Solving Systems of Linear Equations over GF(2) on FPGAs

A Major Qualifying Project

Submitted to the Faculty of Worcester Polytechnic Institute

In partial fulfillment of the requirements for the Degree of Bachelor of Science in

Electrical and Computer Engineering

By

James McAleese

Date: 8/13/2021

Project Advisor: Koksal Mus

Acknowledgements

I would like to thank the Electrical and Computer Engineering department and Worcester Polytechnic Institute as a whole for providing the opportunity and resources necessary to conduct the research for this project. I would also like to give a mention of thanks to Professor Jakub Szefer of Yale University for providing me with sample code and insights from a related paper he authored. Lastly, I owe a great deal of gratitude to Professor Koksal Mus for serving as my advisor and for presenting me this opportunity to work on an MQP that fit my academic timeline. His guidance was vital to the completion of this project and I would not have been able to complete it had he not been so active in the research process with me.

Abstract

The focus and scope of this project are to reduce the computational complexity and time complexity required to find solutions to large systems of linear equations with binary coefficients and to implement this reduced method on FPGA hardware. Beginning with a simple exhaustive search to check all possible solutions against every equation in the system, continuous research and calculation resulted in various iterations of a reduced search algorithm. Each version attempted to take advantage of inherent patterns in the input system, or of mathematical principles that arise when working with systems with binary coefficients. The resulting algorithm is the combination of Gaussian Elimination and Partial exhaustive search algorithm with sub-exponential complexity. The provided algorithm makes use of counting 1s coefficients to recursively find portions of valid solutions for each equation in the system and combines those portions to generate full solutions to the system. The C implementation of the final *RecursiveSearch()* function can be found in the Appendices, as well as the C implementations of the *ExhaustiveSearch()* function and the failed search attempts.

Table of Contents

Acknowledgements **1**

Abstract **2**

Table of Contents **3**

Table of Figures **4**

Introduction **5**

The RecursiveSearch Algorithm **7**

Early Search Attempts **14**

Exhaustive Search 14

Using Linear Combinations 17

Effects of Overdetermined Systems 17

Reducing with Linear Combinations 18

Inversion of Coefficients 20

Taking Intersections of the Input System 21

Results and Conclusions **25**

Results 25

Future Research 26

Conclusion 27

References **28**

Appendix **29**

C Implementation 29

Main.c 29

Parameters.h 31

RecursiveSearch.c 32

RecursiveSearch.h 38

VariableExhaustiveSearch.c 39

VariableExhaustiveSearch.h 46

Table of Figures

Figure 1:	The layout and sections of the matrix after sorting with the upper triangle (red), sectioned 1s (green), and remaining data (gray) are shown.	8
Figure 2:	The Equations in our Example System.	10
Figure 3:	The Example System converted into a Matrix, unsorted and unsolved.	10
Figure 4:	The original Example Matrix now sorted with original columns and rows labelled. The data is highlighted in the style of Figure 1 for reference.	11
Figure 5:	Generating a Solution with the Example Matrix after sorting.	11
Figure 6:	The step by step process of verifying a solution.	13
Figure 7:	The Initial method of verifying a solution through successive XOR operations.	15
Figure 8:	Successive XOR operations reduce to 0 or 1 if the number of input 1s is even or odd, respectively.	16
Figure 9:	The updated method of verifying a solution through counting 1s.	16
Figure 10:	The contents of the *VariableExhaustiveSearch()* function.	18
Figure 11:	Inverting the coefficients of an equation does not guarantee the same solutions.	20
Figure 12:	Bit states of valid solutions can be predicted from the arrangements of 1s in an equation.	22
Figure 13:	The number of solutions can be calculated from the predicted bit arrangements.	23
Figure 14:	Finding solutions by assigning variables to columns is inefficient at scale.	23

Introduction

In the world of cryptography, computer security relies on the generation of large "key" numbers to secure data. Various types of schemes exist to solve such systems which break down mainly into 5 categories: Code-based, hash-based, isogeny-based, lattice-based, and multivariate-based schemes. For this paper, we focused on the development of multivariate schemes, which are efficient when implemented on low-resource hardware like an FPGA. When attempting to solve such systems with computers, we treat the system as a matrix to take advantage of the various linear algebra properties which arise. Additionally, working with low-level computer hardware restricts the values within a matrix, and within the solutions to be found, to a Galois Field (or Finite Field) size of 2 denoted GF(2). In general terms, this means the possible matrix and solution values are limited to a set of two terms. For the purposes of computing, we restrict specifically to the binary digits 0 and 1.

In theory, finding solutions to such matrices is an easy task. However, when working in GF(2) many of the normal rules of Linear Algebra and mathematics, in general, may not produce results as expected. Under normal conditions, a system of equations can have no solutions, one solution, or infinitely many solutions. When restricted to binary digits a system can still have no solutions or one solution, but rather than the third option being an infinite amount, the number of possible solutions is limited to 2^v where v is the number of variables in the system.

Additionally, as the level of security needed increases the size of these matrices must increase accordingly, often to a point that finding solutions becomes costly and prohibitive both in terms of the time required and the computational power needed.

Current proposals attempt to use various forms of Gaussian Elimination (GE) or matrix inversion to find solutions (Bardet et al., 2013; Keinänen et al., 2005; Wang et al., 2016). However, these methods have their own drawbacks.

In the papers authored by Keinänen et al. and Wang et al., both methods proposed breaking up an input matrix into blocks to be solved piece by piece as a way of accommodating matrices otherwise too large for their methods. Wang et al. specifically mention employing systolic architecture to repeatedly perform GE on smaller subsections which are then used to solve the entire input matrix. Within the bounds of GF(2) however, the multiplication and addition operations required to perform a full Gaussian Elimination either results in matrix values beyond 0 and 1. Or when using boolean operations to multiply and add, relevant data is removed in the process, resulting in false solutions being generated.

In this paper, we propose a different method of finding solutions entirely. Rather than attempting to perform full GE on a system as in some of the other proposals, we instead take elements of the GE process to sort a matrix into an upper triangular form and perform partial exhaustive searches of the partial solutions that satisfy the 1s in each equation. These partial solutions are chained together to create full solutions to the system. This simultaneously eliminates multiple solutions at once during the search while avoiding the data loss issues that arise when performing GE within GF(2). In doing so, we believe that we have found a novel method of solving large systems of binary linear equations and that with further improvements it could be a possible future method of solving quadratic equations as well.

The RecursiveSearch Algorithm

Having successfully found a method of finding solutions that work well at a small scale, after experimenting with different ways of interpreting and modifying the system. Eventually, we created our final algorithm *RecursiveSearch()*, which makes use of recursion to find partial solutions without exponential scaling. Normally, performing GE on a matrix involves performing scalar multiplication on rows, or adding them together. However, when working in GF(2), to keep values restricted to 0s and 1s we must substitute algebraic Boolean operations to accomplish the same effect.

For most purposes AND operations and XOR operations, each serves the same function as multiplication and addition, but when trying to use them for GE issues arise. Our observation is that using row operations in GE introduces irrelevant solutions to the system or excludes some solutions from the solution set. Because AND operations require both inputs to be 1 for a 1 output, any scalar row multiplication involving a 0 and a 1 ends up removing data from the system. Likewise with XOR operations, because row value is a single digit, independent number, any operation with two 1s also removes data because there's no second digit for the result to carry-over to.

With all this taken into account, rather than looking at the entire system of equations at once in an attempt to modify GE for these restrictions, our new *RecursiveSearch()* instead divides the system into independent and dependent pieces. Solutions of the independent pieces are solved by partial searches and inserted into dependent parts to keep the whole system consistent. Independent parts are represented by green blocks and dependent parts are represented by gray blocks in Figure 1.

Sort the Matrix to Create an Upper Triangle of 0s and a Section of Grouped 1s on Each Row

S_n	...	S_3	S_2	S_1	RHS	
{0, 0, 0, ... , 0}	{0, 0, 0, ... , 0}	{0, 0, 0, ... , 0}	{0, 0, 0, ... , 0}	{1, 1, 1, ... , 1}	{1 or 0}	E_1
{0, 0, 0, ... , 0}	{0, 0, 0, ... , 0}	{0, 0, 0, ... , 0}	{1, 1, 1, ... , 1}	{1s & 0s}	{1 or 0}	E_2
{0, 0, 0, ... , 0}	{0, 0, 0, ... , 0}	{1, 1, 1, ... , 1}	{1s & 0s}	{1s & 0s}	{1 or 0}	E_3
{0, 0, 0, ... , 0}	{1, 1, 1, ... , 1}	{1s & 0s}	{1s & 0s}	{1s & 0s}	{1 or 0}	...
{1, 1, 1, ... , 1}	{1s & 0s}	{1s & 0s}	{1s & 0s}	{1s & 0s}	{1 or 0}	E_n

Figure 1:The layout and sections of the matrix after sorting with the upper triangle (red), sectioned 1s (green), and remaining data (gray) are shown.

Before *RecursiveSolve()* can work the matrix first has to be sorted, with the end goal being a group of 1s in each row we can later increment through to find solutions as seen in the green blocks in Figure 1 above. Any 1s in positions in the grey blocks beneath the triangle are irrelevant to the solution, the states of those bit positions will be fully determined by the 1s in green. The *SortArray()* function sorts the matrix to form an upper triangle of 0s, bounded by 1s much like with GE. Each row is assigned a weight equal to the number of 1s and then resorted from heaviest to lightest going down. Then within each row, any column with its first 1 in that row is shifted to the right as far as the end of the group of 1s in the previous row. Once sorted, the *RecursiveSolve()* function is then implemented to begin the search process.

Within each row, *RecursiveSolve()* increments through partial solutions only relevant to the grouped 1s, staying within the bounds of the S_1- S_n columns defined in Figure 1. To check the validity of each solution, each individual element in the solution is multiplied using AND operations by its respective value in the row. Normally, these products would then be added together with XOR operations to get a single digit sum to be compared against the Right Hand Side (RHS) value. However, we can bypass this ANDing and successive XORing by instead counting how many 1s are in the tested solution and checking if that count is even or odd. With successive XORing if the number of 1s is even XOR operations will always result in 0, and if the number is odd the XOR operations will always result in 1, as seen in Figure 8. And because we're always AND multiplying by a set of only 1s, the product set will always be identical to the input set.

As further solutions beyond the first row are iterated through, each one is added to the valid partial solutions for the previous equations before being tested against the current RHS value. Adding the partial solutions to each other in this fashion restricts new ones to the bounds set by the previous ones, ensuring the solutions remain valid for the whole system as they're built. When a full solution has been built, the *tempsol[]* array is tested against any possible remaining rows in the system. The *bin2dec()* function then converts *tempsol[]* to its decimal equivalent and appends the number to a list of solutions (This step is purely for making the tracking of solutions easier for people to read, it has no actual effect on finding solutions).

Finally, after the solution has been converted and stored, *RecursiveSolve()* resets to the top level of recursion and resumes iterating through partial solutions for the first row to find the next full solution. This process continues back up and down through the equations until all

possible combinations of valid partial solutions have been found and all their resulting full solutions recorded. We can see an example of this entire process below in Figures 2 - 6 as we take an example system, convert it to a matrix, and find one of its solutions.

The Equations in the System are as Follows:

$$E_0 : x_6 + x_4 + x_2 + x_1 + x_0 = 0$$

$$E_1 : x_8 + x_7 + x_6 + x_1 = 0$$

$$E_2 : x_5 + x_3 + x_2 + x_0 = 0$$

$$E_3 : x_9 + x_8 + x_6 + x_5 + x_0 = 0$$

$$E_4 : x_8 + x_2 = 0$$

Figure 2: The Equations in our Example System.

	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	RHS
E_0	0	0	0	1	0	1	0	1	1	1	0
E_1	0	1	1	1	0	0	0	0	1	0	1
E_2	0	0	0	0	1	0	1	1	0	1	0
E_3	1	1	0	1	1	0	0	0	0	1	0
E_4	0	1	0	0	0	0	0	1	0	0	1

Figure 3: The Example System converted into a Matrix, unsorted and unsolved.

	x_3	x_7	x_9	x_8	x_5	x_6	x_4	x_2	x_1	x_0	RHS
E_0	0	0	0	0	0	1	1	1	1	1	0
E_3	0	0	1	1	1	1	0	0	0	1	0
E_1	0	1	0	1	0	1	0	0	1	0	1
E_2	1	0	0	0	1	0	0	1	0	1	0
E_4	0	0	0	1	0	0	0	1	0	0	1

Figure 4: The original Example Matrix now sorted with original columns and rows labelled. The data is highlighted in the style of Figure 1 for reference.

Our Sample System from our Coding Process sorted and solved with RecursiveSearch():

S_4	S_3	S_2	S_1	RHS	
{0}	{0}	{0, 0, 0}	{1, 1, 1, 1, 1}	{0}	E_0
{0}	{0}	{1, 1, 1}	{1, 0, 0, 0, 1}	{0}	E_3
{0}	{1}	{0, 1, 0}	{1, 0, 0, 1, 0}	{1}	E_1
{1}	{0}	{0, 0, 1}	{0, 0, 1, 0, 1}	{0}	E_2
{0}	{0}	{0, 1, 0}	{0, 0, 1, 0, 0}	{1}	E_4
{0}	{1}	{1, 1, 0}	{0, 0, 0, 1, 1}	Solution	

Figure 5: Generating a Solution with the Example Matrix after sorting.

We start with our example system (Figure 2) which we convert into an unsorted example matrix (Figure 3). Once sorted, the matrix is laid out as shown in Figure 4, and grouped as shown in Figure 5, at which point we can test for solutions.

When testing our sample solution$\{0, 0, 0, 1, 1\}$against our S_1column $\{1, 1, 1, 1, 1\}$ we can see that were we to AND them like so:

$$\{(1\&0), (1\&0), (1\&0), (1\&1), (1\&1)\} = \{0, 0, 0, 1, 1\}$$

We would get our sample solution again as described above, proving that we can skip this step. When we then count the 1s, we can see that we have two 1s meaning if we XOR them together $1 \oplus 1 = 0$, or take the parity of the count we get 0 for our sum. Lastly for this row if we check the RHS value, we can see that the RHS and the sum are both 0, meaning$\{0, 0, 0, 1, 1\}$ is a valid partial solution to equation 1. For subsequent rows, this process continues in much the same way but with the sums from the previous rows added to the current before checking the RHS value to keep partial solutions relevant for the rows above the current one.

Testing a Sample Solution to the System:

1. $\{0, 0, 0, 1, 1\} \Rightarrow two\ 1s \Rightarrow sum = 0 \Rightarrow sum = E_0\ RHS \Rightarrow Partial\ Sol.\ is\ Valid$

2. $\{1, 1, 0\} \Rightarrow two\ 1s \Rightarrow sum = 0 \Rightarrow sum = E_3\ RHS \Rightarrow Partial\ Sol.\ is\ Valid$

3. $\{1\} \Rightarrow one\ 1 \Rightarrow sum = 1 \Rightarrow sum = E_1\ RHS \Rightarrow Partial\ Sol.\ is\ Valid$

4. $\{0\} \Rightarrow zero\ 1s \Rightarrow sum = 0 \Rightarrow sum = E_2\ RHS \Rightarrow Partial\ Sol.\ is\ Valid$

5. $All\ Partial\ Sols.\ Valid,\ Total\ Sol.: \{0, 1, 1, 1, 0, 0, 0, 0, 1, 1\}$

6. $Total\ Sol.\ Against\ Last\ row: \{(0\&0), (0\&1), (0\&1), (1\&1), (0\&0), (0\&0), (0\&0), (1\&0), (0\&1), (0\&1)\}$

$\Rightarrow \{0, 0, 0, 1, 0, 0, 0, 0, 0, 0\} \Rightarrow one\ 1 \Rightarrow sum = 1 \Rightarrow sum = E_4\ RHS \Rightarrow Total\ Sol.\ is\ Valid$

7. $Total\ Solution\ \{0, 1, 1, 1, 0, 0, 0, 0, 1, 1\}\ is\ valid\ for\ the\ whole\ system,\ Decimal\ Index = 483$

Figure 6: The step by step process of verifying a solution.

By basing the search of partial solutions for the current equation on the ones already found for the previous equations, we not only ensure that the partial solutions being found remain valid for all equations before and after the current one, but we find solutions more quickly than *ExhaustiveSearch()* by checking fewer invalid ones. Once a partial solution is found to be invalid none of the possible combinations that could follow it are even tested. This eliminates a large number of invalid solutions altogether without having to fully test them.

Early Search Attempts

Exhaustive Search

As already established, by working exclusively in GF(2), any possible solution to the system will consist only of 1s and 0s. Therefore, no matter the system size it is possible to treat possible solutions as unsigned binary numbers of a length 2^n where *n* is the number of variables in the system, designated within the C code as MaxBin. This has the combined benefit of being able to increment through solutions one by one, and allowing solutions to be tracked and sorted by assigning their decimal equivalent as an index number. With this in mind, I designed two functions, *BinArrayAdd()*, and *ExhaustiveSearch()* to check and record every possible solution.

BinArrayAdd() takes an input array and uses internal variables and a special array called *AddOne[]* containing only a 1 in the Least Significant Bit (LSB) position to function as a ripple carry adder. In a loop starting with the LSB, the function uses XOR operations to add together the current bits of the input and AddOne as well as a carry bit and assigns it to a temporary array while the carry for the next iteration is assigned based on the current input and carry bits. After the carry bit assignment, the temporary value is written back into the input array.

ExhaustiveSearch() contains a large FOR loop with a limit of MaxBin. In each loop iteration, *BinArrayAdd()* is called to generate a solution. The solution is then tested with the RHS of each equation, and if the solution satisfies all equations in the system, a 1 is marked at the current loop position in the array *Valid[]* so the decimal index of the solution can later be printed.

Initially, possible solutions within *ExhaustiveSearch()* were checked by multiplying each bit of the solution with its respective bit coefficient bit in the Left Hand Side (LHS) of the current equation using AND operations, and these products are then XORed together to a single bit sum. That bit was then compared against the RHS, and if the two matched then the solution was valid for that equation.

```
1  EqSum[i] = 0; //making sure EqSum is zero before calculating for the current row
2  for (j = 0; j < Variables; j++) //for each element/variable in each row
3  {
4      //multiplying the coefficients in the LHS with the solution bits
5      CoefProd = LHS[i][j] & x[j];
6      //reapeatedly adding the current equation sum with the CoefProd from the row above
7      EqSum[i] = EqSum[i] ^ CoefProd;
8  }
```

Figure 7: The initial method of verifying a solution through successive XOR operations.

However, by taking advantage of the inherent rules of XOR operations this process could be accomplished with simpler logic than successive XORing.

A standard XOR operation takes two inputs and outputs a 1 if only one of the two inputs is equal to 1, otherwise it outputs a 0. Since XOR operations are both commutative and distributive, an XOR operation with more than two inputs can be interpreted as successive XORing of additional bits with the result of previous XOR operations, and the logic of the two input scenario can be generalized as the output will be 1 if an odd number of the inputs is 1. Any even number of 1s *n* functions as a set of *(n/2)* $1 \oplus 1$ pairs that each cancel to 0. Any odd number of ones *m* would be equivalent to *(n+1)* 1s adding up to *(n/2)* $1 \oplus 1$ pairs with a single 1 remaining. With this in mind, all that is needed to check if a solution is valid for an equation is to count the number of 1s (onesCount) in the sum of products stage of the initial method. If onesCount modulo 2 equals the RHS of the equation, the solution is valid.

When $(X_1, X_2, X_3, X_4 \ldots X_n) = 1$:

If n is even:

$$X_1 \oplus X_2 \oplus X_3 \oplus X_4 \oplus \ldots \oplus X_n = (X_1 \oplus X_2) \oplus (X_3 \oplus X_4) \oplus \ldots \oplus (X_{(n-1)} \oplus X_n)$$

$$\Rightarrow (1 \oplus 1) \oplus (1 \oplus 1) \oplus \ldots \oplus (1 \oplus 1) \Rightarrow 0 \oplus 0 \oplus \ldots \oplus 0 \Rightarrow 0 \oplus 0 = 0$$

If n is odd:

$$X_1 \oplus X_2 \oplus X_3 \oplus X_4 \oplus \ldots \oplus X_n = (X_1 \oplus X_2) \oplus (X_3 \oplus X_4) \oplus \ldots \oplus (X_{(n-2)} \oplus X_{(n-1)}) \oplus X_n$$

$$\Rightarrow (1 \oplus 1) \oplus (1 \oplus 1) \oplus \ldots \oplus (1 \oplus 1) \oplus 1 \Rightarrow 0 \oplus 0 \oplus \ldots \oplus 0 \oplus 1 \Rightarrow 0 \oplus 1 = 1$$

Figure 8: Successive XOR operations reduce to 0 or 1 if the number of input 1s is even or odd, respectively.

```
1   sameones = 0;
2   EqSum[i] = 0; //making sure EqSum is zero before calculating for the current row
3   for (j = 0; j < Variables; j++)
4   {
5       if ((LHS[i][j] == 1) && (x[j] == 1))
6       {
7           //if the bit in the equation and solution are both 1, increment sameones
8           sameones += 1;
9       }
10  }
11  if (sameones % 2 == 1)
12  {
13      EqSum[i] = 1; //if sameones is odd, the equation sum is odd
14  }
15  else if(sameones % 2 == 0)
16  {
17      EqSum[i] = 0; //if sameones is even, the equation sum is even
18  }
```

Figure 9: The updated method of verifying a solution by counting 1s.

With a working Exhaustive Search method finalized, we not only had a way of checking our results, and initial methods involved exploring ways to manipulate the main ExhaustiveSearch function and the initial system to streamline the process.

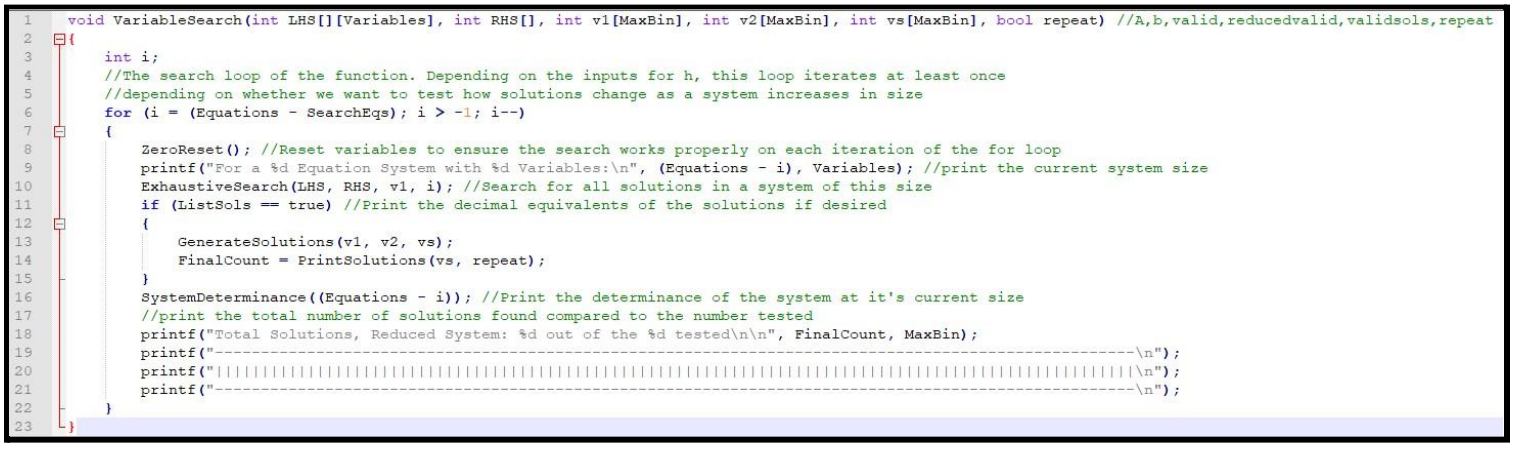
Using Linear Combinations

The first attempts at simplifying the search process all involved variations of taking linear combinations of the initial system and finding their solutions. As is true under normal conditions, within GF(2) the solutions to a system of two equations are also solutions to the linear combination of those two equations. Unlike normal conditions, however, standard addition cannot be used to create these linear combinations, as this would result in integer values other than 0 and 1 in code output. Therefore, XOR operations must be used in place of addition, like when verifying solutions during an exhaustive search.

Effects of Overdetermined Systems

To ensure that linear combinations would still result in solutions under GF(2), we first had to design a series of functions that would create linear combinations of the input equations by XOR adding together individual elements of equations and then append those combinations to the initial system to make a larger array. This process was broken down such that instead of one large function setting everything up and doing the calculations, separate functions handled the steps. The combinations calculations were handled by *LinCombs()* while the appending them was handled by *LoadArrays()*. Additional functions beyond these were also written to support tasks such as printing test results to text files and tracking if the system is over- or under-determined, among other things.

All of these were used in support of a separate main search function called *VariableExhaustiveSearch()*, created to calculate solutions of this new system as equations were repeatedly added. By giving *VariableExhaustiveSearch()* a starting size less than the number of initial equations and having it increase the system size by one equation in each loop iteration, we could see how the number of solutions to a system decreased as the system shifted away from being under-determined. Additionally, when the system shifts from an equal number of equations and solutions into an overdetermined state with the linear combinations added the new system retains the solutions of the initial system.



```
void VariableSearch(int LHS[][Variables], int RHS[], int v1[MaxBin], int v2[MaxBin], int vs[MaxBin], bool repeat) //A,b,valid,reducedvalid,validsols,repeat
{
    int i;
    //The search loop of the function. Depending on the inputs for h, this loop iterates at least once
    //depending on whether we want to test how solutions change as a system increases in size
    for (i = (Equations - SearchEqs); i > -1; i--)
    {
        ZeroReset(); //Reset variables to ensure the search works properly on each iteration of the for loop
        printf("For a %d Equation System with %d Variables:\n", (Equations - i), Variables); //print the current system size
        ExhaustiveSearch(LHS, RHS, v1, i); //Search for all solutions in a system of this size
        if (ListSols == true) //Print the decimal equivalents of the solutions if desired
        {
            GenerateSolutions(v1, v2, vs);
            FinalCount = PrintSolutions(vs, repeat);
        }
        SystemDeterminance((Equations - i)); //Print the determinance of the system at it's current size
        //print the total number of solutions found compared to the number tested
        printf("Total Solutions, Reduced System: %d out of the %d tested\n\n", FinalCount, MaxBin);
        printf("-----------------------------------------------------------------------------------------------------------------\n");
        printf("|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||\n");
        printf("-----------------------------------------------------------------------------------------------------------------\n");
    }
}
```

Figure 10: The contents of the VariableExhaustiveSearch() function.

Reducing with Linear Combinations

With this in mind, the next attempt to take advantage of this property was to assign weight to each equation and take advantage of equations with fewer 1s coefficients. Rather than taking every linear combination, each combination was assigned a weight based on the number of coefficients its two parent equations had in common, so when they canceled out very few 1s would remain. Combinations that met this threshold, along with the combination of their RHS

values, were stored in new arrays which would then serve as the system to be solved. However, once this was implemented two main problems arose.

First, it is possible for the new system to be larger than the input system. Because the number of 2 equation combinations for any system with *n* equations will always be larger than *n*, depending on the weight threshold the new system could end up containing more equations than the initial system. For the 5 equation test system used over the course of this project, 8 out of the 10 ($5\ Choose\ 2$) combinations met the threshold of 4 unique 1s. Alternatively, it is also possible for no equations to pass the threshold, and in either scenario, for a random system of equations, it is not possible to know beforehand what that threshold would need to be to prevent these issues (if such a value exists).

The second problem is that of the solutions found. While it is true that the solutions to the initial system also satisfy the linear combinations, the reverse does not hold true. Within the set of solutions to the system of linear combinations there exist solutions not found in the solution set of the initial system. If one were to look for solutions this way, they would find false and true solutions with no way to know the difference without solving the initial system to confirm the results, defeating the purpose of generating linear combinations, to begin with. To combat this we attempted to repeatedly reduce the system in the hopes that false solutions would disappear, but that wasn't the case, and repeat reduction attempts still fell victim to the issues with weight thresholds and the idea was likewise abandoned.

Inversion of Coefficients

While the idea of using linear combinations ultimately failed, the concept of performing operations to reduce 1s coefficients led to other new ideas one of which was a coefficient inversion. Rather than combining equations to reduce 1s, we attempted to do so by swapping 1s with 0s and vice versa in any equation where more than half the coefficients were 1, with the hope being that the solutions would still be the same. This would then allow inversion as a portion of whatever the final algorithm would be.

But as established above, the outcome value of an equation and a solution is dependent on the number of coefficients to add with XOR operations, which in turn is based on the bitwise ANDing equation and solution together. By inverting the coefficients of the equation we change not just the bit positions of successful AND operations, but the number of successful operations. In some, but not all scenarios this can result in a different value to compare against the RHS.

Testing a Solution on a Sample Equation:

Eq : {1, 1, 0, 0, 0, 1, 1, 0, 1, 0 | 1}

$TestSol$: {0, 1, 1, 0, 1, 0, 1, 1, 1, 0 }

$Products$: {0, 1, 0, 0, 0, 0, 1, 0, 1, 0 } $\Rightarrow 1 \oplus 1 \oplus 1 \Rightarrow 1 = RHS$

Testing a Solution on an Inverted Sample Equation:

$InvEq$: {0, 0, 1, 1, 1, 0, 0, 1, 0, 1 | 0}

$TestSol$: {0, 1, 1, 0, 1, 0, 1, 1, 1, 0 }

$Products$: {0, 0, 1, 0, 1, 0, 0, 1, 0, 0 } $\Rightarrow 1 \oplus 1 \oplus 1 \Rightarrow 1 \neq RHS$

Figure 11: Inverting the coefficients of an equation does not guarantee the same solutions.

Taking Intersections of the Input System

With linear combinations and coefficient inversion having both ultimately failed as reduced search methods, we instead turned back to the first simplification made within the original *ExhaustiveSearch()* function of counting 1s to predict outcomes and explored from there. Rather than counting 1s to predict a solution's validity for a single equation, solutions themselves could be derived by counting positions of 1s across multiple equations.

Instead of looking at the equations of a system, we instead looked at the weight and arrangement of 1s in the columns. For simplicity, we experimented with a 2 equation system, as seen in Figure 12. Rather than directly taking an intersection that could result in coefficients canceling out, each column/bit was assigned a variable based on the column's values in the two equations according to the truth table in Figure 12. By then taking those variables and placing them in an array such that variables line up with their respective system columns we can begin to build a partial solution.

Expanding on the logic outlined in Figure 8, we know that only 1s in the solution which lines up with the 1s in an equation determine validity. By default, any valid solution must have 1s at A variables and 0s at B variables since all equations have 1s and 0s, respectively, at those locations. With definite bits in place, all that remains is to assign values to the bits which only affect one of the two equations. Looking at the definition of C variables, they line up only with 1s in Equation 1, and only with 0s in Equation 2. Therefore, any number of Cs required to satisfy Equation 1 will satisfy Equation 2, as they will merely cancel out. The reverse effect holds true for D variables. Since they will always cancel out of Equation 1, any number of Ds required to satisfy Equation 2 will satisfy Equation 1.

With the relevant bit positions identified for each equation, the RHS values can be used to determine how many relevant 1s are needed in our partial solutions. In the sample case below, the RHS value of Equation 1 is 1, so all valid solutions need the total number of 1s at A and C positions to be odd. At the same time the RHS value of Equation 2 is 0, so all valid solutions also need the total number of 1s at A and D positions to be even. With both A bits predetermined to be 1, that means any one or any combination of three of the Cs must be 1. And either no Ds or any two of the Ds must be 1. Incrementing through all possible combinations that satisfy those conditions will in turn give us all possible solutions to the system. Using these rules, we can also preemptively calculate the number of possible solutions by repeatedly adding through the probabilistic combinations of each scenario. In our example case shown below, there would be 32 valid solutions to this system, as seen in Figure 13.

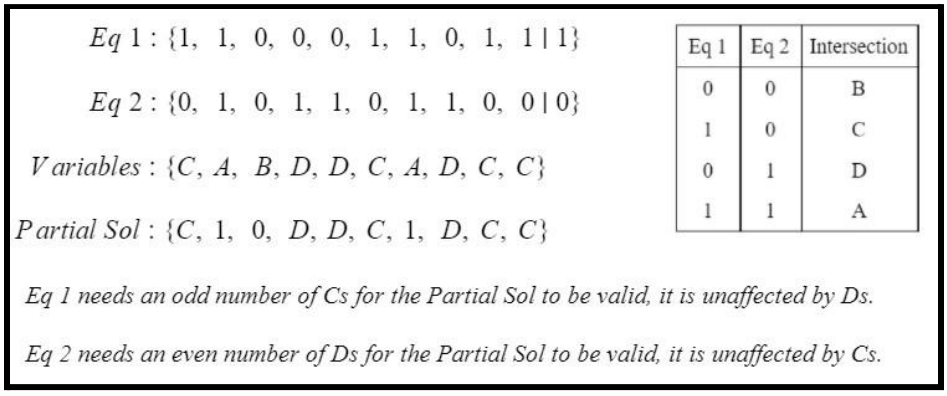


Figure 12: Bit states of valid solutions can be predicted from the arrangements of 1s in an equation.

Total Solutions:

$$[(\binom{4}{1} + \binom{4}{3}) * \binom{3}{0}] + [(\binom{4}{1} + \binom{4}{3}) * \binom{3}{2}]$$

$$\Rightarrow [(4 + 4) * 1] + [(4 + 4) * 3]$$

$$\Rightarrow 8 + (8 * 3)$$

$$\Rightarrow 8 + 24 = 32$$

Figure 13: The number of solutions can be calculated from the predicted bit arrangements.

But although this method of solving technically works, it has a large and serious flaw. When expanding this method beyond a small handful of equations, it quickly becomes unwieldy as more and more variables are needed to represent different column arrangements. Categorizing columns this way is essentially the same as assigning a unique binary number to each of them, so as the number of equations in the system increases, the number of variables increases exponentially at a rate of $2^{(Number\ of\ Equations)}$

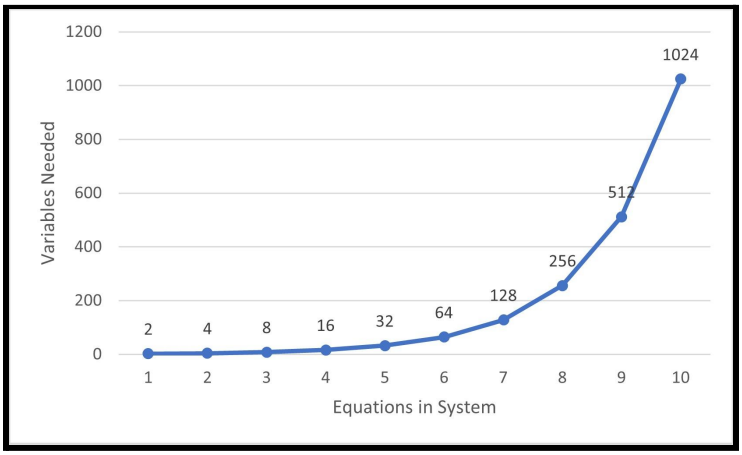


Figure 14: Finding solutions by assigning variables to columns is inefficient at scale.

Unlike previous attempts, examining intersections to predict solutions is conceptually a valid method, and on a small scale is faster than an exhaustive search. The only failure was in the particular implementation. With proof that the concept worked, all that was left now was to work out an algorithm that could apply the concept to a system without relying on positions within columns.

Results and Conclusions

Results

Because the *RecursiveSearch()* function is dependent only on the 1s in each equation, and in a random binary system the distribution of 0s and 1s should be even between the two, we can easily calculate the complexity of this search method. For each S_i column (Figure 1), the complexity of solving that column is equal to:

$$2^{\frac{s_i}{2}}$$

where s_i is equal to the number of bits in S_i. We can then extend this complexity to all the columns for a total complexity of:

$$2^{\frac{s_1+\cdots+s_k}{2}} = 2^{\frac{n}{2}}$$

Where *n* is equal to the total number of variables in the system. Compared to the basic *ExhaustiveSearch()*, complexity is reduced from 2^n to $2^{\frac{n}{2}}$. To be more specific, exhaustive search capacity is increased from n bits to 2n bits. If we consider the exhaustive search limit as 80 bits then *RecursiveSearch()* can solve the binary system with 160 variables. But although this drop to sub-exponential complexity is a significant improvement, perhaps future research could reduce the complexity further, hopefully even dropping the complexity into polynomial time.

Future Research

While developing our *RecursiveSearch()* we discovered a way to possibly modify the existing search algorithm such that it would be able to solve systems of binary quadratic equations. As outlined in section 11.3.1 of Algebraic Cryptanalysis (Bard, 2009, p. 191), under GF(2) any polynomials x^n where $n>1$ will reduce to x since 0 and 1 raised to any power will remain 0 and 1, respectively. Additionally, in section 12.3 (Bard, 2009 , p. 211-213) Bard tells us that in a large quadratic polynomial, the unique monomials can all be represented with single identifier variables. With these two properties, a quadratic system can be condensed into a linear system mostly solvable through the existing *RecursiveSearch()*.

But solving the system in this reduced form introduces solutions not present in the initial system. By assigning variables to monomials which would reduce to 0 when uncondensed we remove information from the system, which then allows for additional solutions to be found which are invalid for the initial system. In our proposed but untested method, by ordering the columns of the matrix to identify the single 0 variables and the condensed variables they go into, one could design a system that still searches for solutions in the normal fashion while keeping the invalid condensed variables out of the calculations.

However, as we began exploring this method we realized the time it would take to fully develop and sort out any issues, it looked to possibly extend beyond the time we had left. So with the time constraints and the scope of this project in mind, we stopped trying to implement it and now we instead leave it as an avenue for future research.

Conclusion

The goal of this project was to develop a new method of solving large systems of binary linear equations. To do so, we designed an algorithm that sorts a matrix into an upper triangle like in Gaussian Elimination, then performs an exhaustive search of partial solutions within each row of the matrix to find full solutions faster than an exhaustive search of the full matrix. In addition to this algorithm being a measurable improvement, our additional failed attempts along with their reasons for failure have also been discovered, and with them possible avenues of future research have been revealed. It is our hope that our new *RecursiveSearch()* can be of value on its own, and that hope that it along with the aforementioned failed attempts can together serve as a starting point and guidelines for future researchers.

References

Bard, G. V. (2009). *Algebraic cryptanalysis*. Springer Science & Business Media.

Bardet, M., Faugère, J.-C., Salvy, B., & Spaenlehauer, P.-J. (2013). On the complexity of solving quadratic Boolean systems. *Journal of Complexity*, *29*(1), 53–75. 10.1016/j.jco.2012.07.001

Keinänen, M., De, U., Keinänen, M., & Oy, M. (2005). *SOLVING BOOLEAN EQUATION SYSTEMS*.

Wang, W., Szefer, J., & Niederhagen, R. (2016). Solving large systems of linear equations over GF(2) on FPGAs. *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 1–7. 10.1109/ReConFig.2016.7857188

Appendix

For reference purposes the code developed for this paper can be found below. This code can also be found through the following github link or by contacting the paper author at jrmcaleese@wpi.edu.

Github Repository:

https://github.com/jrmcaleese/Solving-Systems-of-Linear-Equations-over-GF-2-on-FPGAs.git

C Implementation

Main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include <time.h>
#include "parameters.h"
#include "VariableExhaustiveSearch.h"
#include "SimilarityCheck.h"
#include "ArrayHandling.h"
#include "RecursiveSearch.h"

//Usage bools
bool RandSystem = false; //toggles whether the program uses a preset system or generates a random one
bool doLinCombs = false; //toggles whether A and b are fully populated with their linear combinations
bool ReduceSystem = false; //toggles whether the SimCheck function is used to reduce the system
bool ReduceSearch = false; //toggles whether the ExhaustiveSearch function compares the number of 1s to
skip certain solutions
bool RepeatReduce = false; //toggles whether the system reduction happens once or happens repeatedly by
LoopReduce times


//Debug bools
bool ShowBinaryValids = false; //toggles whether the just the decimal equivalents of the solutions are
shown or the full binary solution sets.
bool ShowFullDebug = false; //toggles detailed print statements for each tested solution. When false
only valid solutions are displayed.
bool ListSols = true; //toggles whether the full list of valid solutions is printed together at the end
of each search
bool csvFriendly = false; //lists the solutions in a format that is copyable to a csv or excel file
bool SimCheckDebug = false; //toggles the debug print statements for the SimCheck2d function
```

```
//The Input system is stored below with the Left Hand Side coefficients stored in InitSystem, and the
Right Hand Side stored in InitSols
int InitSystem[Equations][Variables] =
{

    {0,0,0,1,0,1,0,1,1,1,0},//0
    {0,1,1,1,0,0,0,0,1,0,1},//1
    {0,0,0,0,1,0,1,1,0,1,0},//2
    {1,1,0,1,1,0,0,0,0,1,0},//3
    {0,1,0,0,0,0,0,1,0,0,1},//4
};
int InitSums[InitEqs] = { 0,1,0,0,1 };

int main()
{
    printf("Start Program\n"); //Marking the Start of the output file
    int i, j;
    AddOne[Variables - 2] = 1;

    //print the initial system
    for (i = 0; i < Equations; i++)
    {
        for (j = 0; j < Variables; j++)
        {
            printf("%d,",InitSystem[i][j]);
        }
        printf("\n");
    }
    printf("==========\n");

    //sort the system
    SortArray(InitSystem, RowWeights);
    //print the sorted system
    for (i = 0; i < Equations; i++)
    {
        for (j = 0; j < Variables; j++)
        {
            printf("%d,", InitSystem[i][j]);
        }
        printf("\n");
    }
    printf("==========\n");

    //find solutions
    RecursiveSearch(InitSystem);

    //print the solutions
    printf("The solutions to this system are:\n");
    for (i = 0; i < answercount; i++)
    {
        printf("%d,", answers[i]);
        if (i == answercount - 1)
        {
            printf("\n\n");
        }
        else if (i % 10 == 0)
```

```
        {
            printf("\n");
        }
    }

    printf("\nEnd Program");
    return 0;
}
```

Parameters.h

```
#ifndef PARAMETERS_H
#define PARAMETERS_H


#define InitEqs 5 //The number of initial Equations in the system
#define MaxCombs ((InitEqs*(InitEqs-1))/2) //the maximum number of 2 equation linear combinations from a system the size of InitEqs
#define Equations 5//(InitEqs+MaxCombs) //The maximum number of equations in the combined system
#define SearchEqs 5 //The number of Equations for the program to start looking through solutions for
#define Variables 11 //The number of variables in the system
#define MaxBin 1023 //The largest binary number possible with "Variables" number of bits.
#define MinSame ((Variables/2)-(Variables/2)%1) //The minimum number of coefficients two equations must have in common to be reduced
#define LoopReduce 5 //If reducing the system, this sets the number of times you wish to repeatedly reduce it

//Usage bools
extern bool RandSystem;
extern bool ReduceSystem;
extern bool ReduceSearch;
extern bool doLinCombs;
extern bool RepeatReduce;


//Debug bools
extern bool ShowBinaryValids; //toggles whether the just the decimal equivalents of the solutions are shown or the full binary solution sets.
extern bool ShowFullDebug; //toggles detailed prextern int statements for each tested solution. When false only valid solutions are displayed.
extern bool ListSols;
extern bool csvFriendly;
extern bool SimCheckDebug;

#endif
```

RecursiveSearch.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include <time.h>
#include "VariableExhaustiveSearch.h"
#include "parameters.h"
#include "SimilarityCheck.h"
#include "ArrayHandling.h"

int RowWeights[Equations] = { 0 };
int CoefIndices[Variables - 1] = { 0 };
int stopPoint[Equations] = { 0 };
int depth = 0;
int tempsum[Equations] = {0};
int tempsol[Variables - 1] = { 0 };
int tempones = 0;
int answers[MaxBin] = { 0 };
int answercount = 0;
int binsize = 0;
int fullbinsize[Equations] = { 0 };
int row[Equations][Variables] = { 0 };
int temprow[Variables] = { 0 };
int looplimit[Equations] = { 0 };

void GetRowWeights(int arr1[Equations][Variables], int Weights[Equations])
{
    //get the number of ones for each row
    int i, j;
    int sum;
    for (i = 0; i < Equations; i++)
    {
        sum = 0;
        for (j = 0; j < (Variables - 1); j++)
        {
            if (arr1[i][j] == 1)
            {
                sum++;
            }
        }
        Weights[i] = sum;
    }
}

int countOnes(int arr1[],int limit)
{
    //a function for counting the number of ones in an array
    int i;
    int onescount = 0;
    for (i = 0; i < limit; i++)
    {
        if (arr1[i] == 1)
        {
            onescount++;
```

```
        }
    }
    return onescount;
}

void SortArray(int InputArray[Equations][Variables], int rw[Equations])
{
    //sorting the array into the format required for finding solutions
    int h, i, j, k;
    int temprow[Variables] = { 0 };
    int tempcoef[Equations] = { 0 };
    int tempweight = 0;
    int temp = 0;
    stopPoint[0] = Variables - 1;
    GetRowWeights(InputArray, rw);
    //Sort the equations so the number of coefficients per matrix row decreases from top to bottom
    for (i = 0; i < (Equations-1); i++)
    {
        //Bubble sorting each row of the matrix
        for (j = 0; j < Equations - 1 - i; j++)
        {
            if (rw[j] < rw[j + 1])
            {
                for (k = 0; k < (Variables); k++)
                {
                    temprow[k] = InputArray[j+1][k];
                    InputArray[j+1][k] = InputArray[j][k];
                    InputArray[j][k] = temprow[k];
                }
                tempweight = rw[j+1];
                rw[j+1] = rw[j];
                rw[j] = tempweight;
            }
        }
    }
    //Rearrange the columns in an attempt to create an upper triangle of zeroes in the matrix
    for (i = 0; i < Equations; i++) //Increment through each row
    {
        if (i == 0)
        {
            for (j = 0; j < Variables-1; j++)
            {
                for (k = 0; k < Variables - 2 - j; k++)
                {
                    //If the value at this index in the current row is greater than the next value
                    //then swap the entire columns
                    if (InputArray[i][k] > InputArray[i][k + 1])
                    {
                        for (h = 0; h < Equations; h++)
                        {
                            tempcoef[h] = InputArray[h][k];
                            InputArray[h][k] = InputArray[h][k + 1];
                            InputArray[h][k + 1] = tempcoef[h];
                        }
                    }
                }
            }
```

```
            //In the current row, find the first index with a one so we have a new stopping point when
sorting the next row
            for (j = 0; j < Variables - 1; j++)
            {
                if (InputArray[i][j] == 1)
                {
                    stopPoint[i] = j;
                    break;
                }
            }
        }
        else
        {
            if (stopPoint[i - 1] <= 0)
            {
                break;
            }
            //Sort the columns based on the 1s in the current row of outer loop
            for (j = 0; j < stopPoint[i - 1]; j++)
            {
                for (k = 0; k < stopPoint[i - 1] - 1 - j; k++)
                {
                    //If the value at this index in the current row is greater than the next value
                    //then swap the entire columns
                    if (InputArray[i][k] > InputArray[i][k + 1])
                    {
                        for (h = 0; h < Equations; h++)
                        {
                            tempcoef[h] = InputArray[h][k];
                            InputArray[h][k] = InputArray[h][k + 1];
                            InputArray[h][k + 1] = tempcoef[h];
                        }
                    }
                }
            }
            //In the current row, find the first index with a one so we have a new stopping point when
sorting the next row
            for (j = 0; j < Variables - 1; j++)
            {
                if (InputArray[i][j] == 1)
                {
                    stopPoint[i] = j;
                    break;
                }
            }
        }

    }
}

int bin2dec()
{
    int i;
    int tempval=0;
    for (i = 0; i < Variables - 1; i++)
    {
        if (row[depth][i] == 1)
```

```
        {
            tempval += 1 << i;
        }
    }
    return tempval;
}

void RecursiveSearch(int InputArray[Equations][Variables])
{
    int i, j, k;
    int sameones = 0;
    //calculate the loop limit for the given row as
    //( (2^S_i) - 1 ) where S_i is the size in bits of the clustered column
    //Bitshifting to the appropriate depth serves as a quick shortcut for raising to the correct power
of 2
    int size;
    if (depth == 0)
    {
        size = (Variables - 1) - stopPoint[depth];
    }
    else
    {
        size = stopPoint[depth - 1] - stopPoint[depth];
    }
    looplimit[depth] = (1 << size)-1;
    for (i = 0; i <= looplimit[depth]; i++)
    {
        if (looplimit[depth] == 0 && depth != (Equations - 1)) //if we've found a full solution but we
aren't at max depth, we need to check remaining rows against the full solution
        {
            for (j = depth; j < Equations; j++)
            {
                sameones = 0;
                int remainingrows = Equations - depth;
                EqSum[j] = 0; //making sure EqSum is zero before calculating for the current row so only
the correct answer for this iteration is recorded
                for (k = 0; k < Variables - 1; k++)
                {
                    if ((InputArray[j][k] == 1) && (tempsol[k] == 1))
                    {
                        sameones += 1; //if a coefficient bit in the current equation and its
corresponding bit in the solution are both 1, then increment sameones
                    }
                }
                if (sameones % 2 == 1)
                {
                    EqSum[j] = 1; //if sameones is odd, the equation sum is odd
                }
                else if (sameones % 2 == 0)
                {
                    EqSum[j] = 0; //if sameones is even, the equation sum is even
                }
                if (EqSum[j] == InputArray[depth][Variables - 2]) //if the EqSum equals the RHS value
the solution is valid
                {
                    if (j == remainingrows)
                    {
```

```
                        answers[answercount] = bin2dec(tempsol); //find the decimal equivalent of the
answer and append it to an output array
                        answercount++;
                        continue;
                    }
                    else
                    {
                        continue;
                    }
                }
                else
                {
                    break;
                }
            }
        }

        if (looplimit[depth] > 0)
        {
            if (looplimit[depth] = 1)
            {
                //if we only have a 1 bit wide group
                //instead of iterating and testing both 0 and 1, we can just set the tempsum equal to
the RHS value and guarantee the valid value.
                //only 1 of the two options can be valid.
                tempsum[depth] = InputArray[depth][Variables - 1];
            }
            else
            {
                //Increment through the possible partial solutions for this row
                if (depth == 0)
                {
                    BinArrayAdd(Variables - 1, stopPoint[depth], row[depth], temprow);
                }
                else
                {
                    BinArrayAdd(stopPoint[depth - 1], stopPoint[depth], row[depth], temprow);
                }


                //multiply the possible partial solution with the relevant portion of the current row
                if (depth == 0) //if on the first equation, solve from stop point to the end
                {
                    for (j = stopPoint[depth]; j < Variables - 1; j++)
                    {
                        tempsum[depth] = tempsum[depth] ^ (row[depth][j] & InputArray[depth][j]);
                    }
                }
                else //if not on the first equation, solve from the stop point to the stop point of the
last equation
                {
                    for (j = stopPoint[depth]; j < stopPoint[depth - 1]; j++)
                    {
                        tempsum[depth] = tempsum[depth] ^ (row[depth][j] & InputArray[depth][j]);
                    }
                }
                //add the tempsum of the previous rows to this one
```

```
                tempsum[depth] = tempsum[depth] ^ tempsum[depth - 1];
            }

            if (tempsum[depth] == InputArray[depth][Variables - 1]) //if the tempsum of this row summed
with that of all previous rows is equal to the RHS of the current row, the partial solution is valid, so
the function should recurse
            {
                //copy the partial solution
                if (depth == 0) //if on the first row, stop copying the partial sol at the last index
point
                {
                    for (j = stopPoint[depth]; j < Variables - 1; j++)
                    {
                        tempsol[j] = row[depth][j];
                    }
                }
                else //otherwise stop at the point from the last row
                {
                    for (j = stopPoint[depth]; j < stopPoint[depth - 1]; j++)
                    {
                        tempsol[j] = row[depth][j];
                    }
                }
                if (depth == (Equations - 1)) //if we're at the max depth, then there's no more solution
to find after this point
                {
                    answers[answercount] = bin2dec(tempsol); //find the decimal equivalent of the answer
and append it to an output array
                    answercount++;
                    continue;
                }
                depth += 1;
                RecursiveSearch(InputArray);
            }

            else
            {
                continue;
            }
        }

        //otherwise the function will return to the top of the for loop
    }
}
```

RecursiveSearch.h

```
#ifndef RECURSIVESEARCH_H
#define RECURSIVESEARCH_H

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include "parameters.h"


extern int RowWeights[Equations];
extern int CoefIndices[Variables - 1];
extern int stopPoint[Equations];
extern int depth;
extern int tempsum[Equations];
extern int tempsol[Variables - 1];
extern int answers[MaxBin];
extern int answercount;
extern int binsize;
extern int fullbinsize[Equations];
extern int row[Equations][Variables];
extern int temprow[Variables];
extern int looplimit[Equations];

void GetRowWeights(int arr1[Equations][Variables], int Weights[Equations]);
int countOnes(int arr1[], int limit);
void SortArray(int InputArray[Equations][Variables], int rw[Equations]);
int bin2dec();
void RecursiveSolve(int InputArray[Equations][Variables]);
#endif
```

VariableExhaustiveSearch.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include <time.h>
#include "VariableExhaustiveSearch.h"
#include "parameters.h"
#include "SimilarityCheck.h"
#include "ArrayHandling.h"

int InvCount = 0; //tracking the number of invalid solutions
int SolCount = 0;
int FinalCount = 0;
int CarryBit=0;
int CoefProd = 0;
int xOnes = 0;
int LHSOnes = 0;
int ValidRows = 0; //tracking the number of equations for which a possible solution is valid

void ZeroReset(void)
{
    /*
    Resets relevant global variables to required initial conditions
    to avoid possible data errors when solving systems of multiple size
    or when variables don't rewrite properly between loop iterations
    */
    int i;
    for (i=0;i<Variables;i++)
    {
        x[i]=0;
        y[i]=0;
        invalid[i] = 0;
        valid[i] = 0;
    }
    for (i=0;i<Equations;i++)
    {
        EqSum[i]=0;
    }
    for (i = 0; i < MaxCombs; i++)
    {
        samecoef[i] = 0;
    }
    for (i=0;i<Variables-1;i++)
    {
        if(i<(Variables-2))
        {
            AddOne[i]=0;
        }
        else
        {
            AddOne[i]=1;
        }
    }
    InvCount=0;
```

```
    ValidRows=0;
    SolCount=0;
    FinalCount = 0;
    CarryBit=0;
}

void BinArrayAdd(int length, int arr1[], int arr2[])
{
    /*
    Generates the possible solution sets.
     In this for loop the Variables are treated as bits in a ripple carry adder so we can increment
through all solutions.
     Outside this for loop the carry bit has no effect and the sets are treated as arrays of individual
values again.
    */
    int i;
    for (i = 0; i < Variables - 1; i++) //clear arr2 before adding
    {
        arr2[i] = 0;
    }
     for (i=length;i>-1;i--) //this loop is a 1-bit ripple CarryBit adder looped into a multi-bit ripple
CarryBit adder
    {
        //XOR addition of the current elements in x and AddOne and then with the carrybit, all stored in
arr2[i]
        arr2[i]=(arr1[i]^AddOne[i])^CarryBit;

         if (arr1[i]==1 && AddOne[i]==1) //if x and AddOne were both 1, the CarryBit bit will be 1 next
loop
        {
            CarryBit=1;
        }
         else if (arr1[i]==1 && CarryBit==1) //if x and the CarryBit bit were both 1, the CarryBit bit
will be 1 next loop
        {
            CarryBit=1;
        }
        else if (CarryBit==1 && AddOne[i]==1) //if AddOne and the CarryBit bit were both 1, the CarryBit
bit will be 1 next loop
        {
            CarryBit=1;
        }
        else //otherwise the CarryBit bit will be 0 next loop
        {
            CarryBit=0;
        }
        //now that the CarryBit bit has been assigned,
         //the value from y can be moved into x so the values are in place for the next iteration of the
outermost loop
        arr1[i]=arr2[i];

    }
}

void SystemDeterminance(int CurrentEqs)
{
    /*
```

```
    Prints whether or not the current System is under or over determined
    */
    if ((CurrentEqs)<Variables)
    {
        printf("This is an underdetermined system.\n");
    }
    else if((CurrentEqs)>Variables)
    {
        printf("This is an overdetermined system.\n");
    }
    else
    {
        printf("This is a determined system.\n");
    }
}

int GenerateSolutions(int v1[MaxBin],int v2[MaxBin], int vs[MaxBin])
{
    /*
    Listing all together the decimal representations of the solutions for the given system.
    Makes it easier to actually track patterns in solutions.
    */
    int i,c = 0;
    for (i = 0; i < MaxBin; i++)
    {
        if ((v1[i] == 1)&&(v2[i] == 1))
        {
            vs[c] = (i + 1);
            c++;
        }
    }
    return c;
}
int PrintSolutions(int arr1[MaxBin],bool repeat)
{
    int i, count;
    if(csvFriendly==false)
    {
        if (SolCount > 0)
        {
            if (repeat == false)
            {
                count = 1;
            }
            else
            {
                count = 0;
            }
            printf("\nThe valid solutions for this system are:\n");
            for (i = 0; i < SolCount; i++)
            {

                if (arr1[i] == 0) //if the value is 0 then we've printed all solutions
                {
                    printf("\n\n");
                    return count;
                }
```

```
                printf("%d", arr1[i]);
                 if (i == (SolCount - 1)) //if i is one less than SolCount then we've reached the end of
the solutions
                {
                    printf("\n\n");
                    return count;
                }
                else if ((i + 1) % 20 == 0) //start a new line every 20 solutions printed
                {
                    printf(",\n");
                }
                else //print a comma between solutions
                {
                    printf(",");
                }
                count++;
            }
        }
         else if (SolCount == 0) //Printing a fail message is a clearer output than leaving blank space
in the event of no solutions
        {
            printf("\n-----NO SOLUTIONS FOUND-----\n\n");
        }
    }
    else
    {
        if (SolCount > 0)
        {
            int count;
            if (repeat == false)
            {
                count = 1;
            }
            else
            {
                count = 0;
            }
            printf("\nThe valid solutions for this system are:\n");
            for (i = 0; i < SolCount; i++)
            {

                if (arr1[i] == 0) //if the value is 0 then we've printed all solutions
                {
                    printf("\n\n");
                    return count;
                }
                printf("%d", arr1[i]);
                 if (i == (SolCount - 1)) //if i is one less than SolCount then we've reached the end of
the solutions
                {
                    printf("\n\n");
                    return count;
                }
                else //print a comma between solutions
                {
                    printf("\t");
                }
```

```
                count++;
            }
        }
         else if (SolCount == 0) //Printing a fail message is a clearer output than leaving blank space
in the event of no solutions
        {
            printf("\n-----NO SOLUTIONS FOUND-----\n\n");
        }
    }

}

void ExhaustiveSearch(int LHS[][Variables],int RHS[], int v[MaxBin], int CurrentEqs)
{
    int i, j, k;
    /*
    1. The central search function of the whole program. On each loop, BinArrayAdd() is used to generate
a new possible solution set as an array equal in length to the number of variables in the system,
       incrementing all the way up to the MaxBin value.

     2. The values at each index in the tested solution are then AND multiplied by their respective
values in the first equation of matrix arr1[] and those products are XOR added together
        with the final sum stored in an array called EqSum[] at the index equal to the equation number
being tested.

     3. This sum is then compared against the value at the same index in b[], and if they are equal the
ValidRows count is incremented by 1.
        Otherwise the InvCount (invalid solution count) is incremented by 1 to show that the solution is
not valid for this equation and therefore the system.

    4. Steps 2 and 3 are repeated for all remaining rows in the system. If at the end ValidRows is equal
to the number of equations, then the index number of the tested solution (k+1)
        is stored in the first available spot in the array valid[] and SolCount (a tally of the total
number of valid solutions) is incremented by 1.

    5. Steps 1-4 are repeated until all possible solutions have been incremented through and tested.

     6. When debugging is on the index numbers of each solution are printed along with the SolCount and
the number of solutions tested.
    */
    int sameones;
     for(k=0;k<MaxBin;k++) //this largest loop controls the whole process, running through steps 1-4
until all possible solutions have been tested
    {
        BinArrayAdd((Variables-1),x,y);
        if (ShowFullDebug==true) //shows the specific solution set being tested
        {
            printf("Solution Tested = %d\n",k+1);
            printf("x = {");
            for(i=0;i<Variables;i++)
            {
                printf("%d",x[i]);
                if (i==(Variables-1))
                {
                    printf("}\n\n");
                }
                else
```

```
                {
                    printf(",");
                }
            }
        }
        xOnes = 0;
        for (i = 0; i < Variables; i++)
        {
            if (x[i] == 1)
            {
                xOnes = xOnes + 1;
            }
        }

        for (i=0;i<(Equations-CurrentEqs);i++) //for each row/equation in A
        {
            sameones = 0;
            EqSum[i] = 0; //making sure EqSum is zero before calculating for the current row so only the correct answer for this iteration is recorded
            for (j = 0; j < Variables; j++)
            {
                if ((LHS[i][j] == 1) && (x[j] == 1))
                {
                     sameones += 1; //if a coefficient bit in the current equation and its corresponding bit in the solution are both 1, then increment sameones
                }
            }
            if (sameones % 2 == 1)
            {
                EqSum[i] = 1; //if sameones is odd, the equation sum is odd
            }
            else if(sameones % 2 == 0)
            {
                EqSum[i] = 0; //if sameones is even, the equation sum is even
            }

             if (EqSum[i] != RHS[i]) //if the equation sum does not equal the b value for this equation, save the current k value + 1 to the array 'invalid', then increment invalidcount
            {
                invalid[InvCount]=(k+1);
                InvCount+=1;
                ValidRows=0;
                break;
            }
              else //otherwise if the equation some does equal the solution for this equation, increment the ValidRows count
            {
                ValidRows+=1;
            }
        }
        if (ValidRows == (Equations-CurrentEqs)) //if the solution is valid for all rows/equations, then record the k+1 value into 'valid' at the index equal to the current SolCount
        {
            v[k]=1;

            if (ShowBinaryValids==true) //print the full binary readout of the solution just found
            {
```

```
                printf("VALID SOLUTION FOUND = %d\n",(k+1)); //FIX THIS LINE. VALID SOLUTION FOUND WILL
NOW JUST PRINT A 1 OR 0 IN CURRENT STATE
                if (ShowFullDebug==false)
                {
                    printf("x = {");
                    for(i=0;i<Variables;i++)
                    {
                        printf("%d",x[i]);
                        if (i==(Variables-1))
                        {
                            printf("}\n\n");
                        }
                        else
                        {
                            printf(",");
                        }
                    }
                    printf("----------------\n\n");
                }
            }
            SolCount+=1;
            if (ShowFullDebug==true) //print the number of valid solutions
            {
                printf("\nVALID ANSWERS = %d\n\n----------------\n\n",SolCount);
            }
            ValidRows=0;
        }
    }
}

void VariableSearch(int LHS[][Variables], int RHS[], int v1[MaxBin], int v2[MaxBin], int vs[MaxBin],
bool repeat) //A,b,valid,reducedvalid,validsols,repeat
{
    int i;
    //The search loop of the function. Depending on the inputs for h, this loop iterates at least once
    //depending on whether we want to test how solutions change as a system increases in size
    for (i = (Equations - SearchEqs); i > -1; i--)
    {
         ZeroReset(); //Reset variables to ensure the search works properly on each iteration of the for
loop
        printf("For a %d Equation System with %d Variables:\n", (Equations - i), Variables); //print the
current system size
        ExhaustiveSearch(LHS, RHS, v1, i); //Search for all solutions in a system of this size
        if (ListSols == true) //Print the decimal equivalents of the solutions if desired
        {
            GenerateSolutions(v1, v2, vs);
            FinalCount = PrintSolutions(vs, repeat);
        }
        SystemDeterminance((Equations - i)); //Print the determinance of the system at it's current size
        //print the total number of solutions found compared to the number tested
        printf("Total Solutions, Reduced System: %d out of the %d tested\n\n", FinalCount, MaxBin);

printf("----------------------------------------------------------------------------------------------------
----\n");

printf("||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||||\n");
```

```
printf("-----------------------------------------------------------------------------------------------
----\n");
    }
}
```

VariableExhaustiveSearch.h

```
        #ifndef VARIABLEEXHAUSTIVESEARCH_H
#define VARIABLEEXHAUSTIVESEARCH_H

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include "parameters.h"

void ZeroReset(void);
void BinArrayAdd(int length, int arr1[], int arr2[]);
void SystemDeterminance(int CurrentEqs);
int GenerateSolutions(int arr1[MaxBin], int arr2[MaxBin], int arr3[MaxBin]);
int PrintSolutions(int arr1[MaxBin], bool repeat);
void ExhaustiveSearch(int LHS[][Variables], int RHS[], int v[MaxBin], int CurrentEqs);
void  VariableSearch(int  LHS[][Variables],  int  RHS[],  int  v1[MaxBin],  int  v2[MaxBin],  int  vs[MaxBin],
bool repeat);

extern int InvCount; //tracking the number of invalid solutions
extern int SolCount; //tracking the number of valid solutions for the initial system
extern int FinalCount;
extern int CarryBit; //carry bit needed in the binary counter
extern int CoefProd; // product of a polynomial coefficient with the test variable
extern int xOnes;
extern int LHSOnes;
extern int ValidRows; //tracking the number of equations for which a possible solution is valid
#endif
```