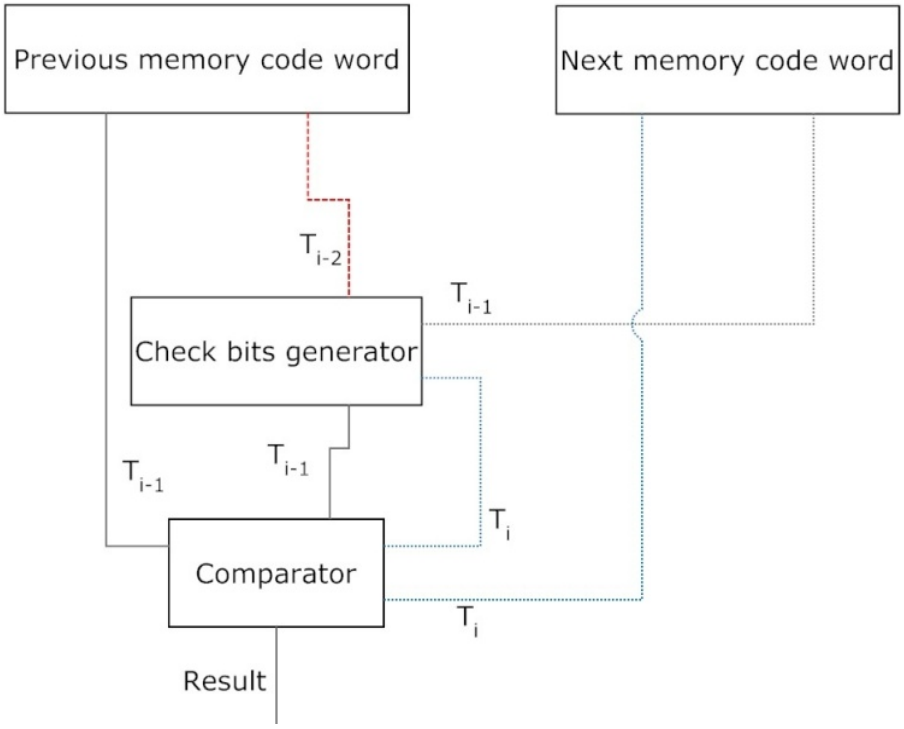


Figure 2.37: Pipelining Operation.

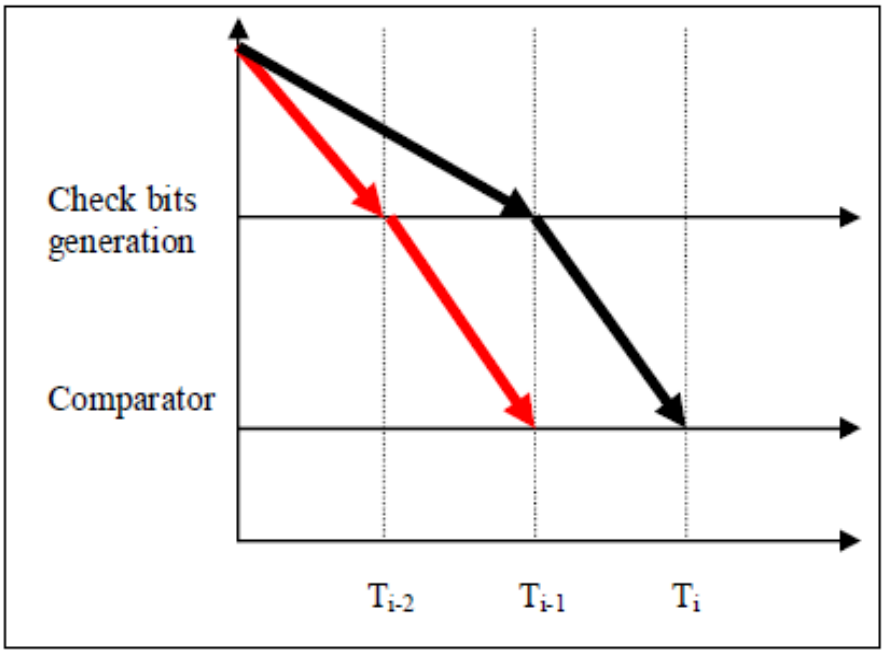


Figure 2.38: Timing results in the proposed pipeline architecture.

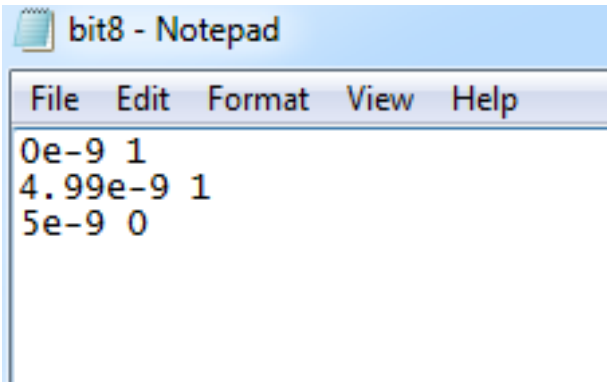


Figure 2.39: Example of .txt stimulus file containing values for one bit of the inputs of an implementation.

The *code redundancy* is computed as the ratio between the number of check bits divided by the total number of bits (information plus check bits). Further explanations are in sub-chapter 2.6.2. *Area overhead* is calculated in terms of number of FAs which are needed for each level of the tree-shaped parallel design, in comparison with the ones needed for the Berger code, for different word sizes. First, the number of FAs needed for each code is compared, then the percentage of each proposed code FAs out of the Berger code FAs is calculated.

Because the CS-K Berger code offer error detection, localization and correction three metrics have been defined and evaluated: *error localization ambiguity*, *error correction* and *error escapes*. Each of these are further discussed and analyzed in the following sub-chapters. A C++ program has been developed which calculates and evaluates each metric exhaustively.

The power consumption is calculated in Cadence environment, by using an implementation for nine information bits for a library of 65nm technology node. The inputs of each implementation are fed with ".txt" stimulus files which contain values of the information bits at specific time intervals. An example of an stimulus file is shown in Figure 2.39. The format for the input values is the following:

$$< time >< value\ of\ the\ information\ bit > \tag{2.5}$$

The simulations start with a transient analysis of each circuit implementation, ran for a specific time (depending on the number of transitions which are set in the stimulus file). Note that in Figure 2.39, only one transition from $1 \rightarrow 0$ is set, in order to simulate one random error in the eighth information bit. During simulation the power consumption is calculated by measuring the intensity of current which

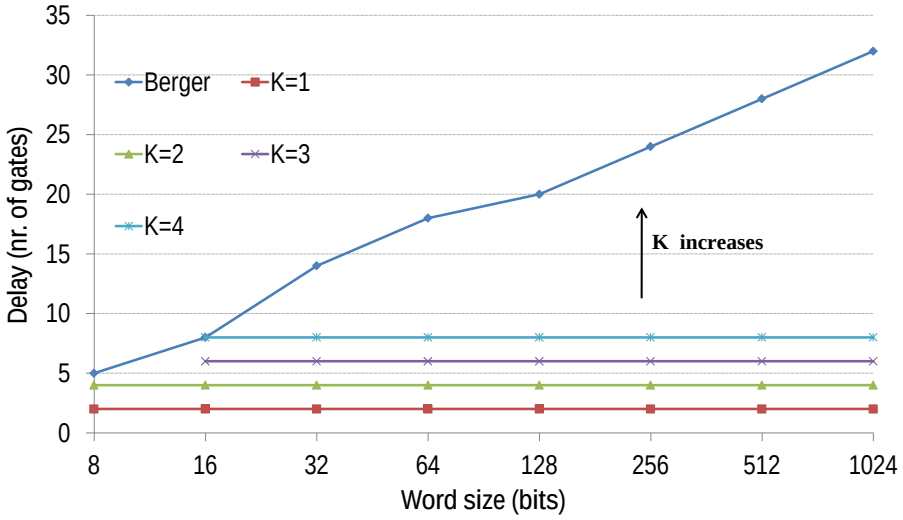


Figure 2.40: *Code delay* of CS-K Berger codes.

flows through the power supply line during transitions multiplied by V_{dd} voltage (set to 1.0 V by default). The results are presented in sub-chapter 2.6.7.

2.6.1 Code delay

The *code delay* of CS-K Berger codes (for the first four levels of FAs) can be observed in Figure 2.40. The code delay is expressed as number of FA layers, for each information word size raging from 8 to 1024 bits. The values for the Berger code are from reference [41]. For the proposed codes, the delay remains constant regardless of the word size and is indicated by the value of K (i.e. level of FAs).

2.6.2 Code redundancy

For the original Berger code, the *code redundancy* can be calculated by the following expression (n_i is the information size):

$$C_R = \frac{\lceil \log_2(n_i + 1) \rceil}{n_i + \lceil \log_2(n_i + 1) \rceil} \cdot 100 \tag{2.6}$$

Figure 2.41 illustrates the *code redundancy* for information word sizes from 8 to 1024 bits and $K = 1, 2, 3, 4$, Berger code is also added for comparison.

For $K = 1$ and *segments* of 3 bits (the first level of FAs), the code redundancy is about 40%. This is because a FA generates a code redundancy of 40% (3 inputs

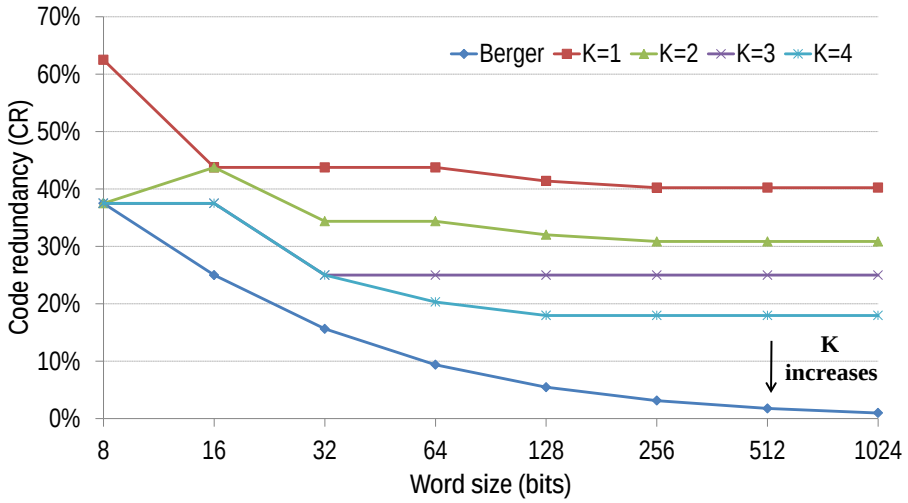


Figure 2.41: Code redundancy for CS-K Berger codes and original Berger codes.

for information bits and 2 outputs for check bits, which gives a redundancy ratio of $2/5$). But because a word contains the information and check bits, some values are not exactly a power of 2 giving small variations to the code redundancy.

For $K = 2$, the minimum amount of information bits is 9 and generates 4 check bits. So, the code redundancy is $4/13$. But it can fluctuate as explained in the above paragraph.

2.6.3 Error localization ambiguity

As explained in sub-chapter 2.4.3, the error localization is quantified with the metric *error localization ambiguity*. The ambiguity expresses the number of possible error locations divided by the total number of possible locations. In Figure 2.42 this metric is assessed for CS-1 Berger code considering a number of simultaneous errors from 1 to 5. First we see that, as expected, more simultaneous errors worsen this metric. However, when the word size increases the metric improves (decreases its value). This result is specially interesting for memories where the size of word could be made significantly large like in cache memories.

2.6.4 Error correction

The graph in Figure 2.43 illustrates the error correction metric for a maximum of 5 simultaneous randomly distributed errors. Depending on the number of errors

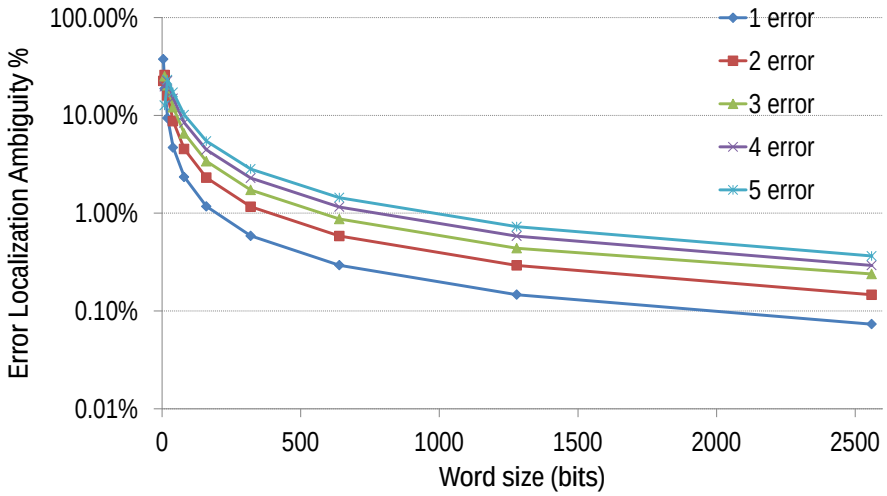


Figure 2.42: *Error localization ambiguity* for the first level of FAs in CS-1 Berger code. Simultaneous errors are changed from 1 to 5.

that occur in a word, one or more segments can have errors. All possibilities of error occurrences are considered. As was explained in sub-section 2.4.4 not all error patterns can be corrected. This is visible in the figure, through the maximum percentage for error correction metric. It reaches a value close to 45% for the one error case and/or the largest word sizes. Like in the previous metric, the trend of the *error correction* that improves with the enlargement of the word size is particularly interesting for cache memories.

2.6.5 Error escapes

A special case of errors are the escapes (for the B_1 coding scheme), when the errors occur in both information and check bits as explained in the sub-chapter 2.4.5.

The *error escape* metric, EES, is calculated as a probability over 100 according to the following expression:

$$EES = n_{esc} \cdot \frac{1}{2^{n_i}} \cdot \left(\frac{1}{n_w}\right)^{n_e} \cdot 100 \tag{2.7}$$

In the above expression, n_{esc} is the exhaustive number of possible error escapes for each word size , n_i is the number of information bits, n_w is the word size in bits and n_e is the number of errors.

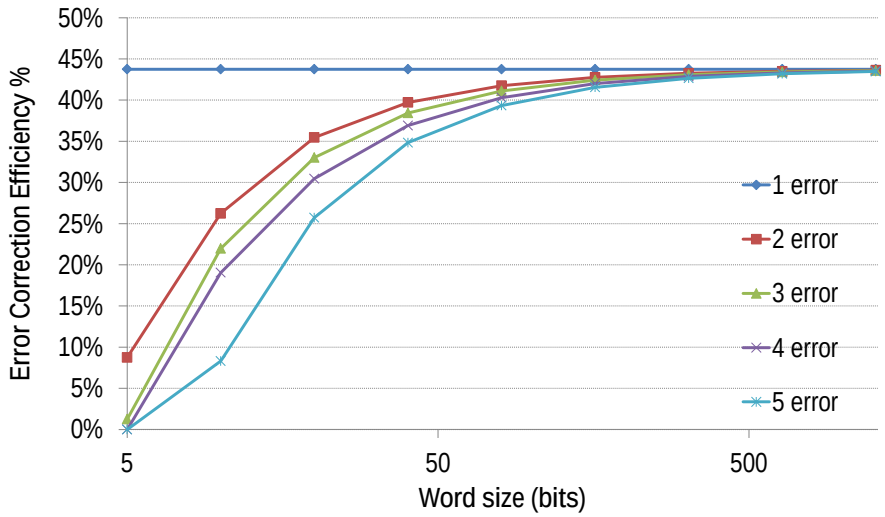


Figure 2.43: *Error correction* per segments for the CS-1 Berger code, errors from 1 up to 5 bits.

In Figure 2.44 *error escapes* are shown for CS-1 Berger code with 2, 3 and 5 multiple errors. Note that for 1 or 4 multiple errors, *error escapes* are not possible. All word sizes not represented in the graph have an EES probability of 0.

2.6.6 Area of the code generator and memory resources

The area of the code generator can be calculated by using the number of CMOS transistors (derived from the number of FAs) needed for each level and compared with the code generator for the Berger code, for a range of specific word sizes. We used the 8 transistor Full Adder design from [69]. Figure 2.45 exposes the area or the code generator (expressed as number of transistors) calculated for each code and Figure 2.46 exposes the area of the code generator in percentage occupied by each code design in comparison with the Berger code design.

Please note that the comparison circuit has not been taken into consideration because all of these codes use the same type of circuit.

The memory resource needed for different K levels is computed with the help of code redundant bits. Each level generates different amounts of redundant bits and are compared with the Berger code check bits. Furthermore, their redundant bits are scaled for SRAM 6T and DRAM 1T cell types. The word sizes are similar to the previous two figures. Figure 2.47 illustrates the memory resource necessary for each code. Finally, in Figure 2.48 is displayed both code generator and memory

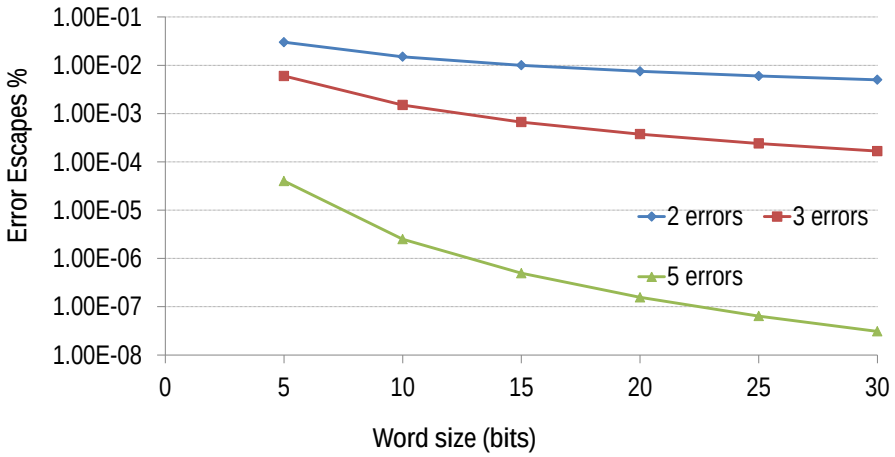


Figure 2.44: *Error escapes* probability for CS-1 Berger code for 2,3 and 5 multiple errors.

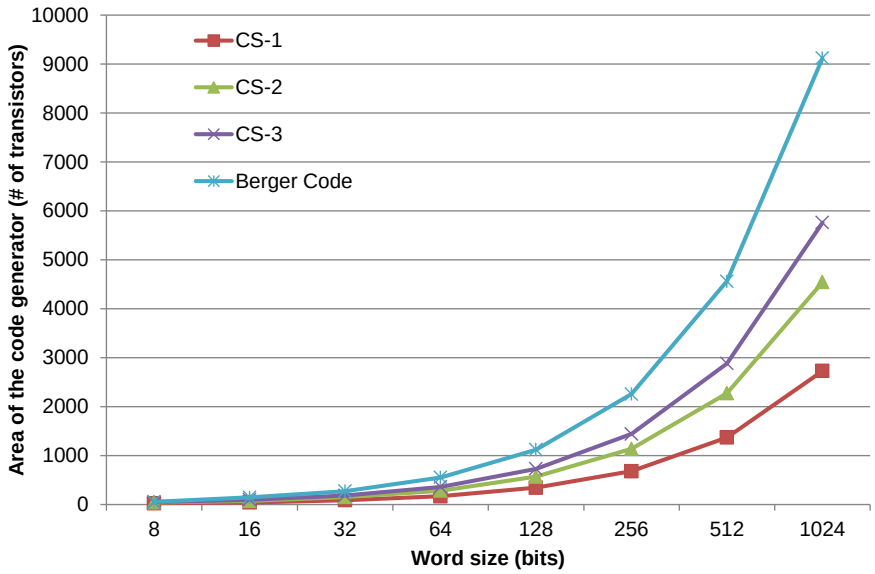


Figure 2.45: Area of the code generator computed for CS-1,2,3 and the original Berger codes and different word sizes.

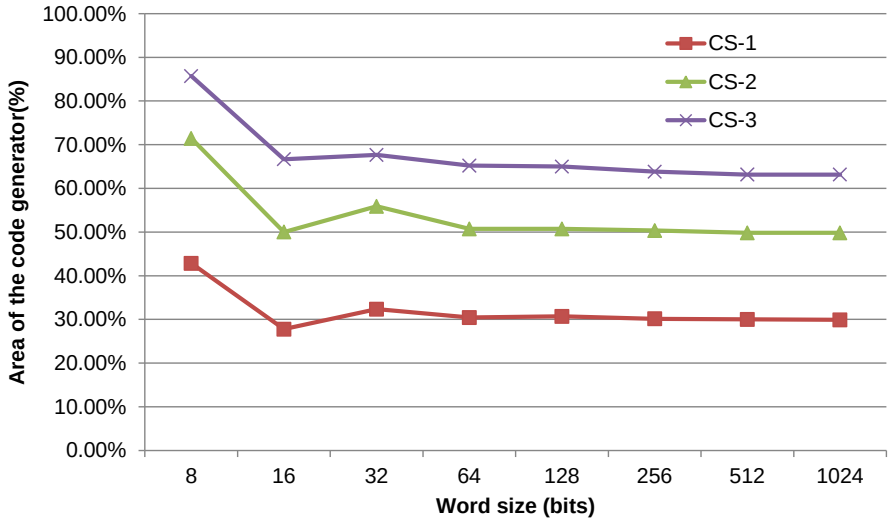


Figure 2.46: Area of the code generator computed as percentage from the Berger code area for the first three levels of Full Adders.

resources needed for CS-1,2,3 and compared to the Berger code.

2.6.7 Power consumption

The power consumption is calculated also in comparison with the Berger code. The results show that the 1st level of FAs use only ≈28% of the Berger code implementation, while the 2nd level of FAs uses ≈63% of the Berger code implementation, for 9 bits of information. Figure 2.49 displays a simple comparison of the power consumption between the first two levels of the FA tree and the Berger code. Note that only 2 transitions of $1 \rightarrow 0$ are assumed during the whole simulation (to imitate 2 random errors), in order to clearly see the lower power consumption of the proposed codes. Thereby, the 1st level of FAs uses ≈25 μW in each transition, the 2nd levels of FAs uses 40 μW for the first and ≈70 μW for the second transition because more FAs are involved in generating the check bits than in the 1st level of FAs , while the Berger code has higher power consumption than the first two: ≈60 μW for the first and ≈110 μW for the second transition.

2.6.8 Delay

The delay for the implementation of the proposed code was computed as the time needed to obtain the check bits from 9 bits of information. In order to simulate

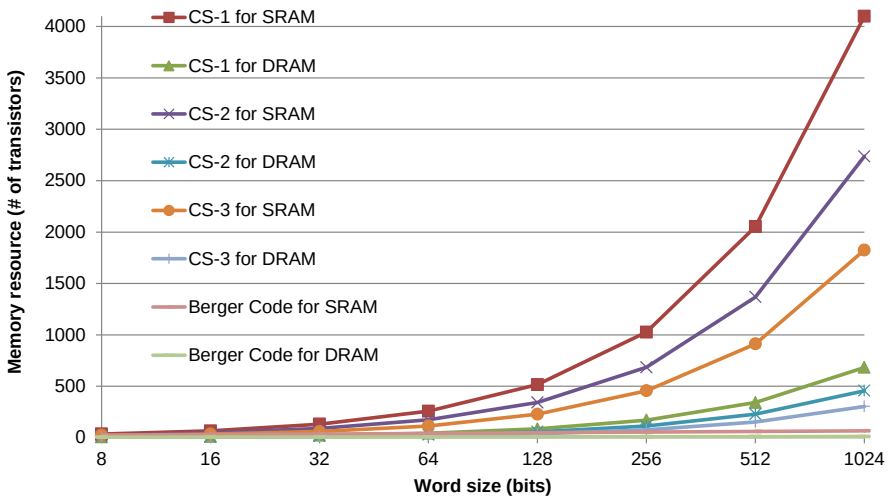


Figure 2.47: Memory utilization area for codes CS-1,2,3 and Berger code, when SRAM and DRAM cells are used.

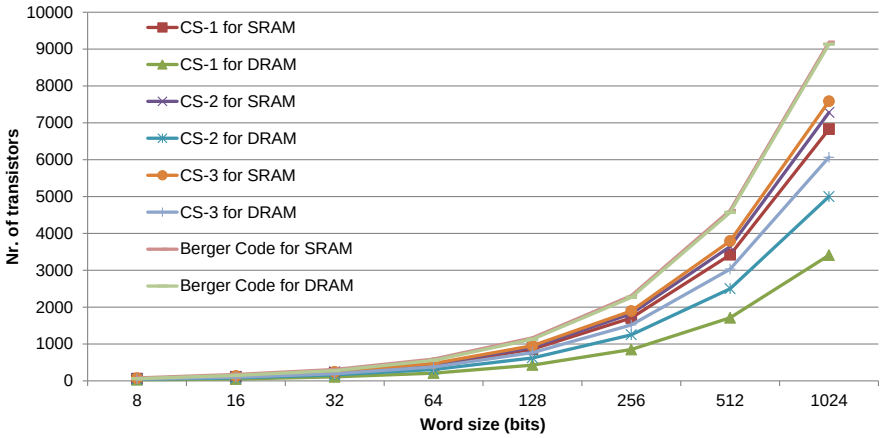


Figure 2.48: Area of the code generator and memory resource necessary for CS-1,2,3 and Berger codes.

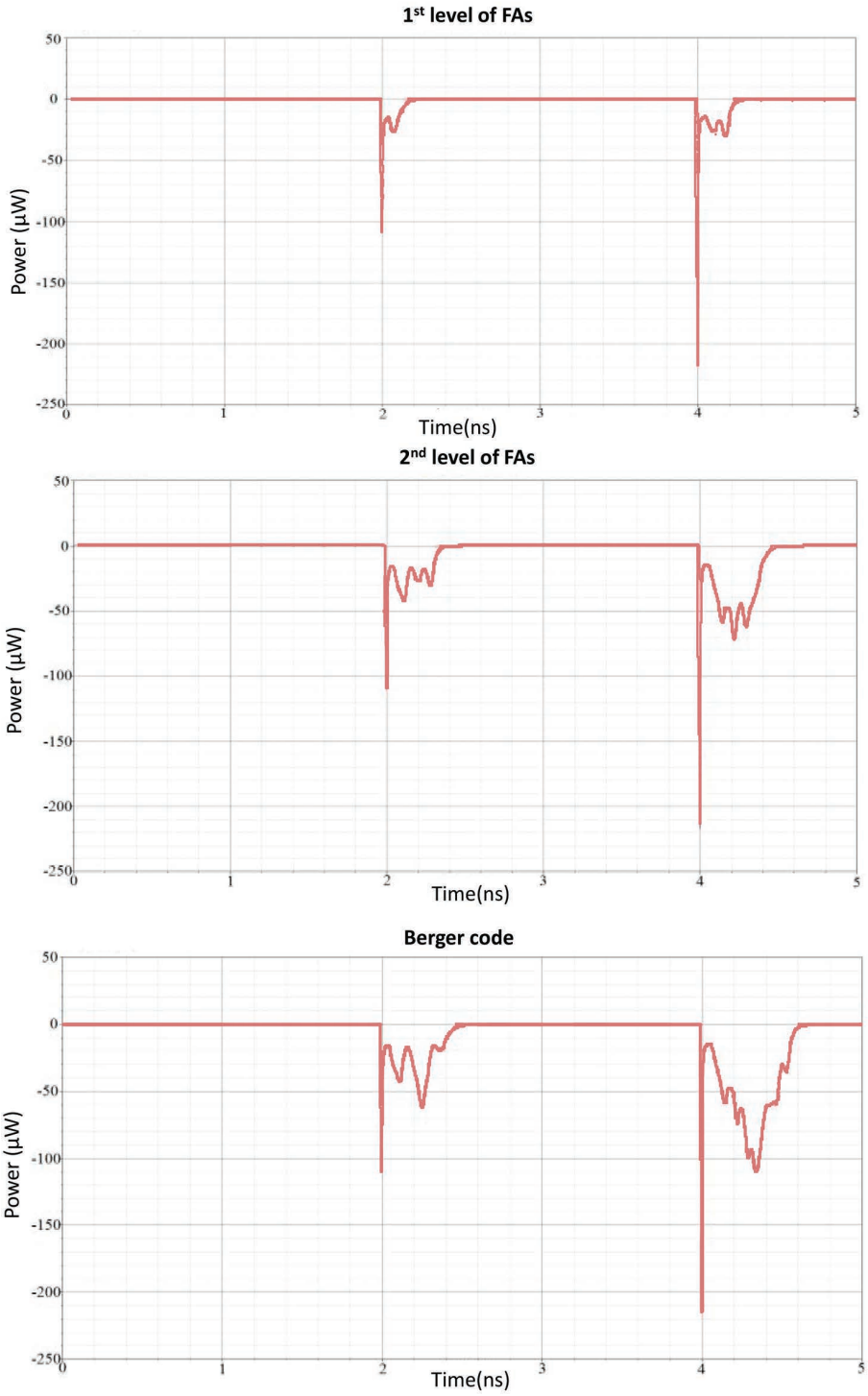


Figure 2.49: Power consumption comparison between the 1st and 2nd level of the FA tree and Berger code (the values are negative because the electric current is measured at the ground node of the circuit).

this, the inputs were activated at a specific time (2 ns and 4 ns) and the transitions are visible in Figure 2.50 below. Note that the comparison is between the first and second level of FAs and the Berger code. The logical voltage level for 1 is 1 V, while for 0 is 0 V. As illustrated, the 1st level of FAs needs 0.1 ns and 0.2 ns to generate 6 check bits, the 2nd level of FAs requires 0.3 ns and 0.4 ns to produce 4 check bits, while the Berger code demands 0.4 ns and 0.6 ns to generate 4 check bits.

2.6.9 Overall evaluation

As displayed in the graphs above, the proposed codes are clearly much faster at generating the check bits, consume less power than the Berger code implementation, have a lower area overhead and have error localization and correction (in comparison with the Berger code which has none). However, the code redundancy is higher than the original Berger code.

The simulations were run for all possible values of the information word and the results show improvements over the common Berger code scheme. Note that if the size of the information bits increases, the improvement is even higher, due to the fact that the first or second level of FAs has a constant delay, while the Berger code tree complexity increases simultaneously with the size of the information bits (as visible in Figure 2.40). Regarding the power consumption, the results show that the 1st level of FAs use only ≈28% of the Berger code implementation, while the 2nd level of FAs uses about ≈63% of the Berger code implementation, for 9 bits of information. The proposed codes have error correcting capabilities and perform better if higher code redundancy (best for the first level of FAs), Also, the codes provide error localization capabilities which has an ambiguity of ≈5% for word sizes over 512 bits.

The area overhead is also lower that the original Berger code implementation. For a word size of 1024 bits, the code from the 3rd level of FAs occupies ≈63% of the Berger code area, the code from the 2nd level of FAs occupies ≈50% of the Berger code area and the code from the 1st level occupies ≈30% of the Berger code area. Code redundancy when high if compared to Berger code, but if considering high word sizes, a trade-off between speed and which proposed code should be implemented is needed.

The error localization ambiguity expresses the total possible locations for errors, if any occur in the word. As the word size increases, the error localization ambiguity decreases very fast, going under 5% for a word size of 512 bits. This means that the higher the word size gets, there are less possible locations for errors, and thus the errors can be localized.

Error correction is also possible, although the maximum percentage is almost

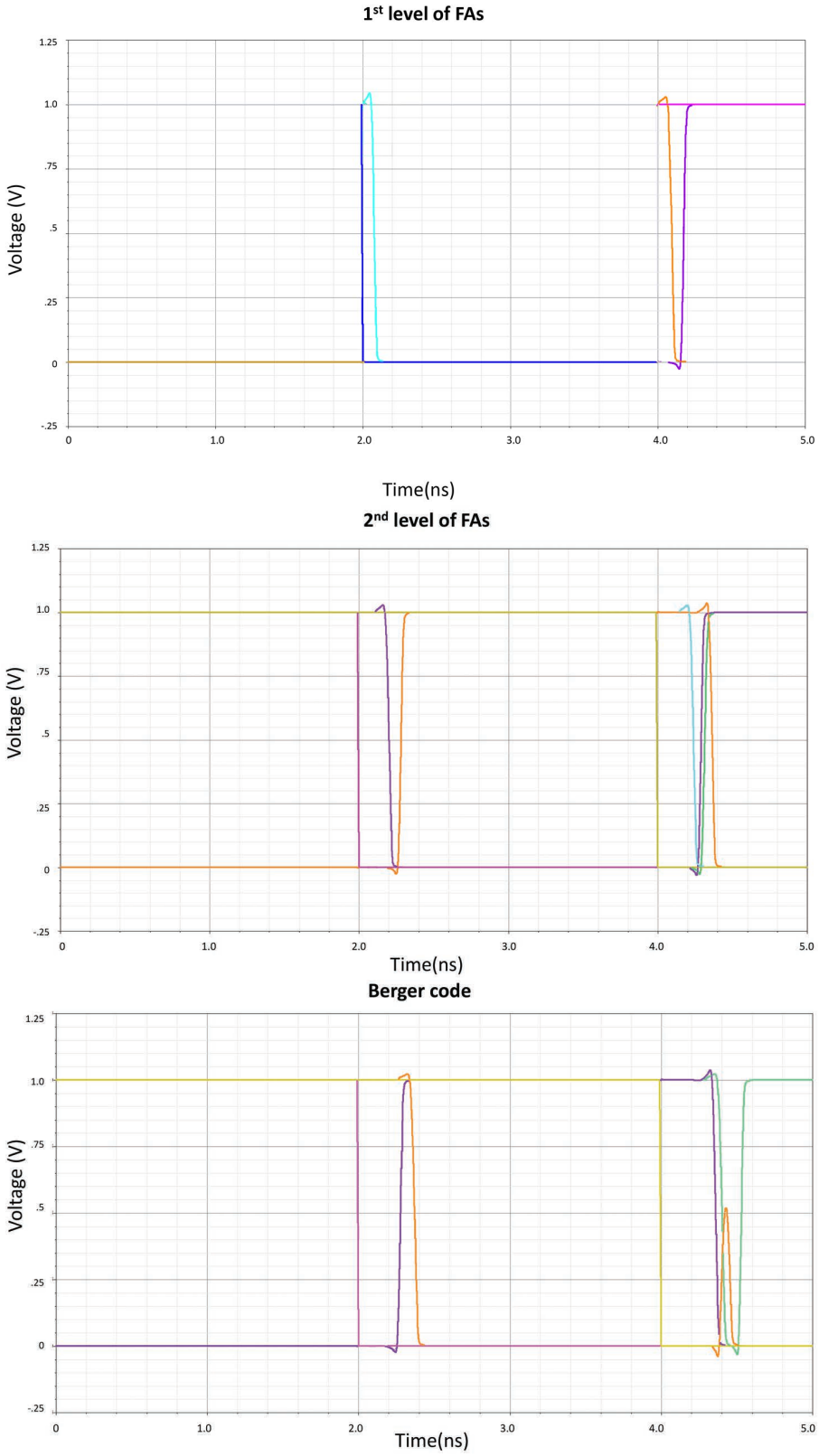


Figure 2.50: Delay comparisons between the 1st and 2nd level of the FA tree and Berger code.

45%. Thus, almost half of the errors detected and localized can be corrected, regaining the original information.

Error escapes appear only when using the B_1 coding scheme and if errors occur in both information and check bits. However, the trend is below $10^{-2}\%$, because there are few patterns that provoke error escapes and the errors must occur random, in the same segment (highly unlikely).

The proposed codes can detect, localize and correct multiple unidirectional errors that appear in DRAMs and trade-off speed for code redundancy. However, the code redundancy can be reduced if the number of FA stages is increased. All in all, high-speed caches implemented with DRAM cells are advisable to be used.

2.7 Conclusions

Self-healing systems are becoming more and more important due to the numerous error sources, which translates into keeping the information as accurate as possible. Any system with such techniques must have a well defined architecture, capable of monitoring, planning and must adapt to errors. Self-healing techniques and methods must be implemented to run autonomously, with no human intervention, being capable of modifying its structure and run-time parameters in real-time operation.

Because there are multiple architectural designs for systems, self-healing techniques as well, the final objective/scope of the system is the most important one. For memory systems, the accuracy of the stored data is crucial, because the data gets used later on by the processing unit. The self-healing techniques for such systems include error detection and correction codes and are usually used with other replacing methods (using spare components), in order to reduce the computing time and high data redundancy. The monitoring operation is also important (e.g. sensors, background software process), as well as the detection/correction part, which can run when the system is idle or during run-time.

Any systems that implements self-healing techniques and methods must be evaluated from the performance perspective (the normal run-time operation of the system must not be highly/severely diminished), the additional circuitry must have a low area overhead and a low power consumption (the latter two are modern trends). The performance of such a system includes processing speed (how fast can it detect and correct errors) and the number of errors that can be detected and corrected (the more, the better). All evaluations must be compared with existing proposals and solutions.

The design and implementation of the self-healing techniques must be done for the first time in a virtual simulation environment. There are a lot of software solutions that provide a good and complex simulating environment. Fine tuning can

be done afterwards (e.g. logic synthesis to adjust the design), in the perspective of obtaining the best results in the considered metrics. Also, because the self-healing concept implies autonomous computing, it is important that the system can execute these techniques and methods during normal run-time operation, with no human intervention. The latter is a necessity when designing and implementing self-healing techniques and methods.

In the present work, a new family of error detecting and correcting codes are proposed and evaluated from the perspective of speed, code redundancy, area overhead, power consumption and several specific evaluations: error localization, error correction and error escapes. Because of the low delay, they can be considered very efficient for fast cache DRAM memory, where the code overhead is not important. The proposed codes are efficient from almost each point of view evaluated in this work. As regarding the power consumption, the design is less power-hungry in comparison with the original Berger code implementation. In order to achieve a memory system with low-power consumption and EDC/ECC, the error detection and correction can be done either when the memory is idle or inactive, either when a word is read. The operations can be executed in a refresh-time period, with high consistency, at the expense of more check bits. Also, when comparing the values of the partial check bits, if more comparators are used, the detection speed improves with the cost of higher area overhead and power consumption. The operations can also be pipelined so than no time or power is wasted.

Chapter 3

Security in cache memories (IST)

3.1 Introduction

Modern day ICs contain significant information that must be highly secured, so that it stays hidden. In the last 5 years, there have been reported a large variety of attacks against ICs and memories, mostly targeting the vulnerabilities of cache and main memories, smart-cards, etc. This problem broadens when the attacks target credit cards and other kind of legal supports including biometric information [3].

Computer systems have drawn a lot of attention, due to the fact that sensitive information is stored in the memory during runtime. Some attacks on the main memory have been investigated because the stored information is still visible a small period of time after the system loses power (known as memory remanence). These attacks, called cold-boot attacks [21], [22], target the recovery of the last information which was stored in the main memory. Thus, relevant information such as encryption keys could be retrieved. Also, by keeping the memory module at low temperatures, the information can remain intact for as long as 5 minutes [21].

In memory systems, the main memory has usually a low level of protection because it is more accessible than other levels of memory and thus it can be removed by the attacker by simpler technical means. In cold-boot attacks, the information is frozen for a long enough period of time such that the memory modules can be removed and the attacker can download the content into a backup system which stores data in plain text. To avoid this, security is improved by encrypting data while the memory transaction is undertaken [22], [23], [70], [25]. The decrease of bus throughput can be compensated by an increase of L2 cache size. Unlike the main memory nowadays, for performance reasons, the cache memory is placed in the same CPU package, either stacked on or embedded within it. This configura-

tion provides a higher protection degree at a free cost. Because of this and also to avoid strong penalty in the throughput between cache and CPU, the information is stored in plain text in the cache. Even more, some encryption algorithms are designed to run all the rounds using only cache memory and avoiding main memory transfers [26].

Attacks that target the cache memory and retrieve sensitive data have been analyzed and investigated in several works [25], [26], [27], [28]. Most of these are side-channel attacks, which are based on the information leaked by a cryptographic devices, such as power consumption, timings, etc. Attacks against AES algorithms have been reported that target the stored private keys [26], [27], [28]. It cannot be ruled out the possibility of direct reading of the cache content. If cache is conveniently frozen, a power-up sequence can derive to the recovery of the same content existing before the power-up [27].

In this chapter, a new methodology, Interleaved Scrambling Technique (IST), is proposed to conceal the plain text data from the L2 cache memory. It pursues to make data unusable in case of an attacker retrieves it and to avoid the transfer of regular patterns between CPU and cache which leaks significant information through side-channel attacks. A data scrambling technique is selected which scrambles data in the write cycle and descrambles when data is read from the cache. It periodically refreshes the scrambling vectors using a unique random source and a table that keeps track of the expired vectors avoiding the need of updating the whole cache content at a time. This gives a significant reduction of the power consumption if compared to standard scrambling techniques. The elapsed time between refreshes is data dependent and thus it makes more unpredictable than if it were done at constant time intervals.

3.2 Theoretical background

The cache memory is in the middle of any memory system. Every time the CPU requests data from the main memory, it passes through the cache. The L2 cache will store the last accessed information which was requested by the CPU. Also, when the CPU generates new information, the data is first retained in the cache and then passed to the main memory. Considering the above, the cache memory security is a priority because of the sensitive information which can be retained for a period of time.

Physical attacks on static cache memories (SRAM) are possible in powered-off state. A recent paper [71] tries to increase the security of SRAMs by specialized circuitry to eliminate any data remanence.

Protecting against side-channel attacks is also widely explored and discussed.

Authors in [72] propose the use of a dual-rail precharge principle, applied either by cutting the power supply or by cutting the feedback loop. The results show significantly lower variations of supply current compared to standard SRAM designs, thus better resistance against power-analysis.

The authors in [27] state that the traditional way of reading data from RAMs or FLASH memories (invasive attacks using mechanical probing, depackaging the chip, etc.) is becoming very difficult due to the shrinking of the feature size or because of hardware access control circuits. Semi-invasive attacks (chip is still depackaged, but no direct electrical contact) are explored, including optical probing (induced transient faults in some logic gates to cause information leakage) and eddy current attacks (inducing a large transient magnetic field near the surface of the chip), each of them carried out while the chip is in a frozen state.

Despite the reading of a cache memory content (or any RAM) is difficult and hard to perform with basic technical means, in a presentation from Black Hat [73], the author talks about the possibility of memory data acquisition: software-based (additional software, OS-dependent) and hardware-based (PCI/PCMCIA cards, DMA access, OS-independent). However, the hardware-based approach still needs direct access to the machine, while the software-based acquisition is not reliable.

As presented above, retrieving sensitive data from cache memories is possible and several techniques have been proposed to increase its security. Data scrambling is a simple yet efficient technique to obscure plain data and can be employed in cache memories, if specific rules are defined and followed. The next section covers the principles of data scrambling, in respect to cache memories.

3.3 Data scrambling

Scrambling is the function of replacing a character (or byte) of data with a different character (or byte) of data. Simply put, typical scrambling technologies would replace, for example, the letter A with G. While this example is very simple, it would be very easy for someone with a hacker mentality to figure out this simplistic way of scrambling. A modern solution actually uses random generation of characters to replace existing data with scrambled data. Because the characters are random in nature, a hacker will be unable to figure out what the original data was.

In the following, D will denote the *plain data*, as sent by the processor on the bus. The scrambling process transform D into a new value SD by applying an injective (i.e. one-to-one) function f to D (i.e. $SD = f(D)$). The inverse process is done at the receiver's end.

A scrambler consists of two main blocks: a randomizer (a random pattern generator, e.g., a linear feedback shift register- LFSR), that produces a new pattern

S at each new bus cycle, and the block mixer, which performs the modulo-2 addition (i.e., bitwise XOR) between the D and S. Thus, the scrambled word is: $SD = D \oplus S$ [74].

The *additive scrambler* is the simplest method to scramble any information. Basically, it transforms the input data stream by applying a pseudo-random binary sequence (by modulo-two addition). Sometimes, this sequence is stored in a ROM, but more often is generated by a linear feedback shift register (LFSR).

The *multiplicative scrambler*, also known as feed-through, performs the multiplication of the input signal by using a transfer function in Z-space, thus they are discrete linear time-invariant. While the multiplicative scrambler is recursive, the descrambler is non-recursive. In telecommunications, the additive scrambler needs a sync-word before transmitting the scrambled word, while the multiplicative scramblers don't need frame synchronization (also called self-synchronizing). Both types of scramblers/descramblers are defined similarly by a polynomial of theirs LFSRs (i.e. transfer function) and initial state.

Data scrambling is a methodology widely used in communications, to make data unreadable to the human eye. When transmitting, a random (unique) vector is XORed with the original data, thus scrambling it. At the receiving end, the data is descrambled using the same vector and the original information is retrieved. It is assumed that both parts know the unique vector which is usually generated from a random source.

Consider D as the original *plain data* and SD as the *scrambled data*. When transmitting, the scrambling process transforms D into a new value SD by applying an injective function f to D:

$$SD = f(D) \rightarrow D = f^{-1}(SD) \tag{3.1}$$

The inverse process is done at the receiver's end. The function used is generally a modulo-2 addition (i.e. bitwise XOR):

$$SD = D \oplus S \rightarrow D = SD \oplus S \tag{3.2}$$

Proposals and implementations of data scrambling methods and techniques include the same components as the additive or multiplicative scrambler. A US patent from Hsu et. al. [75] in 2002 defines a scrambling circuit to protect data in a read only memory. The implementation includes an initial value generator, a shift register, a logic circuit, an adder and a lock circuit. The general architecture is displayed in Figure 3.1. The *scrambled data* is obtained by adding the data stored in the read only memory with the data stored in the shift register. Thus, no de-scrambling circuit can be provided for this scrambling circuit. This is a basic and simple implementation of an additive scrambler, but has the downside of not being able to

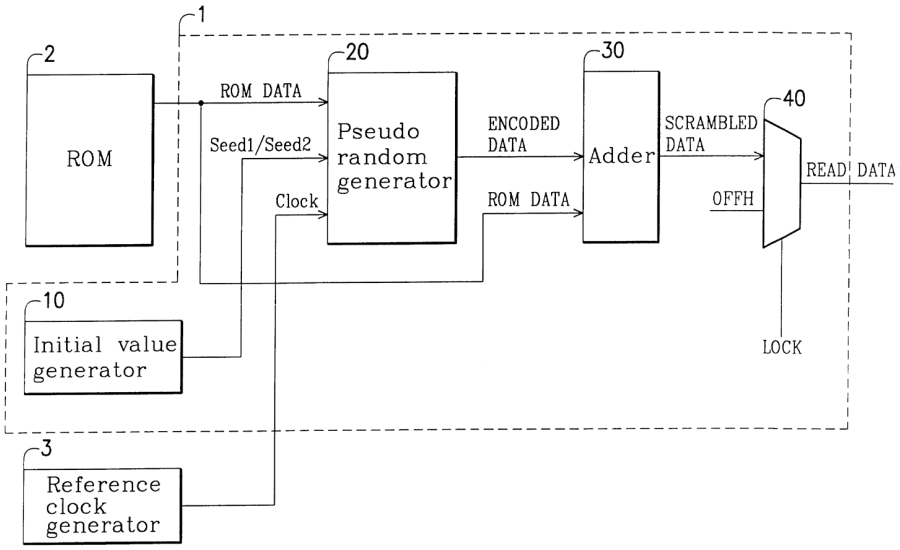


Figure 3.1: Design used in [75].

descramble the information. If the outputs of the pseudo-random generator would be stored in a LUT or something similar, descrambling would be possible.

The authors in [74] focus on protecting the information exchange on the data bus interface, by scrambling the information transmitted. They consider 2 types of scrambling: pure and conditional, both designs are visible in Figure 3.2. A pure scrambler consists of 2 main blocks: a random pattern generator (block randomizer, e.g. LFSR), that produces a new pattern r at each new bus cycle, and the block "$\oplus$", which performs modulo-2 addition (i.e. bitwise XOR) between D (*plain data*) and r. By keeping the sender's and receiver's clocks synchronized, and by using the same initial seed for the two LFSRs, correct decoding is guaranteed. Conditional scrambling is slightly different from pure scrambling because the randomization is controlled by an external signal, called FS (Force Scrambling). The DCM (Duty Cycle Modulator) block works as a switch driven by FS that, at each cycle, determines what operand r' must be supplied to the modulo-2 adder. Thus, if FS equals 1, the r' operand is r, else $r' = 0$.

To improve the energy savings, the authors in [74] consider using bus-invert pure and conditional scrambling. In Bus-Invert, the current word is compared with the previously transmitted one and if the Hamming distance between the two exceeds $N/2$ (for N bus lines), the complement of the word is sent on the bus. However, 2 extra bus lines are required: one signal for the occurrence of scrambling and another to signal the transmission of an inverted pattern. By combining the

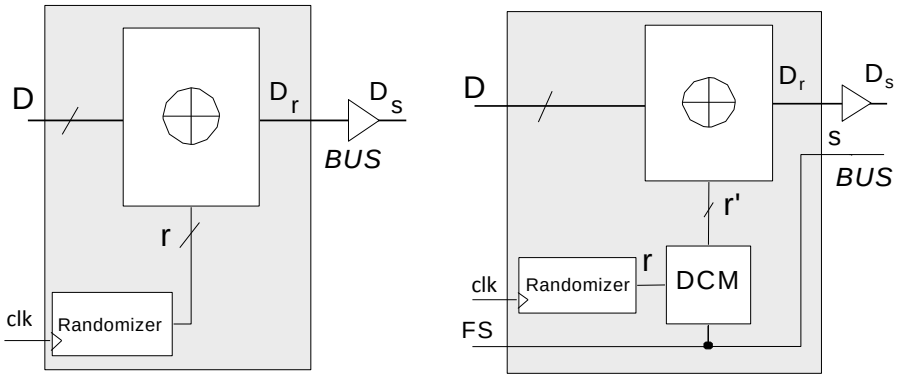


Figure 3.2: Pure (left) and conditional (right) scrambling [74].

two techniques, authors achieve a $\approx 7.8\%$ of energy saving. The proposals from [74] are effective for obscuring the data which is transmitted on the communication bus (e.g. CPU-cache memory bus). For a cache memory, the methods proposed can also be employed, with few minor changes regarding the descrambling part.

Very similar to [74], the authors in [76] propose the same pure and conditional bus scrambling combined with a coupling aware algorithm (they consider the effect of coupling capacitance). The results show an improvement of $\approx 10\%$ energy saving over generic scrambling. Because of the similarities between [74] and [76], the same comments apply.

The authors in [77] address the problem of designing keyed permutations of compact shape, that generate a large set of permutations when the key runs over the key space and offers good properties against chosen plain-text attacks in the context of physical probing. The functions proposed can be used for very fast on-chip data scramblers, which integrate keyed permutations. Most notable results are for a size of 32 bits: the design has only five levels of gates (depth), key size of 80 and 224 multiplexers needed. The proposal basically strengthens the encryption of the data against plain-text attacks (e.g. probing attacks), at a cost determined by the information size, depth, key size and number of multiplexers. It can be viewed as a more complex solution than simple additive scrambling, but simpler than the AES algorithm.

The authors from [78] propose a Memory Encryption Control Unit (MECU) that encrypts all the memory transfers between the Level 2 Cache and main memory, to ensure plain data is never written to the persistent medium. Basically, the additional unit is on the memory bus and all memory operations are mediated by this MECU. The encryption process is based on any encryption algorithm (e.g.

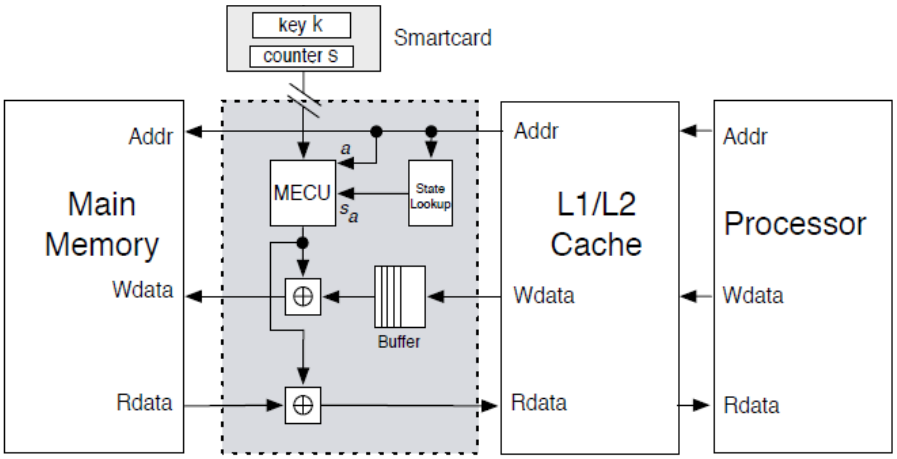


Figure 3.3: MECU architecture from [78].

DES, AES), but with reduced overhead, by using encryption blocks or pads (generated by the MECU) which are afterwards XORed with the *plain data.* Because the MECU itself can be attacked, some vital information of the encryption process (like master key, state counter) is stored in a removable device (i.e. smart-card), which is assumed to be removed on suspend (see Figure 3.3). The overall additional unit has an overhead of 9% for the worst case and less than 2% for average workloads. The proposal can be found useful for protecting the main memory against attacks because the unit basically resides on the memory bus. However, placing the unit on the memory bus between the CPU and L1/L2 cache is somehow problematic because the cache memory is actually on the same chip with the processor and timing delays must be investigated from several points of view, given the fast speed of a cache memory.

Another work that focuses on protecting the main DRAM memory against cold boot attacks is [22]. Here, the authors propose a solution to scramble the data before writing it to the DRAM. The scrambling is done using a XOR circuit, a Galois Field Multiplication of order 128 (GF128) and a Pseudo Random Number Generator (PRNG). The GF128 block will take the address to the DRAM and a session key as inputs to perform arithmetic operation. The outputs of this block will XOR with the *plain data* to form *scrambled data* when writing to the DRAM. The descramble operation is done by XORing the GF128 output with the *scrambled data*. The session key is given by the PRNG and is software invisible, kept in a hardware register until the next reset. All the design blocks are integrated into a single chip and act as an interface between the bus and DRAM controller. Thus,

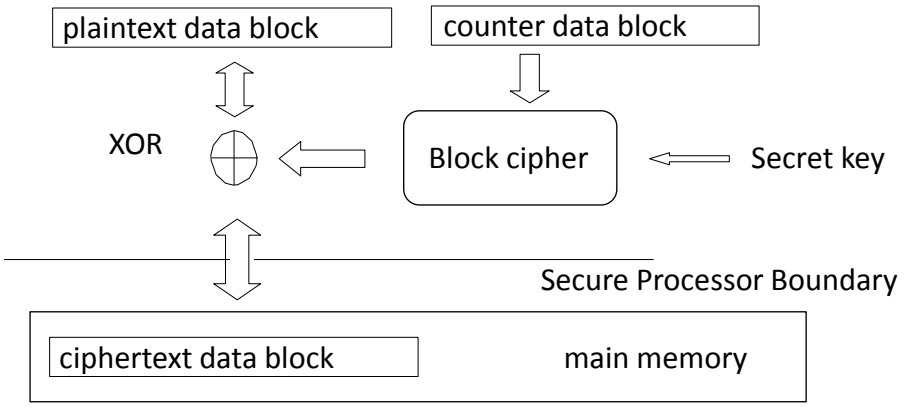


Figure 3.4: Counter-mode encryption used in [23].

any data sent to the DRAM is scrambled. As in [78], the design is good for securing the DRAM against attacks, but reduces the performance of the memory system (not sure if viable for the cache memory) and has an additional cost for implementing the scrambling and descrambling logics.

Phase change memory (PCM) is a new technology for computer memory systems, but has some problems with the lifetime and privacy protection. In this scope, the authors from [23] propose a new mechanism to improve PCM-based main memory systems. Because PRAM is non-volatile, there are privacy concerns over the contents residing in the main memory. Thus, the proposal in [23] uses the counter mode encryption (Figure 3.4), with the secret keys stored inside the processor, but with additional counters for each data block (each cache line already has a counter). This type of encryption technique is good for memories with limited lifetime (like PRAM) because it can improve the lifetime. This encryption can also be used in DRAMS because it needs constant refresh of the data stored, thus, reducing the power for refreshing the information. However, for cache memories, the encryption method using counters is somehow expensive and needs a lot of additional circuitry.

3.4 Statement of the problem

As mentioned in 3.1, as the data in cache is stored in plain text any successful attack could reveal sensitive information. Data scrambling can solve the dissemination requirement of the data stored in the cache at a reasonable performance cost because it uses only a layer of XOR gates to combine the scrambling vector with *plain data*

and thus the performance penalty is limited. Note that cache operates at very high speeds and any reduction in the bus throughput has an immediate effect in the CPU performance. To guarantee a reasonable level of security, the scrambling technique needs to consider the following milestones.

1. **Regular refresh of the scrambling vector**. In order to dodge side channel attacks, the principles for dissemination must be altered regularly. The *scrambling vector* which is used in the data scrambling operation needs to be refreshed periodically. Thus, any possible leaked information retrieved by side-channel attacks will vary from time to time, making it very difficult to catch the true pattern.

2. **Generate scrambling vectors with a high quality random source.** The periodical refresh of the scrambling vectors implies the use of random generators. True random number generators make the modeling of the source very difficult and thus the prediction of the dissemination rules harder. Pseudo-random number generators are classified as fast circuits, while the former are considered slow. By mixing the two types of generators, a respectable speed can be achieved, combined to a high degree of randomness.

3. **Softly refresh of scrambling vectors, producing minimum disturbances in time and power.** Memories containing *scrambled data* have the problem that any change in the *scrambling vector* corrupts the stored data and thus updating strategies are needed. It is vital to avoid long elapsed periods of time or large data transfers, because it could be detected by power or time measurements and, thus, raise the attention of the attacker. This milestone is the most difficult one because it involves the reprocessing of large amounts of data in very short periods of time.

The proposed technique has the objective of securing the cache memory with data scrambling schemes, considering the previous milestones and giving special attention to the third one.

3.5 Proposed solution: Interleaved Scrambling Technique (IST)

For memory systems, data scrambling is crucial when trying to secure and obscure the stored information. In the special case of the cache memory, the speed when generating *scrambled data* is vital, and thus, fast scrambling methods are advisable.

Unidirectional scrambling is not a solution because the protected information must be descrambled when it's needed by the CPU.

Given the previous sections, the data from the cache can be scrambled by using a simple pure scrambler. If conditional scrambling is used (to reduce the power consumption), a more complex logic design is needed because the critical data must be isolated, thus scrambling only the important information. For the rest of this section the proposed solution is presented as for scrambling the entire cache memory. However, the same technique can be enhanced with conditional scrambling.

A global overview of the IST (Interleaved Scrambling Technique) proposed is presented in Figures 3.7 and 3.8, later these two figures will be discussed in more detail. Besides the CPU and cache memory the new module containing the core of the IST is the scrambler table (ST). In Figure 3.5 the main content of ST is shown.

3.5.1 Scrambler Table

The central part of the ST is the **ST internal table**. It contains k rows of five fields each. From left to right: sw is the *youth flag*, C^0 is the *age counter* zero, S^0 is the *scrambling vector* zero, C^1 is the *age counter* one, and S^1 is the *scrambling vector* one. The idea is that when the CPU writes data to the cache they are scrambled using one of the two *scrambling vectors*, S^0 or S^1, depending on the *youth flag* (sw) state. During CPU reading, both S^0 or S^1 can be used for descrambling, it will depend of the information stored in the cache memory. Flag sw always point to the young *scrambling vector* (S^{sw}) which is used for writing while $S^{\overline{sw}}$ is the old *scrambling vector* only used for descrambling data read by CPU from the cache memory. In the same row j the young and old roles of the *scrambling vectors* are always paired.

Age counters (C^0, C^1) account for the number of times the corresponding *scrambling vector* has been used for scrambling data and for the persistence of these data in the cache memory. Thus, if new *scrambled data* is stored using S^{sw}, *age counter* C^{sw} is increased by one. Furthermore, if existing data in cache that was scrambled with $S^{\overline{sw}}$ is overwritten with *scrambled data* using a different *scrambling vector* then $C^{\overline{sw}}$ is decreased by one. Consider NB_{CM} be the number of data blocks in the cache memory that contain *scrambled data*, the following equality will always be satisfied,

$$NB_{CM} = \sum_{i=0}^{k-1} (C_i^0 + C_i^1) \tag{3.3}$$

While this is always true, in the IST the values of the individual *age counters* are

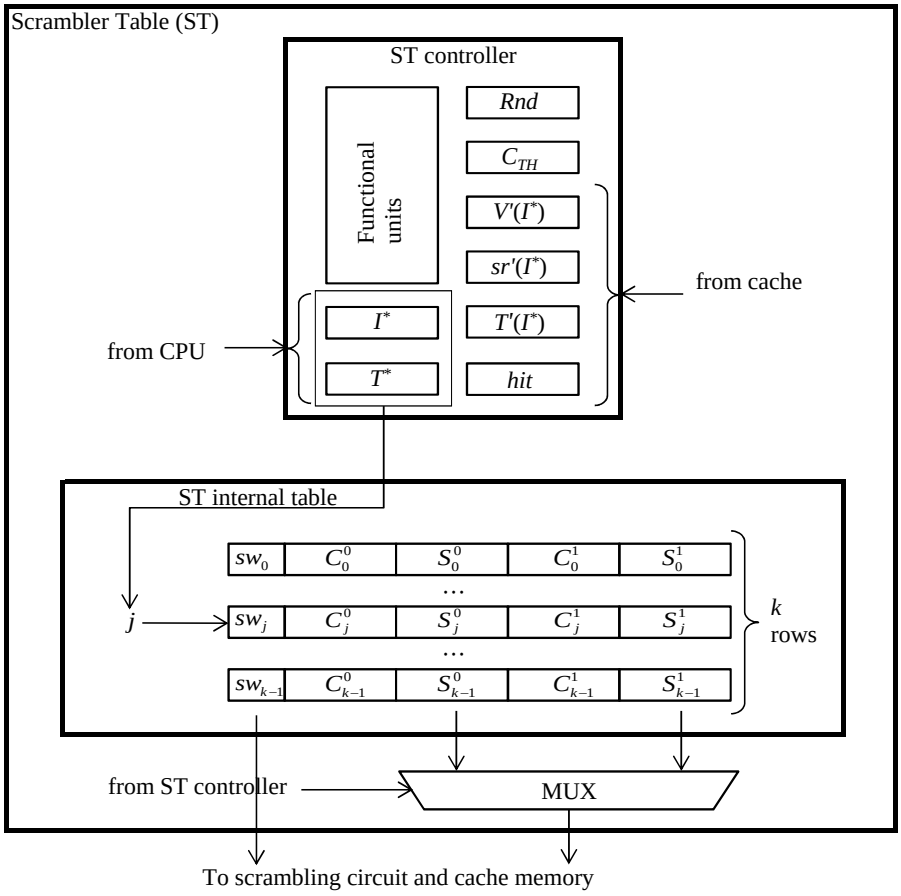


Figure 3.5: The scrambler table block and entries structure.

dynamically changed such that it becomes difficult to know at a given time instant how many data blocks are using a particular *scrambling vector*.

The **ST controller** is the managing unit and contains several key registers that are now explained. I^* and T^* are the index and tag fields obtained from the address generated by the CPU during the write operation. From these two fields the ST internal table index j is generated according to either schemes: the Set Address Unsynchronized (SAU) or the Set Address Synchronized (SAS). In the SAU scheme both I^* and T^* are considered while generating j as follows,

$$j \equiv I^* \oplus T^* \pmod{k} \tag{3.4}$$

which implies that the selection of the *scrambling vector* is a function of these two address fields and the *youth flag*. In the SAS scheme only the I^* is taken to generate the ST internal table index j as follows,

$$j \equiv I^* \pmod{k} \tag{3.5}$$

While the scheme SAS is simpler to work with and as discussed later can generate smaller STs, the SAU scheme will generate more unpredictable behaviors in the selection of the *scrambling vectors*.

Inside the ST controller, register Rnd contains the last random value generated by a TRNG module that is included in the **Functional units** block. The content of this register is refreshed on demand of the ST controller and is triggered each time a new *scrambling vector* is refreshed. Each time an old *age counter* ($C^{\overline{sw}}$) of an old *scrambling vector* ($S^{\overline{sw}}$) is decremented down to 0, the content of the *scrambling vector* is refreshed ($S^{\overline{sw}} \leftarrow Rnd$) and the TRNG is triggered again.

Register C_{TH} is the *age counter threshold* and is used for the management of the *youth flag* (sw). As said before, sw always points to the young *scrambling vector* (S^{sw}), which is always used for scrambling during CPU writing to the cache operations. Consequently, *age counter* (C^{sw}) will tend to increase as different writings succeed. After a number of operations the roles of young and old *scrambling vector* are changed by modifying the state of sw such that $S^{\overline{sw}} \leftarrow S^{sw}$ is not used for writing anymore. Inverting the *youth flag* state occurs if the two following conditions are satisfied,

$$(C_j^{sw} \geq C_{TH}) \wedge (C_j^{\overline{sw}} = 0) \Rightarrow \begin{cases} \text{first.} & S_j^{\overline{sw}} \leftarrow Rnd \\ \text{second.} & sw_j \leftarrow \overline{sw}_j \end{cases} \tag{3.6}$$

The young *age counter* while increasing must equal or surpass the threshold and the old *age counter* while decreasing must reach 0. Once at this point the old *scrambling vector* is refreshed with the content of Rnd register and the *youth flag* state is flipped.

Next flags and registers in the ST controller are necessary for the management of the *age counters*. Flag $V'(I^*)$ has the valid bit corresponding to the data block pointed by I^* for writing, see Figure 3.6. According to the cache controller policy one of the data blocks of line I^* will be selected for writing and therefore is necessary to know if the previous *scrambled data* there was valid. If it wasn't then neither *age counter* is decremented. Furthermore for the same reason, after reset with all data blocks in cache memory invalid, *age counters* will only be incremented. Flag $sr'(I^*)$ is the *youth flag* stored previously in the data block pointed by I^* and at which the writing must proceed, see Figure 3.6. Register $T'(I^*)$ contains the tag of the previous data block stored in the cache memory and pointed by I^* and selected by the cache controller for writing. The flag hit is set if the tags T^* and T' are equal with independence of the *scrambled data* content. This last flag is generated even during writing, since it is necessary for the management of the *age counters*.

As explained so far, after writing there are always two *age counters* that must be updated, one for increment and another for decrement. The rules are specified in the following state change formulas,

$$\begin{aligned} C_j^{sw} &\leftarrow \begin{cases} \overline{hit} \vee (sw \neq sr') \vee \overline{V'}, & C_j^{sw} + 1 \\ \text{else}, & C_j^{sw} \end{cases} \\ C_{j'}^{sr'} &\leftarrow \begin{cases} (\overline{hit} \vee (sw \neq sr')) \wedge V', & C_{j'}^{sr'} - 1 \\ \text{else}, & C_{j'}^{sr'} \end{cases} \end{aligned} \tag{3.7}$$

in which j was defined in Equation 3.4 and $j' \equiv I^* \oplus T' \pmod{k}$ for the SAU scheme or $j' \equiv I^* \pmod{k}$ for the SAS scheme. The young *age counter* is incremented only if the block of data to be written is not valid or if the previous *scrambling vector* was not the same. Otherwise it remains constant. The *age counter* to be decreased, commonly old but in the SAU scheme it can be a young *age counter* too, is decreased if the block of data to be written is valid and the previous *scrambling vector* was not the same, otherwise the *age counter* is kept the constant.

3.5.1.1 ST table resource requirements

The resources in terms of area of the ST is directly related to the ratio of *scrambling vector* refresh expected. Imagine the two opposite situations:

1. ST has only one row, $k = 1$. Thus, a pair of *scrambling vectors* interleave as young and old in order to periodically refresh their value. In this situation the old *scrambling vector* will be able to refresh its content and to become

young again when the cache memory would be fully written back. Therefore, refresh is expected to happen less often but the requirements of area will be minimum.

2. ST has as many rows as number of data blocks in the cache memory, $k = NB_{CM}$. Thus each data block has associated a pair of *scrambling vectors* which interleave to allow refreshment. Therefore, at any write a new *scrambling vector* will be refreshed. In this case the refreshment rate is the highest and the needs of area will be the maximum.

It is seen then that adding more or less rows in ST becomes a trade off between the frequency of *scrambling vector* refreshment and the needs of area. The higher the refreshment rate is wished the larger area for the ST table is needed. Let's assume F_w to be the average writing frequency in cache memory. The average refreshment rate of *scrambling vectors* is,

$$u = \frac{F_S}{F_w} = \frac{k}{NB_{CM}} \tag{3.8}$$

where F_S if the *scrambling vector* average refreshment frequency.

The area requirements can be estimated at a first approximation as the number of bits stored in the ST internal table. One row of the ST table has the following number of bits: *scrambling vectors* have the same length as the data blocks n, and the number of bits of the *age counters* is m. Thus the total number of bits of a ST internal table row is $(2n + 2m + 1)$ including the *youth flag*. Parameter m depends on the scheme selected, SAS or SAU, as discussed previously in Equations 3.4 and 3.5.

In the SAU scheme any *age counter* might accumulate a value equal to the number of data blocks in the cache memory plus one, despite this maximum has very low probability. Therefore, $m_{sau} = \lceil \log_2(NB_{CM} + 1) \rceil$. In the SAS scheme the maximum value expected in any *age counter* is the number of data blocks in the cache memory divided by the number of ways (W) and the number of rows k of the ST internal table plus one. Hence, $m_{sas} = \lceil \log_2(NB_{CM}/(W \cdot k) + 1) \rceil$.

The area occupied by the cache memory A_{CM} will be proportional to the number of bits stored, then $A_{CM} = NB_{CM} \cdot n$. The area occupied by the ST internal table it will depend on k and the scheme selected. Thus, the occupancy ratio of the ST table is,

$$\begin{aligned} a_{sau} &= \frac{A_{ST(sau)}}{A_{CM}} = \frac{(2n + 2\lceil \log_2(NB_{CM} + 1) \rceil + 1) \cdot k}{NB_{CM} \cdot n} \\ a_{sas} &= \frac{A_{ST(sas)}}{A_{CM}} = \frac{(2n + 2\lceil \log_2(NB_{CM}/(W \cdot k) + 1) \rceil + 1) \cdot k}{NB_{CM} \cdot n} \end{aligned} \tag{3.9}$$

where A_{ST} is the total number of bits stored in the ST internal table.

As a good trade off between refresh ratio and occupancy ratio the following design criteria is proposed: $k_{sau} = \sqrt{NB_{CM}}$ and $k_{sas} = \sqrt{NB_{CM}/W}$. As a result, Equations 3.8 and 3.9 become,

$$\begin{aligned}
&\text{(SAU)} \begin{cases} u_{sau} = 1/\sqrt{NB_{CM}} \\ a_{sau} = (2n + 2\lceil \log_2(NB_{CM} + 1) \rceil + 1)/(\sqrt{NB_{CM}} \cdot n) \end{cases} \\
&\text{(SAS)} \begin{cases} u_{sas} = 1/\sqrt{NB_{CM} \cdot W} \\ a_{sas} = (2n + 2\lceil \log_2(\sqrt{NB_{CM}/W} + 1) \rceil + 1)/(\sqrt{NB_{CM} \cdot W} \cdot n) \end{cases}
\end{aligned} \tag{3.10}$$

In Table 3.1 these equations are evaluated for an architecture of 32 bits. Sizes for the cache memory are varied from 1K to 400K data blocks. SAS schemes are assessed for cache memories of 1, 2 and 4 ways. The maximum values are found for the smaller cache memory (1K data block) with a refresh rate of 3.162 meaning that on average one *scrambling vector* is updated after each 32 writes. Also in this case, the area of the ST table represents an 8.4 % of the cache memory area.

Table 3.1: Evaluation of Equations 3.10 for $n = 32$.

			$W = 1$		$W = 2$		$W = 4$	
$\frac{NB_{CM}}{\text{K}}$	$\frac{u_{sau}}{\%}$	$\frac{a_{sau}}{\%}$	$\frac{u_{sas}}{\%}$	$\frac{a_{sas}}{\%}$	$\frac{u_{sau}}{\%}$	$\frac{a_{sas}}{\%}$	$\frac{u_{sau}}{\%}$	$\frac{a_{sas}}{\%}$
1	3.162	8.400	3.162	7.412	2.236	5.171	1.581	3.607
2	2.236	6.079	2.236	5.311	1.581	3.706	1.118	2.585
4	1.581	4.398	1.581	3.805	1.118	2.655	0.791	1.853
7	1.195	3.399	1.195	2.913	0.845	2.034	0.598	1.419
10	1.000	2.906	1.000	2.469	0.707	1.724	0.500	1.203
20	0.707	2.099	0.707	1.768	0.500	1.234	0.354	0.862
40	0.500	1.516	0.500	1.266	0.354	0.884	0.250	0.617
70	0.378	1.169	0.378	0.969	0.267	0.677	0.189	0.472
100	0.316	0.978	0.316	0.810	0.224	0.566	0.158	0.395
200	0.224	0.706	0.224	0.580	0.158	0.405	0.112	0.283
400	0.158	0.509	0.158	0.415	0.112	0.290	0.079	0.203

Cache memory internal structure needs to include an additional flag to make it compatible with the IST technique. Next the main parts are described.

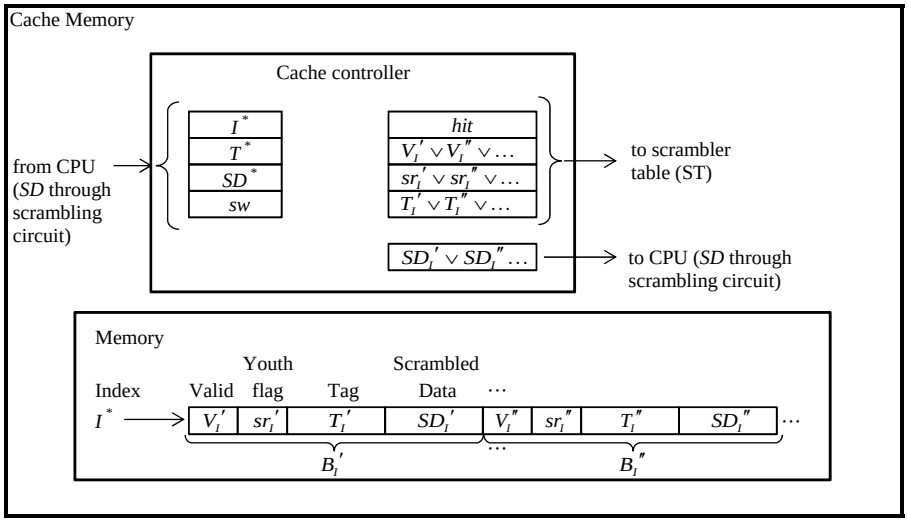


Figure 3.6: Cache memory entries structure and main functional blocks.

3.5.2 Cache Memory

In Figure 3.6 the block diagram is presented. As formerly indicated, the size of the cache memory is NB_{CM} blocks of data. Considering an organization in ways, the number of rows pointed by index I is NB_{CM}/W. At the same time each block of data can be subdivided in words pointed by the less significant bits of the address, but this internal sub-structure is not commented here because is not relevant to the IST.

One block of data has the following internal structure $B = \{V, sr, T, SD\}$ in which V is the valid flag, sr is the *youth flag* of the *scrambling vector* used to scramble SD, T is the tag field and SD is the *scrambled data*. During the writing operation the CPU generates an index I^* which points to blocks B'_I, B''_I and so on. The selection of the $way = \{',''',''', ...\}$ depends on the internal cache controller policy. It is relevant to the ST because some information of the previous data block must be feedback in order to manage the *age counters*. Henceforth, for easing the explanation way is assumed $\{'\}$ unless otherwise specified. During the reading operation the transaction is simpler because the data block is selected according to the matching of the tag field and the *age counters* are not modified.

The **Cache controller** receives index I^* and tag T^* from the CPU. If the operation is for writing, it also receives the *scrambled data* SD^* and the *youth flag* sw from the scrambling circuit. In case of reading operation these two last data are not required.

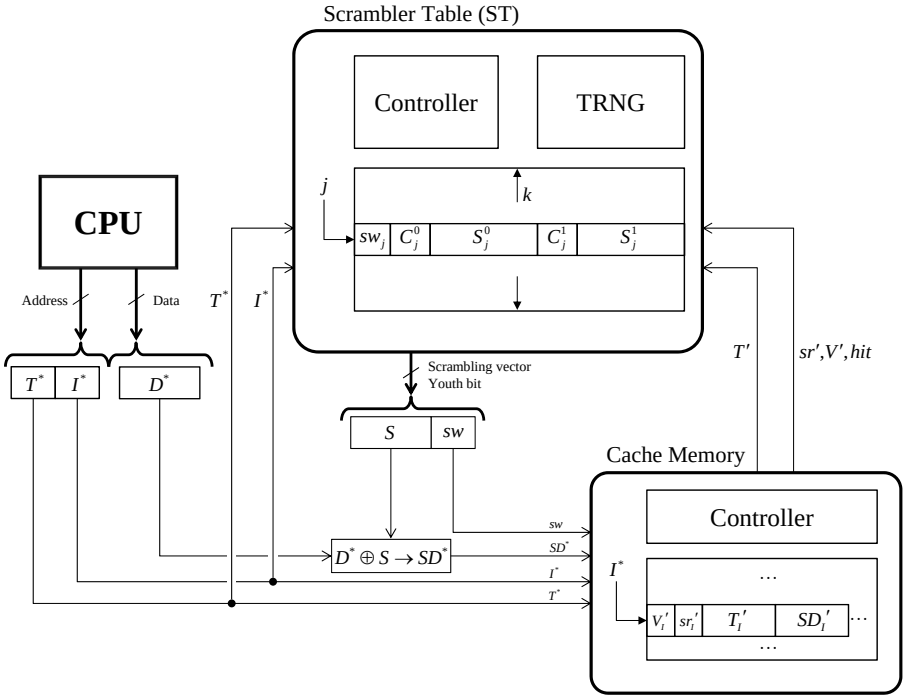


Figure 3.7: IST write cycle.

The following data is sent to ST corresponding to the data block selected during the writing operation. The flag hit, the valid flag V', the *youth flag* sr' and the tag field T', all of them necessary for the *age counters* management as specified in Equation 3.7. During reading, the hit flag is sent to the CPU, the sr' flag and T' field are sent to the ST in order to make the correct selection of the *scrambling vector*, and in its turn the *scrambled data* SD' is sent to the *scrambling circuit* for descrambling and to recover the original *plain data* required by the CPU.

3.5.3 Read and write cycles

In Figure 3.7 the elements involve in the write operation are presented. It begins with the CPU generating the memory address and data, $\{T^*, I^*, D^*\}$. Address $\{T^*, I^*\}$ is sent to the ST and from it the *youth flag* (sw) and the young scrambling vector (S) are retrieved. *Plain data* vector (D^*) is XORed with the scrambling vector $SD^* \leftarrow D^* \oplus S$ and is sent to the cache memory together with the address and the *youth flag* $\{T^*, I^*, SD, sw\}$. At the same time, the cache controller sends previous data flags $\{T', sr', V', hit\}$ to ST to allow the correct management of the

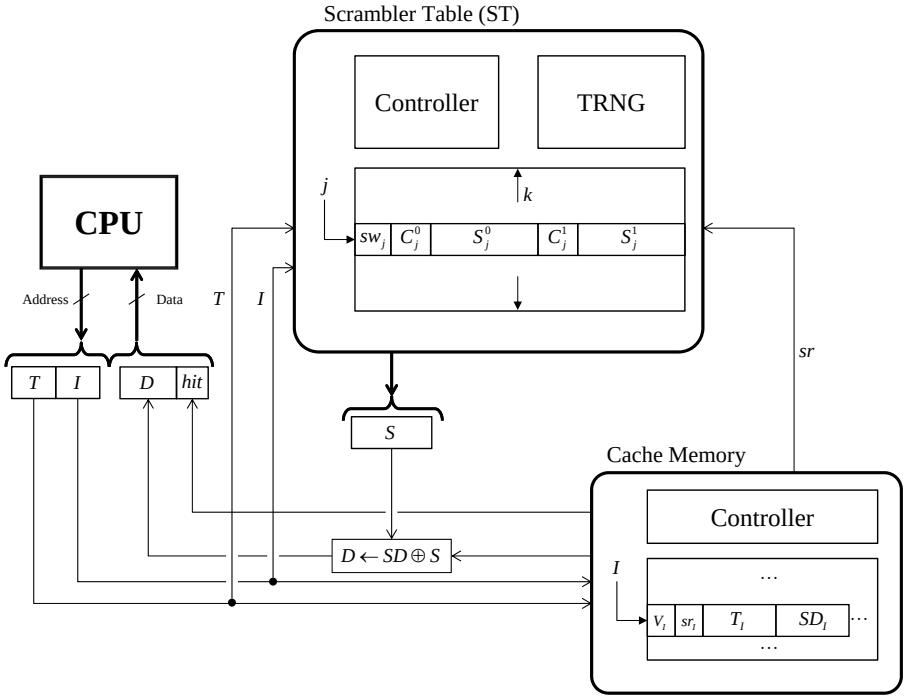


Figure 3.8: IST read cycle.

age counters. The cycle ends with the write of SD^* at the data block selected by the cache controller.

In Figure 3.8 the elements involved in the read operation are presented. It begins with the CPU generating only the memory address $\{T, I\}$ and checking if the hit flag is active. If it is, the following actions are done: the *youth flag* (sr) is read from cache and is sent to the ST and from it the scrambling vector S is retrieved. Once *scrambled data SD* is available from cache it is descrambled, $D \leftarrow SD \oplus S$, and the *plain data* is transferred to the CPU.

In Figures 3.9 and 3.10 two illustrative examples of the IST operation are presented. The dimensions of the cache memory and ST table here are limited to demonstrate the management of the scrambling vectors in detail. The model for the cache memory is simple, 64 data blocks one way. The ST table has four pairs of *scrambling vectors* with the corresponding *age counters* and $C_{TH} = 16$. In both examples 3000 writing operations are simulated following different strategies for the generation of the address.

In Figure 3.9 the CPU generates random address. The lines plotted show the accumulated values of the type zero, $\sum C^0$ (dotted line) and type one, $\sum C^1$ (con-

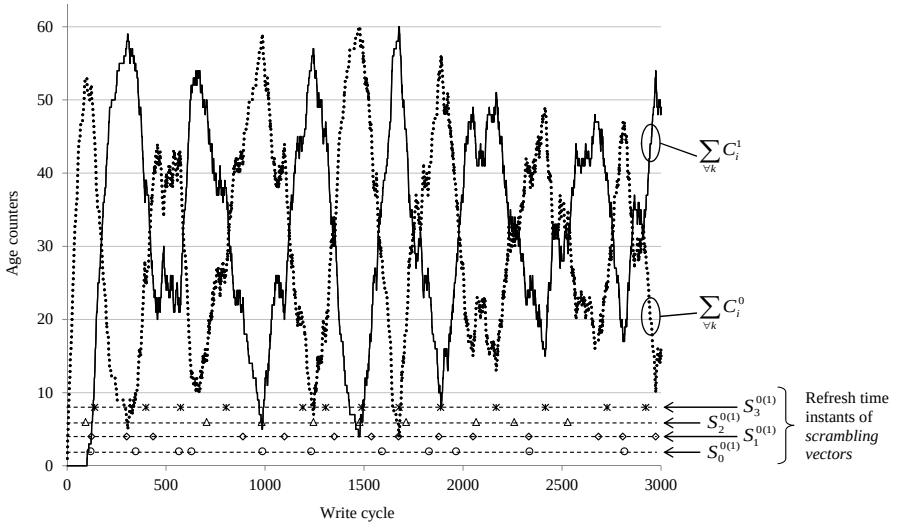


Figure 3.9: Simulation of the ST table management of the *scrambling vectors*. The scheme is SAU, the number of data blocks is 64 and the number of lines in the ST internal table is 4. CPU addresses are generated randomly.

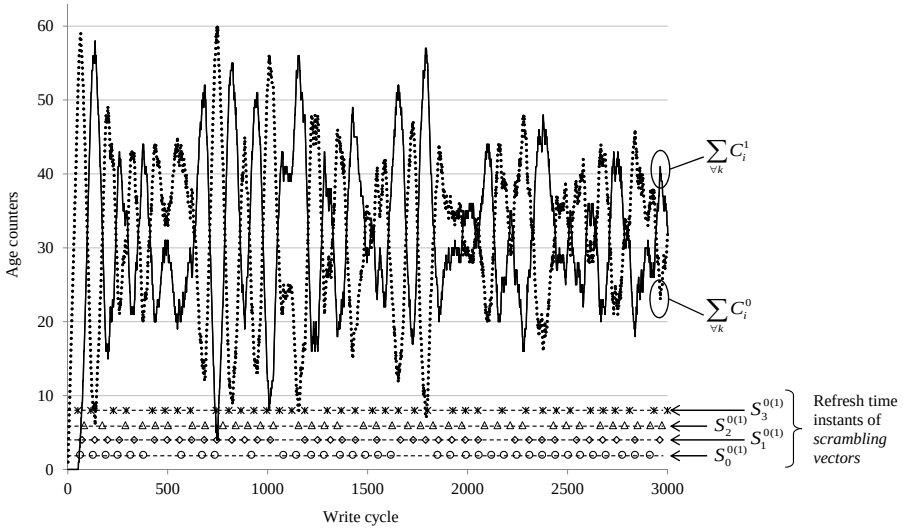


Figure 3.10: Simulation of the ST table management of the *scrambling vectors*. The scheme is SAU, the number of data blocks is 64 and the number of lines in the ST internal table is 4. CPU addresses are generated incrementally with a random increment between 0 and 3.

tinuous line) *age counters* separately. This information indicates after each write how many *scrambled data* vectors are in the cache memory using type zero and one *scrambling vectors* respectively. At any time the sum of these two lines must be equal to 64. For the same reason their average is 32.

Some conclusions can be drawn. It is seen that the amount of *scrambling vectors* of each type tilt over time, following an irregular profile. Never the maximums/minimums reach exactly the same value and neither the fluctuations last the same number of write operations. At the bottom of the Figure the time instants at which each *scrambling vector* is refreshed is indicated, it is marked by the dots. There are four lines for each pair of scrambling vectors. Looking at the time intervals of these refreshes they are irregular and different in each line, that makes the prediction of them difficult from the point of view of an adversary.

In Figure 3.10 the same information is presented but now the addresses are generated with a different profile. CPU generates index incrementally by one at each write operation, and the tag is incremented using random increments between 0 and 3. The results observed demonstrated that the refresh rate is increased because now there are less write operations hitting the same tag. As before, the distribution of type zero and one *scrambling vectors* tilts and is irregular. The time instants of the refreshes are more frequent but again the distribution is irregular, as expected.

3.6 IST performance and efficiency

In this subsection the improvement of performance achieved by IST is estimated. Both, the time performance and the power reduction are considered. IST is compared to non-interleaved scrambling technique, i.e. after writing NB_{CM}/k times into the cache memory using the same *scrambling vector* it is refreshed using the TRNG and the content of the cache is updated completely for the NB_{CM} data blocks by reading and scrambling data from the main memory.

3.6.1 Time performance

Assume that the average time for writing to the cache is t_w and for reading t_r. If the miss ratio is η then the cache reading performance is,

$$\rho = \frac{(1-\eta)t_r}{(1-\eta)t_r + \eta t_w} \tag{3.11}$$

For example for a standard cache with $t_w = 20t_r$ and $\eta = 0.005$ we would have a performance of $\rho = 0.909$. In the IST, times for reading and writing will be some larger amount but the performance will follow the same Equation 3.11 because there aren't special interruptions during reading and writing cycles.

If the standard scrambling technique is applied, after NB_{CM}/k writings the *scrambling vector* is changed and thus additional NB_{CM} writes will be necessary to refresh the content of the cache. As k is increased, $k = 1, 2, 3, ...$ the refresh rate increases similarly to the IST with the same k. Considering this and the previous definitions the reading performance is,

$$\begin{aligned} \rho^* &= \frac{(NB_{CM}/k)(1-\eta)t_r}{(NB_{CM}/k)(1-\eta)t_r + (NB_{CM}/k + NB_{CM})\eta t_w} \\ &= \frac{(1-\eta)t_r}{(1-\eta)t_r + \eta(1+k)t_w} \end{aligned} \tag{3.12}$$

that taking the same data of the IST case it gives $\rho^* = 0.833, 0.768, 0.666, 0.525, 0.369$ for $k = 1, 2, 4, 8, 16$, which are significantly lower performance than the IST.

In Figure 3.11 Equations 3.11 and 3.12 are plotted for reading to writing time ratios between 10 to 39 and for values of $k = 1, 2, 4, 8, 16$. As expected, in the standard scrambling technique, as the refresh frequency increases the reading performance becomes worse.

3.6.2 Power efficiency

Let's assume that p_w is the average power consumption during the writing cycle of the cache and p_r is the average reading power. Then, for a standard cache memory the power efficiency, i.e. the proportion of energy which is already used for reading is,

$$\varepsilon = \frac{(1-\eta)p_r t_r}{(1-\eta)p_r t_r + \eta p_w t_w} \tag{3.13}$$

During the writing cycle the amount of power needed is much higher than during reading cycle because external lines of the chip are switched. As a first approximation we can assume that the same proportion is like in time, i.e. $p_w = 20p_r$. Taking the rest of values from the previous estimation we obtain an efficiency of $\varepsilon = 0.332$. Like before, in the IST technique the model for the efficiency is the same, Equation 3.13, except that the values for the average power will be slightly higher because of the data scrambling and the ST management, but we can expect the same ratio between reading and writing power.

In a standard scrambling technique, during the data update phase also an extra power p_w is consumed for a time of $NB_{CM}t_w$. Thus the energy efficiency is,

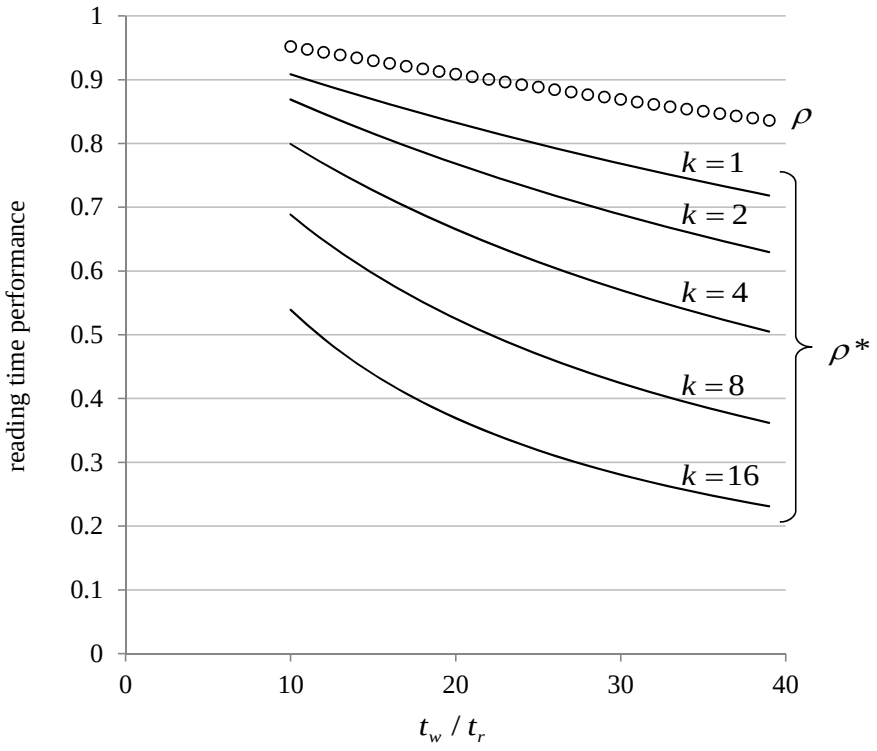


Figure 3.11: Representation of Equations 3.11 and 3.12 assuming a miss ratio $\eta = 0.005$.

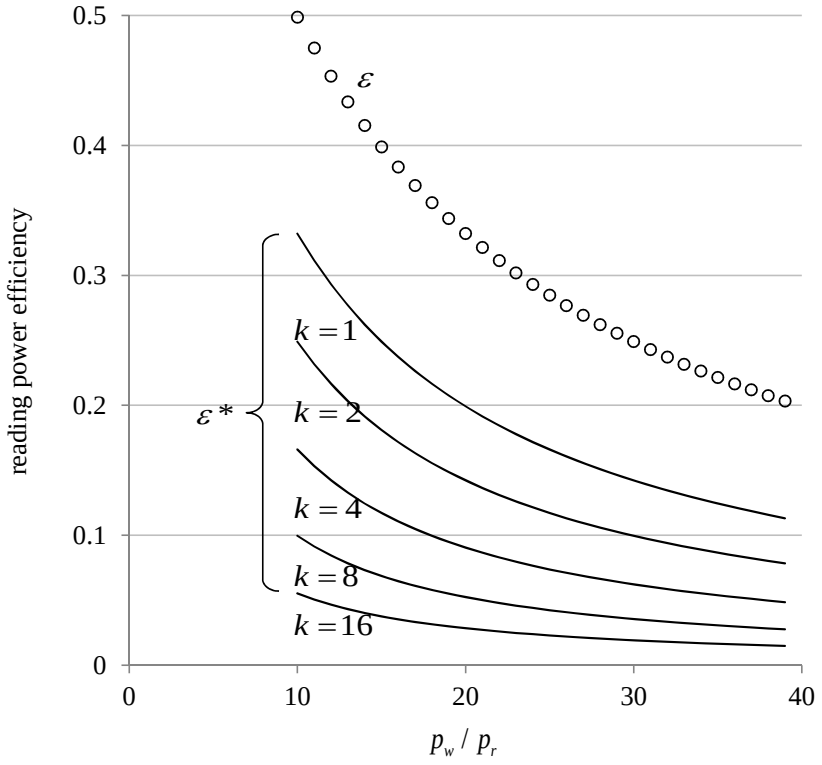


Figure 3.12: Representation of Equations 3.13 and 3.14 assuming a miss ratio $\eta = 0.005$ and $t_w = 20t_r$.

$$\begin{aligned} \varepsilon^* &= \frac{(NB_{CM}/k)(1-\eta)p_r t_r}{(NB_{CM}/k)(1-\eta)p_r t_r + (NB_{CM}/k + NB_{CM})\eta p_w t_w} \\ &= \frac{(1-\eta)p_r t_r}{(1-\eta)p_r t_r + \eta(1+k)p_w t_w} \end{aligned} \tag{3.14}$$

Again, substituting with the numbers of the estimation we obtain an efficiencies of $\varepsilon^* = 0.199,\ 0.142,\ 0.090,\ 0.052,\ 0.028$ for $k = 1, 2, 4, 8, 16$. Like in the evaluation of the time performance, the power efficiency is significantly lower in the standard scrambling technique. This is reason why the IST is considered a low power scrambling technique.

In Figure 3.12 the Equations 3.13 and 3.14 are plotted for $\eta = 0.005$ and $t_w = 20t_r$. Power ratio is varied from 10 to 39 and refresh coefficient in the standard scrambling technique are $k = 1, 2, 4, 8, 16$. For the IST the maximum and

minimum efficiencies are 0.499 and 0.203 respectively. For the standard scrambling technique the maximum efficiency is achieved in the case of $k = 1$ with a value of 0.332 and the minimum is found for $k = 16$ with a value of 0.015.

In the next section the performance degradation introduced by the IST to a standard cache design is shown.

3.7 Evaluation and experimental results

In order to evaluate the proposed technique a comparison between standard L2 cache and IST is made. Area overhead, power consumption and performance are analyzed. CACTI tool [79] is selected for the estimation of the area and power for different size architectures while a Xilinx FPGA implementation is used for the performance evaluation in a 16 KB L2 cache memory.

3.7.1 CACTI tool evaluation

CACTI is a tool from HP that evaluates alternative implementations of caches for different technologies. It includes technology data from different vendors. For this work a 45nm technology has been selected. Different sizes of cache memories have been generated and compared for access times, area occupied and power consumption. The same simulation tool is used to create the scrambler table because the ST is mostly a memory itself. Thus, we can obtain a good approximation of the area occupied and power consumption, in comparison to an actual implementation.

Equation 3.10 dictates the size of the scrambler table, based on the L2 cache size. The results obtained from the CACTI tool are illustrated below, in Tables 3.2, 3.3 and 3.4. We consider cache sizes from 16 to 2048 KB, a line size of 16 bytes, one bank and 1-way, 2-way and 4-way set associative.

It can be observed that as the size of the cache increases, both the area overhead and the power consumption increase is less significant, and the access times follows a similar trend. Area overhead is below 5% for caches larger than 128 KB, while power consumption and access time overhead is below 28% for sizes larger than 128 KB. The access times increase reaches almost 37% for a cache size of 64 KB (for a 1-way cache), but the values decrease down to $\approx$12% when the size of the cache reaches 2048 KB (for 2-way and 4-way caches). CACTI tool wasn't able to simulate very small cache sizes (usually below 0.5 KB), hence the tables can contain some "N/A" (i.e. not available) values.

Thereupon the performance evaluation is considered for the FPGA model.

Table 3.2: CACTI results for different cache sizes, 1-way set associative.

	Size (KB)		Area (mm^2)		Power consumption(mW)		Access time (ns)	
L2 cache	16		0.1479		94.55		0.3652	
IST - SAU	0.34	2.08%	N/A	N/A	N/A	N/A	N/A	N/A
IST - SAS	0.31	1.90%	N/A	N/A	N/A	N/A	N/A	N/A
L2+SAU	16.34	2.08%	0.1494	N/A	94.99	N/A	0.3662	N/A
L2+SAS	16.31	1.90%	0.1494	N/A	94.99	N/A	0.3662	N/A
L2 cache	32		0.2395		97.28		0.402	
IST - SAU	0.5	1.54%	N/A	N/A	N/A	N/A	N/A	N/A
IST - SAS	0.44	1.36%	N/A	N/A	N/A	N/A	N/A	N/A
L2+SAU	32.5	1.54%	0.2419	N/A	97.89	N/A	0.403	N/A
L2+SAS	32.44	1.36%	0.2419	N/A	97.89	N/A	0.403	N/A
L2 cache	64		0.5493		159.38		0.5015	
IST - SAU	0.72	1.11%	0.0243	4.24%	78.32	32.95%	0.2938	36.94%
IST - SAS	0.62	0.96%	0.0237	4.14%	78.53	33.01%	0.2895	36.60%
L2+SAU	64.72	1.11%	0.5549	4.24%	160.51	32.95%	0.5033	36.94%
L2+SAS	64.62	0.96%	0.5549	4.14%	160.51	33.01%	0.5033	36.60%
L2 cache	128		1.1452		280.19		0.6239	
IST - SAU	1.04	0.81%	0.0318	2.70%	81.54	22.54%	0.2845	31.32%
IST - SAS	0.88	0.68%	0.0253	2.16%	78.07	21.79%	0.3005	32.51%
L2+SAU	129.04	0.81%	1.1565	2.70%	281.94	22.54%	0.6257	31.32%
L2+SAS	128.88	0.68%	1.1452	2.16%	280.19	21.79%	0.6239	32.51%
L2 cache	256		1.5605		385.37		0.7614	
IST - SAU	1.49	0.58%	0.0346	2.17%	81.07	17.38%	0.2952	27.94%
IST - SAS	1.27	0.49%	0.033	2.07%	81.24	17.41%	0.2895	27.55%
L2+SAU	257.49	0.58%	1.5704	2.17%	387.17	17.38%	0.7634	27.94%
L2+SAS	257.27	0.49%	1.5704	2.07%	387.17	17.41%	0.7634	27.55%
L2 cache	512		3.1233		643.7		0.9832	
IST - SAU	2.16	0.42%	0.0377	1.19%	76.6	10.63%	0.3024	23.52%
IST - SAS	1.8	0.35%	0.0366	1.16%	80.95	11.17%	0.3018	23.49%
L2+SAU	514.16	0.42%	3.1346	1.19%	645.34	10.63%	0.9843	23.52%
L2+SAS	513.8	0.35%	3.1319	1.16%	644.98	11.17%	0.9843	23.49%
L2 cache	1024		5.6764		1034.14		1.3061	
IST - SAU	3.1	0.30%	0.042	0.73%	78.11	7.02%	0.3135	19.36%
IST - SAS	2.6	0.25%	0.0397	0.69%	77.32	6.96%	0.3083	19.10%
L2+SAU	1027.1	0.30%	5.6963	0.73%	1036.98	7.02%	1.3084	19.36%
L2+SAS	1026.6	0.25%	5.6877	0.69%	1035.73	6.96%	1.3073	19.10%
L2 cache	2048		10.5372		1762.71		1.822	
IST - SAU	4.48	0.22%	0.0707	0.67%	75.33	4.10%	0.3275	15.24%
IST - SAS	3.68	0.18%	0.0494	0.47%	90.38	4.88%	0.329	15.30%
L2+SAU	2052.48	0.22%	11.722	0.67%	2396.03	4.10%	1.7501	15.24%
L2+SAS	2051.68	0.18%	10.5372	0.47%	2391.71	4.88%	1.822	15.30%

Table 3.3: CACTI results for different cache sizes, 2-way set associative.

	Size (KB)		Area (mm^2)		Power consumption(mW)		Access time (ns)	
L2 cache	16		0.1341		127.89		0.5899	
IST - SAU	0.34	2.08%	N/A	N/A	N/A	N/A	N/A	N/A
IST - SAS	0.21	1.30%	N/A	N/A	N/A	N/A	N/A	N/A
L2+SAU	16.34	2.08%	0.136	N/A	128.03	N/A	0.591	N/A
L2+SAS	16.21	1.30%	0.1354	N/A	128.02	N/A	0.5909	N/A
L2 cache	32		0.1949		135.93		0.6297	
IST - SAU	0.49	1.51%	N/A	N/A	N/A	N/A	N/A	N/A
IST - SAS	0.3	0.93%	N/A	N/A	N/A	N/A	N/A	N/A
L2+SAU	32.49	1.51%	0.1971	N/A	136.55	N/A	0.6307	N/A
L2+SAS	32.3	0.93%	0.1963	N/A	136.33	N/A	0.6306	N/A
L2 cache	64		0.375		169.09		0.6771	
IST - SAU	0.7	1.08%	0.0241	10.90%	78.39	36.47%	0.2924	31.68%
IST - SAS	0.43	0.67%	N/A	N/A	N/A	N/A	N/A	N/A
L2+SAU	64.7	1.08%	0.3786	10.90%	170.08	36.47%	0.6783	31.68%
L2+SAS	64.43	0.67%	0.3769	N/A	169.69	N/A	0.6774	N/A
L2 cache	128		0.6825		217.97		0.7793	
IST - SAU	1.02	0.79%	0.0314	4.40%	81.63	27.25%	0.2829	26.63%
IST - SAS	0.61	0.47%	0.0235	3.33%	78.61	26.51%	0.288	26.98%
L2+SAU	129.02	0.79%	0.6879	4.40%	218.79	27.25%	0.781	26.63%
L2+SAS	128.61	0.47%	0.6844	3.33%	218.19	26.51%	0.7797	26.98%
L2 cache	256		1.2159		317.97		0.9134	
IST - SAU	1.46	0.57%	0.0342	2.74%	81.1	20.32%	0.2938	24.34%
IST - SAS	0.86	0.33%	0.0251	2.02%	78.11	19.72%	0.2992	24.67%
L2+SAU	257.46	0.57%	1.2225	2.74%	369.1	20.32%	0.9149	24.34%
L2+SAS	256.86	0.33%	1.2192	2.02%	318.51	19.72%	0.9138	24.67%
L2 cache	512		2.324		460.76		1.2126	
IST - SAU	2.12	0.41%	0.0377	1.60%	76.6	14.25%	0.3024	19.96%
IST - SAS	1.24	0.24%	0.033	1.40%	81.24	14.99%	0.2895	19.27%
L2+SAU	514.12	0.41%	2.3338	1.60%	462.55	14.25%	1.2147	19.96%
L2+SAS	513.24	0.24%	2.3274	1.40%	461.35	14.99%	1.2132	19.27%
L2 cache	1024		5.4559		1291.69		1.5686	
IST - SAU	3.04	0.30%	0.0414	0.75%	77.91	5.69%	0.3122	16.60%
IST - SAS	1.76	0.17%	0.0361	0.66%	80.97	5.90%	0.3005	16.08%
L2+SAU	1027.04	0.30%	5.471	0.75%	1293.02	5.69%	1.5704	16.60%
L2+SAS	1025.76	0.17%	5.4642	0.66%	1292.2	5.90%	1.569	16.08%
L2 cache	2048		10.119		2235.16		2.1584	
IST - SAU	4.39	0.21%	0.0715	0.70%	75.53	3.27%	0.3271	13.16%
IST - SAS	2.54	0.12%	0.0391	0.38%	77.12	3.34%	0.307	12.45%
L2+SAU	2052.39	0.21%	10.1399	0.70%	2237.51	3.27%	2.16	13.16%
L2+SAS	2050.54	0.12%	10.1264	0.38%	2235.83	3.34%	2.1586	12.45%

Table 3.4: CACTI results for different cache sizes, 4-way set associative.

	Size (KB)		Area (mm^2)		Power consumption(mW)		Access time (ns)	
L2 cache	16		0.1637		77.65		0.5926	
IST - SAU	0.33	2.02%	N/A	N/A	N/A	N/A	N/A	N/A
IST - SAS	0.14	0.87%	N/A	N/A	N/A	N/A	N/A	N/A
L2+SAU	16.33	2.02%	0.1659	N/A	78.19	N/A	0.5938	N/A
L2+SAS	16.14	0.87%	0.164	N/A	77.73	N/A	0.5933	N/A
L2 cache	32		0.1963		145.99		0.6309	
IST - SAU	0.48	1.48%	N/A	N/A	N/A	N/A	N/A	N/A
IST - SAS	0.2	0.62%	N/A	N/A	N/A	N/A	N/A	N/A
L2+SAU	32.48	1.48%	0.1985	N/A	146.57	N/A	0.6319	N/A
L2+SAS	32.21	0.62%	0.1972	N/A	146.19	N/A	0.631	N/A
L2 cache	64		0.4032		142.59		0.6906	
IST - SAU	0.68	1.05%	0.0241	5.64%	78.39	35.47%	0.2924	29.75%
IST - SAS	0.29	0.45%	N/A	N/A	N/A	N/A	N/A	N/A
L2+SAU	64.68	1.05%	0.4061	5.64%	143.3	35.47%	0.6918	29.75%
L2+SAS	64.29	0.45%	0.4052	N/A	143.14	N/A	0.6911	N/A
L2 cache	128		0.6799		219.4		0.7715	
IST - SAU	0.99	0.77%	0.0314	4.41%	81.63	27.12%	0.2829	26.83%
IST - SAS	0.41	0.32%	N/A	N/A	N/A	N/A	N/A	N/A
L2+SAU	128.99	0.77%	0.6837	4.41%	219.84	27.12%	0.7722	26.83%
L2+SAS	128.42	0.32%	0.6818	N/A	219.62	N/A	0.7719	N/A
L2 cache	256		1.2171		321.63		0.9153	
IST - SAU	1.43	0.56%	0.0342	2.73%	81.1	20.14%	0.2938	24.30%
IST - SAS	0.59	0.23%	0.0235	1.89%	78.61	19.64%	0.288	23.93%
L2+SAU	257.43	0.56%	1.2237	2.73%	322.8	20.14%	0.9167	24.30%
L2+SAS	256.59	0.23%	1.2187	1.89%	321.9	19.64%	0.9154	23.93%
L2 cache	512		2.2562		480.72		1.2029	
IST - SAU	2.07	0.40%	0.0377	1.64%	76.6	13.74%	0.3024	20.09%
IST - SAS	0.84	0.16%	0.0251	1.10%	78.11	13.98%	0.2992	19.92%
L2+SAU	514.07	0.40%	2.2655	1.64%	482.42	13.74%	1.2042	20.09%
L2+SAS	512.84	0.16%	2.2593	1.10%	481.27	13.98%	1.2031	19.92%
L2 cache	1024		4.9412		1054.92		1.5827	
IST - SAU	2.97	0.29%	0.0414	0.83%	77.91	6.88%	0.3122	16.48%
IST - SAS	1.21	0.12%	0.0326	0.66%	81.3	7.16%	0.288	15.40%
L2+SAU	1026.97	0.29%	5.1758	0.83%	1095.76	6.88%	1.5876	16.48%
L2+SAS	1025.21	0.12%	5.165	0.66%	1093.99	7.16%	1.586	15.40%
L2 cache	2048		10.1175		2239.42		2.1392	
IST - SAU	4.3	0.21%	0.0704	0.69%	75.16	3.25%	0.3256	13.21%
IST - SAS	1.72	0.08%	0.0357	0.35%	80.99	3.49%	0.2992	12.27%
L2+SAU	2052.3	0.21%	10.1385	0.69%	2241.77	3.25%	2.1407	13.21%
L2+SAS	2049.72	0.08%	10.125	0.35%	2240.09	3.49%	2.1394	12.27%

3.7.2 FPGA model evaluation

The evaluation is based on the VHDL model implementation for FPGA device Spartan6 XC6LX16-CS234. Basically, we compare our proposal versus the L2 cache memory. The IST model consists of the following components: L2 cache memory, scrambler table, scrambler table controller, random number generator and two data scrambling circuits.

3.7.2.1 L2 cache memory

The size of the memory selected is 16 KB, 1-way associative, line size is 128-bit wide, while the data bus is 32-bit wide (4 words per line). Also, 4 additional bits are appended to the word line for youth flags (one bit for every word). The address bus is 32-bit wide: 20 tag bits, 10 index bits and 2 word bits.

The tag bits are used for the memory tag (or cache directory), while the index bits select one set out of 1024 total sets. The last 2 bits from the address array are used for pointing out which word is selected out of the 4 words in a line.

3.7.2.2 Scrambler table

For the cache size specified above, we considered 32 entries for the ST, SAS configuration. This means that one entry can be used for 128 words (the specified L2 cache has 1024 sets of 4 words each). Each counter has 6 bits, which means that the attached *scrambling vector* can be used for 32 write cycles. The *scrambling vector* is 32-bit wide, like the data bus. These specifications assure that the *scrambled data* which is stored in the cache is well disseminated. The index of the ST internal table has 5 bits and address a total of 32 rows ($k = 32$). Each row has 75 bits, which means that the size of the ST internal table sums up to a value of 2400 bits.

3.7.2.3 Additional circuits

The random number generator circuit used in the FPGA evaluation model is basically a pseudo-random number generator that produces a new number every clock cycle. However, a new random vector is necessary only when conditions of Equation 3.6 are satisfied. The data scrambling and descrambling is performed by two different 32-bit wide XOR circuits. The address j for the ST internal table is computed from index bits of the memory address, by a separate 5-bit wide XOR gate inside the ST controller.

Table 3.5 below illustrates the area utilization in the targeted FPGA device for the L2 cache and for the IST model (ST Controller plus L2 cache). The numbers

Table 3.5: Area utilization and overhead in the FPGA device implementation.

Design	**Reg.**	**LUT**	**Slice**	**IO**	**Block memory**	**BUFG**
L2 cache	2114	3250	997	103	8	2
IST model	2169	3374	1038	102	8	2
Overhead	**2.6%**	**3.8%**	**4.1%**	**-**	**-**	**-**

Table 3.6: Power consumption comparison in normal operation.

	L2 cache	**L2 cache+IST technique**
Clocks (W)	0.026	0.027
Logic (W)	0.017	0.018
Signals (W)	0.014	0.017
BRAMs (W)	0.007	0.007
IOs (W)	0.043	0.052
Device static (W)	0.021	0.021
Total (W)	0.129	0.141
		9.3%

express how much of the FPGA board resources are used. Note that the only noticeable increase is in the FPGA slices, which are the configurable elements - Configurable Logic Blocks (CLBs) such as: look up tables, flip-flops and latches. Thus, we can induce the area overhead for each resource: 2.6% more registers and 3.8% more LUTs are used by the IST technique, in comparison with a plain L2 cache.

The power analysis of the model is based on the power consumption of the design in normal operation when 480 consecutive write and read operations are performed. To get conclusive results, we used the XPower Analyzer tool [80] from the Xilinx suite which computes power consumption through the whole implemented design, based on timing constraints and simulation read and write operations. Table 3.6 displays the power consumption summary of the plain L2 cache memory and with the IST technique working in normal operation mode. Based on the results, the total power consumption increase is approximately 9.3%, which makes the proposed IST technique a viable low-power solution to scramble the data written in the L2 cache.

The speed evaluation of the proposed technique is based on calculating the average delay of the two most used paths: data input from CPU to cache memory

Table 3.7: Delay for data and clock path.

	L2 cache	**L2 cache + IST technique**
Read operation - Average delay (ns) **Delay increase**	7.544	8.691 **15.2%**
Write operation - Average delay (ns) **Delay increase**	3.228 **22.3%**	4.009
Average delay increase	**18.75%**	

(write operation) and data output from cache memory to CPU (read operation). Both the clock and data paths are considered in the comparison, therefore the delay is calculated as the minimum period of time needed to generate the correct output, since the clock signal appears on the path. For a clock period of 10 ns, the delay on the longest path of the L2 cache memory with the IST technique has an increase of 18.75%, as it is displayed in Table 3.7.

3.8 Conclusions

Data scrambling techniques used on cache memories are a viable solution to improve the security of the sensitive data with a minimal impact in the power consumption and area overhead. In this chapter, the Interleaved Scrambling Technique is proposed and explained, it can be employed in any cache memory, independently of the size or type of implementation. Essentially, this technique solves the problem of high volume of data update required in memories when the scrambling vectors are refreshed. It is demonstrated that the impact in average reading time, area overhead and power consumption is very low if compared to a standard scrambling technique. Also, using the CACTI tool and FPGA implementations, the impact of the IST methodology on caches is evaluated. The size of the scrambler table scales similar to the square root of the cache memory size (for the SAS scheme, as explained in a previous section), which means that for higher cache sizes, the impact of area overhead is small or insignificant. Also, power consumption and delay overhead percentage decrease with almost the same ratio (see CACTI and FPGA results tables).

Chapter 4

Defeating SPEMA and DPEMA

4.1 Introduction

Some algorithms running with compromised data select cache memory as a type of secure memory where data is confined and not transferred to main memory. However, some attacks, like cold-boot attack, which was designed for main memories, can also target cache memories. It exploits data remanence, i.e. a parasitic effect present in DRAMs and even more present in SRAMs often used in caches. Thus, a sudden power shutdown may not delete data entirely if after a short time lapse power is up again, giving the opportunity to steal data. Many systems use cache memory firmware cleanup but this may not be sufficient, as said below. The biggest challenge for any technique aiming to secure the cache memory is performance penalty. CPU and cache memory speeds are closely tied, and therefore any significant reduction of the latter becomes catastrophic to the former. Techniques based on data scrambling have demonstrated that security can be improved with a limited reduction in performance. However, they still cannot resist side-channel attacks like power or electromagnetic analysis. This chapter presents a review of known attacks on memories and countermeasures proposed so far and an improved scrambling techniques for defeating SPEMA and DPEMA. The methods are designed to protect the cache memory against cold-boot attacks, even if these are boosted by side-channel techniques like power or electromagnetic analysis. Furthermore, the proposed solutions can be integrated as part of a wider security solution for the memory system. The techniques are analyzed in terms of area, power and speed whereas the level of security is evaluated using adversary models and simulated attacks.

4.2 Motivation

In a recent report [2], the author pointed out that 62% of companies worldwide were subject to payment fraud in 2014 and that credit/debit cards are the second most frequent target of payment fraud. Mobile payments are a relatively new payment method, but this trend is increasing among large companies and organizations. However, there are several uncertainties about it such as disclosure of sensitive information or secure transfer of information. Ensuring the confidentiality of sensitive information is becoming more and more crucial [3]. Very often general purpose devices like desktops, laptops or smartphones are used for private transactions with financial entities or health-care issues, among others. In the case of devices without specialized hardware, all cryptographic operations are executed in software, resulting in an intensive use of memory [4]. This poses sensitive data at risk, including that stored in the cache memory [20].

Research on SRAMs has demonstrated that data can be maintained almost intact for a couple of minutes if the chip is kept at low temperatures or even at room temperature and without power supply [81]. This phenomenon is known as data remanence and results show that chip manufacturers do not control memory retention time as part of their manufacturing quality process. However, memory retention time varies between devices from the same manufacturer and of the same type but of different subtype or series. Also, low power versions of the same chips always seem to have longer retention times. This has opened up a whole new domain that has been widely investigated in the last decade. The scope of this work is in this realm.

When the CPU is running, it needs to work with sensitive data in plain form and, depending on the operating system, it may generate several copies in memory, exposing data to different kinds of attacks [4]. The problem with data remanence is that an adversary can use a cold-boot attack to extract critical data from memory by completely circumventing the software controlling the CPU [21]. This problem is discussed in depth in the following paragraphs.

4.3 Theoretical background

4.3.1 Attacks on memory

A simplified model of a computer memory system is presented in Fig. 4.1. Only cache and main memories are shown. Cache memory usually has two levels, L1 and L2, but this work focuses on the L2 cache because it is the larger of the two and its size allows enough data to be stored for full computations, being considered in some cases as a secure memory [82]. Cache memory is always integrated in the

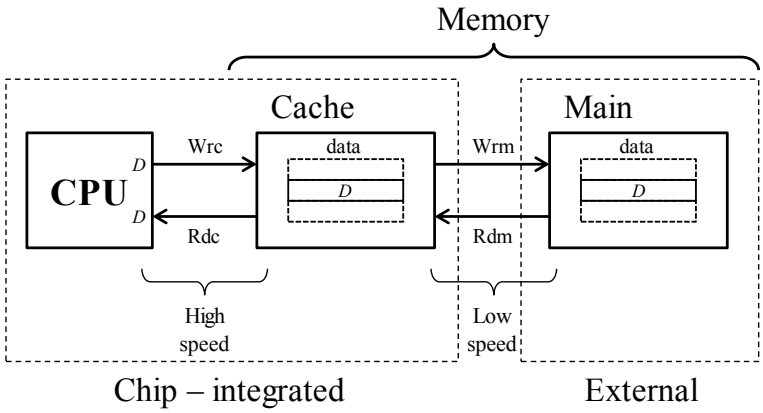


Figure 4.1: Simplified model of a computer memory system.

CPU package, even embedded in the CPU chip or assembled with it using 3D technology in the most advanced versions. This makes it feasible to use a high speed bus for reading and writing data, keeping CPU performance at the maximum level. Main memory is commonly external; it can be expanded and communicates with cache through a lower speed bus. It reads data only in the event of cache miss and writes according to some policies like write-through or write-back.

Since the CPU reads more frequently than it writes, and often does so in similar address ranges, the main memory bus is much less used than the cache memory bus, typically 20 times less. For this reason, loss in transmission speed in the main memory bus has less impact on CPU performance than in the cache memory bus.

4.3.2 Cold-boot attacks

An overview of a cold-boot attack on main memory is illustrated in Fig. 4.2. While the software that stores sensitive data in memory is running, the power supply is suddenly disconnected and the main memory is rapidly removed, connected to a backup system and powered up again. Then, the victim's memory content is downloaded into a backup machine and from there critical data like encryption keys or any other type of sensitive data is extracted [21]. By cooling the memory modules, degradation of volatile memory is slowed down; hence, an adversary has more time to act. The results of the attacks in [21] and [83] show that by cooling a memory chip at -50°C, decay within a 1MB region is 0.13% (after 60 minutes). Also, after a single minute without power supply, 99.9% of bits were recovered correctly.

This attack is not feasible on a cache memory because the latter cannot be

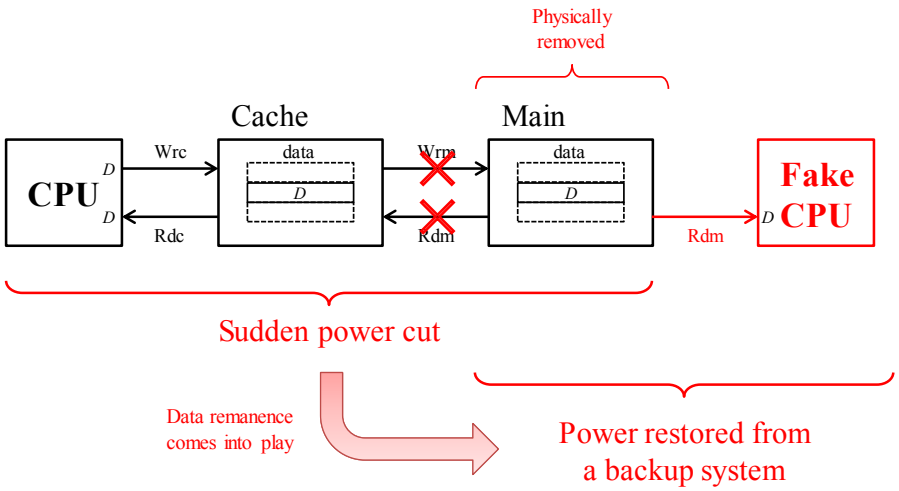


Figure 4.2: Cold-boot attack on main memory.

physically removed. However, a variation in Fig. 4.3 illustrates how an adversary using cold-boot attack could steal data.

Once the CPU has run the software of interest and sensitive data has been stored in memory, power is suddenly cut off and immediately afterwards the system is booted up with an ad-hoc fake program which makes a backup copy of the memory. Next, data is analyzed and sensitive information extracted. As an example, this kind of cold-boot attack was conducted on several smartphones [29]. A simple reboot from an ad-hoc operating system (FROST) immediately made a backup copy of the memory content and the secret keys of encrypted flash data and other sensitive data were extracted with specialized software tools. The technique used managed to recover email messages, contact lists, credit card data and other login credentials after cooling the device to 5°C. These smartphones had a boot locking mechanism that ensured the deletion of data in the user's partition and cache memory. However, not all models had this option activated by default, which might have been unknown to users.

In a recent work [82], several techniques to improve the security of smartphones and tablets were discussed. In these architectures, two types of memories internal to the CPU package can execute basic functionalities without accessing the main RAM, i.e. iRAM and L2 cache. They are considered secure mainly because after reboot, and this includes unexpected power cut offs, firmware cleans them up completely. Unfortunately, as said in [84], firmware can be attacked in many different ways, and thus it cannot be regarded as a strong security pillar.

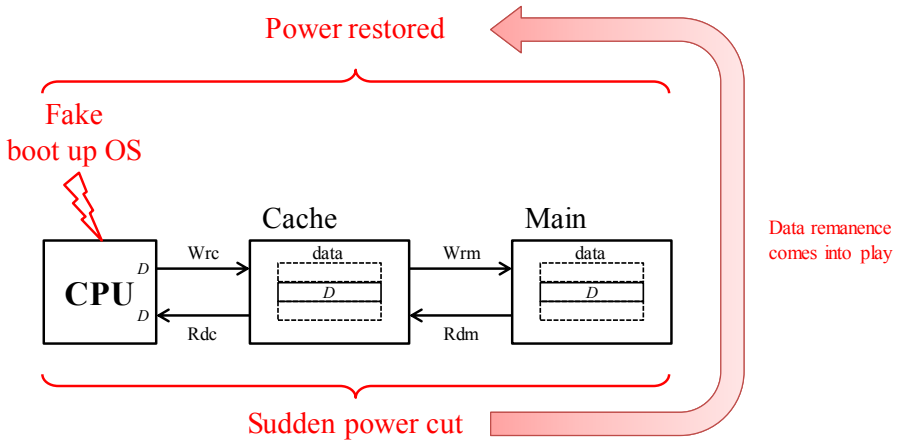


Figure 4.3: Cold-boot attack without removing memory chips.

4.4 Securing memory at hardware level

In order to provide stronger security against cold-boot attacks, more close to the hardware solutions are needed, as pointed out in [82]. These can be complemented by other firmware and software techniques, leading to a whole solution reaching different abstraction levels of the system. A review of existing hardware solutions is first presented, and then other possible alternatives are proposed. Most can be placed in the general scheme of Fig. 4.4.

4.4.1 Main memory

The most secure way to protect main memory data is to encrypt it in real time. A session key is generated and a strong, secure algorithm like AES [85] or a low-latency one like PRINCE [86] is used to encrypt and decrypt according to CPU demand. The main advantage of this method is that the circuit providing the session key will change it if a reset condition occurs, which will invalidate all memory data completely, thwarting any attempt to read the memory after boot. Memory bus encryption makes use of time clearance provided by the cache memory such that the performance loss is buffered [87, 88, 89, 90, 91]. The drawback of this solution is that encryption must be executed by an independent co-processor to reach enough speed and robustness, and for power consumption to remain below reasonable limits. That is why portable devices are not expected to use this strategy mainly owing to power consumption. Some smartphones and tablets use a kind of memory encryption but this action is only applied during user locking / unlocking

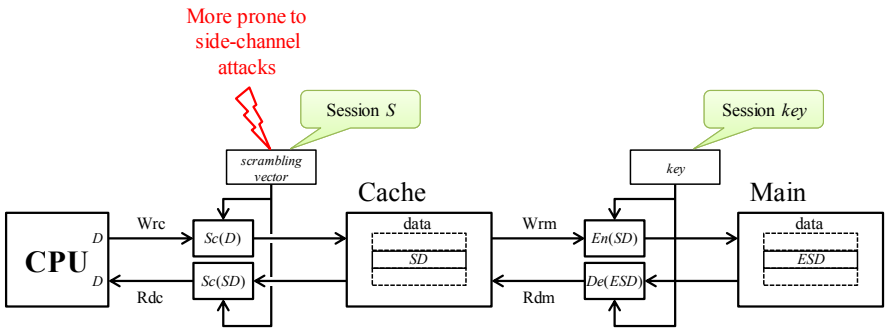


Figure 4.4: Published proposals securing memory at hardware level against cold-boot attacks.

[82]. In [88], a refresh mechanism was added to the above scheme which changes the key periodically, thus strengthening protection against side-channel attacks. The key is generated by a specialized smart card IP. In [89], this strategy was selected to protect systems with non-volatile memory, like ferromagnetic RAM. Encryption is executed incrementally and, in case of unexpected reboot, the whole memory is encrypted in 5 seconds. In [90, 91], memory encryption includes a time stamp to counteract replay attacks.

4.4.2 Cache memory

A similar protection scheme can be used for cache memory. However, no encryption process can be easily selected because of the high speed required by the bus. Scrambling techniques provide a lighter security degree. Data vectors are scrambled/descrambled with a session *scrambling vector* (S) by an XOR operation [92, 93]. As before, the circuit providing the S will change it after each reset, invalidating data in case of attempt to read cache after boot. The advantages are that high speed can be achieved by the scrambling and that the impact on power consumption is negligible. In fact, Intel uses this technique to transmit memory bus data, reducing high current peaks that could aggressively disturb the power supply lines [94]. The main drawback lies in that scrambling is not a securing but an obscuring technique which provides a low degree of security and are prone to side-channel attacks aiming to discover the S. Hence, frequent refresh of the S could improve the level of security, but cache data could become invalidated, and would need therefore to be updated. This would require the cache controller to shut down completely. In [92], scrambling was applied to data and addresses. The technique aims to defend cache memory against a wide range of side-channel attacks.

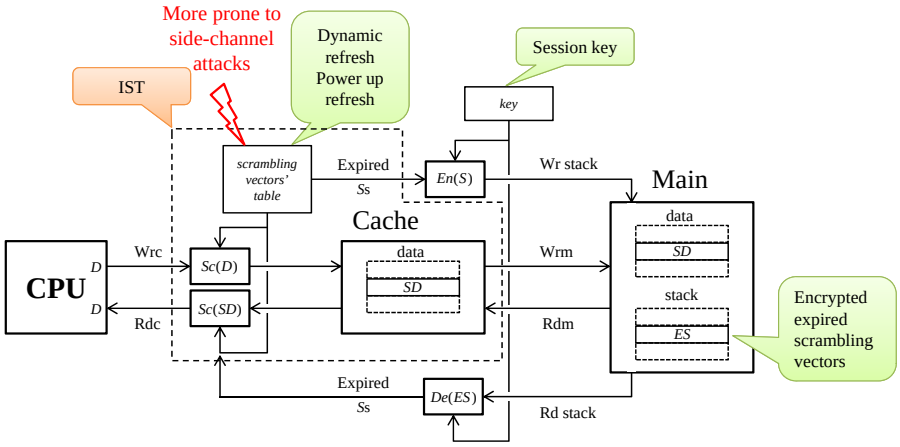


Figure 4.5: Interleaved Scrambling Technique. Hardware protection for cache that can be integrated into a global protection scheme.

It consists in a two-step scrambling process for both data and addresses with two different encryption keys stored in the main memory. Hence, a cold-boot attack on the main memory can disclose the memory section containing the encryption keys. In [93], a data scrambling technique was proposed to protect cache data as follows: the first half of a word is scrambled with the first bit of the first half, and the second half is then XORed with the scrambled result of the first half. The advantage is that since no *scrambling vectors* have to be stored in additional hardware, no data or area overhead has to be added. However, the patterns created by this method can be easily understood and peculiar data samples provide adversaries with a lot of information.

4.4.3 Interleaved Scrambling Technique

Interleaved Scrambling Technique (IST) is a security solution for cache protection against cold-boot attacks presented in Chapter 3 and in [95]. It enhances standard scrambling [92, 93] because the *scrambling vectors* can be refreshed continuously without interrupting communication between the CPU and cache. It can also be integrated into a global protection scheme, as illustrated in Fig. 4.5.

Internally, IST works with pairs of interleaving *scrambling vectors*. One (the young one) is used for writing data to and reading data from the cache memory whereas the other (the old one) is used only for reading. Therefore, when the pair is active, the cache memory becomes filled with data scrambled by the young

scrambling vector while being emptied of data scrambled by the old one. Once the cache is cleared of all data scrambled by the old *scrambling vector*, this vector expires and the pair is renewed such that the young *scrambling vector* becomes old and a newly generated *scrambling vector* becomes the young one. After reset, new *scrambling vectors* are generated again, invalidating cache data completely. IST can be used alone to protect the cache memory, as in [95], or be integrated into a global security solution. Fig. 4.5 shows one possible scheme which maintains high bus speed and a low level of power consumption compared to alternatives that encrypt bus data. In this approach, scrambled data is also written back to main memory. The scrambling and descrambling process is all concentrated in the same unit, which is located close to the CPU. As scrambled data flow through all buses, they help to keep current levels stable, as pointed out by Intel in [94]. Data stored in memory but not copied in cache will need the corresponding *scrambling vectors* to be made available in case the CPU needs them. Thus, whenever a *scrambling vector* expires, it is made available encrypted with a session key in a buffer such that the CPU can store it in a non-cached memory page. In this way, encryption speed requirements are much lower than those of data transfer, *scrambling vectors* generated internally cannot be reverse engineered and cold-boot attacks on main memory data and *scrambling vector*s are thwarted.

4.4.3.1 Side-channel attacks on IST

Even though IST refreshes *scrambling vector*s periodically, memories protected by IST are still prone to side-channel attacks because the scrambling technique is not intrinsically robust against this type of attacks. By using power or electromagnetic radiation analysis, an adversary could discover the *scrambling vector* in use for writing after several attacks. Although this would be difficult in practice, he could theoretically descramble cache or main memory data downloaded after a cold-boot attack. In the coming paragraphs, this weakness is explained and the methodology to improve IST robustness against this type of side-channel attack is presented and evaluated.

4.5 Power (P) and Electromagnetic (EM) Radiation Analysis

For the sake of clarity and without loss of generality, our discussion focuses on a scrambling circuit using a single *scrambling vector*. Let us assume that a data vector D is placed in the data bus. Before it being sent to the cache, IST mixes it with a *scrambling vector* S through XOR gates such that the scrambled data vector

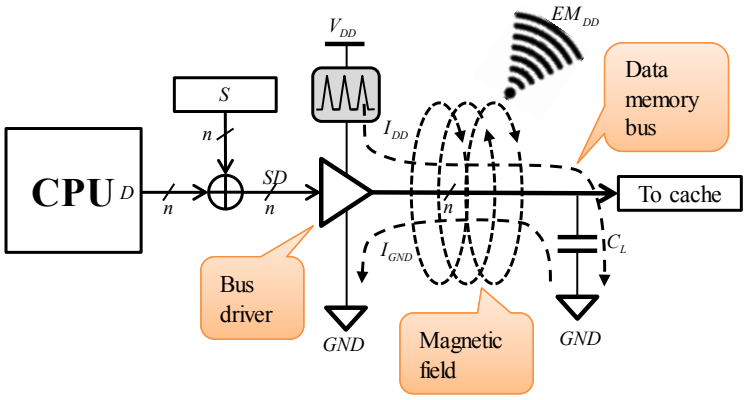


Figure 4.6: Switching drivers and bus line currents excite power peaks and electromagnetic pulses which leak information about data flowing into the drivers and buses.

$SD = D \oplus S$ is generated. Reverting this transformation (descrambling) is simple, $SD \oplus S = D \oplus S \oplus S = D$. A simplified model of the elements involved in information leakage is shown in Fig. 4.6.

Transmitting data flowing in a bus involves charging and discharging load capacitances, the parasitic capacitances of the lines themselves and of the next stage logic. This means that in each clock cycle some of the lines first source current from the power supply line and then drain it to ground. All switching bus lines source currents from the power supply line P_{DD} whose intensity is strongly related to the amount of scrambled data transmitted through the bus. Similarly, the magnetic fields of these currents add up and generate an electromagnetic wave EM_{DD} whose intensity depends on the scrambled data too. Hamming weight is a metric that adds up the contributions of individual bits in a word. Hence, it is often used to find correlations between data and power or radiated electromagnetic intensity [96].

Let us assume that data vector D of n bits is scrambled. Once the vector is in the bus, the hamming weight is

$$HW = \sum_{i=0}^{n-1} SD(i) \tag{4.1}$$

where $SD(i)$ is each one of the individual bits of the scrambled data vector. As a result of this operation, average current intensity and average radiated electromagnetic power are

$$\begin{aligned} I_{DD} &= \Phi(HW) \\ EM_{DD} &= \Psi(HW) \end{aligned} \tag{4.2}$$

Functions $\Phi()$ and $\Psi()$ are unknown to an attacker but he can assume the following powerful hypothesis: 1) they are time invariant, and 2) they are monotonic. With (1) any pre-attack characterization of the function is useful for a later attack, and with (2) he can establish a one-to-one relationship between the physical magnitudes and HW. Moreover, owing to (2) any minimum, maximum and average values of the physical magnitudes can be estimated without confusion by assessing HW and vice versa. By using a model in a simulator, the attacker estimates the values of HW and calculates the physical magnitudes to generate a rough profile of functions $\Phi()$ and $\Psi()$ using measurements of the real system. Based on this, any attack can then be conducted by assessing HW. The attacks are presented in the context of cache memory scrambling technique in the following paragraphs.

4.5.1 Simple P or EM Radiation Analysis Attack

In a Simple P or EM radiation Analysis (SPEMA), the adversary sends profiling data vectors D_p to the cache memory bus and by measuring I_{DD} or EM_{DD} he selects the data vector maximizing HW [96]. For an n-bit bus, the maximum hamming weight is $HW_{mx} = n$ when all bits of the scrambled data vector are ones, $SD^* = (11..1)$. Then, once the adversary knows the maximizing data vector D_p^*, he can extract S as follows:

$$\begin{aligned} D_p^* \rightarrow HW_{mx} \rightarrow HW(SD^*) = n \rightarrow SD^* = (11..1) \\ S^* \oplus D_p^* = (11..1) \rightarrow S^* = \overline{D_p}^* \end{aligned} \tag{4.3}$$

where S^* is the *scrambling vector* estimated by the adversary, which coincides with the real S if the attack has been successful. Because the scrambling circuit is linear, the attack can be conducted by subsets of bits which reduce the number of profiling data vectors to try significantly. In this case, the *scrambling vector* is estimated by subsets S_i^* for $i = 0, 1, 2, \ldots$; for example, S_0^* may contain the first subset of three bits, S_1^* the second subset and so on. In the end, the final S^* is the intersection of all subsets:

$$S^* = \bigcap_{\forall i} S_i^* \tag{4.4}$$

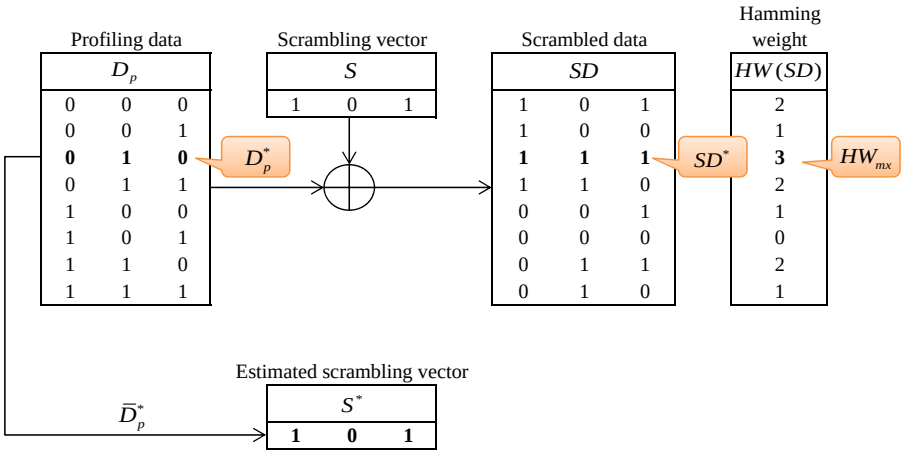


Figure 4.7: Example of SPEMA attack.

4.5.1.1 Example

For illustrative purposes, let us suppose a simple case of three bits, $n = 3$. The numerical analysis is shown in Fig. 4.7.

The true *scrambling vector* is $S = 101$. The adversary performs the attack by trying all possible profiling data vectors D_p shown at the left side of the figure. The scrambling circuit internally generates the scrambled data vectors *SD* shown at the right side. By measuring the leakage, the adversary discovers the combination producing the maximum hamming weight, printed in bold, which is $D_p^* = 010$. Finally, D_p^* is inverted to estimate the *scrambling vector* giving $S^* = 101$, which matches the true *scrambling vector* S.

4.5.2 Differential P or EM Radiation Analysis Attack

DPEMA is an even more powerful attack than SPEMA despite requiring a longer time to be completed. In the context of cache memory, this attack does not discover the whole *scrambling vector* at once but estimates it on a bit by bit basis [96].

Let us assume that the adversary attempts to estimate bit s_j of the *scrambling vector*. He first builds two subsets of profiling vectors (data vectors): subset $D(d_j = 0)$ contains all combinations of values at which data bit located at the same position, d_j, is constant at 0 and subset $D(d_j = 1)$, which is similar but now bit d_j is constant at 1. Then he applies the first subset $D(d_j = 0)$ in a continuous loop and estimates the average hamming weight $HW_{avg}(d_j = 0)$ of the scrambled data. He repeats the same action again with the second subset $D(d_j = 1)$ and estimates

the average hamming weight $HW_{avg}(d_j = 1)$. Finally, he guesses bit s_j^* of S^* as follows:

$$s_j^* = \begin{cases} 0, if\ HW_{avg}(d_j = 0) < HW_{avg}(d_j = 1) \\ 1, if\ HW_{avg}(d_j = 0) > HW_{avg}(d_j = 1) \end{cases} \tag{4.5}$$

This operation is repeated for each bit and once all bits are estimated, the *scrambling vector* is built as the concatenation of:

$$S^* = (s_{n-1}^*, s_{n-2}^*, \ldots, s_1^*, s_0^*) \tag{4.6}$$

The strength of this attack lies in the fact that it is not necessary to apply all combinations of the rest of bits which are not kept constant. It is even better to change them randomly, thus reducing the amount of vectors significantly and obtaining a tighter estimate of the average hamming weight with very low noise. Furthermore, the hamming weight is not really necessary since the comparison in eq. (4.5) can be made directly with the physical magnitudes $P_{DDavg}(d_j = 0)$ and $P_{DDavg}(d_j = 1)$ or $EM_{DDavg}(d_j = 0)$ and $EM_{DDavg}(d_j = 1)$ measured by the external instruments. One of the most dangerous points is that the physical magnitude can be averaged over thousands of samples, which would greatly reduce the effect of (accidental or intentional) noise and would strengthen the signal induced by the sought information.

4.5.2.1 Example

Fig. 4.8 shows an example of a DPEMA attack in which bit number 2 if a *scrambling vector* of 3 bits is estimated. In the first phase (top of figure), the bit in the profiling vectors is kept constant at 0 while the rest of bits are changed in a continuous loop. Meanwhile, the hamming weight is estimated from the physical measurements of P or EM. In the example, this value is $HW_{avg}^0(2) = 2$. In the second phase (bottom of figure), the run is repeated but keeping the bit constant to 1 and again the average hamming weight is estimated, $HW_{avg}^1(2) = 1$. Finally, the comparison is made and the estimated bit is $S^*(2) = 1$, which matches the bit of the true *scrambling vector* $S(2)$.

4.5.3 Attack model

For the rest of the chapter it is assumed that the adversary can measure current consumption using sensors attached to the power supply or measure electromagnetic power radiation by means of antenna probes. These can detect internal activity in

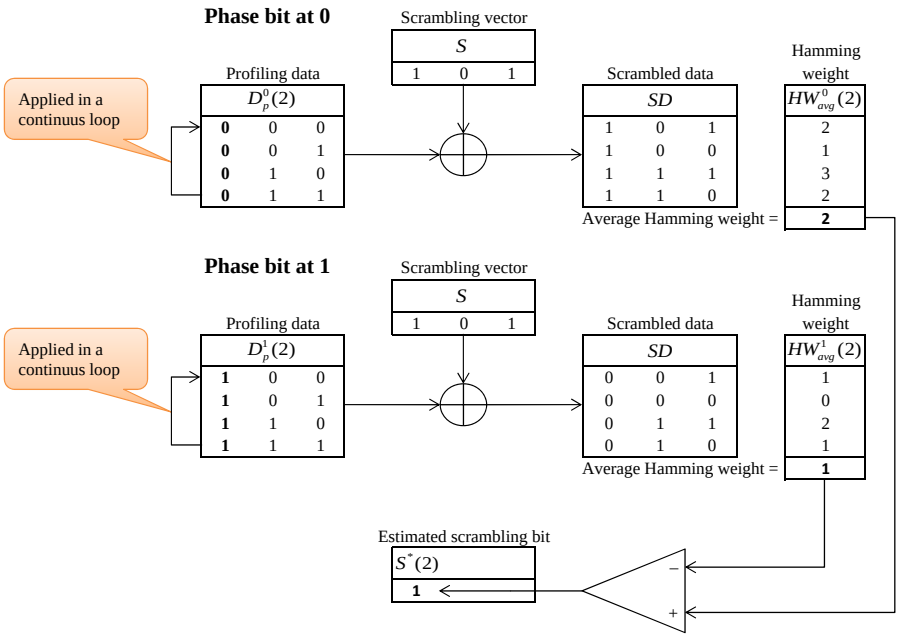


Figure 4.8: Example of DPEMA attack.

the memory bus. He knows the model of the L2 cache and understands the IST process/operation. Moreover, he has control over some data vectors generated by the CPU, which allow him to estimate the *scrambling vector* in use. He cannot read cache memory sensitive data directly from the CPU, since it is assumed that the operating system blocks protected memory addresses when the critical program is in operation, therefore to do so the system must undergo a cold-boot attack such as those described above. The adversary cannot read cache content from the outside of the CPU because the former cannot be detached from the latter.

4.6 Statement of the Problem

Cold-boot attacks against cache memories protected by scrambling techniques fail because this kind of attacks requires a brief interruption of the power supply. This action always places the system in a reboot sequence and forces a change in the *scrambling vector* that immediately invalidates the reading of cache data.

Let us consider a data word D_1. Once it is stored, it becomes the scrambled data vector $SD_1 = D_1 \oplus S_1$. Now the cold-boot attack is applied and the *scrambling vector* changes to S_2. If the scrambled data vector SD_1 is then read from the cache,

the adversary will obtain $D_2 = SD_1 \oplus S_2 = D_1 \oplus S_1 \oplus S_2 \neq D_1$, which is different from the original data.

However, using SPEMA or DPEMA the *scrambling vector* can be estimated before and after the cold-boot attack. In this way, the adversary would know S_1 and S_2, which would allow him to produce the following result: $D_2 \oplus (S_1 \oplus S_2) = SD_1 \oplus S_2 \oplus (S_1 \oplus S_2) = D_1 \oplus S_1 \oplus S_2 \oplus (S_1 \oplus S_2) = D_1$, and therefore he would recover the original data.

In Chapter 3 and [97] a countermeasure against Simple Power Analysis (SPEMA) attacks on caches using the scrambling technique, in particular the IST, was proposed. SPEMA [96] aims to discover S by doing a trace of the P or EM radiation and from the picks and valleys the S is estimated. Using this side-channel leakage an adversary can undergo the cache memory to a Simple Power or Electromagnetic Analysis (SPEMA) or even to a more powerful Differential Power or Electromagnetic Analysis (DPEMA). The aim of the attack is to recover the internal S with which the cold-boot attack can succeed afterwards.

Assume that the victim cache has sensitive data in it, scrambled with vector S_1. Immediately, the adversary attacks the victim with an SPEMA or DPEMA, estimates the *scrambling vector* S_1^* and carries out the cold-boot attack by abruptly disconnecting the supply and booting the system with an ad-hoc operating system. Promptly after, the adversary reads out the sensitive data from the victim cache and stores them in a backup memory. However, if the scrambling technique is like IST from Chapter 3, after the cold_boot attack the *scrambling vector* will change to a new one, e.g. S_2 such that the stolen data recovered by the adversary will be now $SD = D \oplus S_1 \oplus S_2$, in which D is the sensitive data wanted. Therefore, the adversary will have to perform a second SPEMA or DPEMA attack to estimate S_2^*. In the final step, the double scrambling is undone by the adversary on the stolen data $(SD = D \oplus S_1 \oplus S_2) \oplus S_1^* \oplus S_2^* \rightarrow D$ and the sensitive data is recovered provided that the estimations of S_1^* and S_2^* are correct.

However, it was seen that the countermeasure proposed was not resistant against differential P or EM analysis attacks (DPEMA) which is explained in this chapter. In sub-chapter **??**, the limitation of this SPEMA countermeasure against DPEMA is compared with the efficiency of the solution proposed in this work. In this chapter, the SPEMA countermeasure is extended to fight against DPEMA attacks too, so that the complete solution becomes resilient against both types of side channel attacks. The rest of the work is focused on the DPEMA attacks, however the solution proposed and the results presented will include SPEMA attacks too.

4.6.1 Objective

The objective of this chapter is presenting two techniques for the defeating of the SPEMA and DPEMA side-channel attacks. In an initial phase SPEMA combines the error detection correction and localization technique presented in Chapter 2 with the IST technique to pursue this objective. In the next phase the same SPEMA is improved including a masking strategy to extend the protection to DPEMA attacks. Therefore, the final defeating technique proposed for DPEMA is in fact also defeating SPEMA attacks. However, they are presented in these two steps for better understanding the roles of each one of the techniques proposed.

4.7 Proposed solution for defeating SPEMA

In this section, the protection against SPEMA attacks is achieved by combining an *Error Detection, Localization and Correction technique* (eDLC) presented in Chapter 2 with the securing *Interleaved Scrambling Technique* (IST) presented in Chapter 3. In the next paragraphs the eDLC technique is quickly reviewed and how it is combined with the IST is explained.

4.7.1 eDLC review and integration with IST

Permanent faults in memories are difficult to mitigate, but are barely encountered. On the other hand, transient faults that generate soft errors due to radiation are more likely to happen and the main causes are usually high-energy neutrons and alpha-particles. Dynamic RAMs are susceptible to radiation-type errors that manifest as unidirectional bits flips (due to the capacitor inside the DRAM cell). eDLC creates a full adder tree that generates redundant bits (carry and sum) at each subset of three data bits, during the writing operation of the cache memory as it is illustrated in Figure 4.9. The redundant bits count the number of 1ns in the data subset and are appended to the data bits when they are stored in the cache memory. During the reading phase, the redundant bits are checked in the eDLC unit and in presence of an error they are used for localizing and correcting the data transferred to the CPU.

When data include eDLC bits the IST is organized as follows. After the CPU sends data, the eDLC unit adds redundant bits, data vector D becomes $(D, \psi(D))$ where $\psi(D)$ are the redundant bits. In the scrambler table of the IST technique, the *scrambling vectors* are also extended with additional bits to scramble the redundant bits of the data. Thus we have (S, S_ψ) being S the bits scrambling data D and S_ψ the bits scrambling the redundancy $\psi(D)$. The scrambled data stored in the cache memory, (SD, SR) are finally generated as follows,

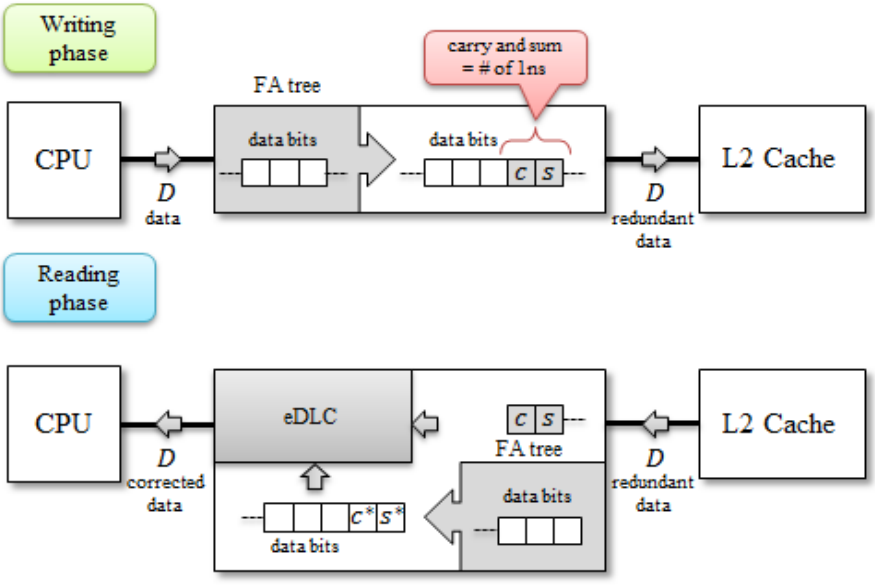


Figure 4.9: The error detection, correction and localization technique (eDLC).

$$\begin{aligned} SD &= D \oplus S \\ SR &= \psi(D) \oplus S_{\psi} \end{aligned} \tag{4.7}$$

As was explained before, when the SPEMA attack is conducted it searches for the maximum *hamming weight* in the cache memory data bus carrying the scrambled data, $(SD, SR)^*$. However, now the search that the adversary makes in the data space cannot be exhaustive because the redundant bits $\psi(D)$ are internally calculated from D and cannot be freely imposed. During the search, e.g. think of the brute force strategy, two possible end points can be reached: 1. only one maximum, $(SD, SR)^*_U$ or 2. More than one maximum are found, $(SD, SR)^*_M$.

1. In the case of $(SD, SR)^*_U$ the attack is straightforward. If the number of bits in SR is less than the number of bits in SD the unique maximum must be $((11\ldots1), SR)$ being SR of no relevance for the attack. Then the *scrambling vector* is found with $S = \overline{D}^*$.

2. In the case of $(SD, SR)^*_M$ the maximum is not unique and this means that different values of D^*_i may give the same hamming weight $HW^* = HW(D^*_i \oplus S) + HW(\psi(D^*_i) \oplus S_{\psi})$ for $i = 1, 2, \ldots$. This introduces an uncertainty to the adversary because now the reversing function $S_D = \overline{D}^*$ may not be valid. In

Scrambling vector

S			S_ψ	
1	0	1	1	0

Data with eDLC

D			$\psi(D)$	
0	0	0	0	0
0	0	1	0	1
0	**1**	**0**	**0**	**1**
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Scrambled data

SD			SR	
1	0	1	1	0
1	0	0	1	1
1	**1**	**1**	**1**	**1**
1	1	0	0	0
0	0	1	1	1
0	0	0	0	0
0	1	1	0	0
0	1	0	0	1

Hamming weight

H	
3	
3	
5	*
2	
3	
0	
2	
2	

Table 4.1: Example of unique maximum.

general, there will be several values to consider and in particular even none of them may include the expected $SD = (11\ldots1)$.

4.7.1.1 Example of unique maximum

Table 4.1 presents a numeric example of case 1.

At the left we have 3 data bits with 2 check bits counting the number of ones in data. All possible values of data bits are listed. At the top we have the *scrambling vector* $(101, 10)$ and below that, the scrambled data (SD, SR). At the right, the *hamming weight* is evaluated. We can see that a unique maximum exists with a value of 5 and it corresponds to $SD = (111)$. Therefore, the *scrambling vector* can be discovered by complementing the data $D^* = (010)$ giving $S = (101)$.

4.7.1.2 Example of multiple maximums

Table 4.2 presents the numeric example of case 2. The content of the Table is the same except that now $S_\psi = (01)$ instead of (10). When the scrambled data is calculated, two maximums are found with *hamming weights* equal to 4, being the corresponding data $D^* = \{(011), (110)\}$. None of them contain the scrambled data $SD = (111)$ and as a consequence the *scrambling vector* cannot be obtained inverting any of both data.

The fact that case 1 or 2 happen it depends on the *scrambling vector*, and in particular of the S_ψ segment of it. The countermeasure against SPEMA attack is based on the following principle. Segment $S\psi$ is not completely generated at

Data with eDLC					Scrambling vector					Scrambled data					Hamming weight	
D			$\psi(D)$		S			S_ψ		SD			SR		H	
0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	3	
0	0	1	0	1						1	0	0	0	0	1	
0	1	0	0	1						1	1	1	0	0	3	
0	**1**	**1**	**1**	**0**						**1**	**1**	**0**	**1**	**1**	**4**	*
1	0	0	0	1						0	0	1	0	0	1	
1	0	1	1	0						0	0	0	1	1	2	
1	**1**	**0**	**1**	**0**						**0**	**1**	**1**	**1**	**1**	**4**	*
1	1	1	1	1						0	1	0	1	0	2	

Table 4.2: Example of multiple maximums.

random but a filter is applied to the random generator such that case 2 always happens.

4.7.2 Scrambling vector redundancy filter

The eDLC coding subdivides data in subsets of three bits and adds two additional bits corresponding to the Berger code of them, like the configuration shown in the examples of tables 4.1 and 4.2. Let's assume that $D^3 = (d_2, d_1, d_0)$ is one of these subdivisions. eDLC coding adds the carry and sum bits of the addition of these bits such that we have the codeword space $D^5 = (d_2, d_1, d_0, c, s)$, where $(c, s) = \sum_{all} d_i$. To guarantee that after scrambling we will reach case 2, *scrambling vectors* are generated in the non-codeword space such that never all 1ns are found in the scrambled data. Let's assume that our subset of bits scrambling D^5 is $S^5 = (s_2, s_1, s_0, s_c, s_s)$. The following generating function assures this condition, that set $S^5 \notin D^5$ and that multiple maximum *hamming weights* will always exist in the scrambled data.

$$\begin{aligned} (s_2, s_1, s_0, s_c) &= rand\,() \\ s_s &= \overline{s_2 \oplus s_1 \oplus s_0} \end{aligned} \tag{4.8}$$

where $rand\,()$ is the random generator function and bit s_0 is the $NXOR$ of the three most significant bits. Table 4.3 shows the codeword space of D^5 and the non-codeword space of S^5

Table 4.3: Codeword and non-codeword spaces for D^5 and S^5 respectively.

	D^5				
	b_2	b_1	b_0	c	s
0	0	0	0	0	0
5	0	0	1	0	1
9	0	1	0	0	1
14	0	1	1	1	0
17	1	0	0	0	1
22	1	0	1	1	0
26	1	1	0	1	0
31	1	1	1	1	1

	S^5				
	s_2	s_1	s_0	s_c	s_s
1	0	0	0	0	1
3	0	0	0	1	1
4	0	0	1	0	0
6	0	0	1	1	0
8	0	1	0	0	0
10	0	1	0	1	0
13	0	1	1	0	1
15	0	1	1	1	1
16	1	0	0	0	0
18	1	0	0	1	0
21	1	0	1	0	1
23	1	0	1	1	1
25	1	1	0	0	1
27	1	1	0	1	1
28	1	1	1	0	0
30	1	1	1	1	0

4.7.2.1 How it works

Consider the case where the adversary attacks the system by scanning all possible values of the data in a subset base. First, he takes the three data bits D_0^3 and generates all possible combinations, detects the maximums of the energy and records the values giving these maximums D_0^{3*}. Then moves to the next subset D_1^3, repeats the procedure and identifies the maximums D_1^{3*}. He continues until all subsets are scanned and the maximum subsets are created. Finally, he combines all the maximum subsets to create the set of data D^* giving the maximum *hamming weighs* which are the candidates to extract the scrambling vector S using the equation $S = \overline{D}^*$. However, most of the times this equation will fail because of the multiplicity of the maximum *hamming weights.*

In Table 4.4 this multiplicity number is presented, all possible *hamming weights* for the scrambled data $SD^5 = D^5 \oplus S^5$ are shown. At the top and at the left we have all possible values for S^5 and D^5 respectively. When the adversary scans all possible values for the data vector, this is equivalent to sweep the matrix columnwise. For each of the columns we count the number of maximums found. This information is summarized below the table.

In a non-protected cache using a scrambling technique the number of elements

Table 4.4: Hamming weights for the data vector and scrambling vector spaces.

$HW(SD^5)$ / D^5	S^5 = 1	3	4	6	8	10	13	15	16	18	21	23	25	27	28	30
0	1	2	1	2	1	2	3	**4**	1	2	3	**4**	3	**4**	**3**	**4**
5	1	2	1	2	3	**4**	1	2	3	**4**	1	2	3	**4**	**3**	**4**
9	1	2	3	**4**	1	2	1	2	3	**4**	3	**4**	1	2	**3**	**4**
14	**4**	**3**	2	1	2	1	2	1	**4**	3	**4**	3	**4**	3	2	1
17	1	2	3	**4**	3	**4**	3	**4**	1	2	1	2	1	2	**3**	**4**
22	**4**	**3**	2	1	**4**	3	**4**	3	2	1	2	1	**4**	3	2	1
26	**4**	**3**	**4**	3	2	1	**4**	3	2	1	**4**	3	2	1	2	1
31	**4**	**3**	**4**	3	**4**	3	2	1	**4**	3	2	1	2	1	2	1
max HW^*	4	3	4	4	4	4	4	4	4	4	4	4	4	4	3	4
# of	**4**	**4**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**4**	**4**

Top number of maximums = 4 (for 4 cases)
Bottom number of maximums = 2 (for 12 cases)
Average number of maximums = 2.5

in the set D^* is one and then S can be extracted immediately. With the protection in the most favourable situation any D_i^5 will have at least 2 maximums and therefore the total number of elements in the set D^* will be the combination of all the subset values, that is 2^l where l is the number of subsets. In the worst case the number of maximums in all the subsets will be 4 and thus the total number of elements in D^* will be 4^l. In the Table 4.5 these two limits are calculated for different sizes of data words. The average number of maximums is 2.5^l as indicated in Table 4.4. It can be seen that after 32 bits the number of elements that the hacker has to investigate is significantly large and for higher number of bits it becomes impractical.

A program has been made in C++ that emulates the eDLC and ITS units using the non-codeword generation for the *scrambling vectors*. For 12 data bit lengths, brute force attacks have been programmed scanning all the data space. One attack means that a *scrambling vector* is generated and data is scanned exhaustively while the maximum *hamming weights* are saved and counted. These results have been registered for 3,000 attacks and are shown in the histogram of Figure 4.10.

As it can be observed, in 1.008 attacks, the number of maximum *hamming weights* have been 16 while in 15 attacks the number of maximums have increased to 256, as predicted in Table 4.5. The average number of maximums is 37.92, very close to the predicted average. In any attack a unique maximum is not found which confirms the validity of the technique against SPEMA attack.

In Fig. 4.11 an overview of the SPEMA cache protection is shown and in Figure 4.12 a detailed scheme for the writing cycle can be seen. From now on this

Table 4.5: Number of elements in the set of maximums D^*.

Data word length	# subsets l	# Elements in D^*		
		min	avg	max
8	3	8	15	64
10	4	16	39	256
12	4	16	39	256
14	5	32	97	1024
16	6	64	244	4096
20	7	128	610	16384
24	8	256	1525	65536
28	10	1024	9536	1048576
32	**11**	**2048**	**23841**	**4194304**
40	14	16384	372529	268435456
48	16	65536	2328306	4,295E+09
56	19	524288	3,638E+07	2,749E+11
64	22	4194304	5,684E+08	1,759E+13

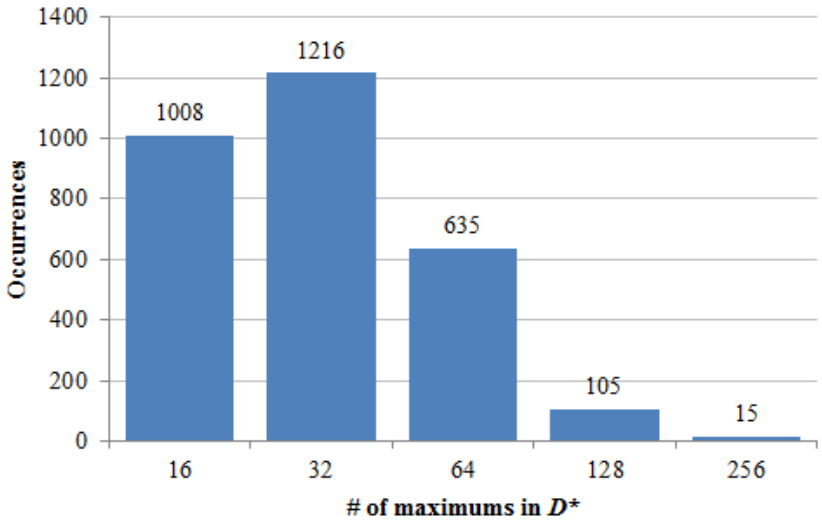


Figure 4.10: Histogram of number of maximums found in the D^* set for 12 data bits after 3.000 attacks. Average is 37.92.

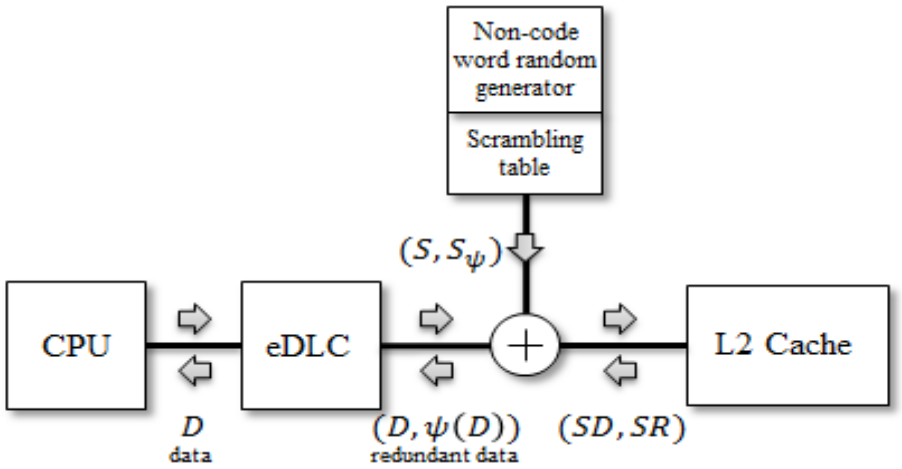


Figure 4.11: Overview of the IST with SPEMA protection.

combined technique IST-eDLC will be refered as ISTe.

4.8 Proposed solution for defeating DPEMA

This section presents the solution to protect the cache memory against cold-boot attacks boosted with DPEMA. The idea is to reduce the amount of information leaking from the system and to modify the correlations such that confusion prevents proper operation of the attack model, that is to prevent the analysis with DPEMA from revealing the true *scrambling vector*. This solution is build over the countermeasure ISTe explained in Section 4.7. In order to extend the ISTe technique to the protection of DPEMA attacks a small modification of Equations 4.8 is applied as it is explained in the next paragraphs. The purpose of this modification is to reduce to the half the non-codeword map shown in Table 4.3 but by keeping the properties of the technique. With this reduction we will obtain some additional protection as it will be shown in the experimental results Section, particularly for the extension of the DPEMA countermeasure.

As a quick remind of the ISTe technique, remember that data vector bits (D^3) are separated in groups of three bits to which two bits are appended corresponding to the sum and carry bits. Therefore, each data vector $D^3 = \{\ldots, (d_2, d_1, d_0)^i, \ldots\}$ is first appended with the error detection and correction bits and transformed to the new vector $D^5 = \{\ldots, (d_2, d_1, d_0)^i, c^i, s^i, \ldots\}$. Then the scrambling transformation is applied as in any regular scrambling technique, cf. Figure 4.12. In correspondence, the *scrambling vector* provides redundant bits $S^5 = \{\ldots, (s_2, s_1, s_0)^i,$

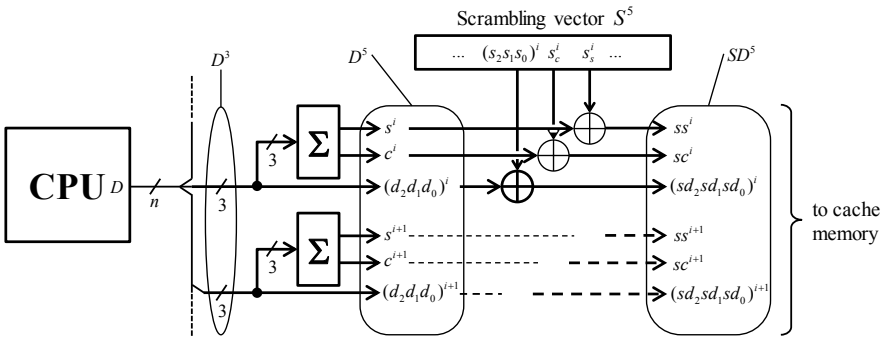


Figure 4.12: Overview of the writing cycle in the IST – eDLC technique.

$s_c^i, s_s^i, \ldots\}$ to protect data redundancy.

These bits s_c^i, s_s^i are generated in such a way that the set of *scrambling vectors* belong to the non-codeword of the data vectors. This can be achieved using the Equations 4.8 which generate a set of 16 non-codewords and is the case proposed for the ISTe when it is applied standalone. This non-codeword set can be reduced to the half if the following equations are applied

$$\begin{aligned}(s_2, s_1, s_0) &= rand() \\ s_c &= not(carry(s_2, s_1, s_0)) = \overline{s_2s_1 + s_2s_0 + s_1s_0} \\ s_s &= not(sum(s_2, s_1, s_0)) = \overline{s_2 \oplus s_1 \oplus s_0}\end{aligned} \tag{4.9}$$

in which now the bit s_c is also calculated from the random bits (s_2, s_1, s_0) and that gives additional protection. Contrary to the intuition, generating s_c^i and s_s^i randomly would otherwise compromise the security. This last facts will be illustrated in the results sub-chapter 4.9.

In the following subsection it is demonstrated that ISTe is not secure enough against DPEMA attacks.

4.8.1 Example of DPEMA attack on ISTe

In a DPEMA attack, differences in the physical magnitude (current consumption/EM radiation power) are correlated to the state of internal bits [96]. With regard to cache memories, the internal bits to be correlated with the physical magnitude are the scrambled data ones, SD^5 (Figure 4.12).

For simplicity and without loss of generality, consider a bus of three bits, i.e. the three data bits plus two redundant bits as illustrated in Figure 4.12. First the adversary prepares data vector sets for the attack (refer to Subsection 4.5.2), the

Table 4.6: Zero and one data vector sets for the DPEMA attack in a three bit bus example.

$D^5(d_2 = 0)$

d_2	d_1	d_0	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0

$D^5(d_1 = 0)$

d_2	d_1	d_0	c	s
0	**0**	0	0	0
0	**0**	1	0	1
1	**0**	0	0	1
1	**0**	1	1	0

$D^5(d_0 = 0)$

d_2	d_1	d_0	c	s
0	0	**0**	0	0
0	1	**0**	0	1
1	0	**0**	0	1
1	1	**0**	1	0

$D^5(d_2 = 1)$

d_2	d_1	d_0	c	s
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$D^5(d_1 = 1)$

d_2	d_1	d_0	c	s
0	**1**	0	0	1
0	**1**	1	1	0
1	**1**	0	1	0
1	**1**	1	1	1

$D^5(d_0 = 1)$

d_2	d_1	d_0	c	s
0	0	**1**	0	1
0	1	**1**	1	0
1	0	**1**	1	0
1	1	**1**	1	1

zero vector set $D^5(d_j = 0)$ and the one vector set $D^5(d_j = 1)$ as shown in Table 4.6. It includes the sets for each one of the data bits.

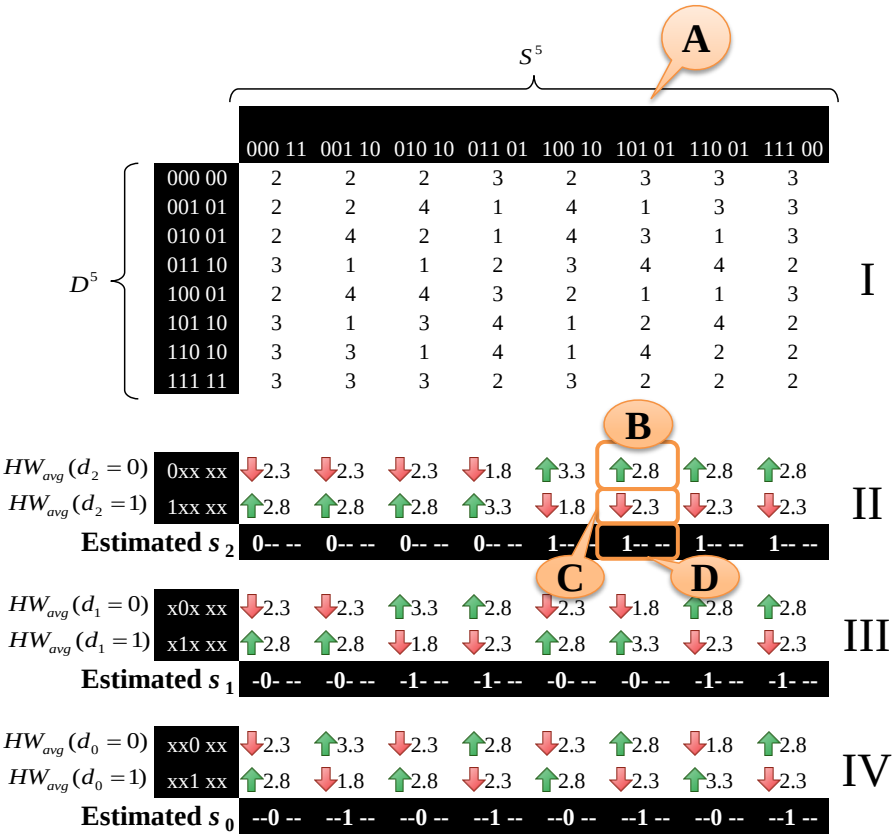
The attack consists in estimating the average *hamming weights* for each one of these sets. The adversary activates the system and, for example, starts by applying the zero set $D^5(d_2 = 0)$ in a continuous loop such that he excites current consumption and EM radiation power at a stable rate. He calculates the average of these physical magnitudes and then estimates the average *hamming weight* $HW_{avg}(d_2 = 0)$. By repeating the same process with the one set, he obtains $HW_{avg}(d_2 = 1)$. Finally, with Equation 4.5 he guesses bit s_2 of the scrambling vector as,

$$s_2^* = \begin{cases} 0, if HW_{avg}(d_2 = 0) < HW_{avg}(d_2 = 1) \\ ?, if HW_{avg}(d_2 = 0) = HW_{avg}(d_2 = 1) \\ 1, if HW_{avg}(d_2 = 0) > HW_{avg}(d_2 = 1) \end{cases}$$

Following the same methodology bits s_1 and s_0 are guessed too, it is unnecessary to do predictions for bits s_c and s_s because they are only part of the error correction scheme.

In Table 4.7 the complete calculations are shown. It is divided in four subtables: (I) has the matrix of hamming weights generated in the data bus after scrambling the data vector D^5 (shown at the left side of the matrix) with all possible scrambling vectors S^5 (at the top side of the matrix), (II) has the average hamming weights $HW_{avg}(d_2 = 0)$ and $HW_{avg}(d_2 = 1)$ generated after running

Table 4.7: Example of DPEMA attack applied to a three bit data bus with ISTe scrambling technique. (I) is the matrix of *hamming weights* generated in the data bus after the scrambling. (II), (III) and (IV) are the average *hamming weights* used to estimate bit s_2, s_1 and s_0 respectively.



the sets $D^5(d_2 = 0)$ and $D^5(d_2 = 1)$ respectively. From these averages the estimation of bit s_2 is made and shown at the bottom of this sub-table. Sub-tables (III) and (IV) show the same information but for bits 1 and 0 of the data and *scrambling vectors* giving the estimations of s_1 and s_0 respectively.

Let's focus on one particular case. Assume that the internal scrambling vector is $S^5 = (101\ 01)$, cf. label **(A)** in the Table. When the CPU sends data vectors $D^5 = \{000\ 00, 001\ 01, ..., 111\ 11\}$ to the scrambling circuit, the hamming weights $HW = \{3, 1, 3, 4, 1, 2, 4, 2\}$ are excited in the data bus, see column under label **(A)**. Based on this fact, the adversary starts exciting the scrambling circuit with the zero data vector set $D^5(d_2 = 0)$ that will excite in the data bus only the hamming weights $\{3, 1, 3, 4\}$. After averaging them the adversary obtains the average hamming weight, $!HW_{avg}(d_2 = 0) = 2.8$, cf. label **(B)**. Next, he excites the scrambling circuit with the one data vector set $D^5(d_2 = 1)$ that will produce in the data bus only the hamming weights $\{1, 2, 4, 2\}$ and obtains the new average hamming weight $HW_{avg}(d_2 = 1) = 2.3$ cf. label **(C)**. Finally, since $HW_{avg}(d_2 = 0) > HW_{avg}(d_2 = 1)$, bit s_2 is correctly estimated as $s_2^* = 1$, according to Equation 4.5, cf. label **(D)**. The estimates of bit s_2 for the rest of scrambling vectors are provided along the same rows. The smallest of the average hamming weights is the one that always determines the corresponding scrambling vector bit, cf. red arrows. Sub-tables III and IV repeat the attack for bits s_1 and s_0. In all cases, the estimate of the *scrambling vector* matches the bits of S^5.

Next, the countermeasure against DPEMA attacks is presented.

4.8.2 DPEMA countermeasure

Random masking ISTe (RM-ISTe) is a strategy for balancing average hamming weights $HW_{avg}(d_j = 0)$ and $HW_{avg}(d_j = 1)$ to make them equal or randomly unequal in order to render Equation 4.5 useless. The circuit schematic of this countermeasure is shown in Figure 4.13 in which the red part shows the modifications to include this *random masking*.

A random bit generator providing bit r is added to the previous design. In fact this generator already exists and only an extra function is added to it. This bit changes randomly after each CPU write cycle. For $r = 0$, the scrambling vector is taken as it is whereas for $r = 1$ the scrambling vector is inverted before scrambling the data vector. Once the scrambled data is stored in the cache, bit r is appended to it. Therefore, the scrambled data vector which is sent to the cache memory is $SD_r^5 = (SD^5, r)$, where bit r equally affects all scrambled data bits.

To better understand how the countermeasure works, Figure 4.14 presents the flowchart of the write cycle including an example on 9 bits. When the write cycle begins, the CPU generates the data vector D^3 that needs to be stored in the L2

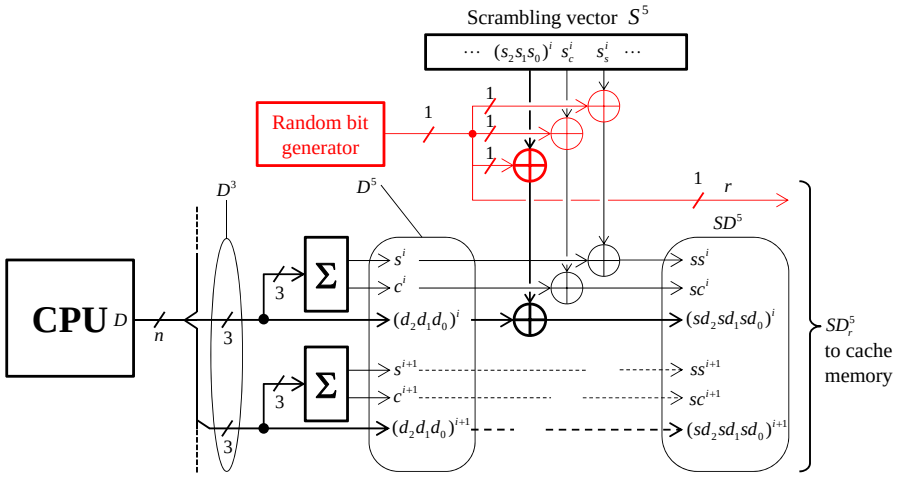


Figure 4.13: Overview of the RM-ISTe technique.

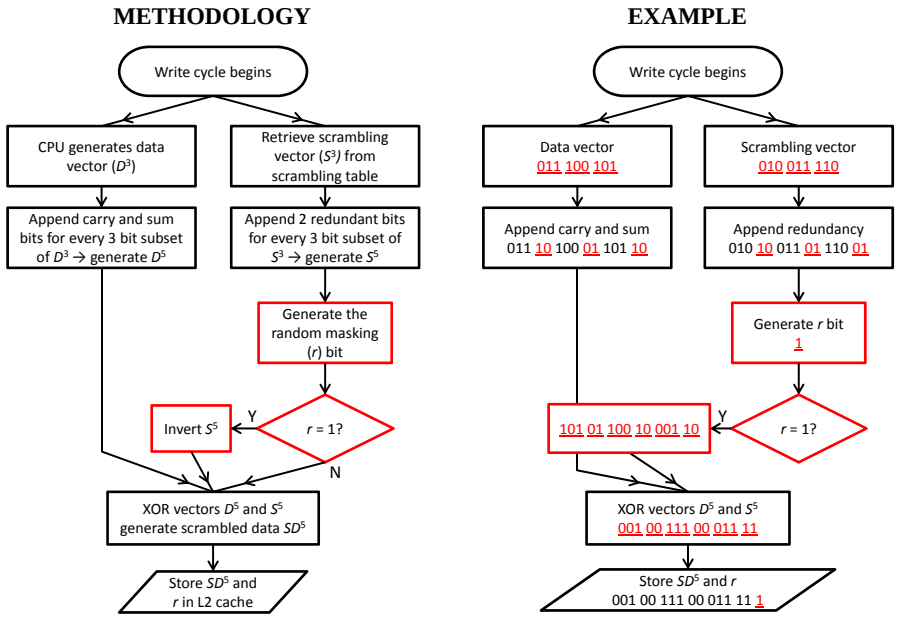


Figure 4.14: Architecture flowchart of the write cycle in the RM-ISTe technique. An example on 9 bits is included.

cache. At the same time, a scrambling vector S^3 is retrieved from the scrambler table. The redundancy is calculated for the data vector (for every 3 bits of data, a carry and sum bit are appended) and D^5 is generated. Similarly, the redundancy for S^3 is computed according to Equation (4.9) and S^5 is generated. Then, the *random masking* bit r is obtained from the random generator and its value is checked. If it is 1, S^5 is inverted and XORed with D^5 generating the scrambled data SD^5 otherwise the inversion of S^5 is not performed. Finally, SD^5 is stored in the cache, together with the random masking bit r.

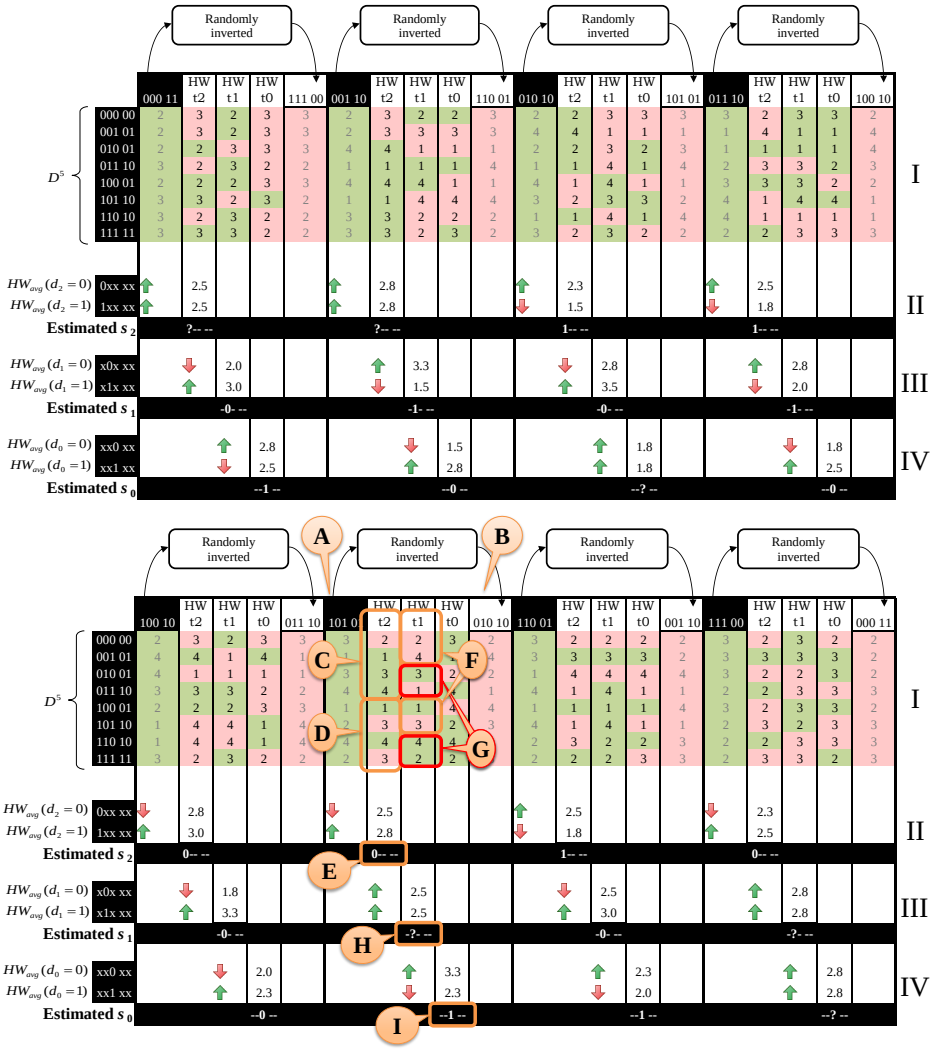
4.8.3 How it works

Table 4.8 presents the same DPEMA attack example as in Table 4.7 but for the RM-ISTe technique. It exemplifies how the countermeasure works and as before it is made on a three bit data bus architecture. Notice that, since the scrambling technique is bit-wise, the conclusions of this example can be extended to any data bus size.

The table is divided in the same four sub-tables (I)–(IV) as before. Sub-table (I) contains the matrix of the *hamming weights* generated in the data bus by the scrambling operations and sub-tables (II) to (IV) have the average *hamming weights* used by the adversary to estimated the scrambling vector bits. Unlike in the previous example (Table 4.7) now the top row of matrix (I) has not only all possible scrambling vectors S^5 for three bits but their inversions $\overline{S^5}$. That is because each time the CPU writes, the data vector D^5 can be randomly scrambled with S^5 or $\overline{S^5}$. A second difference in matrix (I) is that the *hamming weights* are printed in three columns (HW / t2, HW / t1, HW / t0) for each *scrambling vector* S^5. Each of these columns indicate a random selection of the scrambling vector S^5 or $\overline{S^5}$ so at three different time instants ($t2$, $t1$ and $t0$) the *hamming weight* obtained after the scrambling operation can be different. The background color of each number indicates what *scrambling vector* has been used (green S^5) or (pink $\overline{S^5}$).

As before, we will examine the case for the scrambling vector $S^5 = (101\ 01)$, cf. label **(A)** whose inverted value is $\overline{S^5} = (010\ 10)$, cf. label **(B)**. Suppose that the adversary starts, at time instant $t2$, attacking bit s_2 and for this purpose he first excites once the scrambling circuit with the zero set $D^5(d_2 = 0)$. He captures the hamming weights $\{2, 1, 3, 4\}$, cf. label **(C)**, and obtains the average *hamming weight* $HW_{avg}(d_2 = 0) = 2.5$, where three out of four values correspond to the use of S^5 and one to the use of $\overline{S^5}$. Then, he applies the one set $D^5(d_2 = 1)$ and captures the new values for the hamming weights $\{1, 3, 4, 3\}$, cf. label **(D)**, and estimates the average *hamming weight* $HW_{avg}(d_2 = 1) = 2.8$. In this second estimation, two out of four *hamming weights* are generated by S^5 while the other two are generated by $\overline{S^5}$. He finally applies eq. 4.5 and estimates $s_2^* = 0$, cf. label

Table 4.8: DPEMA attack applied to RM-ISTe countermeasure.



(E), which becomes wrong because $s_2 = 1$.

Next, the adversary repeats the procedure to obtain bit s_1, but now when he applies the zero and one sets at time instant $t1$, he will obtain different selections for the S^5 and $\overline{S^5}$ *scrambling vectors*, cf. labels **(F)** and **(G)**. After calculating the averages, he will obtain two equal values, $HW_{avg}(d_1 = 0) = HW_{avg}(d_1 = 1) = 2.5$, and consequently it will not be possible to reliably estimate the corresponding scrambling vector bit $s_1^* =?$, cf. label **(H)**. Finally, by repeating the process at the time step $t0$ for the last bit, the estimation obtained is $s_0^* = 1$, which in this case is correct by chance, cf. label **(I)**.

DPEMA are attacks applied in loops because they benefit from the reduction of noise. Note that, in RM-ISTe coding, if the sets are applied in loops or similarly over thousands of data samples, the estimates of the *hamming weights* will balance each other for the two sets tending to reach the same value, i.e. $HW_{avg}(d_2 = 1) \rightarrow 2.5$ and $HW_{avg}(d_2 = 0) \rightarrow 2.5$. Therefore, *random masking* hides the sensitive information as the attack extends over time.

The metric used in the results section to evaluate countermeasure leakage is reviewed in the following section.

4.9 Evaluation and experimental results

The proposed technique RM-ISTe is evaluated on a virtual implementation of the scrambled cache memory where several switches configure different kinds of countermeasures. The experiments are compared to the previous techniques IST and ISTe presented in Sections 3.5 and **??** [95, 97].

The evaluation of the security is done by calculating the information that leaks through the *hamming weight*. The leakage function $L(s)$ which is based on information entropy [98], evaluates how much close are the estimated scrambling bits s_j^* from the real ones s_j. This function is bounded in the interval $0 \leq L(s) \leq 1$ in which 0 means an always wrong estimation while 1 corresponds to always correct estimation. The derivation of this function is presented in the following subsection.

4.9.1 Leakage function

In information theory, the term entropy refers to Shannon entropy [99], which quantifies the expected value of the information contained in a message and is typically measured in bits, nants (natural logarithms or powers of e) or bans (decimal digits, base 10 logarithms). Basically, it is a measure of the uncertainty in a random variable, the average unpredictability, which is also equivalent to its information content (with the opposite sign). If the information can be represented as

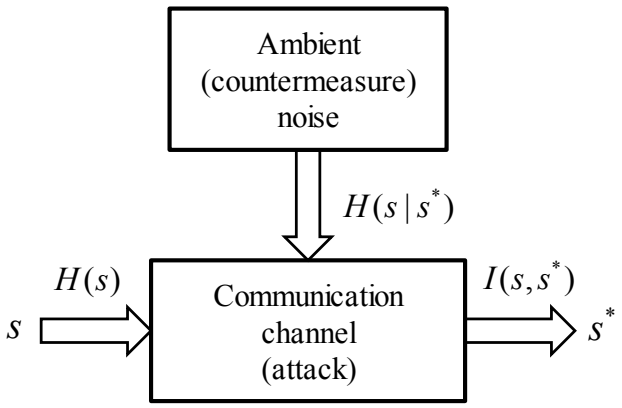


Figure 4.15: Communication channel model used to evaluate the leakage.

a sequence of independent and identically distributed random variables, the limit of Shannon's entropy provides the best lossless encoding or compression. With respect to the proposed model, the entropy can be used to create a metric to evaluate the attack success. The success of an attack relies on the energy dissipated during the attack, therefore the *hamming weight* in the scrambled data. This metric must evaluate how close the adversary is from revealing the original *scrambling vector*. We consider the *hamming weight* as an accurate depiction for the radiated energy since a bit value of 1 symbolizes a transition. Hence, the consumed radiated energy is proportional to the *hamming weight* and this metric will be considered in the analysis for the rest of the chapter.

By leakage it is meant the amount of information extracted from the scrambling vector to produce an accurate estimate of it. Information entropy is a metric for the measurement of the amount of information carried by a set of symbols and is commonly used as an indication of leakage in secure systems. The evaluation of the leakage exploited by an adversary during an attack can be modeled as the communication channel shown in Figure 4.15.

Assume that s is a scrambling vector bit and s^* is its estimate obtained by the attack. The attack can be viewed as a communication channel through which two types of information are sent: signal values (scrambling vector bits) and noise produced by the countermeasures in use. The ability of the attack to separate the signal from the noise will determine the degree of leakage achieved. According to the definitions of communication channels in [57], entropy $H(s)$ is the amount of information contained in scrambling vector bits. Conditional entropy $H(s|s^*)$ is

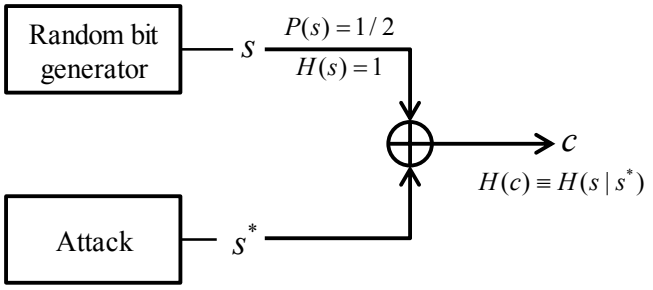


Figure 4.16: Evaluation of entropies in an attack scenario.

the information responsible for errors in the communication channel. In our problem, it represents the noise injected by the countermeasures, which make estimates of s^* more or less correlated with s. Finally, $I(s, s^*)$ in the output channel is the entropy of the bit ensemble s, s^* and represents the amount of information from the input channel s that reaches the output channel s^*. This entropy represents the amount of leakage achieved by the attack and will be renamed as leakage function $L(s)$ or simply leakage. According to [98], this leakage function is evaluated in the channel as

$$L(s) = I(s, s^*) = H(s) - H(s|s^*) \tag{4.10}$$

which represents the balance of information in the channel. The evaluation of entropies $H()$ proceeds as follows.

Scrambling vector bits s are obtained from a random source. Hence, it can be assumed that their probability is $P(s) = P(s = 1) = 1/2$. Since entropy is the average of all information (log function) contained in binary symbols, we have that $H(s) = -[P(s) \cdot log_2 P(s) + P(\overline{s}) \cdot log_2 P(\overline{s})]$, where base 2 log is used, and thus $H(s) = 1$, Figure 4.16. This indicates that scrambling vectors are generated with the maximum amount of information possible, and accordingly have the highest uncertainty.

Conditional entropy $H(s \mid s^*)$ is calculated from the bit probability using the following equation:

$$H(s \mid s^*) = - \sum_{\substack{i=s,\overline{s} \\ j=s^*,\overline{s^*}}} P(i,j) \cdot log_2 P(i \mid j) \tag{4.11}$$

where $P(i, j)$ and $P(i \mid j)$ are the joint and conditional probabilities, respectively. These probabilities can be assessed in real experiments using the XOR

logic function, cf. Figure 4.16. If the probability in one of the inputs of the XOR is $P(s) = 1/2$, then the following symmetries are observed in these probabilities:

$$\begin{aligned} P(\overline{s}, \overline{s}^*) &= P(s, s^*) = P(\overline{c})/2 \\ P(\overline{s}, s^*) &= P(s, \overline{s}^*) = P(c)/2 \\ P(\overline{s} \mid \overline{s}^*) &= P(s \mid s^*) = P(\overline{c}) \\ P(\overline{s} \mid s^*) &= P(s \mid \overline{s}^*) = P(c) \end{aligned} \tag{4.12}$$

whose substitution in Equation 4.11 results in output c entropy of the XOR:

$$H(s \mid s^*) = H(c) = -[P(c) \cdot log_2 P(c) + P(\overline{c}) \cdot log_2 P(\overline{c})] \tag{4.13}$$

By considering Equations 4.10 and 4.13 and taking the boundary conditions in Figure 4.16, the final expression for the leakage function is

$$L(s) = 1 - H(c) \tag{4.14}$$

For an open system, the evaluation of the leakage function starts by collecting the individual bits of the scrambling vector S and estimated scrambling vector S^*. Since the attack will be repeated many times using different (usually random) data vectors, after each one the result of the comparison vector C must be generated, probability $P(c)$ is estimated from its bits and then the leakage function is evaluated. To improve the accuracy of the estimate, several attacks can be performed keeping the same scrambling vector S and then all the bits of the set of vectors C are collected together to estimate the leakage function. Finally, if the strength of the countermeasure needs to be tightly assessed, then the same procedure must be applied but now including some experiments where the scrambling vector is changed too.

In order to illustrate Equation 4.14, a short Monte-Carlo simulation was made for the circuit in Figure 4.16. A total of 2000 random bits were generated for each s and s^* pair and the correlation between them in the $[-1, 1]$ interval was modified. Figure 4.17 summarizes the results of Equation 4.14, including 200 simulations and the theoretical curve derived from Equation 4.10. It is interesting to see that the maximum amount of information is leaked by predicting not only the same bit $s = s^*(cor(s, s^*) = 1)$ but also the inverted one $s = \overline{s^*}(cor(s, s^*) = -1)$.

4.9.2 Results

An ad hoc simulation environment is programmed in C++ which includes the control of the data bus, the scrambling circuit and the cache memory. A wrapper emulates the behavior of the attack in the virtual environment and virtual instruments

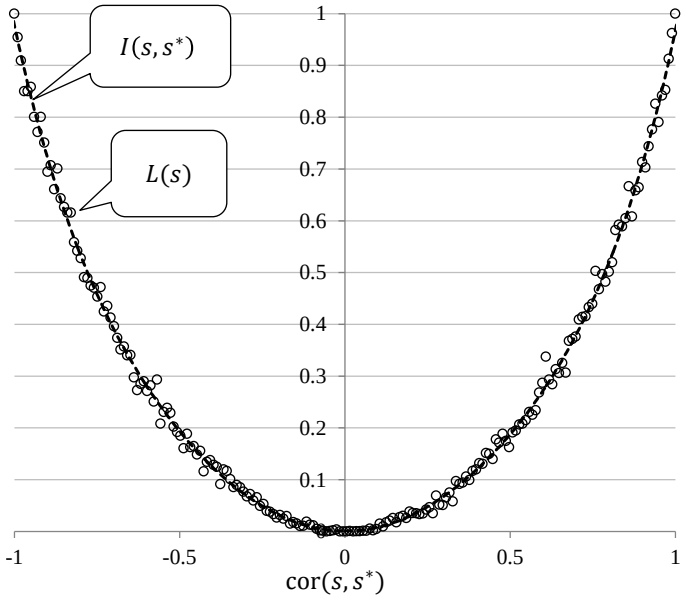


Figure 4.17: Simulation of the circuit in Figure 4.16 and Equation 4.14, and theoretical curve derived from Equation 4.10.

monitor the performance of the different countermeasures. The wrapper also gives us full control over the generation of scrambling vectors and observation of scrambled data. This environment runs on a Supermicro workstation with the following specifications: 64 AMD cores of 64 bits, 256 GB of memory, 6 TB of disc space and with CentOS Linux operating system.

The virtual environment allows the modification of architecture size. Users have full control over the data bus through which bursts of data vectors can be repeatedly sent to carry out attacks. An internal monitor measures the *hamming weight* generated during data transfer to the cache, including transformations made by the scrambling circuit. Monitoring the *hamming weight* as a direct metric for the attack is a conservative way to evaluate the strength of countermeasures because it is an upper bound of the physical measures that the adversary would achieve by the observation of power consumption or EM radiation intensity, as has been explained in previous sub-chapter 4.5. Furthermore, periodical refreshes of *scrambling vectors* that are made by IST and the derived techniques in order to limit the effectiveness of attacks is disabled. In these experiments the same *scrambling vector* is kept in use throughout the attack because we are more interested in evaluation of the random masking scheme strength than in the whole scheme effectiveness.

Therefore results must be understood as an upper bound of the attack success.

The emulated configurations are:

- S3 - IST scrambling technique (Section 3.5) [95].
- S5R5 - ISTe″ SPEMA countermeasure, but where the two redundant bits of the scrambling vector s_c^i, s_s^i are generated randomly. This configuration is presented to illustrate the comment made in Subsection 4.7.1 that full random generation of the redundancy decreases the security instead of increasing it.
- S5R4C1 - ISTe′ SPEMA countermeasure, but where bit s_c^i is generated randomly while s_s^i is calculated according Equation 4.8. This configuration is a middle step between S5R5 and S5R3C2 (Section **??**) [97].
- S5R3C2 - ISTe SPEMA countermeasure as proposed in Equation 4.9 [100]
- S3M - same as S3 but with random masking RM-IST. No figure illustrates explicitly this configuration but consider Figure 4.13, where bits $\{\ldots, c^i, s^i, \ldots\}$ are not added to the data vector and bits $\{\ldots, s_c^i, s_s^i, \ldots\}$ are not generated for the *scrambling vector*. However, a random bit generator flips the content of the *scrambling vector* randomly and bit r is stored in the cache together with scrambled data *SD*.
- S5R5M - Same as S5R5 but with random masking RM-ISTe″.
- S5R4C1M - Same as S5R4C1 but with random masking RM-ISTe′.
- S5R3C2M - Same as S5R3C2 but with random masking, Figure 4.13. This is the full RM-ISTe proposed in this work for defense against SPEMA and DPEMA attacks which renders the minimum leakage.

4.9.2.1 SPEMA attack

In SPEMA, one scrambling vector is set up and then the attack is carried out by generating data vectors and capturing the *hamming weight*. The data vectors producing the maximum *hamming weight*, HW_{mx}, are selected to estimate the *scrambling vector*, cf. Figure 4.13, and leakage is estimated according to Equation 4.14. Because the scrambling technique is linear, data vectors can be generated in such a way that all maximums can be found independently of the number of bits in the configuration. Each data vector is divided into subsets of bits, the HW_{mx} is found for each subset and then combinations of subsets are produced to build the final set of maximums. This attack was simulated for 1000 scrambling vectors selected

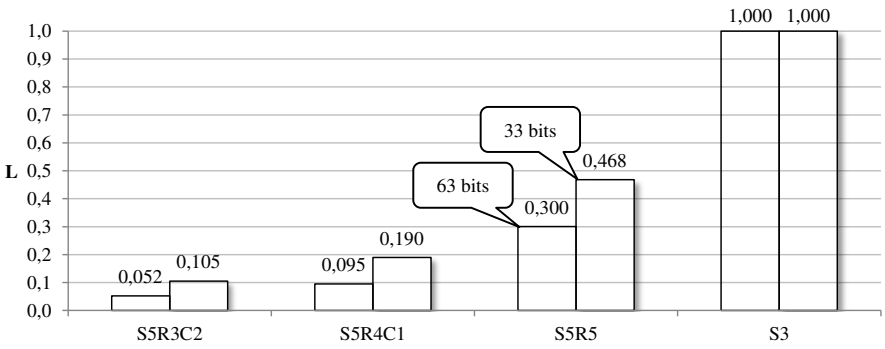


Figure 4.18: Information leakage achieved with SPEMA attacks.

randomly following the rules of each technique. Then, the average information leakage was evaluated. Figure 4.18 shows leakage levels for a 33 and 63-bit architectures for each of the four scrambling configurations in which *random masking* is not used.

The basic scrambling technique S3 was found to be totally vulnerable to SPEMA attack. All attacks for the 1000 scrambling vectors estimated the scrambling vector correctly, and therefore leakage is 1.000. When error detection was applied and boosted by the scrambling technique (ISTe), some protection was found if all scrambling vector bits were generated randomly, including those that scrambled error correction redundancy (S5R5). Although this reduces leakage to 0.300 for the 63-bit architecture, 2 accurate estimates of the scrambling vector can still be found. Nonetheless, in a real attack the adversary has no way to distinguish which of the estimated *scrambling vectors* is the good one, hence the lower leakage. On the other hand, configurations S5R4C1 and S5R3C2 exhibited 0 correct estimations. Therefore, information leakage is significantly lower but not 0 because subsets of bits are still correctly guessed. It should also be noted that S5R3C2, where the scrambling vector redundant bits were not randomly generated, shows the lowest leakage, 0.052, as opposed to 0.095 for S5R4C1.

4.9.2.2 DPEMA attack in non-random-masking techniques

DPEMA attack results are first presented for an architecture of 63 bits. Data vector sets $D^{3/5}(d_i = 0/1)$ used during the attacks are of lengths from 20 to 1000 vectors. The attacks are repeated for 1000 different *scrambling vectors* and the leakage function $L(s)$ is averaged between them. Figure 4.19 plots the results for the techniques without random masking, {S3, S5R5, S5R4C1, S5R3C2}. The x-

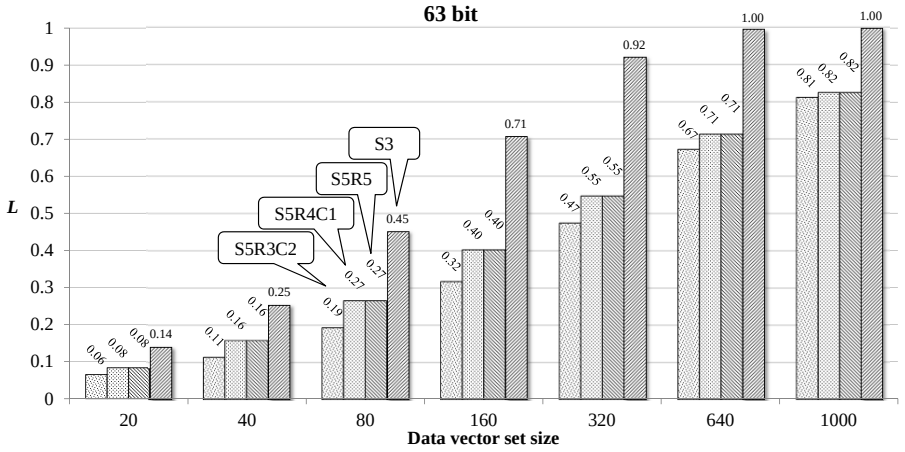


Figure 4.19: Information leakage achieved by DPEMA attack applied on techniques without random masking for a 63-bit architecture.

axis shows the number of vectors in the attacking data set and the y-axis gives the values of the leakage metric $L(s)$.

The low effectiveness of the techniques without random masking against DPEMA is worth nothing. The left column of each group shows that leakage $L(s)$ increases from 0.06 to 0.81 for the most effective technique in this group, S5R3C2, when the data vector set is changed from 20 to 1000 vectors. For the leakage level of 0.81 the number of correctly estimated scrambling vectors is 131 over 1000. The other techniques exhibit a similar trend but with higher leakage levels, giving in the worst case 1 which means that all the estimations were correct. In particular the worse behavior of S5R4C1 and S5R5 is caused by the random generation of the redundancy in the scrambling vectors. The case S3 without SPEMA countermeasure is the one that also presents the poorest results under DPEMA attacks.

Another significant trend observed is that the leakage increases when the attacking data vector set enlarges, independently of the technique used. Notice that for a set of 20 the leakages of the four techniques are $\{0.06, 0.08, 0.08, 0.14\}$ while for a set of 1000 the leakages are $\{0.81, 0.82, 0.82, 1\}$. This is an important information for the adversary because he knows that for a large enough set he can break the system completely independently of the internal countermeasure.

4.9.2.3 DPEMA attacks in random-masking techniques

The previous experiments are repeated but with the techniques using random masking {S3M, S5R5M, S5R4C1M, S5R3C2M}, results are plotted in Figure 4.20.

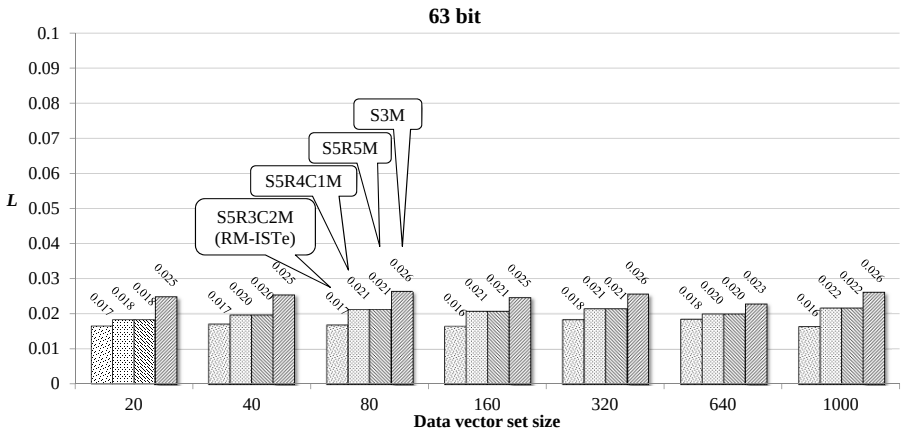


Figure 4.20: Information leakage achieved with DPEMA attacks applied on random masking techniques for 63-bit architecture size.

For the attacking set of 1000 vectors, a significant decrease of the leakage from 0.81 to 0.016 is observed for S5R3C2M (the proposed RM-ISTe technique). It is remarkable that, among all predictions none of the *scrambling vectors* are correct. It is also worth noting that the leakage is approximately the same for all data vector set sizes without following any expected trend. This is particularly interesting because one of the procedures that an adversary can use to drive the attack is to predict the variation of the leakage vs. the number of data vectors applied. In the techniques using *random masking*, the above prediction does not provide any useful feedback. Finally, the combination of techniques designed against SPEMA attacks in [97] provide an additional degree of protection when used with random masking. Compare for the case of 1000 data vectors the case without SPEMA countermeasure (S3M) to the case with SPEMA countermeasure (S5R3C2M), which give leakages of 0.026 and 0.016 respectively.

4.9.2.4 DPEMA attacks for different architecture sizes

Table 4.9 summarizes the results for several architecture sizes going from 9 to 63 bits, for attacks carried out with data vector sets of 1000 vectors. All the numbers are averaged for 1000 attacks using different *scrambling vectors*. First column (Arch. size) indicates the size of the bus. Second column (S3 / IST) lists the leakage for all architectures using the S3 technique, which gives a leakage of 1 in all cases. The next three columns (S5R5 / ISTe$''$, S5R4C1 / ISTe$'$, S5R3C2 / ISTe) correspond to the techniques without *random masking* but including a SPEMA countermeasure. In this group, a maximum leakage of 1 is found for all techniques

Table 4.9: Information leakage measured for DPEMA attacks in different architecture sizes and techniques. Attacking data vector set is 1000. Number of *scrambling vectors* tested is 1000. In gray background cells at least one *scrambling vector* has been correctly estimated.

Arch. size	S3 IST	S5R5 ISTe''		S5R4C1 ISTe'		S5R3C2 ISTe		S3M RM-IST		S5R5M RM-ISTe''		S5R4C1M RM-ISTe'		S5R3C2M RM-ISTe	
9	1	1.000	0.0%	1.000	0.0%	1.000	0.0%	0.198	80.2%	0.142	85.8%	0.142	85.8%	0.110	**89.0%**
12	1	0.999	0.1%	0.999	0.1%	1.000	0.0%	0.147	85.3%	0.108	89.2%	0.108	89.2%	0.088	**91.2%**
15	1	0.997	0.3%	0.997	0.3%	0.999	0.1%	0.121	87.9%	0.087	91.3%	0.087	91.3%	0.065	**93.5%**
18	1	0.993	0.7%	0.993	0.7%	0.997	0.3%	0.090	91.0%	0.070	93.0%	0.070	93.0%	0.054	**94.6%**
21	1	0.990	1.0%	0.990	1.0%	0.993	0.7%	0.080	92.0%	0.057	94.3%	0.057	94.3%	0.050	**95.0%**
24	1	0.984	1.6%	0.984	1.6%	0.986	1.4%	0.064	93.6%	0.052	94.8%	0.052	94.8%	0.046	**95.4%**
27	1	0.973	2.7%	0.973	2.7%	0.981	1.9%	0.055	94.5%	0.047	95.3%	0.047	95.3%	0.036	**96.4%**
30	1	0.963	3.7%	0.963	3.7%	0.974	2.6%	0.059	94.1%	0.043	95.7%	0.043	95.7%	0.037	**96.3%**
33	1	0.952	4.8%	0.952	4.8%	0.957	4.3%	0.050	95.0%	0.039	96.1%	0.039	96.1%	0.033	**96.7%**
36	1	0.930	7.0%	0.930	7.0%	0.945	5.5%	0.048	95.2%	0.034	96.6%	0.034	96.6%	0.033	**96.7%**
39	1	0.928	7.2%	0.928	7.2%	0.928	7.2%	0.045	95.5%	0.033	96.7%	0.033	96.7%	0.029	**97.1%**
42	1	0.914	8.6%	0.914	8.6%	0.921	7.9%	0.039	96.1%	0.031	96.9%	0.031	96.9%	0.025	**97.5%**
45	1	0.895	10.5%	0.895	10.5%	0.903	9.7%	0.037	96.3%	0.027	97.3%	0.027	97.3%	0.025	**97.5%**
48	1	0.883	11.7%	0.883	11.7%	0.889	11.1%	0.035	96.5%	0.027	97.3%	0.027	97.3%	0.023	**97.7%**
51	1	0.872	12.8%	0.872	12.8%	0.883	11.7%	0.033	96.7%	0.024	97.6%	0.024	97.6%	0.021	**97.9%**
54	1	0.850	15.0%	0.850	15.0%	0.859	14.1%	0.032	96.8%	0.024	97.6%	0.024	97.6%	0.021	**97.9%**
57	1	0.838	16.2%	0.838	16.2%	0.845	15.5%	0.029	97.1%	0.023	97.7%	0.023	97.7%	0.021	**97.9%**
60	1	0.828	17.2%	0.828	17.2%	0.821	17.9%	0.028	97.2%	0.023	97.7%	0.023	97.7%	0.018	**98.2%**
63	1	0.815	18.5%	0.815	18.5%	0.813	18.7%	0.026	97.4%	0.021	97.9%	0.021	97.9%	0.018	**98.2%**

in the smallest architectures while minimum leakages are 0.815, 0.815 and 0.813 respectively for the largest architecture, representing a reduction of 18.7%. In all the cases at least one scrambling vector has been estimated correctly, this is shown painting the background in gray shadow.

The following four columns (S3M / RM-IST, S5R5M / RM-ISTe$''$, S5R4C1M / RM-ISTe$'$, S5R3C2M / RM-ISTe) correspond to the techniques with *random masking* and therefore all of them are DPEMA countermeasures. Except the first, the rest also include a SPEMA countermeasure. The maximum leakage is found for the smallest architecture size with 0.198 for (S3M / RM-IST), representing a reduction of 80.2% in leakage while the minimum leakage is found in (S5R3C2M / RM-ISTe) for the biggest architecture size with 0.018 that represents a reduction of 98.2%. In all four cases none of the *scrambling vectors* are correctly estimated for architecture sizes larger than 18 bits, this is shown with the white background of the cells.

4.9.3 Implementation costs

The evaluation of real and optimized implementation costs requires the use of silicon compilers which are costly for the architecture sizes evaluated in this work and are not accessible in our case.

To evaluate the implementation costs we have followed the same strategy as in Chapter 3 ([101][102]) which consists in predicting them with the CACTI tool

[79]. This tool generates cost predictions for cache memory architectures that can be tuned for several parameters including line size, associativity, number of banks, technology nodes, etc. It allows to do space exploration of different alternatives during the design phase.

We consider the following methodology. The particular logic implementation whose costs needs to be evaluated is split in sub-blocks whose architecture needs to resemble as close as possible to a cache memory. Then, different cache memories are dimensioned according to the sub-block parameters and technology and their cost predictions obtained with the CACTI. Finally an artifact is created to combine the predictions of the sub-blocks following the rules of the global design from which the final cost predictions are generated.

In our particular case all the scrambling techniques that we present are based on the IST (Interleaved Scrambling Technique) architecture (Section 3.5 [95]). This technology consists of two main blocks, the L2 cache memory itself and the *scrambler table* which contains sets of *scrambling vectors* that are selected according to certain replacement rules and additional auxiliary registers and flags. The other three main sub-blocks are the *redundancy generator* and *checking code modules*, the *scrambling circuit* and the *random generator*. The extraction of parameters and artifacts for the cost estimation are as follows:

- L2 cache memory – In all IST versions each cache line needs one extra flags. In RM-IST an additional flag is included per word. In ISTe, words are extended with data redundancy.

- *Scrambler vector* table – In IST the size of this table grows as the square root of the L2 cache memory size according to the rules from Chapter 3. In the ISTe cases the scrambling vectors (two per line) of the table are extended with the redundancy. With respect to the effect of the delay (access time) of the cache emulating this table, it is added to the L2 cache memory time as a worst case scenario. With respect to the area and power they are added too.

- Redundancy generator and checking code modules, and scrambling circuit – They grow linearly with the size of the data bus so it is assumed that the memory cache ports (L2 cache and *scrambler table*) properly emulate the cost overhead of these three elements too.

- Random generator – we do not consider the cost of the random generator because it is constant and independent of the architecture size. If implemented as a pseudo-random generator its impact is negligible with respect to the other sub-blocks.

Table 4.10: Implementation costs of the previous and current proposed techniques, 1-way set associative cache.

	Size (KB)		Area (mm^2)		Power consumption(mW)		Access time (ns)	
L2 cache	16		0.1479		94.55		0.3652	
IST - SAU	0.34	2.08%	N/A	N/A	N/A	N/A	N/A	N/A
IST - SAS	0.31	1.90%	N/A	N/A	N/A	N/A	N/A	N/A
ISTe - SAU	11.02	40.78%	0.1207	44.94%	83.95	47.03%	0.3577	49.48%
ISTe - SAS	10.98	40.70%	0.1207	44.94%	83.95	47.03%	0.3577	49.48%
RM - IST - SAU	0.47	2.85%	N/A	N/A	N/A	N/A	N/A	N/A
RM - IST - SAS	0.43	2.62%	N/A	N/A	N/A	N/A	N/A	N/A
RM - ISTe - SAU	11.14	41.05%	0.1219	45.18%	84.45	47.18%	0.359	49.57%
RM - ISTe - SAS	11.1	40.96%	0.1207	44.94%	83.95	47.03%	0.3577	49.48%
L2 cache	32		0.2395		97.28		0.402	
IST - SAU	0.5	1.54%	N/A	N/A	N/A	N/A	N/A	N/A
IST - SAS	0.44	1.36%	N/A	N/A	N/A	N/A	N/A	N/A
ISTe - SAU	21.84	40.56%	0.1914	44.42%	79.05	44.83%	0.39	49.24%
ISTe - SAS	21.78	40.50%	0.1914	44.42%	79.05	44.83%	0.39	49.24%
RM - IST - SAU	0.75	2.29%	0.0245	9.28%	78.26	44.58%	0.2952	42.34%
RM - IST - SAS	0.69	2.11%	0.0241	9.14%	78.39	44.62%	0.2924	42.11%
RM - ISTe - SAU	22.09	40.84%	0.1926	44.57%	79.41	44.94%	0.391	49.31%
RM - ISTe - SAS	22.03	40.77%	0.1926	44.57%	79.41	44.94%	0.391	49.31%
L2 cache	64		0.5493		159.38		0.5015	
IST - SAU	0.72	1.11%	0.0243	4.24%	78.32	32.95%	0.2938	36.94%
IST - SAS	0.62	0.96%	0.0237	4.14%	78.53	33.01%	0.2895	36.60%
ISTe - SAU	43.39	40.40%	0.2941	34.87%	111.03	41.06%	0.4269	45.98%
ISTe - SAS	43.29	40.35%	0.2941	34.87%	111.03	41.06%	0.4269	45.98%
RM - IST - SAU	1.22	1.87%	0.0326	5.60%	81.3	33.78%	0.288	36.48%
RM - IST - SAS	1.12	1.72%	0.0322	5.54%	81.46	33.82%	0.286	36.32%
RM - ISTe - SAU	43.89	40.68%	0.2965	35.06%	111.63	41.19%	0.4279	46.04%
RM - ISTe - SAS	43.79	40.63%	0.2965	35.06%	111.63	41.19%	0.4279	46.04%
L2 cache	128		1.1452		280.19		0.6239	
IST - SAU	1.04	0.81%	0.0318	2.70%	81.54	22.54%	0.2845	31.32%
IST - SAS	0.88	0.68%	0.0253	2.16%	78.07	21.79%	0.3005	32.51%
ISTe - SAU	86.38	40.29%	0.6735	37.03%	183.84	39.62%	0.5441	46.58%
ISTe - SAS	86.22	40.25%	0.6735	37.03%	183.84	39.62%	0.5441	46.58%
RM - IST - SAU	2.04	1.57%	0.0372	3.15%	76.41	21.43%	0.301	32.54%
RM - IST - SAS	1.88	1.45%	0.0372	3.15%	76.41	21.43%	0.301	32.54%
RM - ISTe - SAU	87.38	40.57%	0.6792	37.23%	184.94	39.76%	0.546	46.67%
RM - ISTe - SAS	87.22	40.53%	0.6792	37.23%	184.94	39.76%	0.546	46.67%

Table 4.11: Implementation costs of the previous and current proposed techniques, 1-way set associative cache (continued).

	Size (KB)		Area (mm^2)		Power consumption(mW)		Access time (ns)	
L2 cache	256		1.5605		385.37		0.7614	
IST - SAU	1.49	0.58%	0.0346	2.17%	81.07	17.38%	0.2952	27.94%
IST - SAS	1.27	0.49%	0.033	2.07%	81.24	17.41%	0.2895	27.55%
ISTe - SAU	172.16	40.21%	1.3946	47.19%	318.2	45.23%	0.6665	46.68%
ISTe - SAS	171.94	40.18%	1.3946	47.19%	318.2	45.23%	0.6665	46.68%
RM - IST - SAU	3.49	1.34%	0.0482	3.00%	89.93	18.92%	0.3267	30.02%
RM - IST - SAS	3.27	1.26%	0.0465	2.89%	89.47	18.84%	0.3237	29.83%
RM - ISTe - SAU	174.16	40.49%	1.4059	47.39%	319.92	45.36%	0.6683	46.74%
RM - ISTe - SAS	173.94	40.46%	1.4059	47.39%	319.92	45.36%	0.6683	46.74%
L2 cache	512		3.1233		643.7		0.9832	
IST - SAU	2.16	0.42%	0.0377	1.19%	76.6	10.63%	0.3024	23.52%
IST - SAS	1.8	0.35%	0.0366	1.16%	80.95	11.17%	0.3018	23.49%
ISTe - SAU	343.5	40.15%	2.3881	43.33%	352.77	35.40%	0.9027	47.87%
ISTe - SAS	343.14	40.13%	2.3881	43.33%	352.77	35.40%	0.9027	47.87%
RM - IST - SAU	6.16	1.19%	0.0514	1.62%	51.7	7.43%	0.3162	24.33%
RM - IST - SAS	5.8	1.12%	0.0496	1.56%	50.9	7.33%	0.3137	24.19%
RM - ISTe - SAU	347.5	40.43%	2.4083	43.54%	355.82	35.60%	0.9068	47.98%
RM - ISTe - SAS	347.14	40.41%	2.4083	43.54%	355.82	35.60%	0.9068	47.98%
L2 cache	1024		5.6764		1034.14		1.3061	
IST - SAU	3.1	0.30%	0.042	0.73%	78.11	7.02%	0.3135	19.36%
IST - SAS	2.6	0.25%	0.0397	0.69%	77.32	6.96%	0.3083	19.10%
ISTe - SAU	685.77	40.11%	3.9877	41.26%	779.66	42.98%	1.1008	45.74%
ISTe - SAS	685.27	40.09%	3.9877	41.26%	779.66	42.98%	1.1008	45.74%
RM - IST - SAU	11.1	1.07%	0.1207	2.08%	83.95	7.51%	0.3577	21.50%
RM - IST - SAS	10.6	1.02%	0.1178	2.03%	82.95	7.43%	0.3549	21.37%
RM - ISTe - SAU	693.77	40.39%	4.0274	41.50%	785.4	43.16%	1.1052	45.83%
RM - ISTe - SAS	693.27	40.37%	4.0274	41.50%	785.4	43.16%	1.1052	45.83%
L2 cache	2048		10.5372		1762.71		1.822	
IST - SAU	4.48	0.22%	0.0707	0.67%	75.33	4.10%	0.3275	15.24%
IST - SAS	3.68	0.18%	0.0494	0.47%	90.38	4.88%	0.329	15.30%
ISTe - SAU	1369.82	40.08%	7.5687	41.80%	1339.86	43.19%	1.5311	45.66%
ISTe - SAS	1369.02	40.06%	7.5687	41.80%	1339.86	43.19%	1.5311	45.66%
RM - IST - SAU	20.48	0.99%	0.1836	1.71%	76.92	4.18%	0.3851	17.45%
RM - IST - SAS	19.68	0.95%	0.1719	1.61%	101.66	5.45%	0.3821	17.34%
RM - ISTe - SAU	1385.82	40.36%	8.8219	45.57%	1359.36	43.54%	1.4971	45.11%
RM - ISTe - SAS	1385.02	40.34%	8.8219	45.57%	1359.36	43.54%	1.4971	45.11%

Table 4.12: Implementation costs of the previous and current proposed techniques, 2-way set associative cache.

	Size (KB)		Area (mm^2)		Power consumption(mW)		Access time (ns)	
L2 cache	16		0.1341		127.89		0.5899	
IST - SAU	0.34	2.08%	N/A	N/A	N/A	N/A	N/A	N/A
IST - SAS	0.21	1.30%	N/A	N/A	N/A	N/A	N/A	N/A
ISTe - SAU	11.01	40.76%	0.1207	47.37%	83.95	39.63%	0.3577	37.75%
ISTe - SAS	10.88	40.48%	0.1207	47.37%	83.95	39.63%	0.3577	37.75%
RM - IST - SAU	0.4	2.44%	N/A	N/A	N/A	N/A	N/A	N/A
RM - IST - SAS	0.27	1.66%	N/A	N/A	N/A	N/A	N/A	N/A
RM - ISTe - SAU	11.07	40.89%	0.1207	47.37%	83.95	39.63%	0.3577	37.75%
RM - ISTe - SAS	10.94	40.61%	0.1207	47.37%	83.95	39.63%	0.3577	37.75%
L2 cache	32		0.1949		135.93		0.6297	
IST - SAU	0.49	1.51%	N/A	N/A	N/A	N/A	N/A	N/A
IST - SAS	0.29	0.90%	N/A	N/A	N/A	N/A	N/A	N/A
ISTe - SAU	21.83	40.55%	0.1914	49.55%	79.05	36.77%	0.39	38.25%
ISTe - SAS	21.63	40.33%	0.1902	49.39%	78.7	36.67%	0.389	38.19%
RM - IST - SAU	0.62	1.90%	0.0237	10.84%	78.53	36.62%	0.2895	31.49%
RM - IST - SAS	0.42	1.30%	N/A	N/A	N/A	N/A	N/A	N/A
RM - ISTe - SAU	21.96	40.70%	0.1926	49.70%	79.41	36.88%	0.391	38.31%
RM - ISTe - SAS	21.75	40.47%	0.1914	49.55%	79.05	36.77%	0.39	38.25%
L2 cache	64		0.375		169.09		0.6771	
IST - SAU	0.7	1.08%	0.0241	6.04%	78.39	31.68%	0.2924	30.16%
IST - SAS	0.43	0.67%	N/A	N/A	N/A	N/A	N/A	N/A
ISTe - SAU	43.37	40.39%	0.2941	43.95%	111.03	39.64%	0.4269	38.67%
ISTe - SAS	43.1	40.24%	0.2918	43.76%	110.44	39.51%	0.4259	38.61%
RM - IST - SAU	0.95	1.46%	0.0258	6.44%	78	31.57%	0.3031	30.92%
RM - IST - SAS	0.68	1.05%	0.0241	6.04%	78.39	31.68%	0.2924	30.16%
RM - ISTe - SAU	43.62	40.53%	0.2941	43.95%	111.03	39.64%	0.4269	38.67%
RM - ISTe - SAS	43.35	40.38%	0.2941	43.95%	111.03	39.64%	0.4269	38.67%
L2 cache	128		1.2159		317.97		0.9134	
IST - SAU	1.02	0.79%	0.0314	2.52%	81.63	20.43%	0.2829	23.65%
IST - SAS	0.59	0.46%	0.0235	1.90%	78.61	19.82%	0.288	23.97%
ISTe - SAU	86.36	40.29%	0.6735	35.65%	183.84	36.64%	0.5441	37.33%
ISTe - SAS	85.93	40.17%	0.6735	35.65%	183.84	36.64%	0.5441	37.33%
RM - IST - SAU	1.52	1.17%	0.0346	2.77%	81.07	20.32%	0.2952	24.42%
RM - IST - SAS	1.09	0.84%	0.0318	2.55%	81.54	20.41%	0.2845	23.75%
RM - ISTe - SAU	86.86	40.43%	0.6792	35.84%	184.94	36.77%	0.546	37.41%
RM - ISTe - SAS	86.43	40.31%	0.6735	35.65%	183.84	36.64%	0.5441	37.33%

Table 4.13: Implementation costs of the previous and current proposed techniques, 2-way set associative cache (continued).

	Size (KB)		Area (mm^2)		Power consumption(mW)		Access time (ns)	
L2 cache	256		1.5605		385.37		0.7614	
IST - SAU	1.46	0.57%	0.0342	2.14%	81.1	17.39%	0.2938	27.84%
IST - SAS	0.86	0.33%	0.0251	1.58%	78.11	16.85%	0.2992	28.21%
ISTe - SAU	172.13	40.21%	1.3946	47.19%	318.2	45.23%	0.6665	46.68%
ISTe - SAS	171.53	40.12%	1.3946	47.19%	318.2	45.23%	0.6665	46.68%
RM - IST - SAU	2.46	0.95%	0.0391	2.44%	77.12	16.67%	0.307	28.73%
RM - IST - SAS	1.86	0.72%	0.037	2.32%	80.94	17.36%	0.3031	28.47%
RM - ISTe - SAU	173.13	40.34%	1.4059	47.39%	319.92	45.36%	0.6683	46.74%
RM - ISTe - SAS	172.53	40.26%	1.3946	47.19%	318.2	45.23%	0.6665	46.68%
L2 cache	512		2.324		460.76		1.2126	
IST - SAU	2.12	0.41%	0.0377	1.60%	76.6	14.25%	0.3024	19.96%
IST - SAS	1.21	0.24%	0.033	1.40%	81.24	14.99%	0.2895	19.27%
ISTe - SAU	343.46	40.15%	2.3881	50.68%	352.77	43.36%	0.9027	42.67%
ISTe - SAS	342.55	40.09%	2.3881	50.68%	352.77	43.36%	0.9027	42.67%
RM - IST - SAU	4.12	0.80%	0.0694	2.90%	74.77	13.96%	0.3242	21.10%
RM - IST - SAS	3.21	0.62%	0.0591	2.48%	70.54	13.28%	0.305	20.10%
RM - ISTe - SAU	345.46	40.29%	2.3881	50.68%	352.77	43.36%	0.9027	42.67%
RM - ISTe - SAS	344.55	40.23%	2.3881	50.68%	352.77	43.36%	0.9027	42.67%
L2 cache	1024		5.4559		1291.69		1.5686	
IST - SAU	3.04	0.30%	0.0414	0.75%	77.91	5.69%	0.3122	16.60%
IST - SAS	1.76	0.17%	0.0361	0.66%	80.97	5.90%	0.3005	16.08%
ISTe - SAU	685.71	40.11%	3.9877	42.23%	779.66	37.64%	1.1008	41.24%
ISTe - SAS	684.43	40.06%	3.9791	42.17%	778.4	37.60%	1.0997	41.21%
RM - IST - SAU	7.04	0.68%	0.0914	1.65%	85.28	6.19%	0.3379	17.72%
RM - IST - SAS	5.76	0.56%	0.0493	0.90%	50.77	3.78%	0.313	16.63%
RM - ISTe - SAU	689.71	40.25%	4.0075	42.35%	782.53	37.73%	1.103	41.29%
RM - ISTe - SAS	688.43	40.20%	3.9989	42.29%	781.27	37.69%	1.1019	41.26%
L2 cache	2048		10.119		2235.16		2.1584	
IST - SAU	4.39	0.21%	0.0715	0.70%	75.53	3.27%	0.3271	13.16%
IST - SAS	2.47	0.12%	0.0391	0.38%	77.12	3.34%	0.307	12.45%
ISTe - SAU	1369.73	40.08%	7.5687	42.79%	1339.86	37.48%	1.5311	41.50%
ISTe - SAS	1367.81	40.04%	7.5601	42.76%	1338.6	37.46%	1.5299	41.48%
RM - IST - SAU	12.39	0.60%	0.1293	1.26%	86.99	3.75%	0.366	14.50%
RM - IST - SAS	10.47	0.51%	0.1178	1.15%	82.95	3.58%	0.3549	14.12%
RM - ISTe - SAU	1377.73	40.22%	8.777	46.45%	1353.67	37.72%	1.4931	40.89%
RM - ISTe - SAS	1375.81	40.18%	7.6005	42.89%	1344.41	37.56%	1.5347	41.56%

Table 4.14: Implementation costs of the previous and current proposed techniques, 4-way set associative cache.

	Size (KB)		Area (mm^2)		Power consumption(mW)		Access time (ns)	
L2 cache	16		0.1637		77.65		0.5926	
IST - SAU	0.33	2.02%	N/A	N/A	N/A	N/A	N/A	N/A
IST - SAS	0.14	0.87%	N/A	N/A	N/A	N/A	N/A	N/A
ISTe - SAU	11	40.74%	0.1207	42.44%	83.95	51.95%	0.3577	37.64%
ISTe - SAS	10.81	40.32%	0.1195	42.20%	83.45	51.80%	0.3563	37.55%
RM - IST - SAU	0.36	2.20%	N/A	N/A	N/A	N/A	N/A	N/A
RM - IST - SAS	0.17	1.05%	N/A	N/A	N/A	N/A	N/A	N/A
RM - ISTe - SAU	11.03	40.81%	0.1207	42.44%	83.95	51.95%	0.3577	37.64%
RM - ISTe - SAS	10.84	40.39%	0.1195	42.20%	83.45	51.80%	0.3563	37.55%
L2 cache	32		0.1963		145.99		0.6309	
IST - SAU	0.48	1.48%	N/A	N/A	N/A	N/A	N/A	N/A
IST - SAS	0.2	0.62%	N/A	N/A	N/A	N/A	N/A	N/A
ISTe - SAU	21.82	40.54%	0.1914	49.37%	79.05	35.13%	0.39	38.20%
ISTe - SAS	21.54	40.23%	0.1902	49.21%	78.7	35.03%	0.389	38.14%
RM - IST - SAU	0.55	1.69%	0.0231	10.53%	78.9	35.08%	0.2845	31.08%
RM - IST - SAS	0.26	0.81%	N/A	N/A	N/A	N/A	N/A	N/A
RM - ISTe - SAU	21.88	40.61%	0.1926	49.52%	79.41	35.23%	0.391	38.26%
RM - ISTe - SAS	21.6	40.30%	0.1902	49.21%	78.7	35.03%	0.389	38.14%
L2 cache	64		0.4032		142.59		0.6906	
IST - SAU	0.68	1.05%	0.0241	5.64%	78.39	35.47%	0.2924	29.75%
IST - SAS	0.28	0.44%	N/A	N/A	N/A	N/A	N/A	N/A
ISTe - SAU	43.36	40.39%	0.2941	42.18%	111.03	43.78%	0.4269	38.20%
ISTe - SAS	42.95	40.16%	0.2918	41.99%	110.44	43.65%	0.4259	38.15%
RM - IST - SAU	0.81	1.25%	0.0249	5.82%	78.16	35.41%	0.2979	30.14%
RM - IST - SAS	0.41	0.64%	N/A	N/A	N/A	N/A	N/A	N/A
RM - ISTe - SAU	43.48	40.45%	0.2941	42.18%	111.03	43.78%	0.4269	38.20%
RM - ISTe - SAS	43.08	40.23%	0.2918	41.99%	110.44	43.65%	0.4259	38.15%
L2 cache	128		0.6799		219.4		0.7715	
IST - SAU	0.99	0.77%	0.0314	4.41%	81.63	27.12%	0.2829	26.83%
IST - SAS	0.4	0.31%	N/A	N/A	N/A	N/A	N/A	N/A
ISTe - SAU	86.33	40.28%	0.6735	49.76%	183.84	45.59%	0.5441	41.36%
ISTe - SAS	85.74	40.11%	0.6735	49.76%	183.84	45.59%	0.5441	41.36%
RM - IST - SAU	1.24	0.96%	0.033	4.63%	81.24	27.02%	0.2895	27.29%
RM - IST - SAS	0.65	0.51%	0.0239	3.40%	78.45	26.34%	0.291	27.39%
RM - ISTe - SAU	86.58	40.35%	0.6792	49.97%	184.94	45.74%	0.546	41.44%
RM - ISTe - SAS	85.99	40.18%	0.6735	49.76%	183.84	45.59%	0.5441	41.36%

Table 4.15: Implementation costs of the previous and current proposed techniques, 4-way set associative cache (continued).

	Size (KB)		Area (mm^2)		Power consumption(mW)		Access time (ns)	
L2 cache	256		1.2171		321.63		0.9153	
IST - SAU	1.43	0.56%	0.0342	2.73%	81.1	20.14%	0.2938	24.30%
IST - SAS	0.58	0.23%	0.0235	1.89%	78.61	19.64%	0.288	23.93%
ISTe - SAU	172.1	40.20%	1.3946	53.40%	318.2	49.73%	0.6665	42.14%
ISTe - SAS	171.25	40.08%	1.3946	53.40%	318.2	49.73%	0.6665	42.14%
RM - IST - SAU	1.93	0.75%	0.0374	2.98%	80.93	20.10%	0.3044	24.96%
RM - IST - SAS	1.08	0.42%	0.0318	2.55%	81.54	20.22%	0.2845	23.71%
RM - ISTe - SAU	172.6	40.27%	1.3946	53.40%	318.2	49.73%	0.6665	42.14%
RM - ISTe - SAS	171.75	40.15%	1.3946	53.40%	318.2	49.73%	0.6665	42.14%
L2 cache	512		2.2562		480.72		1.2029	
IST - SAU	2.07	0.40%	0.0377	1.64%	76.6	13.74%	0.3024	20.09%
IST - SAS	0.82	0.16%	0.0251	1.10%	78.11	13.98%	0.2992	19.92%
ISTe - SAU	343.41	40.15%	2.3881	51.42%	352.77	42.32%	0.9027	42.87%
ISTe - SAS	342.16	40.06%	2.3881	51.42%	352.77	42.32%	0.9027	42.87%
RM - IST - SAU	3.07	0.60%	0.058	2.51%	101.87	17.49%	0.3114	20.56%
RM - IST - SAS	1.82	0.35%	0.0366	1.60%	80.95	14.41%	0.3018	20.06%
RM - ISTe - SAU	344.41	40.22%	2.3881	51.42%	352.77	42.32%	0.9027	42.87%
RM - ISTe - SAS	343.16	40.13%	2.3881	51.42%	352.77	42.32%	0.9027	42.87%
L2 cache	1024		4.9412		1054.92		1.5827	
IST - SAU	2.97	0.29%	0.0414	0.83%	77.91	6.88%	0.3122	16.48%
IST - SAS	1.18	0.12%	0.0326	0.66%	81.3	7.16%	0.288	15.40%
ISTe - SAU	685.65	40.10%	3.9877	44.66%	779.66	42.50%	1.1008	41.02%
ISTe - SAS	683.85	40.04%	3.9791	44.61%	778.4	42.46%	1.0997	41.00%
RM - IST - SAU	4.97	0.48%	0.0453	0.91%	48.82	4.42%	0.3044	16.13%
RM - IST - SAS	3.18	0.31%	0.042	0.84%	78.11	6.89%	0.3135	16.53%
RM - ISTe - SAU	687.65	40.17%	3.9989	44.73%	781.27	42.55%	1.1019	41.05%
RM - ISTe - SAS	685.85	40.11%	3.9877	44.66%	779.66	42.50%	1.1008	41.02%
L2 cache	2048		10.1175		2239.42		2.1392	
IST - SAU	4.3	0.21%	0.0704	0.69%	75.16	3.25%	0.3256	13.21%
IST - SAS	1.67	0.08%	0.0357	0.35%	80.99	3.49%	0.2992	12.27%
ISTe - SAU	1369.64	40.08%	7.5687	42.79%	1339.86	37.43%	1.5311	41.72%
ISTe - SAS	1367.01	40.03%	7.5601	42.77%	1338.6	37.41%	1.5299	41.70%
RM - IST - SAU	8.3	0.40%	0.1051	1.03%	78.46	3.38%	0.3422	13.79%
RM - IST - SAS	5.67	0.28%	0.049	0.48%	50.64	2.21%	0.3123	12.74%
RM - ISTe - SAU	1373.64	40.15%	7.5919	42.87%	1343.15	37.49%	1.5335	41.75%
RM - ISTe - SAS	1371.01	40.10%	7.5773	42.82%	1341.12	37.46%	1.5323	41.73%

Results from the CACTI simulation tool are presented in Tables 4.10, 4.11, 4.12, 4.13, 4.14, 4.15. Column (Size (KB)) contains the capacity of the base L2 cache memory and the equivalent size extension of the cache necessary to implement each one of the scrambling techniques. At the right side of each number the increment in percentage is shown with respect to the base L2 cache memory. Eight base L2 cache memory sizes have been considered: 16, 32, 64, 128, 256, 512, 1024 and 2048 KB and in each one of them the four scrambling techniques are implemented {IST, ISTe, RM-IST, RM-ISTe}, but each one having two different addressing schemes: SAU and SAS. We considered 1-way, 2-way and 4-way set associative cache implementations, with a single memory bank.

In the rest of columns three costs are shown: (Area occupied), (Power consumption) and (Access time). All of them are obtained for a technology node of 45 nm. It is remarkable to see that the overheads decrease for larger cache memory sizes, which is caused by the slower increase of the *scrambler table* as indicated above. As denoted in Chapter 3, some values in these tables are "N/A" (i.e. not available) due to the limitations of the CACTI tool (i.e. cache sizes below 0.5 KB).

As expected, the RM-IST techniques have higher area overhead, power consumption and access times than the regular IST solutions because of the additional stored bits. The area overhead is below 7% when cache sizes is above 64 KB, power consumption has a maximum value of 45% for a cache size of 32 KB, but decreases to almost 2% when the cache size is 2048 KB, and the access times also decrease from 43% (32 KB cache, 1-way set associative) to 13% (2048 KB cache, 4-way set associative).

When including the eDLC code to either IST or RM-IST, the overhead increases dramatically due to the high number of redundant bits. All four ISTe and RM-ISTe solutions have almost similar overhead for all three metrics, but the RM-ISTe techniques are a little bit more costly. The results for area overhead are almost the same and they don't follow a specific pattern, they are independent of the cache size, but with some exceptions. Thus, the highest area overhead is 53%, while the lowest is 36% (average value is 44.74%). The overhead for power consumption and access time are follow a similar trend as the area overhead, irregular and independent of the cache size. The highest power consumption percentage reaches 51%, while the lowest is 35% (average value is 41.78%). The highest overhead for access time reaches 49%, while the lowest is 37% (average value is 42.68%).

4.10 Conclusions

In this chapter cold-boot attacks on cache memories boosted by static and differential power and electromagnetic analysis (SPEMA and DPEMA) are consid-

ered. While it is known that scrambling techniques, like the Interleaved Scrambling Technique (IST) can be effective against cold-boot attacks it is demonstrated that a SPEMA or DPEMA can be used to discover the internal *scrambling vector* and consequently to make the cold-boot attack effective, thus breaking the security of IST.

In this chapter new strategies are presented that can be added to the IST making this robust against SPEMA and DPEMA. Detection, correction and localization codes presented in Chapter 2 (eDLC) are extended to design a countermeasure against SPEMA, ISTe. Its advantages are demonstrated against SPEMA attacks and also its limitations against DPEMA attacks. This protection is again extended with random masking (RM) and a complete solution is presented RM-ISTe which becomes effective against cold-boot attacks boosted by SPEMA (static) and DPEMA (dynamic) attacks aiming to discover the internal scrambling vectors. Several examples illustrate the operation of the methodology proposed and experiments are presented to evaluate its effectiveness for different architecture sizes. It is seen that the leakage emanating from the power or electromagnetic radiation is reduced to a 98.2% with respect to the plain IST technique. The cost of the implementation considering a technology node of 45 nm, for a cache memory size of 128 KB, 2-way set associative, is: 35% for the area overhead, 36% for the power consumption overhead and 37% for the access time overhead, respectively.

Chapter 5

Conclusions

Self-healing systems are becoming more and more important due to the numerous error sources and threats, which translates into keeping the information as accurate as possible. Any system with such techniques must have a well defined architecture, capable of monitoring, planning and must adapt to errors and attacks. Self-healing techniques and methods must be implemented to run autonomously, with no human intervention, being capable of modifying its structure and run-time parameters in real-time operation. Implementing security and data privacy in memory systems is difficult due to the variety of threats, different attack models must be considered and analyzed.

Because there are multiple architectural designs for systems, self-healing techniques as well, the final objective/scope of the system is the most important one. For memory systems, the accuracy of the stored data is crucial, because the data is used later on by the processing unit. The self-healing techniques for such systems include error detection and correction codes and are usually used with other replacing methods (using spare components), in order to reduce the computing time and high data redundancy. The monitoring operation is also important (e.g. sensors, background software process), as well as the detection/correction part, which can run when the system is idle or during run-time. Security can be enforced in numerous ways, but it mostly depends on the type of threat which is encountered. For memory systems, information leakage is costly, which denotes that such systems must control the data that is leaked. It is almost impossible to develop a memory system that doesn't leak any data due to the variety of attacks, hence a common countermeasure is to leak non-relevant information.

Any systems that implements self-healing techniques and methods must be evaluated from the performance perspective (the normal run-time operation of the system must not be highly/severely diminished), the additional circuitry must have

a low area overhead and a low power consumption (the latter two are modern trends). The performance of such a system includes processing speed (how fast can it detect and correct errors) and the number of errors that can be detected and corrected (the more, the better). Evaluating the security is troublesome and sensitive, and it's usually done by utilizing attack and threat models and patterns. Because information leakage is worrisome, a simple and efficient metric is to measure how much data and what type of information is leaked by the memory systems. All in all, the evaluations must be compared with existing proposals and solutions.

The design and implementation of the self-healing techniques must be done for the first time in a virtual simulation environment. There are a lot of software solutions that provide a good and complex simulating environment. Fine tuning can be done afterwards (e.g. logic synthesis to adjust the design), in the perspective of obtaining the best results in the considered metrics. Also, because the self-healing concept implies autonomous computing, it is important that the system can execute these techniques and methods during normal run-time operation, with no human intervention. The latter is a necessity when designing and implementing self-healing techniques and methods. For security efficiency and information leakage, attacks can be imitated through several models and a monitoring component can track the outcome.

In the present work, a new family of error detecting and correcting codes are proposed and evaluated from the perspective of speed, code redundancy, area overhead, power consumption and several specific evaluations: error localization, error correction and error escapes. Because of the low delay, the can be considered very efficient for fast cache DRAM memory, where the code overhead is not important. The proposed codes are efficient from almost each point of view evaluated in this work. As regarding the power consumption, the design is less power-hungry in comparison with the original Berger code implementation. In order to achieve a memory system with low-power consumption and EDC/ECC, the error detection and correction can be done either when the memory is idle or inactive, either when a word is read. The operations can be executed in a refresh-time period, with high consistency, at the expense of more check bits. Also, when comparing the values of the partial check bits, if more comparators are used, the detection speed improves with the cost of higher area overhead and power consumption. The operations can also be pipelined so than no time or power is wasted.

Regarding security and data privacy for memory systems, a new technique that uses data scrambling is presented. It is designed to minimize the impact of cold-boot attacks on memories, by scrambling the stored data. Experiments include simulations with the CACTI tool and a FPGA model, while the evaluation process consists of measuring area overhead, power consumption and delay. The results demonstrate that the impact of the proposed technique is low when compared to a

standard scrambling technique.

Against simple and differential power and electromagnetic attacks, two novel techniques are proposed. These strategies are based on a family of error detection and correction codes and merged with a data scrambling technique. The evaluation of the security efficiency is achieved by measuring the amount of leaked information. For this, numerous attacks have been simulated. The implementation costs were assessed by simulating different architectures in the CACTI tool and comparing the results for area overhead, power consumption and access time. The proposed solutions greatly reduce the information leakage, while the overhead for area, power consumption and delay is very small when considering large cache sizes.

In Chapter 1, seven questions related to the objectives of the thesis where formulated. Now, they are answered, in order to see if the objectives are fulfilled.

- Which was the main objective of this thesis?

The main objective of the thesis is to design a self-healing and secure technique for memory systems, which favors low-power consumption.

- Is the main objective accomplished?

Yes, the self-healing technique is achieved through the design and implementation of a new family of error detecting and correcting codes, based on the Berger code. The security of the memory system is accomplished through the design and implementation of several strategies that use data scrambling. Each technique and strategy is tested and evaluated from different points of view.

- How is the self-healing and secure memory system achieved?

The proposed codes are able to detect, localize and correct errors that occur in the memory system. Also, memory repair strategies which include the proposed codes are exhibited, explained and evaluated. The security strategies can mitigate cold-boot and SPEMA/DPEMA attacks, by using data scrambling merged with error detecting and correcting codes. The security methodology is explained in detail and evaluated from several points of view.

- Is the low-power perspective fulfilled?

Yes, the proposed solutions favor a low-power consumption, when compared to standard architectures. All of the proposed strategies and techniques are evaluated from the power consumption point of view.

- How is the security in memories dealt with?

The security in caches is strengthen with 3 separate solutions: one for cold-boot attacks, another for SPEMA and the last for DPEMA. The techniques use data scrambling and error detecting and correcting codes and each solution is evaluated from several points of view: performance and efficiency, information leakage, area overhead, power consumption and delay.

- From the evaluation point of view, can the proposed methodology be considered a good solution?

The evaluations from Chapter 2 show that the proposed codes are good from every point of view (speed, power consumption, etc.), except code redundancy, which is higher than the original Berger code redundancy. Regarding the cache security, the results exhibit that the proposed security methodology handles very well cold-boot, SPEMA and DPEMA attacks, when considering the following metrics: time performance, energy efficiency, information leakage and performance estimation.

- Can the self-healing and secure technique be further researched and used in other systems?

Yes, the self-healing technique can be considered a gateway for future research, for example, in the telecommunications domain or in security for integrated circuits. Other systems that exhibit unidirectional soft errors can employ the proposed codes, while the entire self-healing technique can be used also for permanent errors. The security solutions can be further improved, in order to cope with other types of attacks.

5.1 Scientific contributions

In the next paragraphs, the scientific contributions are presented.

- Chapter 1
  - *Introduction*

A complete analysis of the current situation in self-healing memory systems is realized. The analysis consists of types of errors for memory systems and how do they occur, how self-healing is achieved and implemented in self-healing memory systems, methods and techniques for mitigating errors and how low-power is fulfilled in self-healing memory systems. The security and privacy in memories is

also covered, starting with basic security principles and policies, continuing with an analysis of attack types on memories and ending with memory security techniques and methodologies.

- Chapter 2

  - *Unidirectional Error Detection, Localization and Correction Codes for DRAMs*

The proposed codes for achieving a self-healing low-power memory system are presented. They are basically a modification of the Berger code tree-shaped implementation and gain error localization and error correction (not possible by using the Berger code). Thus, the codes are analyzed from several perspectives, especially error localization, error correction and error escapes. For each perspective, a new metric is defined, explained and evaluated. In this chapter, the proposed codes are implemented using Cadence virtual simulation environment. The codes are tested in several ways, in order to obtain low-power consumption and small delay when generating the check bits. Memory repair strategies are exhibited, as well as how pipelining is achieved, in order to accomplish a self-healing low-power memory system. The evaluation of the proposed codes is presented. The metrics evaluated are the following: speed and delay, power consumption, area overhead, error localization, error correction and error escapes. All of the evaluations are done by exhaustive testing and simulation of all possible error patterns that occur in the memory system. The metrics for error detection, correction and escapes have been defined separately, by creating a C++ program which calculates the metrics by generating all possible values for the information bits and all possible error (random and burst) locations. The contributions of this chapter have been disseminated as follows:

1. **Neagu, M.**, Miclea, L., "Modified Berger Codes for On-Line DRAM Repair Strategies", Proceedings of the 2012 18th IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), pp. 296 – 301, 2012, `https://doi.org/10.1109/AQTR.2012.6237720`

2. **Neagu, M.**, Mois, G., Miclea, L.,"On-Line Error Detection for Tuning Dynamic Frequency Scaling", Proceedings of the 2012 18th IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), pp. 290 – 295, 2012, `https://doi.org/10.1109/AQTR.2012.6237719`

3. **Neagu, M.**, Miclea, L., Figueras, J., "Unidirectional error detection, localization and correction for DRAMs: Application to on-line DRAM repair

strategies", Proceedings of the 2011 IEEE 17th International On-Line Testing Symposium (IOLTS), pp. 264-269, 2011, https://doi.org/10.1109/IOLTS.2011.5994540

4. **Neagu, M.**, Miclea, L.,"On-Line Error Detecting Codes for DRAMs", ACAM Journal: Automation, Computers, Applied Mathematics, Vol. 20, Nr. 2, pp. 133 – 138, 2011, http://acam.tucn.ro/pdf/CA_20%282011%29no2.pdf

- Chapter 3

  – *Interleaved Scrambling Technique*

The Interleaved Scrambling Technique (IST) was proposed to increase the security of data from cache memories, especially protection against cold-boot attacks. This attack can extract data from any type of memory, including caches, but it requires unobstructed physical access to the memory itself. The methodology presented in this chapter hides the plain text data by scrambling them with secret keys, thus making cold-boot attacks useless. If such an attack occurs, the extracted data is scrambled and therefore worthless. The methodology presented in this chapter commences with a short review of previous works, then data scrambling is explained and the statement of the problem is addressed. The following section exhibits the proposed methodology, designed to make cold-boot attacks useless. The main block design is exposed and each block is explained in detail, as well as the operation mode of the design when the read and write cycles occur in the cache memory. The proposed solution is then evaluated from several points of view, including: performance estimation, time performance and energy efficiency. The CACTI tool for cache designs was used to compare several implementations of caches, thus comparing a plain L2 cache with the proposed design. For the same comparison, a FPGA implementation is used and the evaluation metrics are the following: area utilization and overhead in the FPGA device, power consumption in normal operation and delay for data and clock path. The contributions of this chapter have been disseminated as follows:

1. **Neagu, M.**, Miclea, L., Salvador, M., "Improving Security in Cache Memory by Power Efficient Scrambling Technique", IET Computers & Digital Techniques, Volume 9, Issue 6, pp. 283 – 292, 2015, http://dx.doi.org/10.1049/iet-cdt.2014.0030

2. **Neagu, M.**, Miclea, L., Salvador, M., "Interleaved Scrambling Technique: A Novel Low-Power Security Layer for Cache Memories", Proceedings of

the 2014 19th IEEE European Test Symposium (ETS), pp. 241 – 243, 2014, https://doi.org/10.1109/ETS.2014.6847844

3. **Neagu, M.**, Sebestyen, G., "Increasing Memory Security through Information Entropy Models", 15th IEEE International Symposium on Computational Intelligence and Informatics (CINTI), pp. 49-53, Budapest, Hungary, 2014, https://doi.org/10.1109/CINTI.2014.7028727

4. **Neagu, M.**, Miclea, L., "Protecting Cache Memories through Data Scrambling Technique", 10th International Conference on Intelligent Computer Communication and Processing (ICCP), pp. 297 – 303, Cluj-Napoca, Romania, 2014, https://doi.org/10.1109/ICCP.2014.6937012

5. **Neagu, M.**, Miclea, L., "Data Scrambling in Memories: A Security Measure", Proceedings of the 2014 19th IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), pp. 1 – 6, Cluj-Napoca, Romania, 2014, https://doi.org/10.1109/AQTR.2014.6857847

- Chapter 4

  – *Defeating SPEMA and DPEMA*

This chapter contains a security methodology designed for defeating simple and differential power and electromagnetic analysis attacks. These types of attacks are powerful and can target cache memories, but the attacker model must also include a cold-boot attack. SPEMA attacks are easier to overcome, in contrast to DPEMA which are very difficult to defeat. The chapter contains a short review of power and electromagnetic radiation analysis and continues with the proposed solutions for defeating these types of attacks. The methodology for SPEMA is presented and evaluated, while for DPEMA, the latter is improved in order to cope with these type of attacks. For each methodology, a leakage function is defined, which is based on information entropy theory and is used to calculate the leaked information from the scrambled data. Independent C++ programs were designed to evaluate and to provide experimental results for both solutions, while the CACTI tool was used to compare several implementations of cache memories. The contributions of this chapter have been disseminated as follows:

1. **Neagu, M.**, Salvador, M., "Defending cache memory against cold-boot attacks boosted by power or EM radiation analysis", Microelectronics Journal, Volume 62, pp. 85-98, 2017, https://doi.org/10.1016/j.mejo.2017.02.010

2. **Neagu, M.**, Miclea, L., Manich, S., "Defeating Simple Power Analysis Attacks in Cache Memories", in Proceedings of the 2015 30th Conference on Design of Circuits and Integrated Systems (DCIS), pp. 1 – 6, Estoril, Portugal, 2015, https://doi.org/10.1109/DCIS.2015.7388557

3. **Neagu, M.**, Miclea, L., Manich, S., "On the use of error detecting and correcting codes to boost security in caches against side channel attacks", Workshop on Trustworthy Manufacturing and Utilization of Secure Devices", Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition: 9-13 March 2015, Grenoble, France, p. 1-6, 2015, http://upcommons.upc.edu/e-prints/handle/2117/26828

5.2 Future research and developments

The proposed self-healing technique for memory systems can be further researched, because they can be used in other domains, like telecommunications or integrated circuits security.

First of all, the proposed codes from Chapter 2, used in the the self-healing technique, can be further researched, by implementing them in real-time memory systems or FPGAs. Because there are plenty of high-speed circuits and chips in the nowadays electronic technology, the proposed codes can scale very well in modern systems and can have even better results. The proposed codes favor low-power consumption, thus they can be employed in devices with limited power source like handhelds or in the wireless communications domain. The security methodology described in Chapter 3 and 4 can be further improved when considering the nowadays variety of attacks. However, the attacker model must be defined and specified prior to the security design, since this domain is overwhelming and attacks are updated continually.

Bibliography

[1] ITRS. ITRS roadmap of 2013. Technical report, ITRS, 2013. URL http://www.itrs2.net/itrs-reports.html.

[2] Morganm J.P. Payments fraud and control survey, 2015.

[3] F. Paget. Financial fraud and internet banking: Threats and countermeasures. Technical report, McAffee Avert Labs, 2009.

[4] David R Piegdon and L Pimenidis. Hacking in physically addressable memory. In *Seminar of Advanced Exploitation Techniques, WS 2006/2007*, volume 12, 2007.

[5] Charles Slayman. Soft errors—past history and recent discoveries. In *Integrated Reliability Workshop Final Report (IRW), 2010 IEEE International*, pages 25–30. IEEE, 2010.

[6] IBM. Ibm unveils new autonomic computing deployment model. Technical report, IBM, 2001. URL http://people.scs.carleton.ca/~soma/biosec/readings/autonomic_computing.pdf.

[7] IBM. Autonomic computing: Ibm's perspective on the state of information technology. Technical report, IBM, 2002. URL http://www-03.ibm.com/press/us/en/pressrelease/464.wss.

[8] Edward Curry and Paul Grace. Flexible self-management using the model-view-controller pattern. *IEEE software*, 25(3), 2008.

[9] Deepak Halan. Autonomic computing without human intervention, 2015. URL http://electronicsforu.com/technology-trends/autonomic-computing-without-human-intervention.

[10] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[11] Goutam Kumar Saha. Software-implemented self-healing system. *CLEI Electronic Journal*, 10(2), 2007.

[12] Andrew Berns and Sukumar Ghosh. Dissecting self-* properties. In *Self-Adaptive and Self-Organizing Systems, 2009. SASO'09. Third IEEE International Conference on*, pages 10–19. IEEE, 2009.

[13] R.C. Dorf. *The Electrical Engineering Handbook*. Electrical Engineering Handbook. CRC-Press, 1998. ISBN 9780849385742. URL `https://books.google.ro/books?id=VGOFngEACAAJ`.

[14] Mark D Hill. A case for direct-mapped caches. *Computer*, 21(12):25–40, 1988.

[15] Thomas Vogelsang. Understanding the energy consumption of dynamic random access memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 363–374. IEEE Computer Society, 2010.

[16] Nam Sung Kim, Todd Austin, David Baauw, Trevor Mudge, Krisztián Flautner, Jie S Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore's law meets static power. *computer*, 36 (12):68–75, 2003.

[17] David Nguyen, Abhijit Davare, Michael Orshansky, David Chinnery, Brandon Thompson, and Kurt Keutzer. Minimization of dynamic and static power through joint assignment of threshold voltages and sizing optimization. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pages 158–163. ACM, 2003.

[18] Hamid Mahmoodi, Vishy Tirumalashetty, Matthew Cooke, and Kaushik Roy. Ultra low-power clocking scheme using energy recovery and clock gating. *IEEE transactions on very large scale integration (VLSI) systems*, 17(1):33–44, 2009.

[19] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R Hanebutte, and Onur Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 31–40. ACM, 2011.

[20] Keith Harrison and Shouhuai Xu. Protecting cryptographic keys from memory disclosure attacks. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 137–143. IEEE, 2007.

[21] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009. doi: 10.1145/1506409.1506429.

[22] Joo Guan Ooi and Kok Horng Kam. A proof of concept on defending cold boot attack. In *Quality Electronic Design, 2009. ASQED 2009. 1st Asia Symposium on*, pages 330–335, July 2009. doi: 10.1109/ASQED.2009.5206245.

[23] Jingfei Kong and Huiyang Zhou. Improving privacy and lifetime of pcm-based main memory. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 333–342, June 2010. doi: 10.1109/DSN.2010.5544298.

[24] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Architecting against software cache-based side-channel attacks. *IEEE Transactions on Computers*, 62(7):1276–1288, 2013.

[25] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Computer Architecture (ISCA'07), 34th Symposium on*, pages 494–505, 2007.

[26] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. Aes power attack based on induced cache miss and countermeasure. In *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on*, volume 1, pages 586–591. IEEE, 2005.

[27] David Samyde, Sergei Skorobogatov, Ross Anderson, and J-J Quisquater. On a new way to read data from memory. In *Security in Storage Workshop, 2002. Proceedings. First International IEEE*, pages 65–69. IEEE, 2002.

[28] Yuemei He, Haibing Guan, Kai Chen, and Alei Liang. A new software approach to defend against cache-based timing attacks. In *Information Engineering and Computer Science, 2009. ICIECS 2009. International Conference on*, pages 1–4. IEEE, 2009.

[29] Tilo Müller and Michael Spreitzenbarth. Frost: Forensic recovery of scrambled telephones. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ACNS'13, pages 373–388,

Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-38979-5. doi: 10.1007/978-3-642-38980-1_23.

[30] Ludger Borucki, Guenter Schindlbeck, and Charles Slayman. Comparison of accelerated dram soft error rates measured at component and system level. In *Reliability Physics Symposium, 2008. IRPS 2008. IEEE International*, pages 482–487. IEEE, 2008.

[31] Norbert Seifert, P Slankard, M Kirsch, Balaj Narasimham, Victor Zia, Chris Brookreson, A Vo, Subhasish Mitra, Balkaran Gill, and J Maiz. Radiation-induced soft error rates of advanced cmos bulk devices. In *Reliability Physics Symposium Proceedings, 2006. 44th Annual., IEEE International*, pages 217–225. IEEE, 2006.

[32] Charles W Slayman. Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations. *IEEE Transactions on Device and Materials Reliability*, 5(3):397–404, 2005.

[33] Balaji Narasimham and Wing K Luk. A multi-bit error detection scheme for dram using partial sums with parallel counters. In *Reliability Physics Symposium, 2008. IRPS 2008. IEEE International*, pages 202–205. IEEE, 2008.

[34] Shu Lin and Daniel J. Costello. *Error Control Coding, Second Edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004. ISBN 0130426725.

[35] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge University Press, 2003.

[36] Jianwu Zhao and Yibing Shi. A novel approach to improving burst errors correction capability of hamming code. In *Communications, Circuits and Systems, 2007. ICCCAS 2007. International Conference on*, pages 1193–1196. IEEE, 2007.

[37] Elaine Ou and Woodward Yang. Fast error-correcting circuits for fault-tolerant memory. In *Memory Technology, Design and Testing, 2004. Records of the 2004 International Workshop on*, pages 8–12. IEEE, 2004.

[38] Sang-uhn Cha and Hongil Yoon. High speed, minimal area, and low power sec code for drams with large i/o data widths. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 3026–3029. IEEE, 2007.

[39] Chris Wilkerson, Alaa R Alameldeen, Zeshan Chishti, Wei Wu, Dinesh Somasekhar, and Shih-lien Lu. Reducing cache power with low-cost, multi-bit error-correcting codes. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 83–93. ACM, 2010.

[40] Chris Wilkerson, Hongliang Gao, Alaa R Alameldeen, Zeshan Chishti, Muhammad Khellah, and Shih-Lien Lu. Trading off cache capacity for reliability to enable low voltage operation. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 203–214. IEEE, 2008.

[41] Arjan C Dam, Michel GJ Lammertink, Kenneth C Rovers, Johan Slagman, and Arno M Wellink. Hardware/software co-design applied to reed-solomon decoding for the dmb standard. In *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, pages 447–455. IEEE, 2006.

[42] Arshad Ahmed, Naresh R Shanbhag, and Ralf Koetter. An architectural comparison of reed-solomon soft-decoding algorithms. In *Signals, Systems and Computers, 2006. ACSSC'06. Fortieth Asilomar Conference on*, pages 912–916. IEEE, 2006.

[43] Sunwook Rhee, Changgeun Kim, Juhee Kim, and Yong Jee. Concatenated reed-solomon code with hamming code for dram controller. In *Computer Engineering and Applications (ICCEA), 2010 Second International Conference on*, volume 1, pages 291–295. IEEE, 2010.

[44] Jay M Berger. A note on error detection codes for asymmetric channels. *Information and Control*, 4(1):68–73, 1961.

[45] Jaydeep P Kulkarni, Keejong Kim, and Kaushik Roy. A 160 mv robust schmitt trigger based subthreshold sram. *IEEE Journal of Solid-State Circuits*, 42(10):2303–2313, 2007.

[46] Prashant Upadhyay, Rajesh Mehra, and Niveditta Thakur. Low power design of an sram cell for portable devices. In *Computer and Communication Technology (ICCCT), 2010 International Conference on*, pages 255–259. IEEE, 2010.

[47] Hooman Jarollahi and Richard F Hobson. Power and area efficient 5t-sram with improved performance for low-power soc in 65nm cmos. In *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, pages 121–124. IEEE, 2010.

[48] J-P Noel, O Thomas, C Fenouillet-Beranger, M-A Jaud, and A Amara. Robust multi-v t 4t sram cell in 45nm thin box fully-depleted soi technology with ground plane. In *IC Design and Technology, 2009. ICICDT'09. IEEE International Conference on*, pages 191–194. IEEE, 2009.

[49] Igor Loi and Luca Benini. An efficient distributed memory interface for many-core platform with 3d stacked dram. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 99–104. European Design and Automation Association, 2010.

[50] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. Pdram: A hybrid pram and dram main memory system. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pages 664–669. IEEE, 2009.

[51] Antonin Bougerol, Florent Miller, and Nadine Buard. Novel dram mitigation technique. In *On-Line Testing Symposium, 2009. IOLTS 2009. 15th IEEE International*, pages 109–113. IEEE, 2009.

[52] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: a large-scale field study. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 193–204. ACM, 2009.

[53] Charles Slayman. Soft error trends and mitigation techniques in memory devices. In *Reliability and Maintainability Symposium (RAMS), 2011 Proceedings-Annual*, pages 1–5. IEEE, 2011.

[54] Ludger Borucki, Guenter Schindlbeck, and Charles Slayman. Impact of dram process technology on neutron-induced soft errors. In *Integrated Reliability Workshop Final Report, 2007. IRW 2007. IEEE International*, pages 143–146. IEEE, 2007.

[55] F Lima Kastensmidt, Luca Sterpone, Luigi Carro, and M Sonza Reorda. On the optimal design of triple modular redundancy logic for sram-based fpgas. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 2*, pages 1290–1295. IEEE Computer Society, 2005.

[56] Masashi Horiguchi and Kiyoo Itoh. *Nanoscale memory repair*. Springer Science & Business Media, 2011.

[57] Robert Chien. Cyclic decoding procedures for bose-chaudhuri-hocquenghem codes. *IEEE Transactions on information theory*, 10(4):357–363, 1964.

[58] Sandip Kundu and Sudhakar M. Reddy. On symmetric error correcting and all unidirectional error detecting codes. *IEEE transactions on computers*, 39 (6):752–761, 1990.

[59] Chi-Sung Laih and Ching-Nung Yang. On the analysis and design of group theoretical t-syec/aued codes. *IEEE transactions on computers*, 45(1):103–108, 1996.

[60] Sulaiman Al-Bassam and Bella Bose. Asymmetric/unidirectional error correcting and detecting codes. *IEEE Transactions on Computers*, 43(5):590–597, 1994.

[61] Dimitris Nikolos, Nicolas Gaitanis, and George Philokyprou. Systematic t-error correcting all unidirectional error detecting codes. In *Fehlertolerierende Rechensysteme*, pages 177–188. Springer, 1984.

[62] Philipp Ohler and Sybille Hellebrand. Low power embedded drams with high quality error correcting capabilities. In *Test Symposium, 2005. European*, pages 148–153. IEEE, 2005.

[63] Philip G Emma, William R Reohr, and Mesut Meterelliyoz. Rethinking refresh: Increasing availability and reducing power in dram for cache applications. *IEEE micro*, 28(6), 2008.

[64] Abdallah M Saleh, Juan J Serrano, and Janak H Patel. Reliability of scrubbing recovery-techniques for memory systems. *IEEE transactions on reliability*, 39(1):114–122, 1990.

[65] Shubhendu S Mukherjee, Joel Emer, Tryggve Fossum, and Steven K Reinhardt. Cache scrubbing in microprocessors: Myth or necessity? In *Dependable Computing, 2004. Proceedings. 10th IEEE Pacific Rim International Symposium on*, pages 37–42. IEEE, 2004.

[66] William Robert Reohr. Memories: Exploiting them and developing them. In *SOC Conference, 2006 IEEE International*, pages 303–310. IEEE, 2006.

[67] Jin-Fu Li, Tsu-Wei Tseng, and Chih-Sheng Hou. Reliability-enhancement and self-repair schemes for srams with static and dynamic faults. *IEEE Transactions on very large scale integration (vlsi) systems*, 18(9):1361–1366, 2010.

[68] Samuel Evain, Yannick Bonhomme, and Valentin Gherman. Programmable restricted sec codes to mask permanent faults in semiconductor memories.

In *On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International*, pages 147–153. IEEE, 2010.

[69] Vinny Wilson. Analysis and performance evaluation of 1-bit full adder using different topologies. *International Journal of Engineering Research and General Science*, 5(1):196–210, 2017. ISSN 2091-2730.

[70] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Architecting against software cache-based side-channel attacks. *IEEE Transactions on Computers*, 62(7):1276–1288, 2013.

[71] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Architecting against software cache-based side-channel attacks. *IEEE Transactions on Computers*, 62(7):1276–1288, 2013.

[72] Vladimir Rožić, Wim Dehaene, and Ingrid Verbauwhede. Design solutions for securing sram cell against power analysis. In *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*, pages 122–127. IEEE, 2012.

[73] Joanna Rutkowska. Beyond the cpu: Defeating hardware based ram acquisition. *Proceedings of BlackHat DC*, 2007, 2007.

[74] Luca Benini, Angelo Galati, Alberto Macii, Enrico Macii, and Massimo Poncino. Energy-efficient data scrambling on memory-processor interfaces. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pages 26–29. ACM, 2003.

[75] Chih-Jen Hsu, Yuh-Chin Huang, and Mu-Chi Hsu. Scramble circuit to protect data in a read only memory, June 18 2002. US Patent 6,408,073.

[76] K.S. Sainarayanan, J.V.R. Ravindra, C. Raghunandan, and M.B. Srinivas. Coupling aware energy-efficient data scrambling on memory-processor interfaces. In *Industrial and Information Systems, 2007. ICIIS 2007. International Conference on*, pages 421–426, Aug 2007. doi: 10.1109/ICIINFS.2007.4579214.

[77] Eric Brier, Helena Handschuh, and Christophe Tymen. Fast primitives for internal data scrambling in tamper resistant hardware. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 16–27. Springer, 2001.

[78] W. Enck, K. Butler, T. Richardson, P. McDaniel, and A. Smith. Defending against attacks on main memory persistence. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 65–74, Dec 2008. doi: 10.1109/ACSAC.2008.45.

[79] HP. Cacti tool v5.3. URL `http://quid.hpl.hp.com:9081/cacti/`.

[80] Xilinx. Xpower analyzer v14.7. URL `http://www.xilinx.com/products/design_tools/logic_design/verification/xpower.htm`.

[81] Sergei Skorobogatov. Low temperature data remanence in static ram, 2002.

[82] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Protecting data on smartphones and tablets from memory attacks. *ACM SIGPLAN Notices*, 50(4): 177–189, 2015.

[83] Michael Gruhn and Tilo Müller. On the practicability of cold boot attacks. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 390–397. IEEE, 2013.

[84] Ang Cui, Michael Costello, and Salvatore J Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *NDSS*, 2013.

[85] Vincent Rijmen and Joan Daemen. Advanced encryption standard. *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology*, pages 19–22, 2001.

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

[87] Lifeng Su, Albert Martinez, Pierre Guillemin, Sébastien Cerdan, and Renaud Pacalet. Hardware mechanism and performance evaluation of hierarchical page-based memory bus protection. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2009.

[88] William Enck, Kevin Butler, Thomas Richardson, Patrick McDaniel, and Adam Smith. Defending against attacks on main memory persistence. In *2008 Annual Computer Security Applications Conference (ACSAC)*. Institute of Electrical & Electronics Engineers (IEEE), 2008. doi: 10.1109/acsac.2008.45.

[89] Siddhartha Chhabra and Yan Solihin. i-nvmm: a secure non-volatile main memory system with incremental encryption. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 177–188. ACM, 2011.

[90] I. Anati, J. Doweck, G. Gerzon, S. Gueron, and M. Maor. A tweakable encrypion mode for memory encryption with protection against replay attacks, 2012. URL `http://www.google.com/patents/WO2012040679A3?cl=en`. WO Patent App. PCT/US2011/053,170.

[91] S. Gueron, U. Savagaonkar, F.X. Mckeen, C.V. Rozas, D.M. Durham, J. Doweck, O. MULLA, I. Anati, Z. Greenfield, and M. Maor. Method and apparatus for memory encryption with integrity check and protection against replay attacks, 2013. URL `http://www.google.com/patents/WO2013002789A1?cl=pt-PT`. WO Patent App. PCT/US2011/042,413.

[92] Boris Dolgunov and Arseniy Aharonov. Memory randomization for protection against side channel attacks, 2014. US Patent 8,726,040.

[93] R Vijay Sai, S Saravanan, and V Anandkumar. Implementation of a novel data scrambling based security measure in memories for vlsi circuits. *Indian Journal of Science and Technology*, 8, 2015.

[94] Intel. 5th generation intel® core™ processor family, intel® core™ m processor family, mobile intel® pentium® processor family, and mobile intel® celeron® processor family, 2015.

[95] Mădălin-Ioan Neagu, Liviu Miclea, and Salvador Manich. Improving security in cache memory by power efficient scrambling technique. *IET Computers & Digital Techniques*, 9(6):283–292, 2015. doi: 10.1049/iet-cdt.2014.0030.

[96] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1): 5–27, 2011. doi: 10.1007/s13389-011-0006-y.

[97] Mădălin Neagu, Liviu Miclea, and Salvador Manich. Defeating simple power analysis attacks in cache memories. In *Design of Circuits and Integrated Systems (DCIS), 2015 Conference on*, pages 1–6. IEEE, 2015.

[98] Robert M Fano. *The transmission of information*. Massachusetts Institute of Technology, Research Laboratory of Electronics, 1949.

[99] Robert M Gray. *Entropy and information*. Springer, 1990.

[100] Mădălin Neagu and Salvador Manich. Defending cache memory against cold-boot attacks boosted by power or em radiation analysis. *Microelectronics Journal*, 62:85–98, 2017.

[101] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. Raidr: Retention-aware intelligent dram refresh. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 1–12. IEEE Computer Society, 2012.

[102] Panagiota Papavramidou and Michael Nicolaidis. Reducing power dissipation in memory repair for high defect densities. In *2013 18th IEEE European Test Symposium (ETS)*, pages 1–7. IEEE, 2013.