ADDIS ABABA UNIVERSITY
ADDIS ABABA INSTITUTE OF TECHNOLOGY (AAiT)
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

MITIGATION OF MEMORY ERRORS ON COMMODITY WORKSTATIONS

BY
YAFET PHILIPOS

ADVISORS
Dr. FITSUM ASSAMNEW
Dr. SALESSAWI FEREDE

A thesis submitted to the School of Electrical and Computer Engineering in partial fulfillment of the requirements for the Degree of Master of Science in Computer Engineering

JUNE, 2023
ADDIS ABABA, ETHIOPIA

ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

The undersigned have examined the thesis titled:

MITIGATION OF MEMORY ERRORS ON COMMODITY WORKSTATIONS

BY

YAFET PHILIPOS

Approval by Boards of Examiners

Dr. Bisrat Derebssa	_____	_____
Dean, SECE, AAiT	Date	Signature
Dr. Fitsum Assamnew	_____	_____
Advisor	Date	Signature
Dr. Sosina Mengistu	_____	_____
Internal Examiner	Date	Signature
Mr. Getachew Teshome	_____	_____
External Examiner	Date	Signature

Declaration

I, Yafet Philipos Israel, declare that this thesis is my original work. All sources of information in this study have been appropriately acknowledged. I further confirm that this thesis has not been submitted either in part or in full for any other requirements to any other learning institution.

Student Name: Yafet Philipos Israel

Signature: _____

Date: _____

JUNE, 2023

Acknowledgments

Firstly, I want to express my deepest gratitude and thanks to the almighty GOD for all the blessings He has bestowed upon me. His guidance, protection, and love have guided me throughout my life and have helped me reach where I am today.

Secondly, I would like to express my heartfelt appreciation to my advisors Dr. Firstum Assamnew and Dr. Salessawi Ferede for their dedication, support, and advice throughout my academic journey. Their guidance, and encouragement have been invaluable to me, and I cannot express my gratitude enough for their impactful teaching and guidance. They have inspired me to achieve great things, and their mentorship has been crucial to my success.

I would also like to express my sincere gratitude to the AAiT ICT department, especially to Mr. Surafel Berhanu, for providing me with the necessary resources, facilities, and technical assistance that have been instrumental in this work.

In addition, I would like to express my sincere gratitude to all of the Computer Engineering MSc students at AAiT. Their hard work, dedication, and enthusiasm have been an inspiration to me. I have learned so much from them, and I am grateful for the opportunity to have worked with them.

Last but not least, I would like to thank my family for their unwavering support, love, and encouragement during all the ups and downs of my academic and personal life. Their belief in me and constant encouragement, even during the most challenging times, have been a tremendous source of strength and motivation for me. Their endless love, support, and care have taught me more than any lessons in the classroom ever could.

Abstract

Bits stored in Dynamic Random Access Memory (DRAM) could flip at random instances for various reasons such as cosmic ray incidence, electrical noise, and temperature fluctuations. In order to handle these bit-flips, Error Correcting Code (ECC) is integrated in many DRAM modules where such DRAMs are referred to as ECC-DRAM. One commonly used algorithm in ECC to detect double bit-flips and correct single bit-flips is the Single Error Correction Double Error Detection (SECDED) algorithm. However, the SECDED is only available on ECC-DRAMs as such we implemented an optimized version of SECDED to make it suitable on non-ECC devices. On the other hand, in order to increase the number of bit-flip detection capabilities, we proposed a novel approach called hash-based software ECC which uses the hash functions. Hash functions provide robust means to ensure the integrity of data due to their deterministic nature and avalanche effect. After a bit flip is detected through our method, a brute-force approach is used to correct the flipped bit/bits. Our implementation of SECDED is up to 6x faster than the direct implementation of SECDED for 1KB of data. The proposed hash-based software ECC is able to detect any number of bit flips with an adjustable number of bit flip corrections. In this work, the hash-based software ECC is set to correct up to 3-bit flips though it can be tuned to correct any number of flips at a cost of performance overhead. We integrated our approach into an in-memory database and the overhead introduced was found to be less than 3% for bit-flip detection.

Keywords: Dynamic Random Access Memory, Software-based Error Correcting Code, Bit-flip, Hash function

Table of Contents

List of Figures

List of Tables

List of Acronyms

CHF Cryptographic Hash Function

CPU Central Processing Unit

DRAM Dynamic Random Access Memory

ECC Error Correcting Code

ERCIM European Research Consortium for Informatics and Mathematics

MD Message Digest

NCHF Non-Cryptographic Hash Function

NIST National Institute of Standards and Technology

NSA National Security Agency

POPCNT Population count

RAM Random Access Memory

RIPEMD RACE Integrity Primitives Evaluation Message Digest

SECDED Single Error Correction Double Error Detection

SHA Secure Hash Algorithm

SHAKE **S**ecure **H**ash **A**lgorithm and **KE**CCAK

SPN Substitution Permutation Network

SSE4 Streaming SIMD Extensions 4

XOF Extendable-Output Function

Chapter 1

Introduction

1.1 Background

Memory is an essential component of any computing system. They are used to store data, instructions, and other information that are required for the device's operation [1, 2]. There are two types of memory: volatile and non-volatile. Volatile memory is temporary and loses its contents when the power is turned off. Non-volatile memory retains its contents even when the power is turned off.

Dynamic Random Access Memory (DRAM) is a type of volatile memory that has served as the main memory in modern computers. DRAMs offer high-density storage capacity and fast access speeds, making them ideal for use in computers and other devices where large amounts of data need to be stored and accessed quickly. It stores each bit of data in a separate capacitor within an integrated circuit. The capacitor can either be charged or discharged, representing a 1 or 0 respectively and must be periodically refreshed to avoid losing the stored information [2, 3].

Occasionally, errors can occur within DRAM, which can lead to data corruption and system instability. These errors can be broadly classified into two types: hard (permanent) errors and soft (transient) errors [1].

A hard error is a permanent issue with the DRAM module that cannot be corrected through normal means. It is commonly caused by physical damage to the module, a manufacturing defect, or device aging. Since hard errors cannot be corrected, the only solution is to replace the faulty module [4, 5]. Hard errors generally occur infrequently and typically only affect a small number of cells within the DRAM module.

On the other hand, a soft error is a temporary issue resulting from external factors, such as background radiation, electrical noise, or temperature fluctuations. Soft errors are caused by external disturbances that interfere with memory. When a soft error occurs, the affected memory cell may flip its value temporarily, causing data corruption [1, 6, 7, 8, 9, 10].

J. F. Ziegler and W. A. Lanford [9] in their study about the effect of cosmic rays on computer memories stated that as the altitude increases the error rates also increase drastically. W.D. Swift [11] also pointed out that higher altitudes have less shielding provided by the atmosphere, which causes the soft-error rate to rise. Sridharan et al. [12] obtained results that support the statement made above by studying two systems at different altitudes. This suggests that areas with different altitude experience different error rates due to variations in particle strikes.

Soft errors are typically corrected automatically by Error Correcting Code (ECC) built into the DRAM module. ECC memory is designed to handle the most common kinds of errors without user intervention. It replaced parity memory which could only detect, but not correct, memory errors [10, 13]. ECC employs multiple parity bits allocated to bigger data chunks to automatically detect and fix single-bit errors. It uses a 7-bit code that is automatically created for every 64 bits of data that is saved in the Random Access Memory (RAM). The system generates a second 7-bit code after reading the 64 bits of data, which is then compared to the first 7-bit code. By comparing the two 7-bit codes, the system can locate and correct any errors if the two generated codes aren't aligned. If the codes are aligned, the data is error-free [14, 15]. An additional 1-bit can be added to allow the detection of two bit-flips. Such types of ECC are called SECDED. SECDED is capable of detecting and correcting single-bit and double-bit flips.

1.2 Problem Statement

DRAM error refers to an unintentional change in the stored data that results in incorrect, corrupted or lost data. It is important to note that even a single DRAM error can have severe consequences, particularly for mission-critical applications where data integrity is crucial. Some outstanding examples of incidents related to software errors are the addition of extra votes in Belgium on May 18, 2003[1], and the disengagement of the plane from the autopilot mode of the Qantas Flight 72 (Airbus A330) on October 7, 2008[2]. In response, many hardware manufacturers employ various measures to detect and correct such errors, such as redundancy and error correction algorithms.

SECDED is the most common of such algorithms which is integrated on ECC-DRAM of high-end workstations to handle single-bit and double-bit flips. Since this feature is limited to high-end workstations, most commodity workstations are vulnerable due to a lack of ECC-DRAM. Commodity workstations are computer systems that are designed to be used for general-purpose tasks. Commodity workstations are typically less expensive than high-end workstations, but they offer a good balance of performance and price. Thus, they require additional memory protection features to provide reliable service to the user.

In addition, SECDED is limited to the detection of double-bit flips, which can't handle more than two bit-flips. This algorithm can't provide reliable memory protection. Hence, a more optimal approach needs to be considered to provide multiple-bit protection.

To solve the problems stated above, this research attempts to respond to the following research questions:

RQ1 What is the performance improvement of software SECDED implementations when hand-tuned for modern Central Processing Units (CPUs)?

RQ2 What is the performance of software implementation of hash functions for efficient detection of multiple bit-flips?

[1] `https://radiolab.org/podcast/bit-flip`

[2] `https://web.archive.org/web/20190616175818/https://www.atsb.gov.au/media/3532398/ao2008070.pdf`

1.3 Objectives

1.3.1 General Objective

The general objective of this research is to implement an optimized algorithm for bit-flip detection on commodity workstations.

1.3.2 Specific Objectives

The following specific objectives are carried out in order to achieve the general objective of this study:

- Optimize SECDED for applicability on devices on commodity workstations
- Implement performance of hash-based software ECC for the detection of single and multiple-bit flips
- Evaluate the performance of the proposed methods

1.4 Contribution

In this work, three main contributions are made.

- Optimizing the existing SECDED algorithm for efficient performance on non-ECC devices. This optimization aims to increase the reliability and security of these devices by adding another layer of error detection and correction measures.
- Use of Cryptographic Hash Functions (CHFs) and Non-Cryptographic Hash Functions (NCHFs) to detect single and multiple bit flips. This proposed method is referred to as hash-based software ECC which is capable of detecting any number of bit-flips.
- Our proposed hash-based software ECC is integrated into the Redis in-memory key-value store database. The integration is implemented into SET and GET commands of the Redis software. This provided protection against single and multiple-bit flips for the Redis server.

Overall, the contributions aimed to improve the reliability and security of non-ECC devices by optimizing the SECDED algorithm and applying hash functions as error detection mechanisms in the in-memory key-value store database.

1.5 Scope and Limitation

Different algorithms have been proposed to address the limitation of SECDED. In this work, we focus on the optimization of SECDED for its applicability in all commodity workstations.

For the hash functions, we consider some selected functions from both CHF and NCHF. 18 hash function from CHF and 4 hash function from NCHF are selected based on their performance previously presented by researcher [16, 17, 18]. Thus, the experiments are based on these 22 hash functions only.

The hash-based software ECC is integrated into the Redis in-memory key-value database. This integration is applied to the SET and GET commands of Redis. The SET command is used to store key-value data in the database, and the GET command is used to retrieve the stored value using the key. Since these two are important commands to store and retrieve data, the proposed method is integrated with these two functions only.

The algorithms used in this research are mainly focused on protecting the data of a user. The bit-flip protection is guaranteed to be delivered to the data that were encoded with the algorithm. This means the protection does not comprehensively cover all the memory cells, it focuses on protecting encoded data. Also, our brute force bit-flip correction algorithm is tested by correcting up to 3 bit-flips. However, it can be extended when needed, considering the trade-off between performance overhead and an increased number of bit flip detection.

1.6 Methodology

To achieve the research objectives, the following procedures were followed:

I. **Literature Review:** Several relevant studies on bit-flip detection, correction, and hash functions have been reviewed in the literature. To clearly describe and comprehend the breadth of our research, a thorough analysis of the literature on memory error detection and correction was conducted. We have acquired a greater understanding of the subject and are better able to identify the gaps in the research.

   After finishing the review, we began our own research work by using a variety of optimization methods on SECDED and implementing the hash-based software ECC for error detection. We were able to optimize the SECDED technique that works well for non-ECC devices and correct multiple-bit flips with hash-based software ECC.

II. **Algorithms:** The optimization of the SECDED algorithm is aimed to provide ECC feature for all devices that don't have a dedicated ECC-DRAM. To do so, bit manipulation and intrinsic functions are used.

   For the hash functions, different kinds of hash functions from both CHFs and NCHFs are used. CHFs play a vital role in cryptography-related applications as they are designed to provide an added layer of security by generating a fixed size, unique output, or digest that is typically used in verifying the accuracy and authenticity of the data. On the other hand, NCHFs are utilized in various applications that don't require cryptographically secure operations. NCHFs are primarily used to generate a hash value for indexing, searching, and in-memory storage of data, and the computation of checksums. In this work, some of the CHFs and NCHFs are investigated for performance on bit flip detection.

III. **Evaluation Metrics:** The results obtained in the research are evaluated mainly running time of the algorithms due to that fact that hash function with same time complexity can have different execution speed [19]. The running time included both the encoding and decoding execution time of the proposed approach. Encoding of the data refers to taking the original data and addition of some information that can later be used to detect the existence of errors. The decoding refers to checking the data for bit-flip and returning the corrected data if there is an error, else it returns the original data.

The performance of the hash-based software ECC on the Redis database is measured using the benchmark that is shipped with Redis. The number of requests served per second and the latency (response time). The requests served per second refers to the number of requests that Redis can handle in one second. This is a measure of Redis's performance and can be used to determine how well it can handle a given workload. Latency refers to the time it takes for Redis to respond to a request. This is a measure of Redis's responsiveness.

1.7 Organization of the study

The remaining of this document is organized as follows: Chapter two presents fundamental concepts related to SECDED and hash functions, including the operational principles of SECDED, various CHFs and NCHFs specifically relevant to this research. Chapter three comprises a literature review that entails a discussion of SECDED and hash functions while focusing on the approaches implemented for detecting and correcting multiple-bit flip. Chapter four consists of the research methodologies utilized. The results of the analysis and their implications are outlined in Chapter Five, and Chapter Six discusses the conclusions and future work.

Chapter 2

Background

This chapter provides a comprehensive theoretical overview of various algorithms employed for the purpose of detecting and correcting errors in computer memory. Additionally, it encompasses fundamental concepts pertaining to hash functions, including but not limited to Blake, SHA, RIPEMD, MD, and other NCHF functions.

2.1 Memory error correcting codes

In this section, we discussed Hamming code and SECDED memory error detection and correction techniques. A Hamming code is a simple error-correcting code that can detect and correct single-bit errors in a block of data. On the other hand, SECDED is used to correct one error and detect two errors in a given data stream. These codes apply to various technical fields, including computer science, digital design, and information theory. We are interested in its application for memory bit-flip detection and correction. Let's see each of these techniques in further detail.

2.1.1 Hamming code

A Hamming code is a type of error-correcting code that can detect and correct single-bit errors in a data stream. It was invented by Richard W. Hamming in 1950 [20], while he was working at Bell Labs. The idea of hamming code is to add extra bits, known as parity bits, to the original message (data) in a way that allows the receiver to detect if an error has occurred during the transmission of the message. The parity bits are computed based on the positions of the data bits and are placed in specific positions in the message [1, 21, 22].

Parity bits are calculated from the data bits and later used to check for errors. Based on the number of parity bits and data bits, there are different types of hamming codes, one such algorithm is the (7,4) hamming code [23]. In this hamming code, 4 bits of data are encoded to be 7 bits by adding 3 parity bits. In the name (7,4) hamming code, 7 refers to the total data bits and 4 refers to the original data bit before encoding. The positions of the parity bits are chosen based on the power of 2. The bit position of 1,2 and 4 are the position of parity bits, and the rest are for the data bit. In hamming code, the 0^{th} bit position isn't used. These parity bits, which are placed in the mentioned position, are later used to check for the existence of single-bit flip and correct it if detected [24].

For example, a (7,4) Hamming code adds three parity bits to four data bits by using the following steps. The first operation is shifting the data bits to the right, leaving space for parity bits. After this operation, the 4 data bits will have the position 3,5,6 and 7 leaving the bit positions 1,2, and 4 for parity bits. The next step is computing the value of parity bits as follows.

For parity at position 1 [P^1] = Even parity of data bits at 7, 5, and 3-bit position

For parity at position 2 [P^2] = Even parity of data bits at 7, 6, and 3-bit position

For parity at position 4 [P^4] = Even parity of data bits at 7, 6, and 5-bit position

When the data stream is decoded, a new parity bit is computed using the above computation method. The newly computed parity bits are XORed with the previously encoded parity bits. If the result of XOR is 0, it indicated there is no bit-flip. If the result is different from 0, then it indicates the position of the flipped bit (in the case of a single-bit flip). In the latter case, the bit located in the indicated position is flipped, and the corrected data is passed to the next layer.

Let's elaborate on the explanation using binary data. Suppose we want to encode the 4-bit data word "0110". To encode it using the (7,4) hamming code, the data bits are shifted to the left, leaving space for the parity bits. After this operation, the bits become "011$\boxed{P^4}$0$\boxed{P^2}\boxed{P^1}$". The $\boxed{P^x}$ indicates the value of the parity bit at the x position. Next, the value of each parity bit is computed using the above computation. Throughout this research, even a parity approach is used.

P^1 = Even parity (7^{th},5^{th},3^{rd}) = (0,1,0) = 1

P^2 = Even parity (7^{th},6^{th},3^{rd}) = (0,1,0) = 1

P^4 = Even parity (7^{th},6^{th},5^{th}) = (0,1,1) = 0

Finally, the encoded data bits and the parity bits became "011**0**0**11**".

The decoder receives the data on retrieval request and computes new parity bits. Let's say a bit in the 5th bit is flipped. Then the retrieved bits become "01*0***0**0**11**". In this case, the newly computed parity bits calculated as:

P^1 = Even parity (7^{th},5^{th},3^{rd}) = (0,0,0) = 0

P^2 = Even parity (7^{th},6^{th},3^{rd}) = (0,1,0) = 1

P^4 = Even parity (7^{th},6^{th},5^{th}) = (0,1,0) = 1

Now the XOR of the new and the previously encoded parity bit is computed.

Parity check: (encoded parity) XOR (new parity) = (0,1,1) XOR (1,1,0) = **101**

The XOR result is "101" which is 5 in decimal. Using this information, the value in the 5th position is flipped and the data can be forwarded. If the error was a double-bit or multiple-bit flip, the hamming code can not correct or detect the bit-flip.

Hamming codes are very efficient at detecting and correcting single-bit errors. They are also, very simple to implement in hardware. This makes them ideal for use in a variety of applications.

They cannot correct multiple-bit errors. They are not as efficient as some other error correcting codes, such as SECDED, Reed-Solomon codes, and others for correcting multiple-bit errors [21].

In summary, hamming code is a simple yet effective error-correcting code that can be used to detect and correct single-bit errors. It achieves this by adding extra parity bits to the original message. The inability of Hamming code to detect double-bit flips can be addressed by using SECDED.

2.1.2 Single Error Correction Double Error Detection (SECDED)

Single Error Correction Double Error Detection (SECDED) is a type of error-correcting code that can detect and correct single-bit errors, and detect double-bit errors. SECDED codes are based on Hamming codes. SECDED code adds additional parity bits at the 0^{th} bit position of a Hamming code to improve its ability to detect double-bit flips. The process of adding parity bits to data is called Error Correcting Code (ECC). The additional parity bits are used to detect double-bit errors [25, 26, 27]. Since SECDED is an extension of Hamming, all the algorithms and methods used to detect and correct single-bit flips are applied here too. The additional feature is using the 0^{th} bit for the detection. This is achieved through computing the parity for all bits (data bits and parity bits) and storing in the 0^{th} bit. To add the parity bit in the 0^{th} position, both the data and the parity needs to be computed using Hamming code. Once it is done, the even parity of all the bits is computed and stored in the 0^{th} bit. Referring to the discussion in section 2.1.1 we have the encoded bits "0110011" before introducing a single-bit flip. The additional computation added in SECDED is computing the parity for "0110011 $\boxed{P^0}$". It can be done as follows:

$$P^0 = \text{Even parity } (7^{th},6^{th},5^{th},P^4,3^{rd},P^2,P^1) = (0,1,1,0,0,1,1) = 0$$

As a result, the totally encoded bits become "01100110".

The SECDED decoding consists of two comparisons. The first is the XOR computation of the parity bits excluding the 0^{th} bit. The 0^{th} bit computation is executed alone. To determine the bit-flip as a single-bit or double-bit flip, the result from the two operations is required. A more detailed explanation of this is discussed in section 4.1. In the case of double-bit flips, the corrupted data is discarded, and a request for new data is sent.

SECDED codes are very efficient at detecting and correcting errors. They are also very simple to implement in hardware. This makes them ideal for use in a variety of applications, such as digital communications and data storage [28].

In computer memory, SECDEDs are used to protect data from errors that can occur during memory access. SECDEDs are a very important tool for protecting data from errors. They are widely used in a variety of applications, and they have helped to make data storage and access more reliable [27, 29].

2.2 Hash functions

A hash function is a mathematical function that takes input data and returns a fixed-size alphanumeric sequence, known as a digest or hash value. It is a method of converting a large number of messages into a smaller number of message digests [17, 30, 31, 32]. This digest can be used to verify the integrity of data, detect changes or data tampering, and provide secure password storage. Cryptographic Hash Functions (CHFs) and Non-Cryptographic Hash Functions (NCHFs) are two types of hash functions commonly used in computing.

2.2.1 Cryptographic Hash Function (CHF)

Cryptographic Hash Functions (CHFs) are algorithms that convert input data into a fixed-size numerical value, known as a hash or message digest. The hash is unique to each input, meaning that even a small change in the input data results in a change in the hash value.

CHFs are used in a variety of applications, including digital signature schemes, digital certificates, password storage, and secure communication protocols. They provide a robust method for ensuring the integrity of data and verifying its authenticity. The basic features of CHFs include:

1. **Determinism:** refers to producing always the same hash output for the same input.
2. **Pre-image resistant:** refers to generating the original input based on the hash output alone as practically impossible.
3. **Collision-resistant:** refers to finding two messages that generate the same hash output practically impossible.
4. **Infeasible to reverse:** refers to reconstructing the original input from the hash output as computationally infeasible.
5. **Avalanche effect:** refers to producing a large change in the hash output under a minimum change in the input.

In this research, five distinct CHFs were utilized. The deterministic and avalanche effect traits assist in identifying any bit-flip made to the data. Meanwhile, the collision-resistance feature is essential for correcting flipped bits.

2.2.1.1 BLAKE

Blake hash function is a cryptographic hashing algorithm that was initially proposed in 2010 by a group of cryptographers named Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. It is based on the concept of permutation-based compression functions and incorporates the use of several rounds to produce an output that is resistant to preimage attacks and collisions. It is designed in such a way that it provides security for various applications, including digital signature schemes, blockchains, and password authentication. Blake hash function is fast, secure, versatile, and open source [16, 33].

Blake works by iteratively applying a compression function to the data. The compression function takes a 64-byte block of data and produces a 64-byte output. The compression function is composed of 16 rounds, and each round consists of a number of different operations, including addition, subtraction, XOR, and rotations.

The output of the compression function is then XORed with the previous state of the hash function, and the process is repeated for the next block of data. This process is repeated until all the data has been processed.

The final output of the Blake hash function can have 256,224,364 or 512 bits output. Based on the number of output bits, the function is referred to as Blake-256, Blake-224, Blake-364, and Blake-512.

Blake3 is the third version of the Blake family of hash functions that were designed by Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O'Hearn in 2020 [34]. Blake3 is a secure, fast, open source, efficient, parallelizable, memory-efficient, and highly versatile cryptographic hash function.

Blake3 works by iteratively applying a compression function to the data. The compression function takes a 1024-byte block of data and produces a 1024-byte output. The compression function is composed of 17 rounds, and each round consists of a number of different operations, including addition, subtraction, XOR, rotations, and bitwise operations.

The output of the compression function is then XORed with the previous state of the hash function, and the process is repeated for the next block of data. This process is repeated until all the data has been processed.

The final output of the Blake3 hash function is a 256-bit value. This value is unique for each input, and it can be used to verify the integrity of the data.

Blake3 is considered to be a good choice for a variety of applications, including data integrity verification, message authentication, fingerprinting, digital signatures, password hashing, and other cryptographic operations.

This study utilized two different versions of the Blake hash function, namely Blake2b, and Blake3. The Blake2b variant, also commonly referred to as Blake2, has been specifically designed for 64-bit platforms and has the ability to generate a message digest with a size range of 1 to 64 bytes. Blake3 is a more optimized and enhanced version of the Blake2 hash function.

2.2.1.2 Secure Hash Algorithm (SHA)

SHA-1 (Secure Hash Algorithm 1) is a cryptographic hash function that was developed by the United States National Security Agency (NSA) in 1993 [35]. The algorithm uses a series of logical operations such as bit-shifting, bitwise operations, and modular arithmetic to transform the input into a new output. It takes an input message or data and generates a fixed-length 160-bit hash value.

The process of generating a SHA1 starts with padding the input message to ensure it is a multiple of 512 bits. This padding process involves adding zeros followed by a one-bit and then a sequence of zeros until the length of the message is 64 bits less than a multiple of 512 bits.

Next, the padded message is divided into 512-bit blocks, and the SHA1 algorithm processes each block in turn. For each block, the algorithm applies a series of data transformations, including bitwise logical operations, exclusive OR (XOR) operations, and modular addition.

Finally, after all, blocks are processed, the SHA1 hash value is generated by combining the outputs of the previous transformations. The resulting hash value is a fixed-length, 160-bit string that represents the original input message.

SHA-1 is a secure hash function, but it has been shown to be vulnerable to collision attacks. In 2005, a collision attack was published that could find two different files that produced the same SHA-1 hash value [35, 36, 37, 38]. This attack was a significant breakthrough, and it showed that SHA-1 is no longer considered to be secure.

SHA-2 is a family of cryptographic hash functions that were developed by the United States NSA in 2001. It is a successor to the SHA-1 hash function, which has been shown to be vulnerable to collision attacks.

The SHA-2 algorithm begins by pre-processing the input message through several rounds of bit manipulation and padding to ensure that the message has a consistent size and format. This pre-processing step includes adding zeros to fill the block size, appending a 1-bit at the end, adding additional bits to align the last block, and placing the length of the original message.

Next, the algorithm processes the pre-processed message through a series of rounds to mix up the bits and ultimately produce a fixed-length output. Each round uses a different combination of operations, including bitwise XOR, AND, OR, and shifts. Each operation modifies the bits in the message in a deterministic way, such that even a single bit change to the input results in a completely different output. Once all rounds have been completed, the final output is generated [36, 39].

SHA-2 has different versions based on the number of output bits. The most common versions of SHA-2 are SHA-256, SHA-384, SHA-512, and SHA-512/224 [36].

SHA-3 (Secure Hash Algorithm 3), also known as *Keccak* [40] is a cryptographic hash function that is developed by the National Institute of Standards and Technology (NIST) in 2012. The SHA-3 family consists of four cryptographic hash functions, called SHA3-224, SHA3-256, SHA3-384, and SHA3-512, and two Extendable-Output Functions (XOFs), called SHAKE128 and SHAKE256 [41, 42]. The SHAKE hash function is discussed in the next section.

SHA-3 is a permutation-based hash function, which means that it works by applying a series of permutations to the input data. A permutation is a mathematical operation that rearranges the elements of a set. The SHA-3 hash function uses a series of 16 rounds of permutations, and each round consists of a number of different operations, including addition, subtraction, XOR, rotations, and bitwise operations.

The process begins by padding the input message with bits until it reaches a multiple of the block size, which for SHA-3 is 1600 bits. This padded message is then divided into 1600-bit blocks and enters the permutation function.

The permutation function consists of a series of rounds, with each round making use of a non-linear function called a ”theta” function, followed by a ”rho” function that rotates the bits of each lane in the state. This is followed by a ”pi” function that rearranges the lanes of the matrix, and finally, a ”chi” function that performs a bitwise operation on the rows of the matrix.

After the final round, the output is obtained by taking the outermost bits of each lane in the matrix and concatenating them together to form the final hash value [41, 42]. SHA-3 is a secure and collision-resistant hash function, and it is not known to be vulnerable to any attacks.

2.2.1.3 Secure Hash Algorithm and KECCAK (SHAKE)

Secure **H**ash **A**lgorithm and **KE**CCAK (SHAKE) is a cryptographic hash function that was developed as part of the SHA-3 competition [40, 42]. It is a permutation-based hash function, which means that it works by applying a series of permutations to the input data. A permutation is a mathematical operation that rearranges the elements of a set. The SHAKE hash function uses a series of 16 rounds of permutations, and each round consists of a number of different operations, including addition, subtraction, XOR, rotations, and bitwise operations.

The SHAKE hash function works by first padding the input message to a fixed length, using the domain separation technique to differentiate between different applications of the function. Then, the message is divided into blocks, which are processed one after the other using a permutation function. The hash value is computed by passing the input through this permutation function multiple times, with each iteration updating the digest with new random bits generated from the input. The output of the SHAKE hash function is a variable-length value. This value can be 128 or 256 bits [40, 42].

2.2.1.4 RACE Integrity Primitives Evaluation Message Digest (RIPEMD)

RACE Integrity Primitives Evaluation Message Digest (RIPEMD) is a family of cryptographic hash functions that was developed by Hans Dobbertin, Antoon Bosselaers, and Bart Preneel [43] in the European Research Consortium for Informatics and Mathematics (ERCIM) in the early 1990s. The original RIPEMD hash function was designed to be a replacement for the MD4 hash function, which had been shown to be vulnerable to collision attacks [44, 45].

RIPEMD works by taking input data of arbitrary length and producing a fixed-size output known as the message digest. The message digest is a 160-bit value in RIPEMD-160, but other variants of RIPEMD with different digest sizes are also available.

To generate the message digest, RIPEMD processes the input data in blocks of 512 bits. The input data is first padded with a certain pattern to make its length a multiple of 512 bits. This padding includes adding a 64-bit sequence that represents the length of the original data before padding.

Each block is then processed using a series of rounds that involve bitwise operations, modular arithmetic, and logical functions. The rounds use a set of constants and a nonlinear function that operates on the data similarly to the Substitution Permutation Network (SPN) used in other cryptographic algorithms.

During each round, the data is transformed using a combination of shifts, XOR operations, and modular addition with a constant. At the end of each round, the result is passed through the nonlinear function. The output of the last round is the message digest. There are three versions of the RIPEMD hash function, namely RIPEMD-128, RIPEMD-160, and RIPEMD-256.

2.2.1.5 Message Digest (MD)

MD5 (Message Digest Algorithm 5) is a cryptographic hash function that was developed by Ronald Rivest in 1991 [46, 47]. It generates a fixed-length output of 128 bits to represent an input message or file. The process of hashing involves taking an input message and passing it through a series of mathematical operations that transform the message into a unique hash value.

At the heart of the process is a ”compression function” that takes as input a 512-bit block of the message, as well as a 128-bit ”chaining value” that represents the output of the previous block. The compression function then applies multiple rounds of transformation to the block and chaining value to generate a new chaining value for the next block. This process continues for each block of the message until a final 128-bit output is produced. The resulting hash value is unique to the input message, meaning that any changes to the message, no matter how small, results in a completely different hash value.

MD5 was once considered to be secure, but it has been shown to be vulnerable to collision attacks. In 1996, a collision attack was published that could find two different files that produced the same MD5 hash value [48]. This attack was a significant breakthrough, and it showed that MD5 is no longer considered to be secure [44, 45].

In this research, selected versions of CHFs were used. Some hash functions with known vulnerabilities were also included. The reason is that the objective of this research is more focused on the detection of bit flips than security. So, these algorithms could be useful if the overhead they introduce is significantly lower than the robust CHF once.

2.2.2 Non-Cryptographic Hash Function (NCHF)

Non-Cryptographic Hash Functions (NCHFs) are an essential tool that is used in a wide range of applications such as search engines, data comparisons, and integrity checks. These functions take any input data, known as the key, and generate a fixed-size output, or hash, that represents that data. The output is a unique fingerprint of the input data, allowing for quick searches and comparisons in large datasets. Unlike cryptographic hash functions, non-cryptographic hash functions do not offer any security guarantees, but they are designed to be fast and efficient while still providing a low chance of collisions. In the following section, we discussed the four best-performing hash functions. The hash functions are selected based on a performance comparison presented by C. Estébanez et al. [49]. C. Estébanez et al. have evaluated the performance of FNV-1, FNV-1a, APartow, DJBX33A, BuzHash, DEK, BKDR, MurmurHash, lookup3, SuperFastHash functions. The reported result shows Lookup3 hash, Murmur3 hash, Superfast hash, and Buzhash functions are performing well than the rest. Based on this finding we considered using these four NCHFs.

2.2.2.1 Lookup3 hash function

The lookup3 hash function is a non-cryptographic hash function that was created by Bob Jenkins. It is a fast and simple hash function that provides good-quality collision resistance and can be used for hash table lookups, checksums, and error detection. Big companies such as Google, Dreamworks, and Oracle have reportedly integrated Jenkins' "lookup3" into their products. This hash has also become a part of PostgreSQL, Linux, Perl, Ruby, and Infoseek's implementations [49].

It works by initializing three 32-bit variables to random values and then processing the message in 12-byte chunks. Each chunk of the message is processed through a series of operations that mix the bytes together and scramble them thoroughly. In each operation, the variables are updated based on the current bytes being processed and then XORed together to produce the final hash value.

At the end of the message, an additional step is taken to apply a final mixing operation that further scrambles the variables and produces the final hash output. This ensures that subtle changes in the input message results in widely varying hash values, making it difficult for attackers to predict or manipulate the output. The Lookup3 hash function can be used in a variety of programming applications for efficient and effective data storage and retrieval.

2.2.2.2 Murmu3 hash function

Murmur3 is a non-cryptographic hash function that is created by Austin Appleby in 2008. It is a fast and efficient hash function that is well-suited for use in a variety of applications, such as data structures and algorithms. The function possesses remarkable avalanche characteristics and is utilized in essential Open Source projects including Apache Hadoop, Maatkit, and libmemcached [49].

Murmur3 works by breaking the input data into 64-bit chunks. Each chunk is then processed by a compression function, which produces a 32-bit hash value. The hash value is then used as the input for the next chunk, and the process is repeated until all the data has been processed. The compression function is a simple algorithm that consists of a series of mathematical operations, including addition, subtraction, XOR, and rotations. The purpose of the compression function is to create a unique hash value for each input.

The algorithm is designed to be fast and produce low collision rates for a wide range of inputs, making it well-suited for hash table and data structure implementations. It also has good distribution properties that help to avoid the clustering of keys.

2.2.2.3 Superfast hash function

Paul Hsieh designed this hash function with the aim of achieving elegance, exceptional speed, and strong avalanche properties. The development of this function was influenced by the principles used in the FNV and the lookup3 hash function. Due to its effectiveness, it has gained popularity in the software industry. Hsieh mentioned that Apple uses Superfast Hash in its open-source WebKit project, which is utilized in various internet browsers such as Safari and Google Chrome. Additionally, this function was included in numerous releases of Macromedia's Flash Player[49].

Superfast Hash works by breaking the input data into 16-byte chunks. Each chunk is then processed by a compression function, which produces a 32-bit hash value. The hash value is then used as the input for the next chunk, and the process is repeated until all the data has been processed.

Overall, superfast hash functions play an important role in many areas of modern computing, from network security to database management and search indexing. Their ability to quickly generate unique identifiers for data streams makes them a valuable tool for developers and researchers alike.

Superfast Hash is a very fast hash function that is well-suited for use in a variety of applications. It is not as secure as cryptographic hash functions, but it is still considered to be secure for most applications.

2.2.2.4 Buzhash function

In 1992, Robert Uzgalis developed a hash function that is versatile and can be used for various purposes. It works by using a substitution table that replaces each byte of input with a random alias, designed so that half of the aliases have one in every bit position while the other half have zero. This function is capable of handling inputs with any distribution, including those that are highly skewed [49].

Each time a new byte is read in, the calculated hash value is shifted left one bit and the new byte is XORed with the most significant bit. This means that the calculation takes into account both the current and previous byte, as well as their order.

This method of shifting and XORing values is repeated until the entire input data has been processed, resulting in a final hash value that can be used for further computation or comparison.

The Buzhash function is known for its efficiency and effectiveness in processing large sets of data, making it a popular choice for tasks such as data storage, compression, and encryption. Its unique approach to generating hash codes also makes it resistant to certain types of data collisions, ensuring that the results produced by the algorithm are consistently accurate.

2.3 Summary

In general, both cryptographic and non-cryptographic hash functions serve their own purpose in the world of computing. Cryptographic hash functions are primarily used for security-related purposes such as message authentication, digital signatures, and password storage because of their one-way nature and ability to produce a fixed-length output. Non-cryptographic hash functions, on the other hand, are less secure but more efficient and commonly used in applications such as database indexing, key-value stores, and bloom filters.

In this research, we are interested in certain features of these hash functions. The required features are speed, low space overhead, collision resistance and Avalanche effect. There is no algorithm that satisfied this all, so we evaluated the performance for most of the above algorithms.

Chapter 3

Literature Review

In this chapter, a number of previous research works that have been done related to DRAM single-bit and multiple-bit flip detection and correction are discussed. First, we began by reviewing research about the existence and frequency of bit flips. Then, previously proposed methodologies about software-based ECCs are discussed. We also have presented a brief review of the hash functions. Finally, researches that focuses on the Redis in-memory key-value store database are discussed.

3.1 DRAM Errors

The research published by Schroeder et al. [6] analyzed measurements of memory errors in a large fleet of commodity servers over a period of 2.5 years. Those servers had DDR1, DDR2, and FBDIMM memories with the capacity of 1GB 2GB, and 4GB. In their work, both soft and hard errors were logged in the study. Their result shows memory errors aren't rare. The average number of correctable errors recorded per year is about 22,000. Their findings show that the error rate observed is significantly higher than previously reported errors. They also stated that the lifetime of the DRAMs has a direct relation with the DRAM being susceptible to bit-flips

Meza et al. [50] studied memory errors in Facebook's servers. Those servers were using 2GB to 24GB DDR3 DRAMs. The study lasted for 14 months. According to the results, there has been a 1.8-fold increase in failure rates in recent DRAM cells compared to the prior generation. Additionally, they found that the kind of workload can affect memory failure rate by up to 6.5X, indicating that particular memory access patterns may lead to greater errors.

The errors that occurred in DRAMs could be affected based on their operational altitude. Sridharan et al. [12] studied faults and errors of DRAM and SRAM using two leadership class high-performance computer systems which are Hopper, a 6,000-node supercomputer, and Cielo, an 8,500-node supercomputer. Both supercomputers had AMD Opteron CPUs, which holds DDR3-DRAM. The data collection lasted for about three and two years for Hopper and Cielo respectively. The two supercomputers had an altitude difference of 2218m (7277 ft). Cielo is located at an altitude of 2231m (7320ft) and Hopper is located at an altitude of 13m (43ft). The result obtained by comparing the soft error rate of DRAM shows that, on average, the DRAMs situated at Cielo experienced 1.83X more errors than the Hopper supercomputer. This provides insights into the effect of altitude on the DRAM fault rate. They infer stronger DRAM resilience schemes are needed.

These researches show that memory bit flip isn't rare, and give an indication of the need for robust means to mitigate the errors. Besides hardware ECC different software based ECCs are proposed. In the following section, the software bases ECCs are reviewed.

3.2 Software-based ECC

To provide software-based error correction D.H Yoon and M. Erez [51] presented an approach similar to virtual memory management to store and manipulate redundant DRAM storage, which adds flexibility to the memory. The system is designed to dynamically assign a location for redundant data. This allocation could make some or all redundant information share the same physical address space as the data it protects. For this study, they used non-ECC DIMMs and double-Chipkill devices. The benchmarks used are SPEC CPU 2006 and PARSEC suites, which have high memory access demand. For devices with ECC capability, the ECC chip is set to store error detection information. The researchers used a two-tiered memory protection approach. The first tier is used to detect 2 symbol errors[1], meanwhile, the second tier is only needed when an error is detected. For devices with non-ECC capability, the OS is modified to assign storage locations for redundant information. The memory controller performs two access, one to DRAM and the other to the ECC information. In this scheme, the average performance degradation is between 3-9%. In the case of reduced power consumption, the degradation is no more than 10-25%.

[1] A chipkill correct used and it referred to as single symbol-error correcting double symbol-error detecting (SSC-DSD) code. It uses Galois Field (GF) arithmetic.

J. Kim et al. [14] proposed an improvement for ECC named Frugal ECC that includes fine-grained compression called coverage-oriented compression to provide versatile protection. This approach provides trade-offs between performance, efficiency, and reliability. The main focus of the paper was the efficient compression of floating points and truly implementing chip-kill. NAS Parallel Benchmarks and SPLASH2X and SPECFP 2006 suites were used to test the performance on floating points. The proposed method had tiers, the first tire detects single bit flip and the second tier can correct these errors with a 6.25%-redundancy code.

Rink and Castrillón [4] proposed flexible memory error detection by combined data encoding and duplication, called flexMEDiC. The study presents a software-based error detection method with customizable overheads that can be used to find different numbers of memory bit flips. This is achieved via AN encoder and Dual Modular Redundancy (DMR) enhanced back-end method. The result shows that the proposed system can detect up to 5 bits flips within a single data word with 1.55X overhead.

Borchert et al. [52] proposed a software-based error protection approach that is implemented on an Embedded Configurable Operating System (eCos). The system is designed to protect only critical sections of the given data using different modules. The number of bit flips to be detected is adjustable with the expense of redundant data. The result shows very low run-time overhead which ranges from 0.09-1.7% with the benefit of being easily pluggable error detection correction schemes.

These proposed algorithms provide a limited number of protection. To enhance the number of bits protected hash functions are implemented based on their feature to provide data integrity. Hash functions take an input string of any length of data and produce a single, fixed-length string known as the hash value. They are made to preserve the integrity of the message [53]. A publication by J. Aumasson et al. [16] and Jack O Connor et al. [34] presented a different variant of Blake hash functions can be used to provide data integrity with minimal performance overhead. Different versions of SHA hash function is also used to provide data integrity. M Stevens et al. [35] showedSHA-1 are vulnerable to collision attacks with the computational effort of $2^{63.1}$. According to Zulfany E Rasjid et al. [54] the SHA-2 is theoretically susceptible to collision attack, it hasn't been shows practically. Federal Information Processing Standards Publications (FIPS PUBS) published SHA-3 as a new standard in 2015 [41]. ZhaoYong-Xia and Zhen Ge [47] showed MD5 hash function is vulnerable in their analysis.

D Rachmawati et al. [19] used the running time and complexity algorithms to compare MD5 and SHA256 hash function. The research shows the complexity of the two hash functions is $\theta(n)$, the running time experiment showed MD5 has better execution time than the SHA256.

3.3 In-memory key-value database

Yin Li H. Wang et al. [55] presented findings related to memory error correction and in-memory key-value stores with specific case studies. The paper studied how practical it is to add strong memory ECC to the software stack for an in-memory key-value store, without causing a significant decrease in speed. For the software-based ECC, they developed a high-speed parallel BCH[2] code encoding and decoding software-based implementation strategy, which can process multiple bytes at one time. The impact of software-based ECC on the speed performance of Memcached and RAMCloud is measured to be less than 6%.

3.4 Summary

This chapter covers a comprehensive literature review of the existing works that focus on determining the error rate of DRAM. Additionally, the various solutions to counter such errors and their effectiveness are examined. Furthermore, a publication of software-based ECC for in-memory key-value database is discussed.

[2]The BCH (Bose–Chaudhuri–Hocquenghem) codes are a type of cyclic error-correcting codes that use polynomials over a finite field (sometimes known as Galois field) to be built.

Chapter 4

Methodology

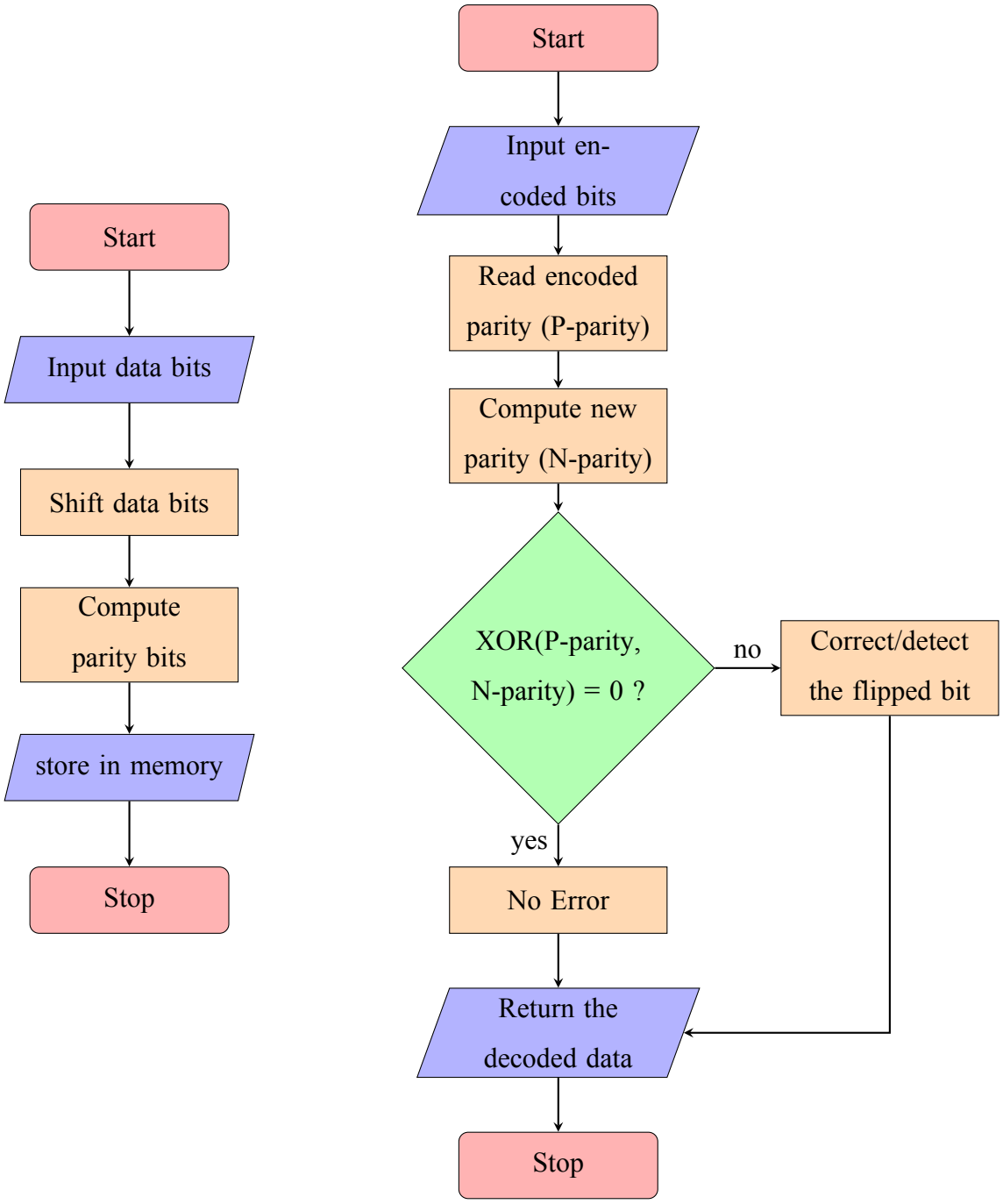
In this chapter, the implemented methodology for the detection and correction of memory bit-flips is discussed. Initially, the optimization techniques applied to the SECDED algorithm are discussed. Next, the detection of multiple-bit flip using hash-based software ECC, and the brute-force error correction algorithm is discussed. The algorithm used to generate bit-flip and the integration of the hash-based software ECC to the Redis in-memory key-value store database is explored respectively. We conclude this chapter with a discussion of the performance evaluation metrics.

As mentioned in the previous chapters, the bit-flip protection is executed using two distinct operations, namely encoding, and decoding. Let's explain these operations in further detail before proceeding to the optimization techniques.

Encoding refers to the process of adding additional bits of information to data in order to detect and correct errors that may occur during the transmission or storage of that data. During the encoding process, redundant bits known as parity or error-correcting codes are added to the data being stored. These bits are calculated based on the values of the original data, and their purpose is to provide a way to verify the accuracy of the data if errors are introduced. The basic operations executed in the SECDED includes receiving data, applying a shift operation on the received bit to leave space for the parity bits, computing parity bits and finally storing the encoded data in the memory. Figure 4.1a presents these sequences of operations in the flow diagram.

Decoding refers to the process of detecting and correcting errors that have occurred during the storage or retrieval of information within a computer's memory system. The added bits (information) in the encoding process are compared with the newly computed one, to determine the existence of error in the data. The SECDED decoding process can be summarized with the flow diagram in Figure 4.1b. After reading the encoded bit, previously computed parity bits (P-parity) are extracted from the encoded data. Also, a new parity bit value (N-parity) is computed from the encoded data bits. Next, a comparison is applied between the two parity bits and the existence of a bit flip is determined. Finally, the decoded data is transferred to the caller.

In general, encoding refers to the insertion of additional bits or information into the data bit meanwhile decoding refers to the detection and correction of memory errors.



(a) Encoding operation (b) Decoding operation

Figure 4.1: SECDED operation flow diagram

4.1 SECDED

The naive implementation of SECDED is designed to work efficiently on hardware ECC. So, this approach needs to be modified for optimal implementation of the software-based SECDED. In this section, we discussed the naive implementation of SECDED and then the optimization features added to make it optimal for software implementation.

The basic concept of SECDED lies in using the bit position of the power of 2 numbers. These numbers are 1,2,4,8,16,32 and 64. The binary representation of these numbers has a single '1' bit value, and all the remaining bits are '0'. For example, the 4-binary representation of 1,2,4 and 8 is '0001', '0010', '0100', and '1000' respectively. These parity values are used to monitor the bit flip on certain blocks. Figure 4.2 shows the parity bits and the block of bits they are monitoring. The green shade indicates the bit positions that are monitored by the parity bit which are highlighted with red color. The 0^{th} bit position is computed after all the other parity bits are computed. Thus, it holds the parity result of all the computed parity bits and data bits.

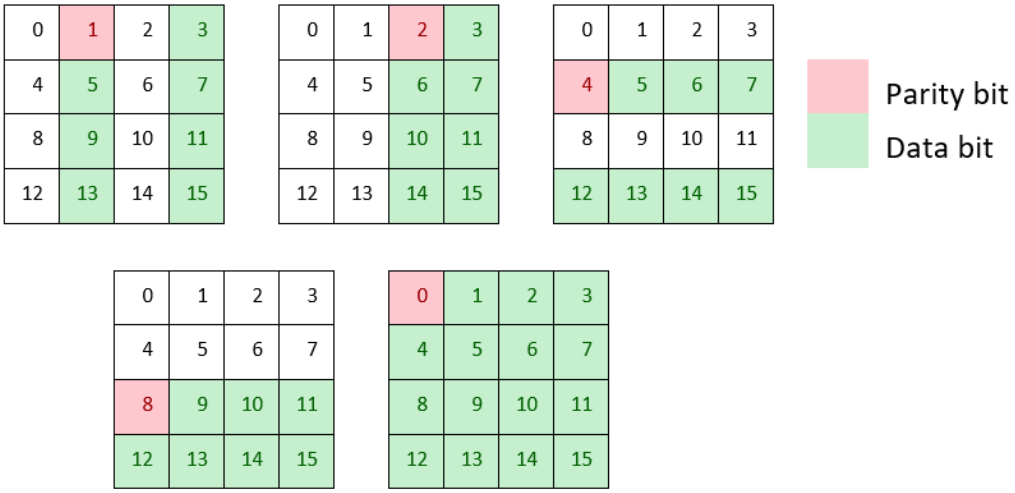


Figure 4.2: Some parity bits and their scope of monitoring for the data bits

If the 6^{th} bit of the encoded bit is flipped as shown in Figure 4.3, this event is detected in the decoding by the newly computed parity bits which are located in the 2^{nd} and 4^{th} bit position. By using the parity bits in these two positions along with the remaining parity bits, it is possible to locate the position of the flipped bit. This is done by applying the XOR operation on the newly computed parity bits with previously encoded parity bits. This process is explained in detail in section 2.1.1.

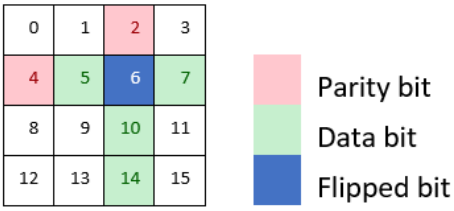


Figure 4.3: Bit-flip in the 6th bit

When the data is provided to the system that supports SECDED, first the encoding operation is executed. This encoding operation consists of two operations. The first is a shift operation and the second is a computation of parity bits operation. The shift operation is required because the original data has to leave a reserved bit position for the parity bit. These reserved positions are the power of two numbers (i.e. 1,2,4,8,… bit positions), including 0th bit position. Hence, the original data bits are shifted leaving these bit positions. Once the shifting is done, the second operation, which is the computation of parity bit value, is executed. This value is obtained by computing either even parity or odd parity of the data bits which are in the scope of that parity bit. Throughout this work, we used the even parity computation. The obtained value is then placed on the corresponding parity bit position.

An illustration of the above encoding operations is explained with an example in the following section. Let's take the letter 'A' as the input data. The binary representation of 'A' is '01000001'. Figure 4.4 shows the data bit with the corresponding bit position. Once these data are received, the shift operation follows to make space for the parity bits. Figure 4.5 shows the data bits after the shift operation. The next step is to compute the even parity value of each parity bit and set it in the corresponding bit position. Based on the discussion in section 2.1.1 and Figure 4.2 the even parities are computed. The fully encoded data is shown in Figure 4.6

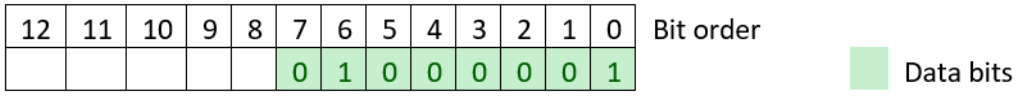


Figure 4.4: The original Data bits of 'A' before encoding

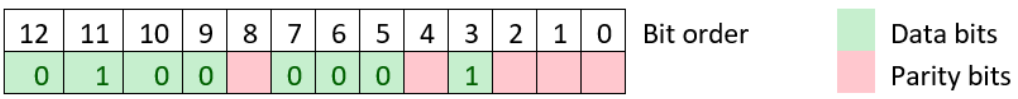


Figure 4.5: The data bits of 'A' after a shift operation

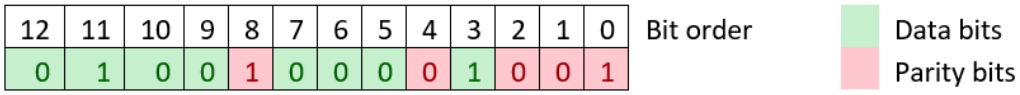


Figure 4.6: The data bits of 'A' after computation of parity bits

When the encoded data is requested for access, the decoding operation is executed. The decoding operation also involves several operations. The first one is the retrieval of the parity bits previously stored in the encoding process. The second operation is a computation of a new parity value based on the data bits only. The previously computed sets of parity bits are referred to as P-parity and the newly computed sets of parity as N-parity. The P-parity can easily be read from the stored address meanwhile the N-parity is computed using the same procedure in the encoding operation. Once these two parities are obtained successive operations follow based on the XOR result of the two.

SECDED treats the 0^{th} parity bit different from the rest. Due to this, both P-parity and N-parity need to have two segments. The first segment holds the 0^{th} bit and the second segment holds the other parity bits excluding the 0^{th} bit flip. The reason behind splitting the parity bits into two segments is for the detection of double-bit flip. The XOR operation is executed between the two 0^{th} parity bits and stored. Similarly, it is executed on the two other parity bits that exclude the 0^{th} parity bit and stored. After having the XOR result from the corresponding two segments, the integrity of the data is determined. To infer the existence of a single-bit or double-bit flip table 4.1 can be used. If the calculated XOR result from both segments is 0, then it indicates the data is free from single-bit and double-bit errors.

Table 4.1: Parity bit XOR result indication

0^{th} bit XOR result	Other parity bits XOR result	Indicates
= 0	= 0	No bit-flip
= 1	= 0	Single bit-flip on 0^{th} bit
= 1	≠ 0	Single bit-flip on other bits
= 0	≠ 0	Double bit-flip

Referring to the previous example in Figure 4.6, the retrieved P-parity bits were “10001”. After applying the same procedure that is used in the encoding operation, the N-parity bit becomes “10001”. The two segments of P-parity become ”1000” and ”1” which correspond to the other parity bits and the 0^{th} parity bit respectively. The segments of N-parity are also ”1000” and ”1”. The XOR of each value becomes 0, which indicates the data is free from single or double bit-flip.

Now, let’s assume the 6^{th} bit of the encoded data is flipped. The flipped bit changed to have the structure shown in Figure 4.7. In this case, the two segments of P-parity are ”1000” and ”1”. The N-parity is computed as follows.

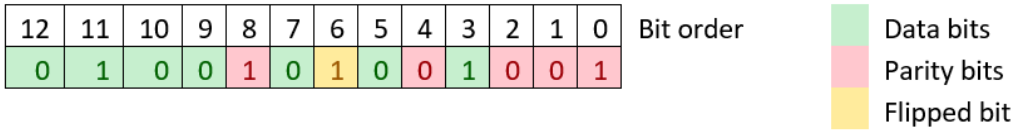


Figure 4.7: Flipped A bit on the 6^{th} bit position

P^1 = Even parity (11^{th},9^{th},7^{th},5^{th},3^{rd}) = (1,0,0,0,1) = **0**

P^2 = Even parity (11^{th},10^{th},7^{th},6^{th},3^{rd}) = (1,0,0,1,1) = **1**

P^4 = Even parity (12^{th},7^{th},6^{th},5^{th}) = (0,0,1,0) = **1**

P^8 = Even parity (12^{th},11^{th},10^{th},9^{th}) = (0,1,0,0) = **1**

P^0 = Even parity (12^{th},11^{th},10^{th},9^{th},P^8,7^{th},6^{th},5^{th},P^4,3^{rd},P^2,P^1)
= (0,1,0,0,1,0,1,0,1,1,1,0) = **0**

From the above computation, the N-parity value becomes ”11100”. Segmenting the N-parity into other parity bits and 0^{th} parity bit, results in ”1110” and ”0” respectively. The result computed so far can be summarized in Table 4.2. The XOR result of the corresponding parity is also shown in the table. Using this result and cross-referencing it with Table 4.1, it shows that the XOR of 0^{th} bit is 1 and the XOR of the other parity bits is not equal to 0 (= the condition in row 3 of Table 4.1). This shows there is a bit flip in the data. The position of the error can be obtained by converting the XOR result of the other parity bits (0110) into decimal, which is 6. So before passing this data, the 6^{th} bit is corrected by flipping it, and then the data becomes ready to be used. If there was a double bit-flip, the 0^{th} bit XOR result becomes 0 and the XOR of the other parity bit is different from 0. In this case, since the data can’t be corrected, the data is discarded and new data is requested.

Table 4.2: Decoding of the received data

	P-parity	N-parity	**XOR(P-parity, N-parity)**
Other parity bits	1000	1110	**0110**
0^{th} bit	1	0	**1**

In this encoding and decoding of the data, the major operations are the shift operations and the parity bit computation. Optimization is applied to these two operations and its implementation is discussed in the next section.

4.2 Modified SECDED

The modified SECDED uses the concept of naive SECDED and optimizes two features of the algorithm. The first modification is a bit manipulation, and the second is the of use an intrinsic function (or built-in function) to compute the parity bit values.

4.2.1 Bit manipulation

The shift operation of SECDED is modified by appending the parity bits to the front of the data bits instead of shifting and inserting the parity bits. This helps to replace the shift operation that is executed on the data bits. The challenge of using this approach is the difficulty to pinpoint the location of flipped bit. As mentioned in the previous section, for naive SECDED, the position for a single bit-flip error can be simply obtained using the XOR result. The change in shift operation makes this value pinpoint to the different bit instead of the flipped bit. This problem is handled by assigning a virtual position for the bits, and later translating the result of the XOR to the correct bit position.

The appending of the parity bits in front of the data bits is executed in the encoding operation. For example, let's take binary data shown in Figure 4.6. After applying the bit manipulation to the data, the final result set to have the structure shown in Figure 4.8. The virtual position indicates the original place for each bit based on the naive SECDED. It is used to convert the result obtain from XOR of P-parity and N-parity to the actual bit position, for the case of a single-bit flip. Considering the bit-flip example shown in Figure 4.7, in this context, the 6th bit in the Figure 4.7 is placed (located) in the 2nd bit of Figure 4.8. If the bit-flip occurs in this bit, then the XOR turns a bit position of 6. This bit position is translated using the virtual position. So, the bit to be corrected becomes the value in the 2nd position.

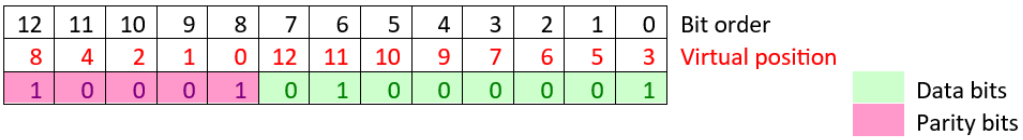


Figure 4.8: The bit arrangement of modified SECDED

To implement this approach, the (72,64) structure definition of C programming language is used. 72 indicates the total number of data bits and parity bits, meanwhile, 64 indicates the number of data bits. The struct definition of parity bit and data bit for the modified SECDED is shown in Listing 4.1. 64-bit data is stored in the $data$ variable, and the parity bits are placed in the $parity$ variable. In the decoding operation, the data is checked for any bit flip and if it is error-free, then the $data$ variable can be passed to the caller without any shift operation. In the case when the incoming data is greater than 64 bits, it is sliced into multiple 64-bit chunks. For example, if the given data is 128 bytes, it is sliced into 64-bit data and an additional 8-bit parity is computed for the 64-bit slice. As a result, the SECDED generates 16 chunks, each holding 64-bit data and 8-bit parity. If the data isn't a multiple of 64, then padding is applied on the last chunk.

Listing 4.1: Definition data and parity bits for modified SECDED

```
1  typedef struct {
2      uint64_t data;
3      uint8_t parity;
4  } dataParity_64;
```

4.2.2 Using intrinsic function

The second optimization added to naive SECDED is the use of intrinsic functions. Intrinsic functions are pre-defined functions available for use in a given programming language, whose implementation is handled specially by the compiler.

Popcount is a built-in feature that was implemented in several computers utilizing some additional hardware. It is an application-targeted accelerator instruction with the goal to determine the number of set bits that are included in a specific computer word. This built-in feature outperformed alternative software solutions like serial shifting. It is built into the system that allows using the specific hardware designed for it along with the machine instructions directly. It is not available in all programming languages, only in a limited number of them like C/C++. Since it is an intrinsic function, the compiler processes it in its entirety and generates inline code that is typically one or two instructions long. Although it is called a function, it does not have an externally visible library function available since it is executed using the circuitry that is designated for it [56, 57].

The serial shifting implementation is presented in the Listing 4.2. This approach is optimized and implemented with the help of hardware support for the Population count (POPCNT) implementation.

Listing 4.2: Serial shifting pseudocode

```
1        Count = 0;
2        While the Number != 0 do
3            If the LSB is 1 then
4                Count +=1
5                Number << 1
6            End If
7        End while
```

The function *__builtin_popcount* utilizes a particular hardware instruction internally. In an x86 design, the *Population count (POPCNT)* instruction is used by the compiler. Intel's Nehalem was the initial processor to introduce the *POPCNT* operation. To utilize *POPCNT*, the compiler must support Streaming SIMD Extensions 4 (SSE4) [58]. Intel Core i7 ("Nehalem"), Intel Atom (Silvermont core), AMD Bulldozer, AMD Jaguar, and subsequent CPUs are among the computer architectures that support SSE4. Additionally, SSE4.2 is supported by almost every Intel Core i3, i5, i7, AMD FX, and Ryzen CPUs. Software developed for Intel 64 and IA-32 architecture microprocessors in prior generations is completely compatible with SSE4. A description of the implementation in Intel processors can be referred from page 156 of Intel® SSE4 Programming Reference[1].

The existence of POPCNT in the intel processor can be obtained by checking the 23rd bit of the ECX register. Execute CPUID with EAX =1 as input to see if the processor is capable of executing the POPCNT instruction. The CPU is capable of supporting the POPCNT instruction if bit 23 of ECX is set. It is not supported by the CPU if the bit is cleared. In addition to the typical needs of the Intel 64 architecture, the operating system does not need any extra support to allow CRC32 or POPCNT.

To compute the even parity value and determine and the number of 1's in a given set of bits, we used the population count intrinsic function ($__builtin\_popcount()$). This intrinsic function counts the number of set bits (bits with a value of 1) in a given value. The function takes 64-bit data and returns an integer value. Since we are only interested in the count being either even or odd, it is possible to apply AND operation on the returned integer value with 1 and get the parity value. If result 1 indicates the value has an odd number of '1's and if not then it has an even number of '1's. Listing 4.3 shows the general syntax used in our work to compute the even parity value for all parity bits.

Listing 4.3: Instrinsic popcount operation

```
1        bit = __builtin_popcountll(bit64) & 1;
```

By utilizing the two optimizations feature, it becomes feasible to implement SECDED on a device that lacks hardware support for ECC. It also helps to achieve single-bit flip correction and double-bit flip detection efficiently, without the need for additional hardware and related costs. The performance measurement of the two algorithms is discussed in the next chapter.

[1] https://www.intel.com/content/dam/develop/external/us/en/documents/d9156103-138479.pdf

Even though single-bit and double-bit flips are detected using SECDED, it doesn't convey any information when multiple-bit flips happen. Therefore, we proposed hashed-based software ECC that is capable of detecting multiple-bit flips.

4.3 Hash-based software ECC

Hash-based software ECC is implemented that utilizes the hash functions to detect any number of bit flips that may occur within a DRAM. This approach offers a more robust and comprehensive means of error detection.

By using the deterministic and collision-resistant features[2] of hash functions, any number of bit flips can be detected. For the correction of the flipped bit, a brute-force algorithm is applied, leveraging the collision-resistant feature of hash functions. This ensures the non-ECC devices remain fully operational and active for detection and correction of bit-flips at all times.

The general structure of the encoding and decoding phases of hash-based software ECC is shown in Figure 4.9. In the encoding operation, a hash (Hash1) is generated based on the input data using a hash function. Then, the Hash1 and the input data are stored together in the memory. The decoding operation beings with generating a new hash value (Hash2) from the stored data. The Hash2 and previously stored Hash1 are compared for similarity. If they are a match, then it indicates the data is error-free. In the case of a mismatch, then the brute-force algorithm is initiated to try to correct the corrupted data.

The hash function used to encode and decode can be either CHF or NCHF. The general background about each of the hash functions used in this research is discussed in section 2.2. In the following section, we discussed the implementation of these algorithms for the purpose of error detection.

[2]Refer to section 2.2.1 for more explanation about these features

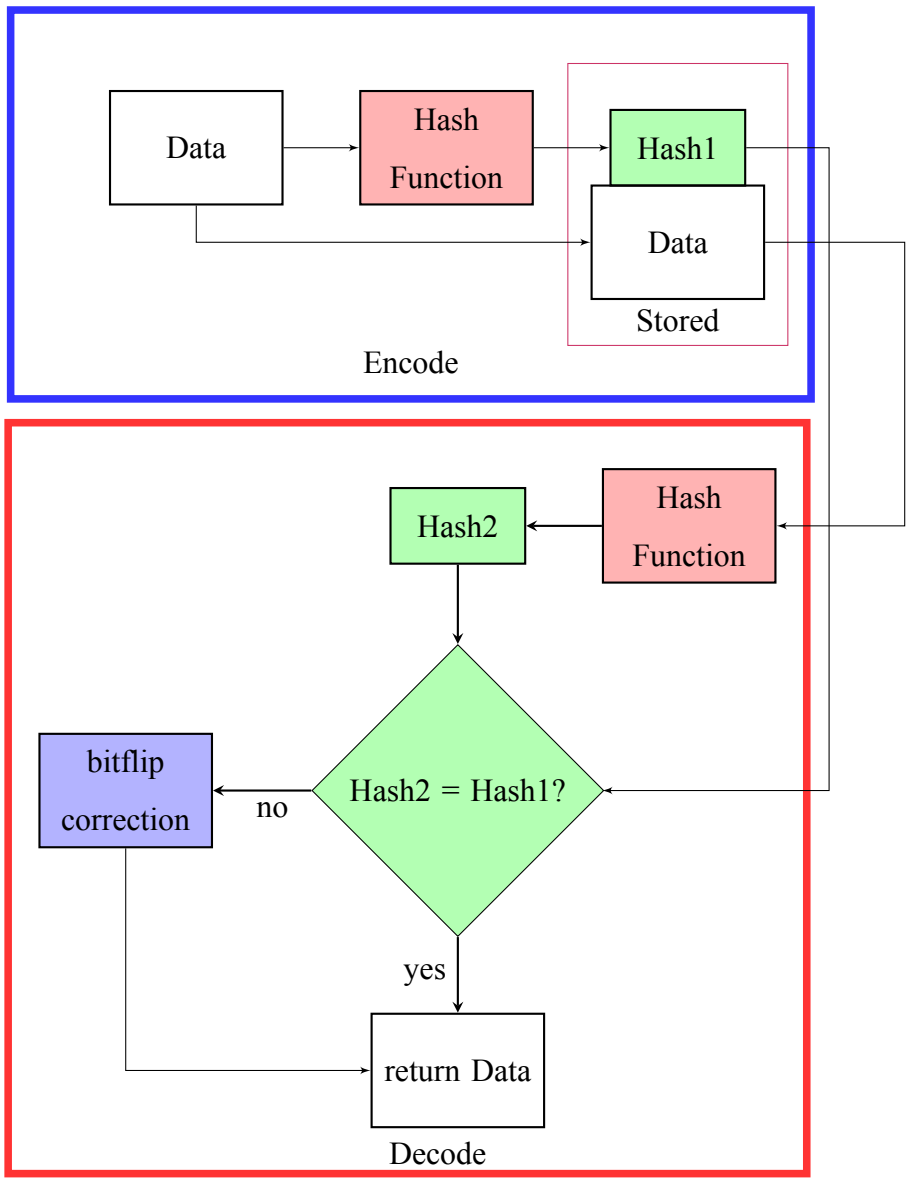


Figure 4.9: Hash based software ECC block diagram

4.3.1 Cryptographic hash function implementation

The Cryptographic Hash Function (CHF) used in this research are Blake, SHA, MD, and RIPEMD. Each of these hash functions has different versions. The following list shows the CHFs used in this work.

- Blake: Blake2b, Blake2s, Blake3
- SHA:
  - SHA1: SHA-1
  - SHA2: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256

- SHA3: SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, SHAKE256

- MD5

- RIPEMD160

For the implementation of the above CHFs except Blake3, the OpenSSL[3] library for C programming language is used. OpenSSL is an open-source software library that provides developers with access to a variety of cryptographic functions, including symmetric and asymmetric encryption, hashing, and digital signatures. It is available on a wide range of platforms, including Linux, Windows, and macOS. With its powerful features and ease of use, OpenSSL has become an essential tool for implementing the listed hash functions.

The Blake3 hash function implementation isn't provided by the official OpenSSL library. consequently, its implementation is adopted from the official Blake3 GitHub[4] repository.

4.3.2 Non-cryptographic hash function implementation

For the Non-Cryptographic Hash Function (NCHF) Lookup3, BuzHash, Superfast hash, and Murmur3 hash functions are used in this work based on the performance evaluation of C. Estébanez et al. [49].

The Lookup3 hash function is adopted from a repository of the Google Git[5] by Bob Jenkins in 2006. The Buzhash function (Hashing by a cyclic polynomial) is implemented using the code in Listing 4.4. The murmur3 hash function is adopted from Peter Scott's murmur3 GitHub repository[6]. Finally, the Superfast hash implementation is taken from Cedric Guillemet's Github Gist repository[7].

Using these CHFs and NCHFs the existence of bit flip can be detected but not corrected. To provide bit flip correction on top of the hash functions, the brute-force algorithm is used.

[3] https://www.openssl.org/

[4] https://github.com/BLAKE3-team/BLAKE3

[5] https://android.googlesource.com/platform/external/jenkins-hash/+/75dbeadebd95869dd623a29b720678c5c5c55630/lookup3.c

[6] https://github.com/PeterScott/murmur3/blob/master/murmur3.c

[7] https://gist.github.com/CedricGuillemet/4978020

Listing 4.4: BuzHash implementation

```
1         uint64_t buzHash_64(const char *s, int length){
2         uint64_t result = 0;
3         int p = 31, m = 1e9 + 7;
4         int hash_value = 0;
5         long p_pow = 1;
6         for (int i = 0; i < length; i++){
7           hash_value = (hash_value + (s[i] - 'a' + 1) * p_pow) % m;
8           p_pow = (p_pow * p) % m;
9         }
10        result = (uint32_t)hash_value;
11        result <<= 32;
12        p = 37;
13        m = 1e9 + 9;
14        hash_value = 0;
15        p_pow = 1;
16        for (int i = 0; i < length; i++){
17          hash_value = (hash_value + (s[i] - 'a' + 1) * p_pow) % m;
18          p_pow = (p_pow * p) % m;
19        }
20        result |= (uint32_t)hash_value;
21          return result;
22      }
```

4.4 Brute-force algorithm

Due to the pre-image resistant nature of CHFs, it is impossible to obtain some information about the input data from the difference between the two hash values (Hash1 and Hash2). The approach used to address this issue is the use of a brute-force algorithm to correct the flipped bit.

The brute-force algorithm is used to iteratively flip different combinations of bit/bits and try to obtain the correct input data. On every iteration, certain bit/bits are flipped and a new hash value (Hash2) is computed. Then, the Hash2 is compared with the previously stored hash value (Hash1). The iteration terminated in two cases. The first is when the comparison of Hash1 and Hash2 is a match. This indicates the original data is recovered (corrected) from the corrupted data. In this case, the corrected data is returned. The flow diagram for this case is shown in Figure 4.10.

The other case is if iteration reaches the maximum number of bit flips set to execute. This indicates either the number of flips that have occurred in the data is more than the maximum number of flips the algorithm is set to compute or the bit-flip has happened in the hashed data. In either case, the data is requested to be reloaded from the disk. The algorithm can be set to correct any number of bit flips. In this work, we implemented the brute-force algorithm to correct up to 3-bit flips.

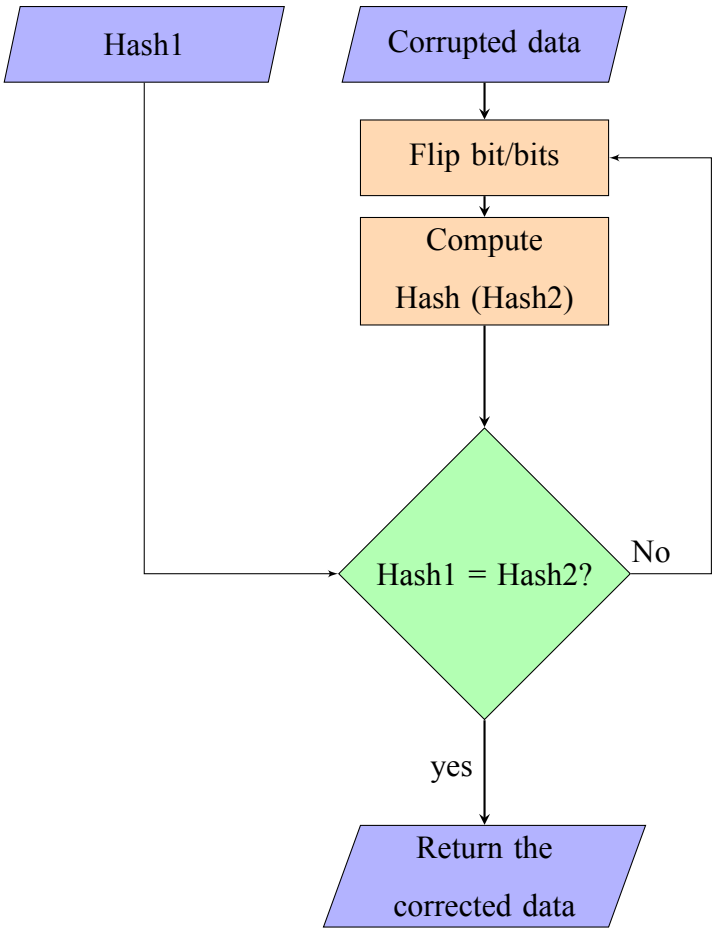


Figure 4.10: Brute-force flow diagram to correct flip bit in the data

4.5 Bit-flip generation

To evaluate the performance of the proposed solution, a bit-flip can be generated on the random bit or it is possible to flip the last bit of the last byte which is the worst case. In this work, both random bit flip generation and the worst-case scenario is tested. To create a single-bit flip, a byte, and bit indices need to be randomly generated. The byte index is used to determine the position of the byte that is selected from the input data. Its value ranges between 0 and the length of the data in bytes. The bit index is used to select the bit position in the selected byte. Its value ranges from 0 to 7.

Using the byte and bit index obtained from the two randomly generated numbers, a bit-flip can be introduced. This execution generates a single-bit flip to the data bits. If a multiple-bit flip is required, the execution can be iterated multiple times. Listing 4.5 illustrates the C code that can create multiple bit-flip to the given data. The function called $multipleBitFlip$ takes two arguments. The first is the data to be flipped which is stored in variable $dataNew$. The second is the number of flips requested which is stored in variable $flip$. The length of the data is computed in Line 3 and the bit and byte numbers are generated in Lines 8 and 9. Finally, the flipping happens in line 10 and the flipped data is returned in line 12. The syntax from Lines 7 to 10 iterates the request number of flip times.

Listing 4.5: Random bit flip generator

```
1   char *mutipleBitFlipRandom(char *dataNew, int flip){
2     srand(time(0));
3     size_t len = strlen(dataNew);
4     int byteNum;
5     int shiftBit;
6     for (size_t i = 0; i < flip; i++){
7       byteNum = rand() % len;
8       shiftBit = rand() % bitsize;
9       dataNew[byteNum] = (dataNew[byteNum] ^ (1 << shiftBit));}
10    return dataNew;}
```

The brute-force algorithm begins flipping bits from the first byte and goes to the end. For this implementation, the worst-case scenario is when a bit flip is in the last byte of the last bits. To simulate this scenario the C code shown in Listing 4.6 is used. The code can flip one or all bits of the last byte.

Listing 4.6: Bit flip generator on last byte

```
1   char *mutipleBitFlipWorst(char *dataNew, int flip){
2     size_t len = strlen(dataNew);
3     int byteNum;
4     int shiftBit = 7;
5     for (size_t i = 0; i < flip; i++){
6       byteNum = len - 1;
7       dataNew[byteNum] = (dataNew[byteNum] ^ (1 << shiftBit));
8       shiftBit -= 1;}
9     return dataNew;}
```

4.6 Redis

Redis is a high-performance in-memory key-value data store that stores data on DRAM for fast access. It is an open-source data structure store that is commonly used as a primary database, cache, message broker, and queue. Redis has several commands that are used to interact with it. From these commands, SET[8] and GET[9] commands are used to store and retrieve data to/from the database, respectively.

The hash-based software ECC is integrated into these SET and GET commands to provide data reliability. When a SET command is issued with a key and value, a hash value (Hash1) is generated from these inputs, and it is stored alongside the key and value. When a GET command is issued, the key, value, and Hash1 are retrieved, and a new hash value(Hash2) is computed based on the newly retrieved key and value. The Hash2 is then compared with the previously stored hash (Hash1) and if they are the same, the data is ready for use.

To demonstrate the feasibility of the proposed solution, the hash-based software ECC is implemented for these two commands only. It can be applied for all commands and be an optional feature integrated.

4.7 Evaluation Metrics

For the bit flip detection and correction algorithms, the evaluation metrics utilized comprise execution time and memory utilization. The average execution time is computed separately for encoding and decoding operations after 1000 iterations. The encoding computation time includes the time it takes

- To access the data to be encoded
- To compute the hash function from the data and
- To store the hash value along with the data

On the other hand, the decoding computation time includes the time it takes:

- To load the stored data and compute the new hash value
- To compare the newly computes hash with the previously stored hash

[8] `https://redis.io/commands/set/`

[9] `https://redis.io/commands/get/`

- To Correct a data with bit-flip/s [if bit flip detected]

The speedup ratio is computed using the formula:

$$speedup = \frac{timeofslow}{timeoffast}$$

The time complexity of both CHFs and NCHFs is dependent on the number of data sizes. As a result, the time complexity becomes $\theta(n)$ [19]. Hence the evaluation metrics rely on the execution time of the encoding and decoding function.

The memory utilization is measured based on the output hash size. All CHFs and NCHFs have fixed size output hash. Thus, a comparison of the memory space required to store the hashed value is tested between the hash functions and also with SECDED

To evaluate the performance of the hash-based software ECC on the Redis in-memory key-value store database, we utilized the built-in Redis benchmarks[10] and latency diagnosis[11] tool. Redis benchmark is a tool that is used to test the performance and capabilities of Redis servers. It is capable of testing various types of Redis commands and transactions on different intensities of workloads. Redis latency diagnosis tool is used to measure the delay between the time a client issues a command and the time the reply to the command is received by the client. These performance measuring tools provide information about the latency, throughput, and responsiveness of Redis data structure stores with and without the integration of hash-based software ECC.

Listing 4.7 shows the two evaluation metrics used for Redis. Line 1 shows the syntax that is used to evaluate the SET and GET commands. The $-t$ flag is used to specify what commands need to be evaluated meanwhile the $-n$ flag is used to provide the total number of requests to be sent. By default is it 100000, but in this work we tested up to 1000000. The $-q$ command is used to quit the operation and show query/sec values.

Line 2 of Listing 4.7 shows the syntax to measure the latency. The argument 100 is the number of seconds the test set be executed. based on the Redis Documentation this value is appropriate to simulate and evaluate the Redis.

Listing 4.7: Redis evaluation metrics

```
1    ./redis-benchmark -t set,get -n 1000000 -q
2    ./redis-cli --intrinsic-latency 100
```

[10] https://redis.io/docs/management/optimization/benchmarks/

[11] https://redis.io/docs/management/optimization/latency/

4.8 Summary

In this chapter, we discussed the working principles of SECDED and the modification implemented to optimize SECDED for non-ECC devices. We also discussed how CHFs and NCHFs are used to detect bit flips and correct the error using a brute-force error correction algorithm. Finally, we discussed the integration of hashed-based software ECC to Redis in-memory key-value store database, and the performance evaluation metrics. In the next section, the result obtained from our experiment is discussed.

Chapter 5

Result and Discussion

In this chapter, we discussed the experiments carried out and the obtained results. We began by discussing the experiment setup and the result obtained from the performance comparison between SECDED and modified SECDED on non-ECC. Next, the evaluation result of hash-based software ECC using both CHF and NCHF is presented. Later, the result of the error correction algorithm with single-bit and multiple-bit flip is discussed. The result on memory utilization of different algorithms and comparison with SECDED follows. Finally, this chapter is concluded by discussing the result obtained from the Redis data structure store and by addressing the research questions given in the first chapter.

5.1 Experimentation Setup

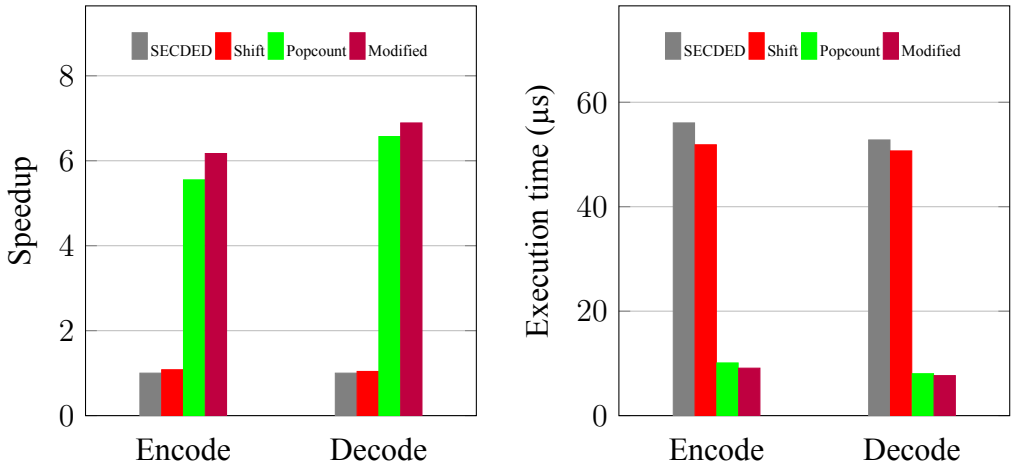
In order to evaluate the performance of the proposed method, a workstation, and a server computing devices were used. The workstation is hp pavilion Laptop 15-cc5xx with Intel® Core™ i7 (7th Gen) processor, 8 GB RAM, and running Ubuntu 20.04 operating system. The server is Huawei 2288H V5 which has a configuration of Intel® Xeon™ Gold 6126 processor, 64GB RAM, and running Ubuntu 18.04.6 operating system. The compiler used in both systems is GCC v9.4.0 and OpenSSL v1.1.1 is used for most CHF implementation. The proposed algorithm is integrated with Redis 7.2 which is the latest stable release of the Redis data structure store at the time of this work.

The Result obtained from the workstation and the server yields similar trends. Due to this, the result from the workstation is presented in this document. The workstation is selected because it has comparatively recent computing units.

5.2 SECDED optimization

The SECDED algorithm is optimized using bit manipulation and intrinsic functions. Figure 5.1 shows the speedup and average execution time obtained for 1KB of data without any bit flip. The vertical axis in Figure 5.1a represents the speedup obtained by the optimizations. The horizontal axis has two labels that represent the operation by which the speedup is measured. The naive SECDED is used as a speedup reference for other optimizations and in the figure it is referred to as just SECDED. The legend shift and Popcount refer to the result obtained by applying bit manipulation and intrinsic functions respectively. The legend Modified represents the optimized SECDED which is the result obtained by applying both the bit manipulation and intrinsic functions. The result shows using intrinsic functions optimizes the algorithms significantly compared to the bit manipulation. Also, the modified SECDED algorithm can execute 6x faster than the naive implementation. The encoding and decoding operation exhibits a relatively similar change in the result.

The execution time for encoding and decoding operation is presented in Figure 5.1b. The y-axis is the average execution time in microseconds, while the x-axis indicates the encoding and decoding operations. The naive SECDED and the bit manipulation takes above 50µs, meanwhile the other two optimizations take less than 10µs. As shown in the figure, the result obtained by optimization of the shift operation is almost the same as the naive one. This is because the shift operation has O(1) time complexity and optimizing such an algorithm generates a small improvement. On the other hand, the POPCNT operation has a time complexity of O(n). The $\_\_builtin\_popcnt$ instruction utilizes a built-in hardware instruction which makes its execution must faster than the serial shifting or other related POPCNT operations. This shows the naive SECDED can be optimized significantly using built-in intrinsic functions.

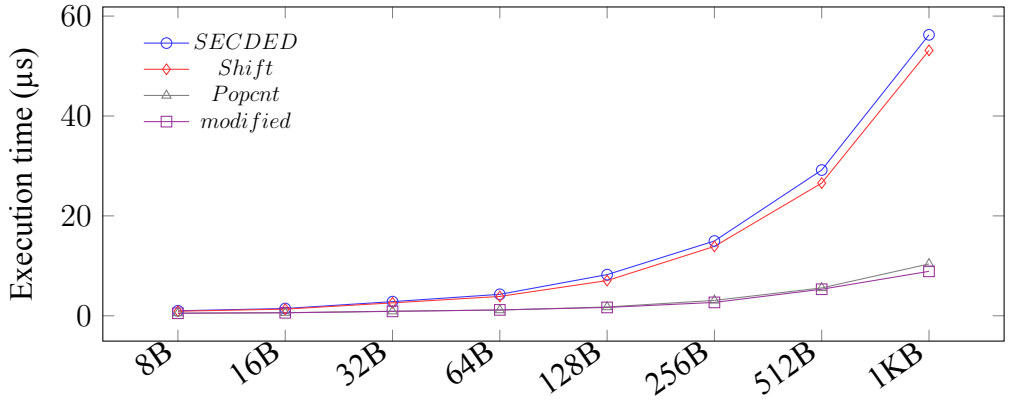


(a) Speedup

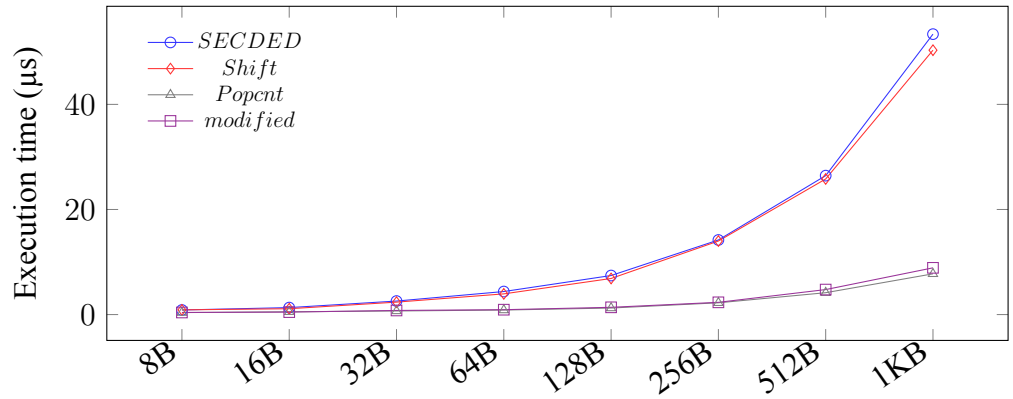
(b) Execution time

Figure 5.1: Comparison of SECDED algorithm with each optimization phases

Besides evaluating the performance optimization using 1KB data the optimization methods were tested for data sizes ranging from 8B to 1KB. This evaluation is also implemented without introducing any bit flip to the data. Figure 5.2 shows the result for encoding and decoding operation. The vertical axis indicates the average execution time in microseconds and the horizontal axis indicates the data sizes used in the experiment. Figure 5.2a illustrates the encoding result and Figure 5.2b shows the decoding result. The execution time for both the encoding and decoding process increase significantly with the increase in data size. This is because the input data needs to be sliced into 64 bits and then an 8-bit parity is added to the sliced data bits. This process repeats until all data is encoded. The reverse process is applied in the decoding operation. The result shows that the optimization features added to the SECDED reduced the execution time. The execution time difference between the naive and modified SECDED increases as the data size gets bigger.

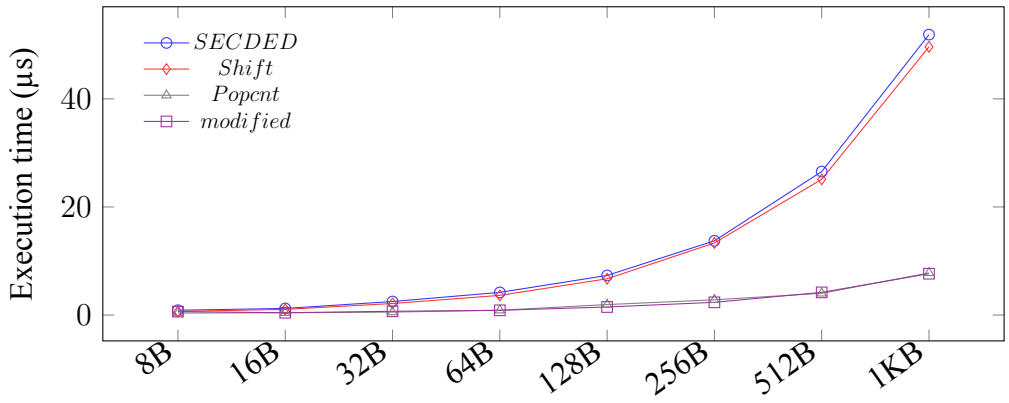


(a) Encoding

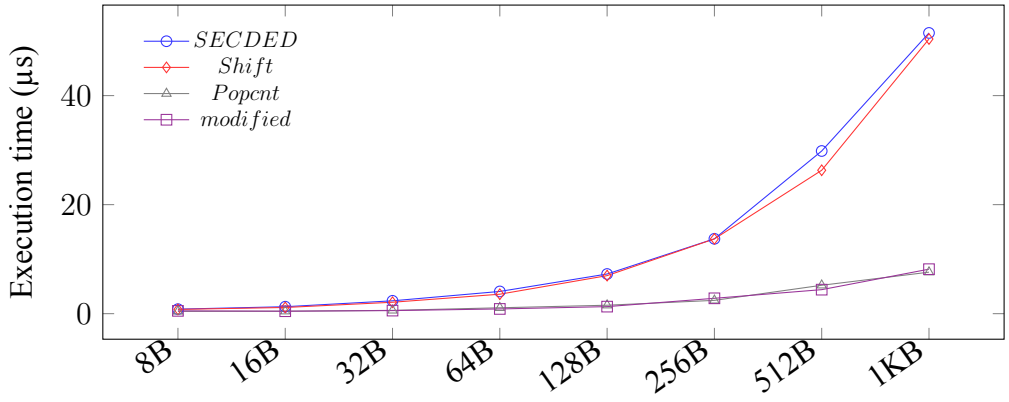


(b) Decoding

Figure 5.2: Execution time comparison without any bit flip



(a) Single-bit flip



(b) Double-bit flip

Figure 5.3: The decoding execution time for single-bit and double-bit flip

The performance comparison is also evaluated for single-bit and double-bit flips. In the case of a single-bit flip, the bit flip occurs in one of the sliced data segments. As a result, the other segments are unaffected. For example, when 1KB of input data is encoded, it is sliced into 128 different segments. When a single-bit flip is introduced to the 1KB data, only 1 segment out of the 128 segments is affected. Due to this, the error correction is executed on this segment only. In the case of a double-bit flip, only the bit flip is detected. This makes the execution time overhead for correction to have minimal effect. The result of decoding for single-bit and double-bit flip is shown in Figure 5.3. The vertical and horizontal axis shows the average execution time in microseconds and the data size used respectively. Figure 5.3a shows the time it took to detect and correct a single-bit flip. Similarly, Figure 5.3b illustrates the average detection time of a double-bit flip. The described result shows that the modified SECDED outperforms the naive SECDED in single and double-event upsets. Since the encoding operation isn't related to or affected by a bit flip, the encoding operation result presented in Figure 5.2a can represent the encoding operation for single-bit and multiple-bit flips.

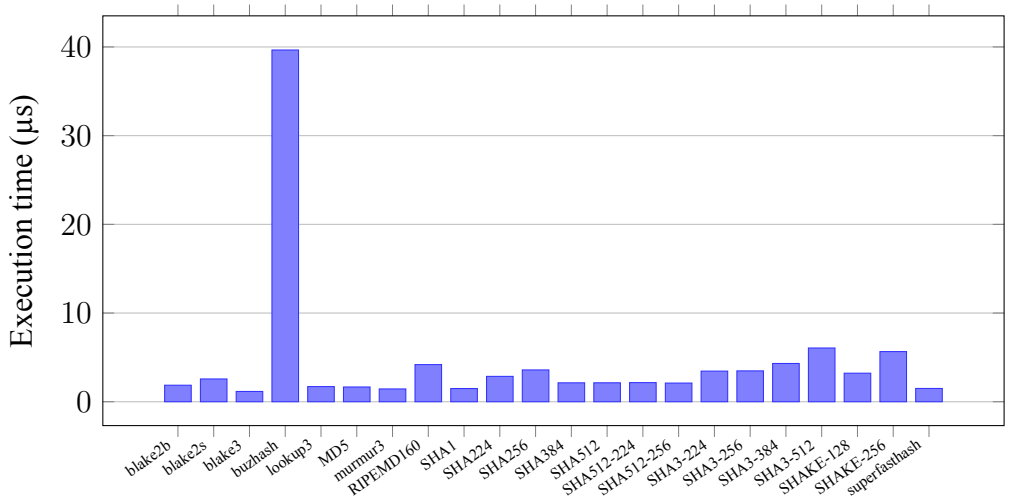
Based on the result obtained above, by optimizing the naive SECDED, it is possible to integrate the error detection and correction capability of SECDED for non-ECC devices to provide fast and reliable service.

5.3 Bit flip detection using hash-based software ECC

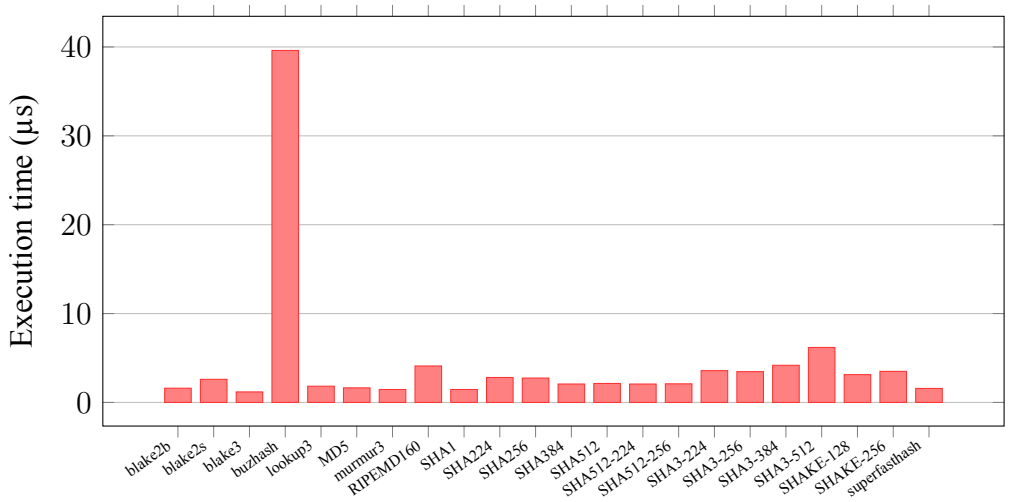
In this work, the CHFs and NCHFs are used to provide error detection for any number of bit flips. The avalanche effect and the collision resistance features of hash functions make them ideal for this task. The CHFs are better in collision-resistant and avalanche effect features, meanwhile, the NCHF are faster and more resource-efficient. Therefore, it is better to test the performance and reliability of both types of hash functions. Figure 5.4 shows the average execution time of encoding and decoding operations for 1KB of data. The vertical axis shows the average execution time in microseconds and the horizontal axis shows the CHFs and NCHFs used in this work.

The result in Figure 5.4a indicates that the buzhash function is performing worse meanwhile Blake3 and murmur3 are performing relatively better than the rest. This is because uses a relatively large number of operations compared to other operations. On the other hand, blake3 uses parallelism for optimal implementation and the other NCHFs aren't computationally intensive as CHFs. The experiment shows that Murmur3, superfast, Blake3, and lookup3 hash functions finished the encoding of 1KB data less than 2µs.

Figure 5.4b shows the average decoding execution time. In this case, Blake3 and murmur3 hash functions finished their execution in less than 2µs. Buzhash took about 39µs to decode 1KB of data, which exceeds 33µs from SHA3-512, which is the next poor-performing hash function. Modern commodity workstation is equipped with dedicated hardware that enhances the computation of CHFs. Utilizing such functionality can generate more fast execution time.



(a) Encoding



(b) Decoding

Figure 5.4: Hash functions execution time for 0-bit flip

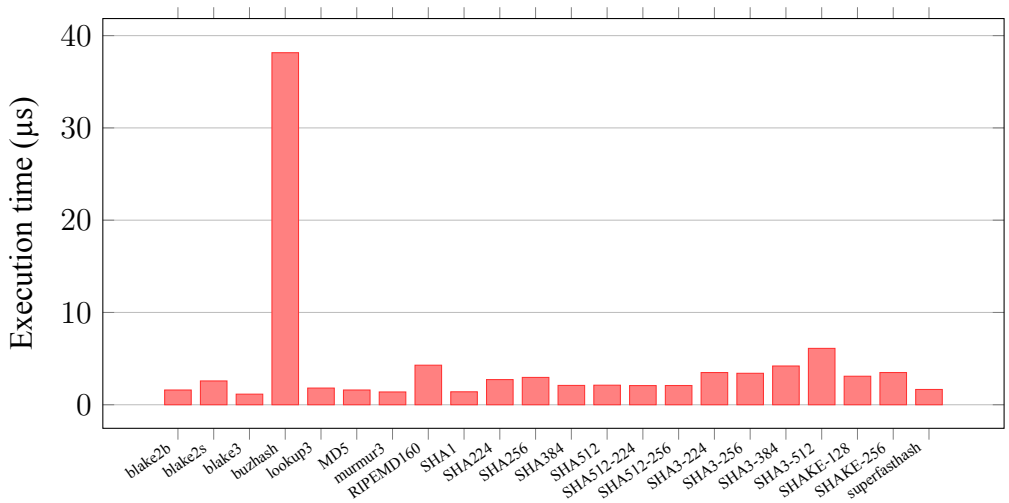


Figure 5.5: Hash function decoding while detecting 50-bit flips

Since a bit flip is a seldom event, the hash function with small encoding and decoding execution time overhead is considered best for the real-world application. The murmur3, superfast, and lookup3 from NCHF; blake3, SHA512-224, and SHA-512 from CHF performed relatively better. The average encoding and decoding time for these hash functions is $<3\mu s$.

The selection of the best-performing algorithm for any given task needs to be selected based on the performance of encoding and decoding of data without the introduction of bit-flip. This is due to the fact that bit-flip is not as frequent as an error-free operation. In addition, different algorithms perform best in certain data sizes. As the data size varied so is the best-performing algorithm. In light of this, it is possible to use a different algorithm to encode and decode based on the input data size.

Due to the avalanche-effect feature of the hash functions, the slightest change in the input data makes the newly computed hash in the decoding operation to be different from the originally computed hash. This helps to detect any number of bit flips easily. Figure 5.5 shows the execution time it takes to detect 50-bit flips. The y-axis indicated the average execution time taken to detect the bit flips. The x-axis shows the hash functions used in this experiment. The result shows the detection execution time for all algorithms is $<7\mu s$. Blake3 has the lowest detection time of $1.125\mu s$. This is because the Blake3 hash function is designed to be a parallelizable hash function [34].

Different numbers of bit flips were introduced to the input data and the result obtained from these experiments shows similar results as described in Figure 5.5. Irrespective of the number of flipped bits the detection time is almost similar. This is because the correctness of the data is determined using the comparison between the two hash functions. The Result described in the figure shows only the bit flip detection time, it doesn't include the time it to correct the bit flip.

5.4 Bit flip correction

A brute-force algorithm is used to trace the position (index) of the flipped bit and correct it. A more detailed discussion of the implementation of the brute-force algorithm is presented in section 4.4. The brute-force algorithm can be set to correct any number of bits, with some effect on the performance. This effect is dependent on the number of bit flips and the data size. Let's first consider Figure 5.6 which shows a single-bit flip correction time for 16B data. The horizontal axis indicates the CHFs and NCHFs used in the experiment while the vertical axis shows the error correction time in microseconds. The result shows Superfast hash, which is NCHF, takes about 5µs to detect and correct the flip from 16B data. Murmur3 and lookup3 follow with the execution time of 11µs and 8µs. From the CHF MD5 and Blake3 performed better with 16µs and 20µs execution time. The brute force algorithm using RIPEMD160 took about 4.7ms to find the flipped bit which is located in the last bit. Since it is performing very worse, the result of RIPEMD160 is excluded for double and triple bit flips.

Double-bit and triple-bit flip correction times are shown in Figure 5.7 and Figure Figure 5.8 respectively. The result shows the correction time significantly increases with the increase in the number of bit flips. The NCHFs expect buzhash are performing better for double and triple bit flips as expected. The result presented here is the outcome of flipping double and triple bits on the last byte of the data (worst-case scenario). As the number of flipped bit increase, the correction time approaches need a second or more. Thus, for multiple-bit flips, it would be more efficient to re-request the data from the disk instead of trying to correct it.

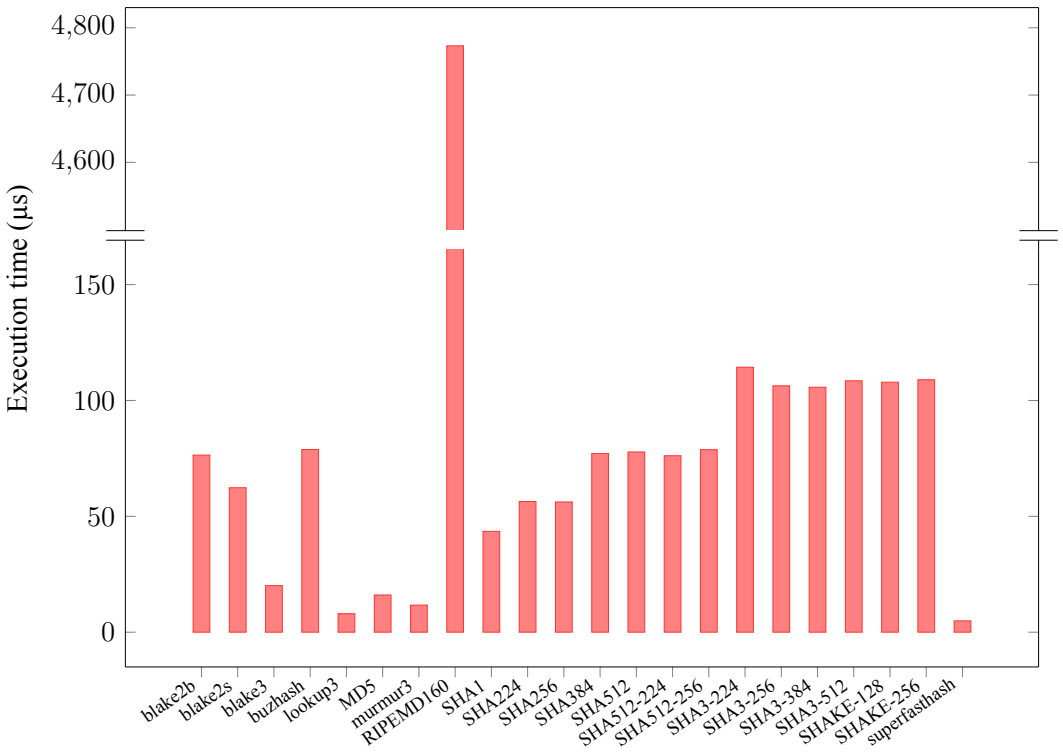


Figure 5.6: Single bit flip correction time of hash functions for 16B data

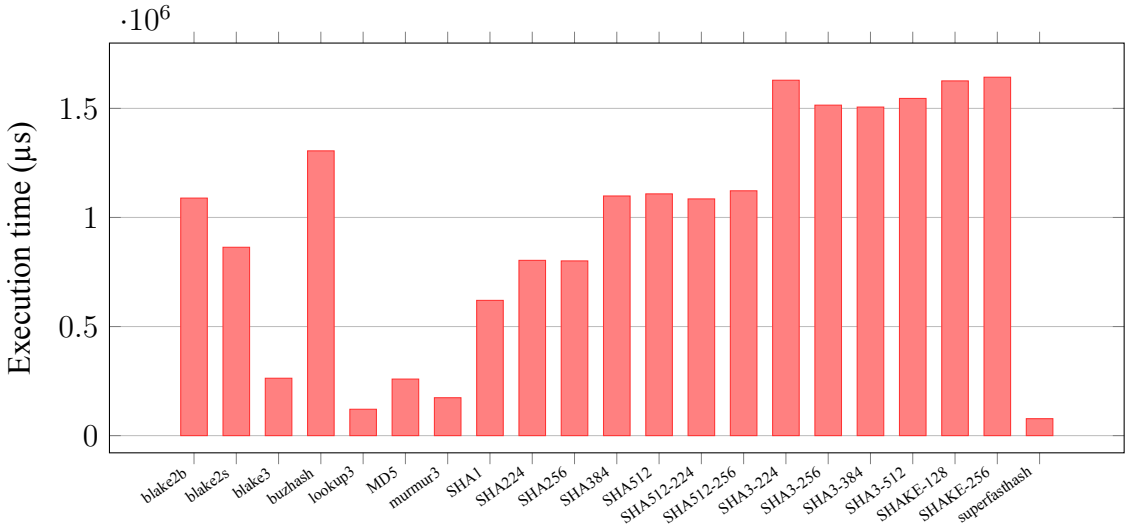


Figure 5.7: double bit flip correction time of hash functions for 16B data

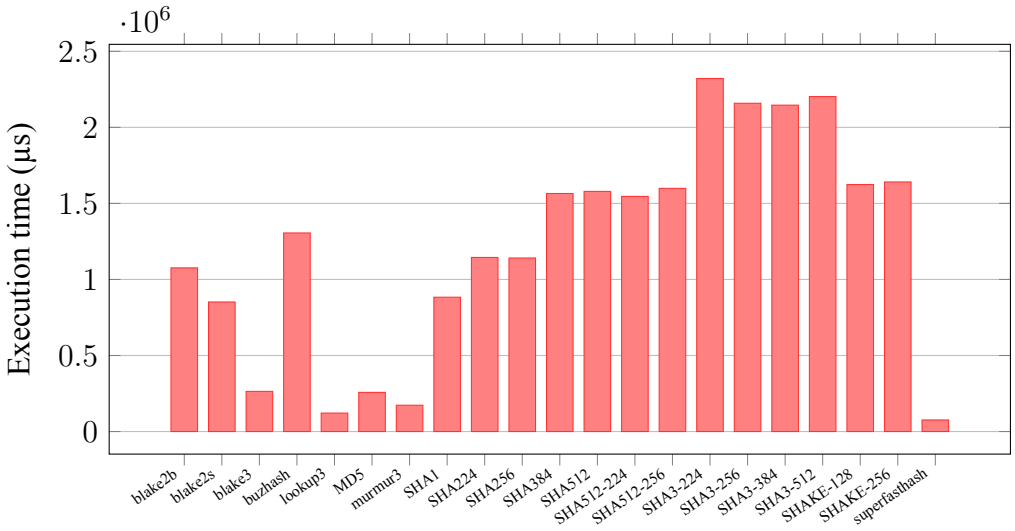


Figure 5.8: three-bit flip correction time of hash functions for 16B data

In order to better understand the effect of input data size on error correction, Figure 5.9 is presented. The y-axis shows the log-scaled execution time in microseconds, and the x-axis shows the data size used in this experiment. The figure consists of the data size from 8B to 1KB. Seven relatively best-performing hash functions were selected. The result shows that as the data size is small, the time it takes to detect and correct the flipped bit is very small. As the data size increases, so does the time for correction. In addition to this, the best-performing hash functions also change based on the input data size. For example, the superfast hash function is performing best up to 256B of data. After 256B, its performance degrades compared to others. Beyond 256B data size, Blake3 and SHA-1 began to outperform other hash functions.

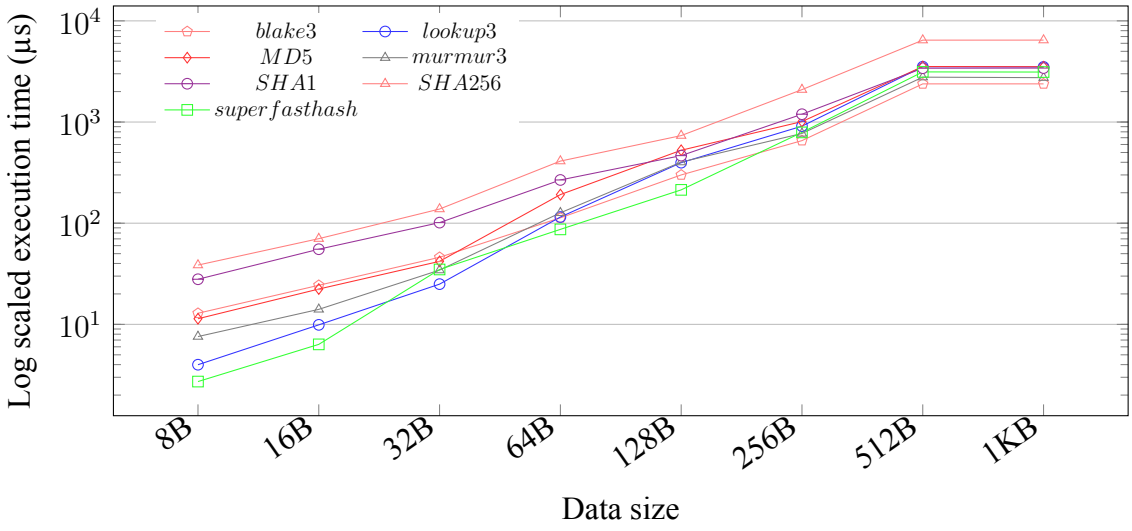


Figure 5.9: Error correction hash function for variable input data

5.5 Memory utilization

One of the critical issues to consider while evaluating memory error detection and correction algorithms beside execution speed is memory space utilization. The memory space utilized by the error correction and detection algorithms needs to be as minimal as possible. The modified SECDED uses the same algorithm as that of the naive SECDED. Consequently, both algorithms require 1 byte of memory for parity bits for every 8-byte data. Therefore, 12.5% of the memory is used to store the parity bit. As the data increase, the amount of parity bit needed to be stored increases, hence the memory requirement of the SECDED increases exponentially with the increase in data size.

The hash-based software ECC generates a hash value that is used to verify the integrity of the message. The output hash size for each algorithm is fixed, irrespective of the input data size. This brings both benefits and shortcomings in the usage of the algorithms. Hash-based software ECC provides great benefits when the input data size is large. Huge bytes of data can be monitored with a few bytes of hash value. The shortcoming becomes greater for small-size data inputs, where the required memory space to store hash may exceed the data size. Figure 5.10 shows a memory space utilization for CHF and NCHF in comparison with SECDED. In the figure '$HashoutXB$' indicates different categories of hash functions, where X indicates the output byte size. The x-axis and the y-axis of the figure show the data size and the memory space required for additional information, respectively. The hash function used in this work with their hash output size in bytes along with their category name is described in Table 5.1.

Up until 32B of data size, the SECDED has minimal memory utilization compared to the other hash functions as shown in Figure 5.10. Blake3 hash function, which is a category of 'Hash out 32B', has equal overhead as that of SECDED for 256B input data. After 256B Blake3 has less overhead than SECDED.

To address the challenge, when the input data size is small, it is recommended to integrate and compute the hash value for multiple input data. This makes memory utilization efficient and decreases the overhead.

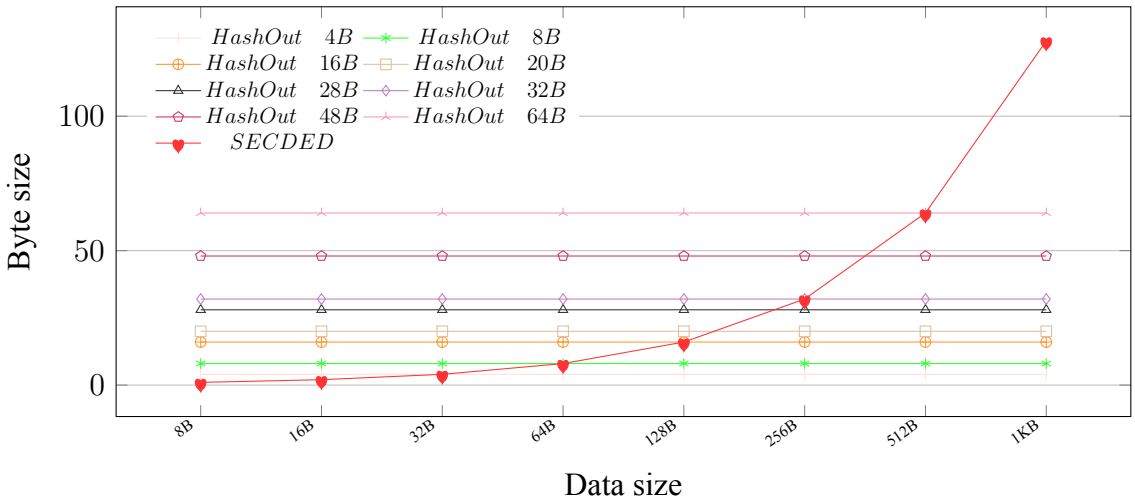


Figure 5.10: Memory space utilization for different function vs SECDED

Table 5.1: Categories of hash functions based on the output hash size

Hash function name	Output size in bits	output size in bytes	Category based on output size
superfast hash	32	4	Hash out 4B
Buzhash	64	8	Hash out 8B
lookup3	64	8	
MD5	128	16	Hash out 16B
murmur3	128	16	
SHAKE 128	128	16	
RIPEMD160	160	20	Hash out 20B
SHA1	160	20	
SHA224	224	28	Hash out 28B
SHA512_224	224	28	
SHA3_224	224	28	
blake2s256	256	32	Hash out 32B
Blake3	256	32	
SHA256	256	32	
SHA512_256	256	32	
SHA3_256	256	32	
SHAKE 256	256	32	
SHA384	384	48	Hash out 48B
SHA3_384	384	48	
blake2b512	512	64	Hash out 64B
SHA512	512	64	
SHA3_512	512	64	

5.6 SECDED vs Hash functions

In this section, we discussed the result obtained by comparing both SECDED algorithms with best-performing hash functions. Figure 5.11 and Figure 5.12 illustrates the encoding and decoding execution time comparison between SECDED and hash-based software ECC. The vertical axis shows the log-scaled execution time in microseconds, while the horizontal axis shows different data sizes used in the comparison. Figure 5.11 shows the comparison of encoding operations. The result indicates, for smaller data sizes, modified SECDED murmur3, superfast hash and Blake3 are well-performing algorithms. When the data size increases, the encoding time for modified SECDED based algorithms is increasing significantly (almost a linear increase). Meanwhile, Blake3 and superfast hash exhibit a relatively small increase in encoding time compared to the other algorithms.

Figure 5.12 shows the same comparison for the decoding operations. In this computation also, Blake3 and superfast hash functions are also performing better in wide ranges of input data size. The performance of SECDED based algorithms becomes worse starting large data size. It is recommended for small data sizes. The other hash functions follow the same trend of an increase in execution time.

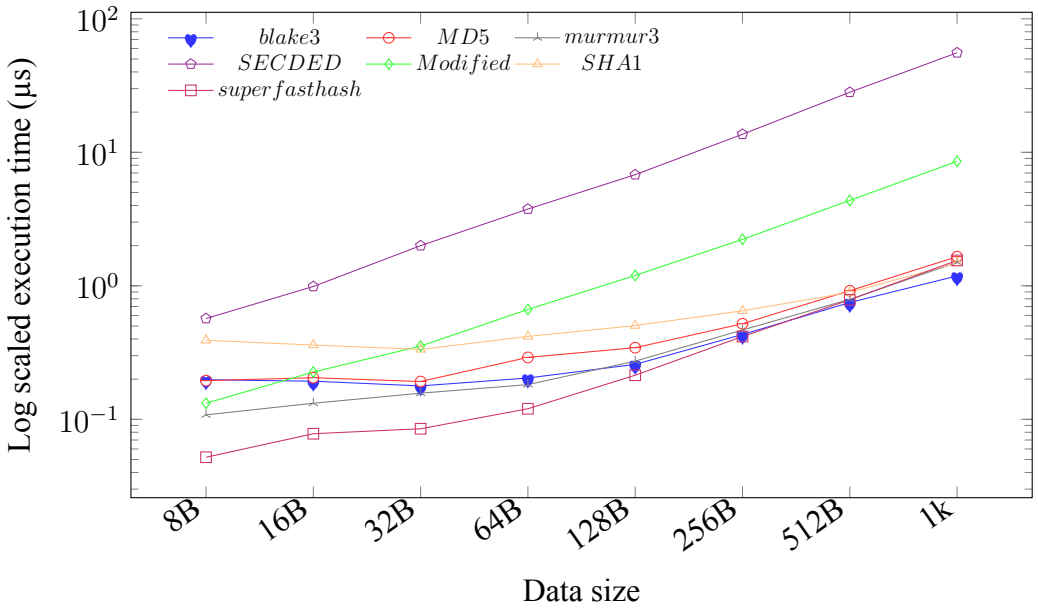


Figure 5.11: Log scaled encoding execution time comparison with 0-bit flip

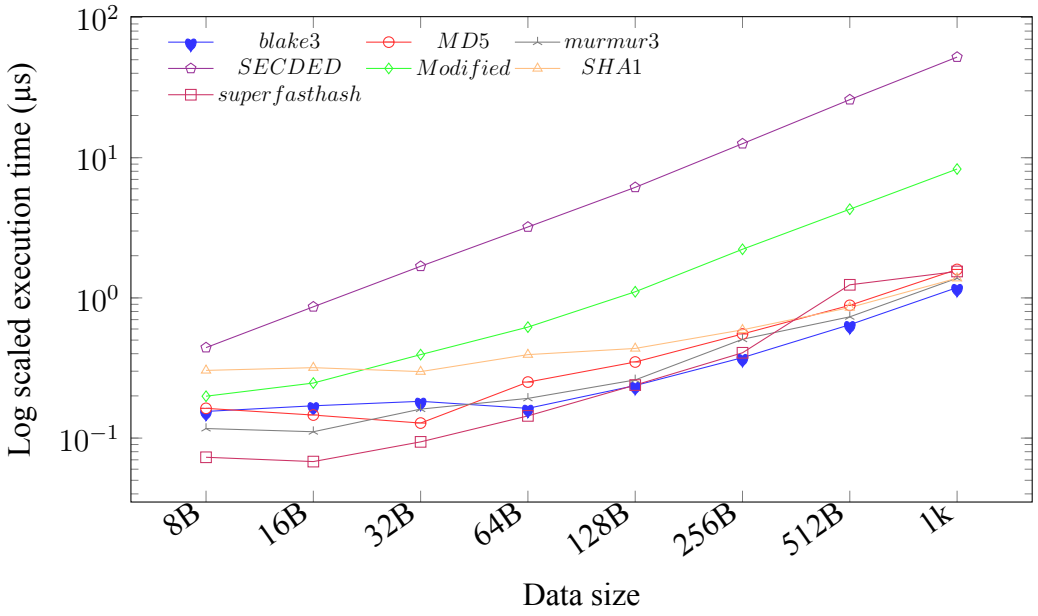


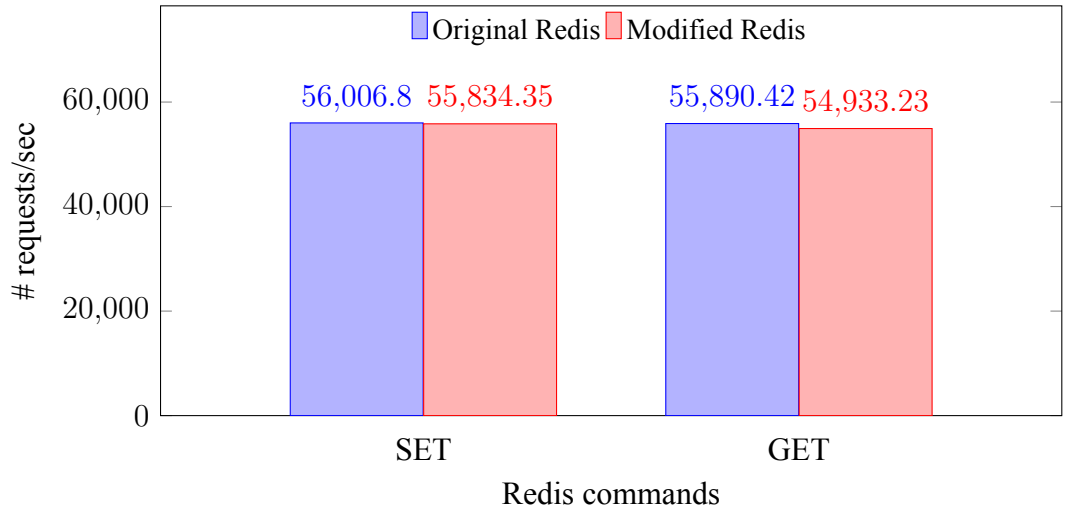
Figure 5.12: Log scaled decoding execution time comparison with 0-bit flip

5.7 Redis with hash-based software ECC

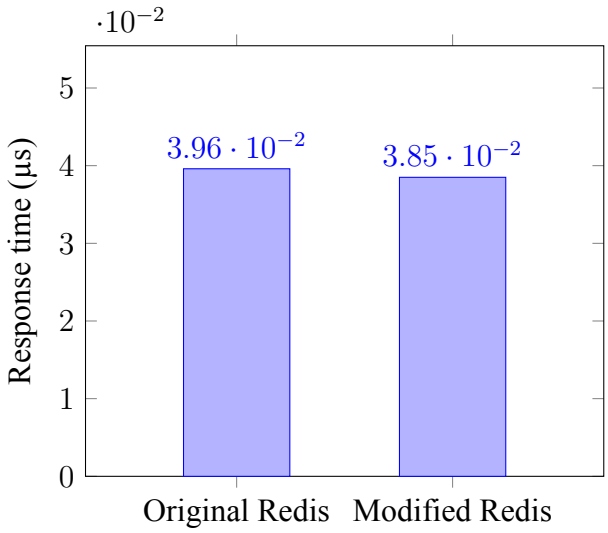
The performance of 22 CHFs and NCHFs is evaluated in the previous section. The evaluation shows that superfast hash is relatively performing better than the rest. In addition, this hash function has a small memory utilization overhead than the rest. Therefore, it is integrated into the Redis in-memory key-value store database, to evaluate the applicability of the hash-based software ECC. As discussed in section 4.6, the Redis server has its own benchmark tool (command) that is used to evaluate its performance on the installed machine. So, from this benchmark response time and the number of requests served per second are the two evaluation metrics that are used to reckon the performance of the Redis server with and without the integration of hash-based software ECC. The encoding and decoding execution of the hash-based software ECC is implemented on SET and GET Redis commands respectively, to provide bit flip protection. The SET command triggers the encoding operation that computes a hash value from the data and stores both the data and the hash. On the GET request, the decoding is executed to check the data isn't corrupted.

Figure 5.13a shows the overhead introduced to Redis SET and GET commands on the number of requests served per second, due to hash-based software ECC without introducing bit flips. The vertical axis shows the number of requests served per second and the horizontal axis shows the two commands used in this work. The original Redis refers to the performance of Redis without the integration of hash-based software ECC meanwhile the modified Redis refers to the result with integrated hash-based software ECC. The result indicates the overhead introduced to the Redis is $<0.5\%$ on SET and $<2\%$ on GET command.

Figure 5.13b shows the response time in microseconds on its y-axis that is computed with and without hash-based software ECC integration. The response time overhead created on Redis due to the implemented error detection feature without introducing a bit flip to the key-value pair is 2.5%.



(a) Number of requests per second



(b) Response time

Figure 5.13: Hash-based software ECC evaluation result on Redis

5.8 Comparison with previous works

The hash-based software ECC proposed in this work, address the gap of previous works by detecting any number of flip with acceptable overhead. Table 5.2 shows the comparison of our hash-based software ECC with previous works. Hash-based software ECC is capable of detecting any number of bit flips and can correct an adjustable number of bits using a brute-force algorithm. It is advisable to use the proposed method for relatively large input data. In the case where the input data is small, concatenating the input data and then applying the proposed algorithm is recommended.

Table 5.2: Comparison with previous works

Authors	Result	Limitation
Yoon and Erez [51]	-Detect any bit flip/chip	-works well for single chip
Rick and Castrillón[4]	-Detected up to 5-bit with 1.55X overhead.	-partial protection
Yin Li, et al. [55]	-Detect up to 3-bit flip	-In conjunction with ECC device -Partial protection
Hash-based software ECC	-Any bit flip detection - adjustable bit flip correction	- Space complexity - Addressed by increasing the data size

5.9 Summary

In this chapter, we have discussed the experiment results for modified SECDED and hash-based software ECC. Based on the results, we can conclude that the modified SECDED can be integrated into a device that doesn't have hardware support for ECC. With the optimization implemented on SECDED, the modified SECDED can efficiently serve for the detection of 2-bit flips and correction of single-bit flips.

The hash-based software ECC along with the brute-force bit flip correction algorithm, can also be used to detect any number of bit flips. This approach servers well when the input data is relatively large. The memory utilization also be reasonable with an increased input data size. The has-based software ECC integrated with Redis in-memory key-value store database and the recorded total overhead is $<3\%$. This indicates the proposed algorithm can be used to provide detection against any number of bit-flips and correction up to the 3 bit-flips in our case.

The two research questions mentioned in section 1.2 are answered as follows:

RQ1 What is the performance improvement of software SECDED implementations when hand-tuned for modern CPUs?

The bit manipulation and the use of intrinsic functions are used to optimize the performance of SECDED. The modified SECDED is up to 6x more efficient than the naive SECDED. Also, using intrinsic functions has a great effect on reducing the execution time of the SECDED algorithm than the bit manipulation. Overall, the result shows that for any device that requires protection from single-bit and double-bit flips, the modified SECDED serves well.

RQ2 What is the performance of software implementation of hash functions for efficient detection of multiple bit-flips?

The CHFs and NCHFs used in this work are used to detect any number of bit flips. Since the change in a single-bit value generates a significant change in the hash value, both CHFs and NCHFs can be used to detect any number of bit flips. Several bit flips were detected in the experiment demonstrating the feasibility of the proposed method. Based on the observed result Blake3 CHF performed better than the other hash functions and it is recommended as the primary choice for hash-based software ECC algorithm. Also, from NCHFs murmur3 and superfast hash function can be used for bit flip detection in the areas where memory utilization is critical. These mentioned three hash functions are capable of detecting a bit flip in less than 2µs from 1KB of data.

Chapter 6

Conclusion and Future Work

There are multiple reasons that can cause a memory bit/bits to flip. To address this issue, the SECDED algorithm has been integrated into the DRAMs. This type of DRAM is referred to as ECC-DRAM and has the capability to detect and correct single-bit flips and only detect but not correct 2-bit flips. As this approach is tailored for hardware implementation, we have significantly optimized the SECDED algorithm to be easily implemented and available on non-ECC devices. In addition, since the SECDED is limited to double-bit flip detection, we proposed the use of hash functions to detect multiple bit-flips and referred to as hash-based software ECC. By using hash-based software ECC, data integrity can be assured regardless of the number of bit-flips that could occur. In the conducted experiment, the Blake3 hash function demonstrated excellent performance for different input data sizes. Once the existence of a bit-flip is detected, the brute-force algorithm is used to identify and correct the flipped bit. In this work, up to three-bit flips were set to be corrected using this algorithm. Furthermore, we have integrated the software-based ECC into SET and GET commands of the Redis in-memory key-value store database and the result shows the total overhead of the proposed system remains below 3%.

To enhance the current state of error correction methods, it is recommended that further research be conducted to explore more efficient techniques for optimal implementation. In addition, integration of hash-based software ECC at a kernel level can be explored to provide more robust multiple-bit flip detection and correction. This process would provide bit-flip protection for the whole memory. The integration of such software is essential because it ensures that all data stored in the system's memory is protected from any possible bit flips that may occur without the need for dedicated ECC-DRAM.

References

[1] Shivani Tambatkar, Siddharth Narayana Menon, V. Sudarshan, M. Vinodhini, and N. S. Murty. Error detection and correction in semiconductor memories using 3d parity check code with hamming code. In *2017 International Conference on Communication and Signal Processing (ICCSP)*, pages 0974–0978, 2017.

[2] Minesh Patel, Jeremie S. Kim, Hasan Hassan, and Onur Mutlu. Understanding and modeling on-die error correction in modern dram: An experimental study using real devices. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 13–25, 2019.

[3] Seong Keun Kim and Mihaela Popovici. Future of dynamic random-access memory as main memory. *MRS Bulletin*, 43(5):334–339, 2018.

[4] Norman A. Rink and Jerónimo Castrillón. flexmedic : flexible memory error detection by combined data encoding and duplication. 2017.

[5] Hajime Kobayashi, Nobutaka Kawamoto, Jun Kase, and Ken Shiraish. Alpha particle and neutron-induced soft error rates and scaling trends in sram. In *2009 IEEE International Reliability Physics Symposium*, pages 206–211, 2009.

[6] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: A large-scale field study. volume 54, pages 193–204, 01 2009.

[7] Xin Li, Kai Shen, Michael Huang, and Lingkun Chu. A memory soft error measurement on production systems. pages 275–280, 01 2007.

[8] Xin Li, Michael C. Huang, Kai Shen, and Lingkun Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 6, USA, 2010. USENIX Association.

[9] JF Ziegler and WA Lanford. Effect of cosmic rays on computer memories. *Science (New York, N.Y.)*, 206(4420):776—788, November 1979.

[10] Adam Neale, Maarten Jonkman, and Manoj Sachdev. Adjacent-mbu-tolerant secded-taec-yaed codes for embedded srams. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(4):387–391, 2015.

[11] W.D. Swift. Memory errors: roll the dice! *IEEE Antennas and Propagation Magazine*, 38(6):124–125, 1996.

[12] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. *ACM SIGPLAN Notices*, 50:297–310, 05 2015.

[13] C.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.

[14] Jungrae Kim, Michael Sullivan, Seong-Lyong Gong, and Mattan Erez. Frugal ecc: efficient and versatile memory error protection through fine-grained compression. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[15] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 55–71, 2019.

[16] Jean-Philippe Aumasson, Willi Meier, Raphael Phan, and Luca Henzen. *The Hash Function BLAKE*. 01 2014.

[17] Rajeev Sobti and Geetha Ganesan. Cryptographic hash functions: A review. *International Journal of Computer Science Issues, ISSN (Online): 1694-0814*, Vol 9:461 – 479, 03 2012.

[18] César Estébanez, Yago Saez, Gustavo Recio, and Pedro Isasi. Performance of the most common non-cryptographic hash□functions. *Software: Practice and Experience*, 44(6):681–698, 2014.

[19] D Rachmawati, J T Tarigan, and A B C Ginting. A comparative study of message digest 5(md5) and sha256 algorithm. *Journal of Physics: Conference Series*, 978(1):012116, mar 2018.

[20] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

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

[22] Anil Kumar Singh. Error detection and correction by hamming code. *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*, pages 35–37, 2016.

[23] Ramadhan J. Mstafa and Khaled M. Elleithy. A highly secure video steganography using hamming code (7, 4). In *IEEE Long Island Systems, Applications and Technology (LISAT) Conference 2014*, pages 1–6, 2014.

[24] U. K. Kumar and B. S. Umashankar. Improved hamming code for error detection and correction. In *2007 2nd International Symposium on Wireless Pervasive Computing*, 2007.

[25] M. Y. Hsiao. A class of optimal minimum odd-weight-column sec-ded codes. *IBM Journal of Research and Development*, 14(4):395–401, 1970.

[26] Valentin Gherman, Samuel Evain, Nathaniel Seymour, and Yannick Bonhomme. Generalized parity-check matrices for sec-ded codes with fixed parity. In *2011 IEEE 17th International On-Line Testing Symposium*, pages 198–201, 2011.

[27] Michael Richter, Klaus Oberlaender, and Michael Goessel. New linear sec-ded codes with reduced triple bit error miscorrection probability. In *2008 14th IEEE International On-Line Testing Symposium*, pages 37–42, 2008.

[28] Adam Neale and Manoj Sachdev. A new sec-ded error correction code subclass for adjacent mbu tolerance in embedded memory. *IEEE Transactions on Device and Materials Reliability*, 13(1):223–230, 2013.

[29] Valentin Gherman, Samuel Evain, Fabrice Auzanneau, and Yannick Bonhomme. Programmable extended sec-ded codes for memory errors. In *29th VLSI Test Symposium*, pages 140–145, 2011.

[30] S Bakhtiari, R Safavi-Naini, and J Pieprzyk. Cryptographic hash functions: A survey. *Centre for Computer Security ...*, pages 1–26, 1995.

[31] Mahima Singh and Deepak Garg. Choosing best hashing strategies and hash functions. In *2009 IEEE International Advance Computing Conference*, pages 50–55, 2009.

[32] Jan Karásek, Radim Burget, and Ondřej Morský. Towards an automatic design of non-cryptographic hash function. In *2011 34th International Conference on Telecommunications and Signal Processing (TSP)*, pages 19–23, 2011.

[33] Mishall Al-Zubaidie, Zhongwei Zhang, and Ji Zhang. Reisch: Incorporating lightweight and reliable algorithms into healthcare applications of wsns. *Applied Sciences*, 10:2007, 03 2020.

[34] Jack O Connor, Jean-Philippe Aumasson, Neves Samuel, and Wilcox-O'Hearn Zooko. BLAKE3 one function, fast everywhere. 2020.

[35] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. pages 570–596, 07 2017.

[36] N. Sklavos and O. Koufopavlou. On the hardware implementations of the sha-2 (256, 384, 512) hash functions. In *Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS '03.*, volume 5, pages V–V, 2003.

[37] Tim Grembowski, Roar Lien, Kris Gaj, Nghi Nguyen, Peter Bellows, Jaroslav Flidr, Tom Lehman, and Brian Schott. Comparative analysis of the hardware implementations of hash functions sha-1 and sha-512. pages 75–89, 10 2007.

[38] Markku-Juhani Saarinen. Cryptanalysis of block ciphers based on sha-1 and md5. volume 2887, pages 36–44, 02 2003.

[39] Henri Gilbert and H. Hanschuh. Security analysis of sha-256 and sisters, selected areas in cryptography. *Ottawa, Canada, Lecture Notes in Computer Science 3006*, pages 175–193, 01 2003.

[40] Shu-jen Chang, Ray Perlner, William E. Burr, Meltem Sönmez Turan, John M. Kelsey, Souradyuti Paul, and Lawrence E. Bassham. NIST Interagency or Internal Reports 7896: Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition. Technical report, National Institute of Standards and Technology, Gaithersburg, MD, nov 2012.

[41] Morris Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions, 2015-08-04 2015.

[42] John Kelsey, Shu-jen Change, and Ray Perlner. NIST Special Publication 800-185 - SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash. *NIST Special Publication*, dec 2016.

[43] Bart Preneel, Antoon Bosselaers, and Hans Dobbertin. The cryptographic hash function ripemd-160. 05 2012.

[44] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the hash functions md4 and ripemd. In *Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'05, page 1–18, Berlin, Heidelberg, 2005. Springer-Verlag.

[45] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. Ripemd-160: A strengthened version of ripemd. In *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*, volume 1039 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 1996.

[46] Eric Thompson. Md5 collisions and the impact on computer forensics. *Digital Investigation*, 2(1):36–40, 2005.

[47] Zhao Yong-Xia and Zhen Ge. Md5 research. In *2010 Second International Conference on Multimedia and Information Technology*, volume 2, pages 271–273, 2010.

[48] Jie Liang and Xuejia Lai. Improved collision attack on hash function md5. *Journal of Computer Science and Technology*, 22:79–87, 01 2007.

[49] César Estébanez, Yago Sáez, Gustavo Recio, and Pedro Isasi. Performance of the most common non-cryptographic hash functions. *Software: Practice and Experience*, 44, 06 2014.

[50] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 415–426, 2015.

[51] Doe Hyun Yoon and Mattan Erez. Virtualized and flexible ecc for main memory. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, page 397–408, New York, NY, USA, 2010. Association for Computing Machinery.

[52] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generative software-based memory error detection and correction for operating system data structures. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2013.

[53] Bok-Min Goi, M.U. Siddiqi, and Hean-Teik Chuah. Computational complexity and implementation aspects of the incremental hash function. *IEEE Transactions on Consumer Electronics*, 49(4):1249–1255, 2003.

[54] Zulfany Erlisa Rasjid, Benfano Soewito, Gunawan Witjaksono, and Edi Abdurachman. A review of collisions in cryptographic hash function used in digital forensic tools. *Procedia Computer Science*, 116:381–392, 2017. Discovery and innovation of computer science technology in artificial intelligence era: The 2nd International Conference on Computer Science and Computational Intelligence (ICCSCI 2017).

[55] Yin Li, Hao Wang, Xiaoqing Zhao, Hongbin Sun, and Tong Zhang. Applying software-based memory error correction for in-memory key-value store: Case studies on memcached and ramcloud. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, page 268–278, New York, NY, USA, 2016. Association for Computing Machinery.

[56] Eyas El-Qawasmeh. Beating the popcount. *International Journal of Information Technology*, 9(1):1–18, 2003.

[57] Chenfan Sun and CC del Mundo. Revisiting popcount operations in cpus/gpus. In *ACM Student Research Competition Posters at the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2016.

[58] Michael E Thomadakis. The architecture of the nehalem processor and nehalem-ep smp platforms. *Resource*, 3(2):30–32, 2011.