Algorithm-Based Fault Tolerance and Matrix Multiplication Project

Justin LaForge
Student in Computer Science
Coastal Carolina University
jmlaforge@coastal.edu

***Abstract*—This report delves into the creation, manipulation, and analysis of 2D matrices through our proprietary programs, covering matrix generation, reading, and addition/multiplication. These programs are designed to handle matrix operations efficiently, ensuring data integrity. We introduce the project's background, emphasizing the significance of modular code design and random number generation. A series of experiments assess the efficiency and accuracy of the implemented programs, focusing on fault creation and tolerance in our corruption simulations. Results demonstrate the programs' effectiveness in generating, reading, and manipulating 2D matrices while maintaining robust error checking. The report concludes that implementing modular, fault-tolerant code is crucial for data manipulation, paving the way for advanced matrix tools in future larger programs.**

I. INTRODUCTION

The use of matrix libraries has been abundant throughout all technology and programs built to this day, and will continue to be used in crucial parts of computer technology. Because matrices are used in all components and are crucial to code production and data transferring, we need to find a way to insure data security and avoid any risk of corruption and error. We will learn in this project how to conduct experiments based on matrix data corruption and see how exactly data can be affected by single bit point corruptions, and we will do so by starting with the creation of matrices using 2D allocated arrays in the C programming language and then implementing the matrix multiplication process on these arrays and finally making programs to simulate data corruption so we can learn how to use matrix multiplication to correct it.

This test is performed because of the possibilities of data corruption which can be caused by solar, or cosmic, rays that pollute the world around us that have a small chance of affecting the electricity and hardware of computers at the microscopic level changing the bit values of our given data. This project acts as an acknowledgment and fixation toward the problem as we delve into how exactly this happens, what we can do to detect it and most importantly how we can fix it. We also look into how we get Masked Benign Errors, and the hamming data protecting process.

This whole process will be completed with the following programs:

- "`make_data_2d.c`": Our program that created our 2D arrays.
- "`cs_rows_data_2d.c`": Our program that check sums the row data of a given array.
- "`cs_cols_data_2d.c`": Our program that check sums the column data of a given array.
- "`cs_data_2d.c`": Our program that check sums the row data and column of a given array.
- "`add_data_2d.c`": Our program which adds the correponding data of two given arrays.
- "`mul_data_2d.c`": Our program that implements matrix multiplication on the two given arrays.
- "`corrupt_data_2d.c`": Our program which creates a fake corruption given a check summed array.
- "`detect_data_2d.c`": Our program that will detect where the corruption occurred.
- "`correct_data_2d.c`": Our program that corrects the corruption and gives us our desired value.
- "`mul_data_2d_faulty.c`": A program which simulates randomness data corruption in the middle of matrix multiplication to show the possibilities of Masked Benign Errors.
- "`read_data_2d.c`": Our program which reads a 2D array from a file.
- "`hamencode4.c`": A hamming encoder.
- "`utilities.c`": Contains all of our methods and functions.
- "`utilities.h`": Contains prototypes of our methods.
- "`timer.h`": Contains the timer function used to create randomness.

With all of these programs we can test and simulate how matrices are affected by corruption and how to fix it given matrix multiplication and summed data.

This paper is organized as follows. Background information is discussed in Section II, while Section III provides a look at the programs themselves.

In Section IV, the experiments are outlined and results are discussed to evaluate the enhancements. Section V concludes the paper.

II. BACKGROUND

Before we delve into this project we need to understand 2D matrix operations and their importance, Algorithm Based Fault Tolerance, what causes Masked Benign Errors (or MBEs). 2D matrix operations play a crucial role in computational tasks and applications in computers. Not only in computer science but also mathematics, 2D matrices are used for representing and processing data. They can be used for a wide range of operations, including addition and multiplication. Some fields that use matrix multiplication include linear algebra, computer graphics, physics simulations, and data analysis. In computer science these operations are fundamental for solving linear systems of equations, optimizing algorithms, and modeling complex systems which we computer scientists deal heavily with. This shows us how important 2D matrices are and it is our job to insure integrity of these data matrices. [1]

Algorithm-Based Fault Tolerance (ABFT) is a robust approach to detect and correct errors in computational tasks, particularly in scenarios where fault tolerance is crucial as mentioned above. ABFT techniques are designed to detect errors that might occur during the execution of algorithms and correct them to maintain the integrity of the results. In matrix multiplication, ABFT methods are used to detect potential errors and corruptions that occur due to hardware errors or other these cosmic rays. These methods often involve encoding the check sums and products of 2D matrices to allow for error detection and correction. ABFT methods enhance the reliability of matrix multiplication which can be crucial given the importance of a project. By implementing ABFT in matrix multiplication, researchers and engineers can ensure that their computations remain accurate and dependable even in the presence of these bit flips caused by hardware issues and cosmic rays, making ABFT a crucial component of scientific testing and data transfer. [2]

Masked benign errors are a quite fascinating occurrence in the context of data corruption. Unlike typical errors, which we implement in our "corrupt data" program, that are caused during matrix multiplication and lead to incorrect results however stay undetected. They occur when a fault, such as a random bit flip or hardware glitch due to cosmic rays, does not impact the final output of a computation. In other words, the error is "masked" and goes undetected by our standard error detection programs. This peculiar characteristic makes masked benign errors challenging to identify and correct. Though our data remains the same, these errors are still critical, whereas when a corruption occurs that you detect, you can figure out what is wrong with it and fix it, however when the error is undetectable in the first place there is nothing you can do to fix it. Understanding how and why these errors occur is crucial for developing fault tolerant programs and improving the reliability of computations, especially in programs where accuracy is key. In this report we will go into more detail on how we find these and how likely they are to occur in our computations.

III. Design

In this section we will go over how exactly our program works from top to bottom, and what we did to implement each step starting with the implementation of 2D Matrices, and finishing with running simulations with corrupt data multiplication. First, to understand 2D Matrices and their

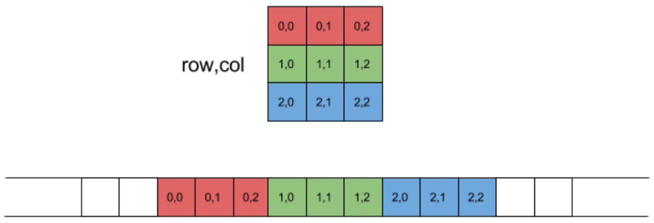


Fig. 1. The above photo represents how how a 2D Array looks in allocated space in the dynamic memory. The rows of the array are broken down to be able to fit in the memory and are aligned in order to allow for proper traversal methods.

importance, we must understand how to implement them. In order to create a 2D Matrix we must first allocate space in our dynamic memory, which is where our code and data is stored physically (all represented in either 32 bit integers), doing so will give us our very own space in the memory which we can then use to create our array, now with this space we can create pointers to our rows and columns. Now given our allocated array, we can add data into these spots in memory by simply using our function to make data, which traverses this allocated space and sets the value at each address to the given value, in our case will be a random number between a given low and high. For reference see Fig. 1. Additionally the pseudo code for our program is as follows:

```
Data: $A$, rows, cols
Result: Allocates space for a 2D array
while condition do
    allocate2d_3$A$, rows, cols
    $header \leftarrow$ rows $\times$ sizeof(int$*$);
    $data \leftarrow$ rows $\times$ cols $\times$ sizeof(int);
    $*A \leftarrow$ malloc($header + data$);
    array_data $\leftarrow$ (int$*$)($*A +$ rows);
    for $i \leftarrow 0$ to $rows - 1$ do
        $(*A)[i] \leftarrow$ array_data $+ i \times$ cols;
    end
end
```

Algorithm 1: allocate2d_3

```
Data: $A$
Result: Frees the allocated space for a 2D array
while condition do
    free2d_3$A$ free($*A$);
    $*A \leftarrow$ NULL;
end
```

Algorithm 2: free2d_3

Matrix row check sums and column check sums are the next step in the process, these are essential when creating our matrix correction programs as we use this data to determine what our correct values should be. To perform a row check sum we are given a 2D matrix and we go through each item in each row and add up the sum of the data, adding a new

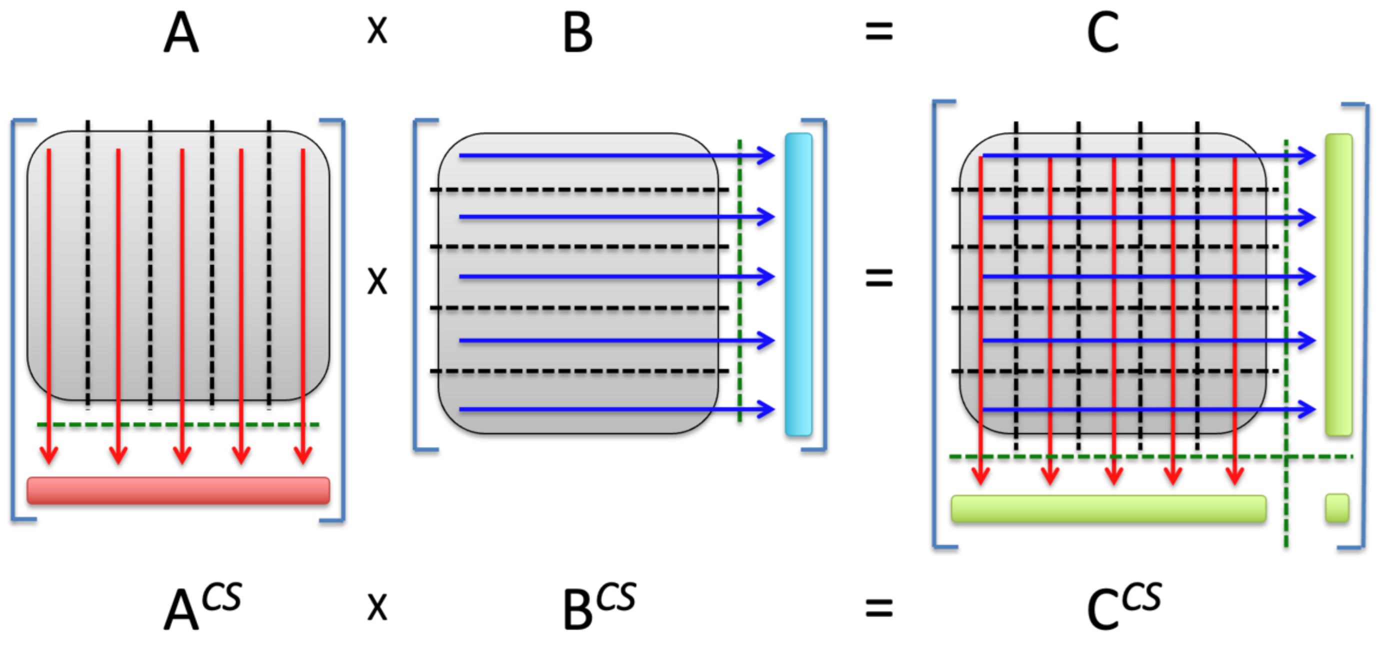


Fig. 2. The row and column checksums are performed by taking the sum of each row, or column, and adding the value of the corresponding rows/columns to a new column or row. The matrix A is performing a column sum and B is performing a row sum, with the red and blue bars being the list with the sums. Matrix C represents the checksummed product, or the result of the multiplication of two checksummed matrices.

column on the right end which includes the sum of the data on the indicated row. The same process is done for column check sums, adding a new row on the bottom to represent the summed data of the corresponding column. To do this in C we simply take the allocated array and traverse it using a nested for loop and add up each element of the rows or columns and add these values, along with our old values, to a new allocated array with an extra row or column. Finally we create a program that does both at the same time we we have the checksum of the rows on the right and the checksum of the columns on the bottom. For a visual representation see Fig. 2.

Once our check sums are complete we can perform matrix multiplication on these two new arrays. Matrix multiplication is the standard way of multiplying two matrices together and it follows this procedure: First the number of columns of the first matrix (B) must match the number of rows of the second matrix (A), then we take each row of B and multiply the values with the corresponding values in the columns of matrix A, for example say B's first row is (1, 2, 3), and the first column of A looks like (7, 9, 11), then we would see result of (1x7, 2x9, 3x11). And this step is repeated through all of the columns of A, and then all of the rows of B. This can be performed on any two matrices of correct size but when we multiply two check summed matrices, we should get a check summed product as seen in Fig. 2 matrix C. [3]

Finally given our new summed matrices, we can find out exactly where, if any, corruptions have occurred and what we need to do to fix it. To do so, we will go through and add up all of the data in each row and check if that sum is equal to the number at the end of the row, we do the same with

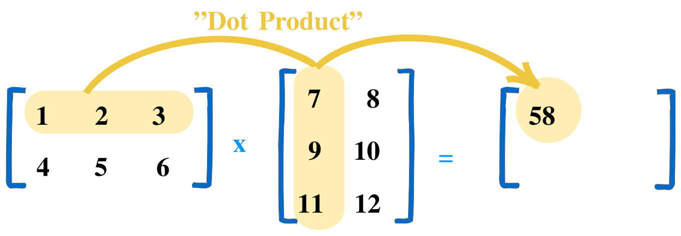


Fig. 3. The process of matrix multiplication is done by multiplying the rows of the first matrix by each column of the second matrix produce a output array with the same number of rows of the first matrix and the same number of columns in the second matrix. Also see [3]

the sums of the columns, and if an error does occur, the new sum of those rows and columns will not equal the last number (the original sum) which means an error has occurred in the corresponding row and column. Now given this data, we are able to calculate what our original value was supposed to be, simply by finding the sum of all the numbers we know are not corrupt and subtracting that from our total sum, and this will give us the number that we desire. At the same time, given our new correct value, and corrupted value, we can also figure out exactly what bit was flipped, to do so, we use the Exclusive OR operator (XOR) on the new and old numbers, which will give us a number when represented in binary, will be the exact bit that was flipped, to find that index we simply take the 2ND base log of that number. It is important to note that the error detection program will only work on a matrix that has already been summed.

In our project we created program drivers to follow through with all of these functions so that we can start experimenting

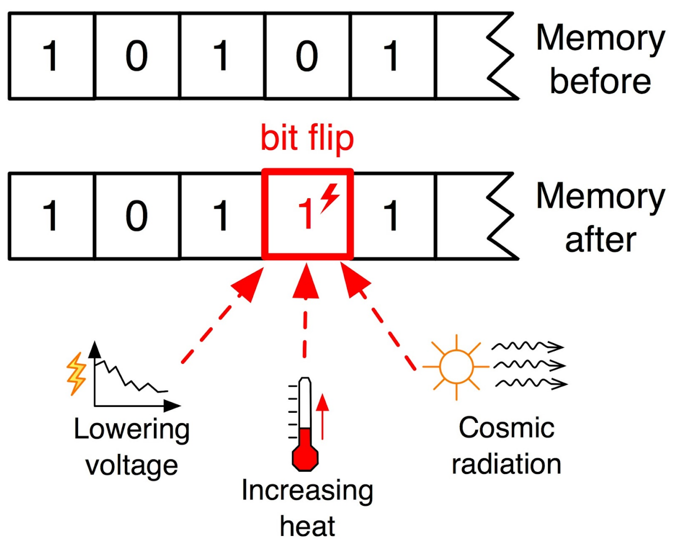


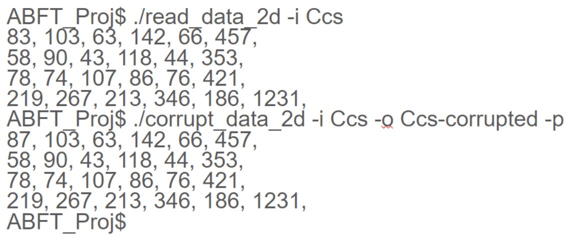
Fig. 4. This is a representation of how cosmic rays cant change the voltage of a given bit in memory and how it will significantly effect the data.

with this data. In all of our programs that involved returning a matrix, we had it print that data to a file which we could then read using our "`read_data_2d.c`" driver which simply takes a binary file with 2D array contents and reads that data to an allocated space and then prints the data to the terminal so we can see a visual of our data.

We also included the use of the OptArg function which provides users to input parameters and values from the command line allowing users to enter data such as file names, array sizes, and specific data requirements. This comes in handy later when we perform our experiments allowing us to experiment faster and easier. When it comes to having users input their own values, we need to account for possible errors which may break our programs. To prevent this from happening we create parameters which specify what data is required from the user assuring they do not enter any values that could cause our data to go out of bounds or mess with files and memory that we do not have access to.

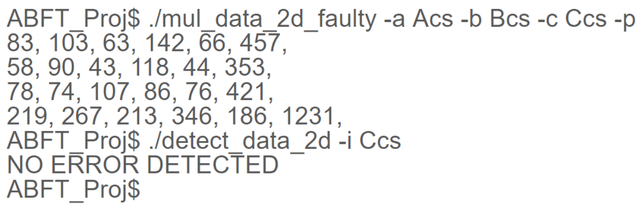
IV. Experiments and Results

Once we have all the essential code for creating matrices and their check sums and product sums we can start creating our program drivers that we will use for our experiments. The first one is the corrupt data function which we will use to fake a corruption in our matrices. To do this we have the user input a file containing the given matrix, and they can enter the exact value and bit they would like to flip given the row, column and bit position (If unspecified we do this at random). Then we traverse that given matrix and find the value at the position they want to corrupt and we take that value and flip the given bit, to do this we take the binary value of 1, and shift that 1 over the amount of times as the position we want to flip, then we use the XOR operator again with the value we want to change and this will create a corrupted value, and when looking at the binary value of the before and after numbers,



```
ABFT_Proj$ ./read_data_2d -i Ccs
83, 103, 63, 142, 66, 457,
58, 90, 43, 118, 44, 353,
78, 74, 107, 86, 76, 421,
219, 267, 213, 346, 186, 1231,
ABFT_Proj$ ./corrupt_data_2d -i Ccs -o Ccs-corrupted -p
87, 103, 63, 142, 66, 457,
58, 90, 43, 118, 44, 353,
78, 74, 107, 86, 76, 421,
219, 267, 213, 346, 186, 1231,
ABFT_Proj$
```

Fig. 5. This figure is a result of our corrupt data program running with random corruption on given matrices. The first matrix is our base data before any sort of corruption. In the second matrix the value of the first element, at index (0,0), was corrupted. (Note: Rows and Column indices start at 0).



```
ABFT_Proj$ ./mul_data_2d_faulty -a Acs -b Bcs -c Ccs -p
83, 103, 63, 142, 66, 457,
58, 90, 43, 118, 44, 353,
78, 74, 107, 86, 76, 421,
219, 267, 213, 346, 186, 1231,
ABFT_Proj$ ./detect_data_2d -i Ccs
NO ERROR DETECTED
ABFT_Proj$
```

Fig. 6. Here is an example of a Masked benign error. The above code was the result of a matrix multiplication with a corruption in the middle of the multiplication and yet the results still ended up being the same, and in this case, the detector did not detect an error because the output looks exactly the same as before the corruption.

we will see that the bit flipped at the exact position specified. See Fig. 5. for a result of our corrupt data program running.

To experiment further, we created a program driver which would corrupt data in the middle of a matrix multiplication. To do this we create a new driver that essentially performs the same task as the "`mul_data_2d.c`" program, only this one will corrupt a value mid multiplication to see what happens. This program allows the user to input the exact row, exact column, exact bit, and which product term will be corrupted. So to implement this we duplicate our original "`mul_data_2d.c`" program, this time we'll call it "`mul_data_2d_faulty.c`" and we'll add getOpt terms for x y z and k, then we will pass these parameters into our newly created faulty function which again performs the same main actions as the original function but at the given row x, the given col y, and the product term of k. It will call a function that performs a multiplication between two numbers however it will flip the given zth bit of the second number, and this will produce an error in our code at the given index.

By looking at it it seems as if this program is essentially doing the same thing as the corrupt data function, however by performing multiple tests of corrupting data and detecting the corruption we will actually see that there are actually occasions where no error is detected, this is not the result of no corruption occurring but it is actually the result of a Masked Benign Error, which occurs when two integers are multiplied and overflow occurs, overflow being the multiplication ended with a value bigger than the possible integer size, causing the rest of the number to be cut off and the remaining numbers end up being the same as the number in comparison. Though our

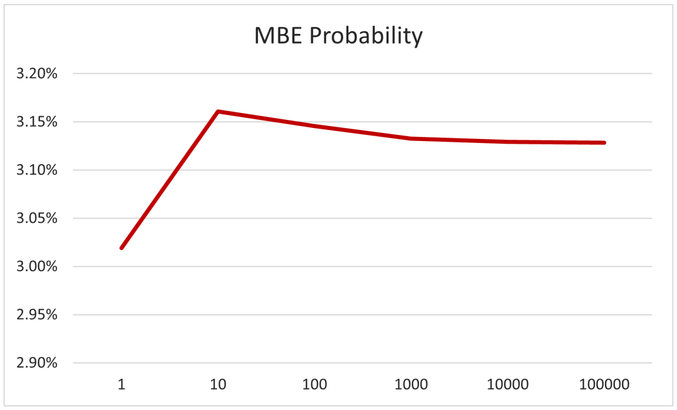


Fig. 7. This is the recorded data from the simple MBE test displaying how many times the flip in each bit position resulted in a Masked Benign Error. The x axis being the bit, and the y axis being the amount of times that bit resulted in an MBE.

data has not changed, a corruption has indeed occurred, this causes a problem in our fault tolerance program. See figure 6 for an example of a MBE.

Furthering our interest in Masked Benign Errors we created small simulation for MBEs. To do this we created a program that would loop through all integers of 32 bits and perform multiplications with another random integer and then we would flip a random bit of that integer and multiply that by the given integer as well and compare the two results to see if there was a masked benign error. To implement this program we first created a way to allow the user to input a given amount of trials. Then we created the outer loop which will loop through integers from 1 to 2^{32} , this being all possible 32 bit unsigned integers (Unsigned being positive integers), then with each of those integers, we created a loop of the given amount of trials, each trial would create a random 32 bit integer, and multiply it by the current integer, then it would take that random integer and flip a random bit, then multiply this by the current integer, then it would compare these results and if they were the same that meant a masked benign error had occurred and we would record this. Then once the program finished iterating all of the integers, it would report the probability of Masked Benign Errors based on the ratio between the amount of errors that occurred and the total number of trials. We used this data file to create a histogram from excel to record what digits, when flipped, were likely to cause an error most and to see how likely a MBE is to occur with a given amount of trials. See Figs. 7 and 8 for the results of this simple MBE test.

In the context of error detection and correction, the Hamming process leverages Hamming codes, a specialized form of error-correcting code devised by Richard Hamming. These codes are designed to enhance the reliability of binary data transmission by introducing additional bits in a systematic manner, forming a specific pattern within the code. This augmented pattern facilitates the identification and correction of errors that may occur during data transmission or storage. The placement of these extra bits strategically creates a

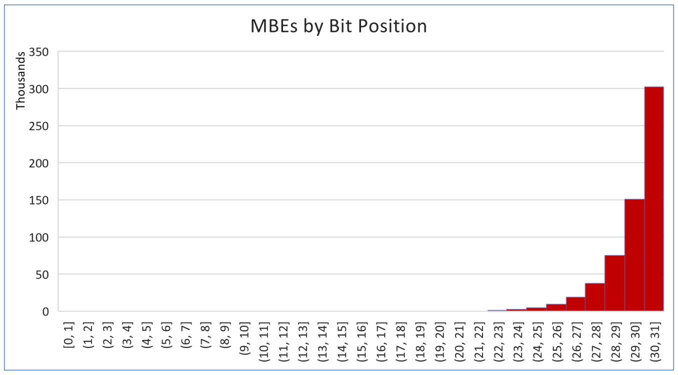


Fig. 8. This is the recorded data from the simple MBE test displaying how many times the flip in each bit position resulted in a Masked Benign Error. The x axis being the bit, and the y axis being the amount of times that bit resulted in an MBE.

$$\mathbf{G^T} := \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \qquad \mathbf{H} := \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Fig. 9. The G-Transpose matrix is a given matrix that was created to perform the hamming process and give us a matrix which represents our original binary value plus 3 new bits that are used to detect and correct any single bit corruption.

protected matrix known as the Hamming matrix, enabling the pinpointing of the bit position where an error has taken place.

The Hamming code method is particularly adept at detecting and correcting single bit errors and can also identify the presence of multiple bit errors within the transmitted or stored data. To implement the Hamming code, a program is employed to take a 4 bit binary number or a hex character, organizing its values into a 4 by 1 matrix. This matrix is then multiplied by the G-Transpose matrix, as illustrated in Figure 9. Subsequently, the resulting matrix undergoes a modulo operation with a base of two, converting the values back into binary digits (1's and 0's). This final encoded value, representing the original data with added error-detection and correction capabilities, is then converted back into a binary string for further transmission or storage [4]. [4]

V. Conclusion

In conclusion, this report has delved into the creation, manipulation, and analysis of 2D matrices using a set of programs designed for matrix generation, reading, and addition and multiplication. These programs have been implemented to handle matrix operations effectively while ensuring data integrity. Additionally, the report introduced the concept of fault creation and tolerance using programs that simulate data corruption caused by random bit flips in crucial conditions.

The background of the project emphasized the importance of modular code design and the use of random number generation for creating data. A series of experiments were conducted to test the efficiency and accuracy of the implemented programs, revealing that they effectively generate, read, and manipulate 2D matrices while maintaining robust error checking mechanisms.

The report underlines the significance of successful implementation of modular, fault tolerant code for data manipulation and processing. This achievement lays the groundwork for the development of more complex matrix tools to be used by larger programs in the future, ensuring the integrity of data in the face of potential corruptions and errors.

In summary, this project acknowledges the importance of data integrity and error detection in the realm of 2D matrices, demonstrating the effectiveness of modular code and fault tolerant mechanisms in addressing these challenges. It sets the stage for the continued development of advanced matrix tools and data manipulation techniques.

To further investigation we could test these scenarios in real life conditions and projects to see how dramatically they can change real world data. The possibilities of fault testing are endless and is important in all tech and science field as digital data is usually something we do not want to change. With these programs we can test and solve many cases of faults in digital data.

References

[1] 2d array: All you need to know about two-dimensional arrays. [Online]. Available: https://www.simplilearn.com/tutorials/data-structure-tutorial/two-dimensional-arrays

[2] Algorithm-based fault tolerance: a review. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S014193319700029X/pdf?md5=5ec3902657df7245a7a28add17b27c92pid=1-s2.0-S014193319700029X-main.pdf

[3] Matrix multiplication. [Online]. Available: https://blog.cambridgecoaching.com/how-to-multiply-matrices-quickly-and-correctly-in-six-easy-steps

[4] Hamming encoder. [Online]. Available: https://www.omnicalculator.com/other/hamming-code