Aenigmagraph: Modifying 2D Arrays Utilizing Shortest Spanning Trees to Augment Crypto graphical Systems

Tyler Szeto (1), Jonathan Lin (2)

Abstract

As technology rapidly evolves, exploring new crypto graphical practices is necessary to maintain the security of our digital files. Here we propose the 'Aenigmagraph', an algorithm that writes information into a 2D array to obscure the order in which information is written. Using a modified version of Prim's algorithm, the process can be done quickly, saving both time and memory. The Aenigmagraph may function either as a symmetrical encryption system or as a trapdoor function, but not both simultaneously. Although the Aenigmagraph has limited functionality on its own, it is designed to augment contemporary and future crypto graphical systems.

1 Introduction

1.1 Motivation

The world is constantly changing, with technology rapidly evolving in ways both expected and unexpected. Whenever abrupt technological growth occurs, digital security must adapt in order to meet the resulting demands. As few security systems have withstood the tests of time, it is paramount that novel methods of improving contemporary systems be explored.

1.2 Intent of the Aenigmagraph

Aenigmagraphs efficiently enhance many crypto graphical systems through its ability to selectively take the role of either a trapdoor function or a symmetric function. By design, Aenigmagraphs obfuscate sensitive information - such as the order data was written in - from many forms of cryptanalysis. It is important to note that the Aenigmagraph is not an independent encryption algorithm, but rather a proposed tool meant to augment other crypto graphical systems.

1.3 Role in Cryptography

When applied as a symmetric algorithm, we can encrypt information into the Aenigmagraph using a replicable series of pseudo random integers. These can later be reversed to change the encrypted Aenigmagraph back to its original, unadulterated state. Paired with another encryption system, obscuring the order of information adds another layer of unpredictability to the system, making it more difficult for an attacker to conduct a clear text attack or cryptanalysis. Using methods similar to Advanced Encryption Standard (AES), one can convolute data such that attempting to reverse the process would be met with great difficulty. However, knowing the seed the Aenigmagraph was generated from allows one to undo the effects quickly by reversing the process.

Alternatively, another feasible application of the Aenigmagraph would be as a one way hashing method. An effectively used pre-image act as the seed that will direct the rest of the obfuscation process. Retrieving sensible information from hashed data is prohibitively difficult, as the seed used to formulate the order of the graph is effectively lost during encryption. Because of this, the Aenigmagraph may act as a trapdoor method.

The proposed Aenigmagraph method excels in the context of either a hashing function or a symmetric encryption system, and can be easily modified for implementation into many cryptographic systems. In both runtime and space, complexity is kept low and predictable, enabling effective functionality in large quantities.

1.4 Other Applications

The Aenigmagraph is not necessarily limited to symmetrical encryption and trapdoor functions; indeed, it can be modified to fulfill many different roles, including uses outside the conventional

boundaries of traditional cryptography. As the Aenigmagraph is highly flexible, it is difficult to foresee all possible uses. However, some proposed modifications and alternative uses involve steganography, multidimensional functionality, reflexive functionality, proof-of-work, and image scrambling.

1.5 Related Work

Methods similar to the Aenigmagraph involve using Sudoku puzzles for both image scrambling and steganography image writing [1] [2]. Just as Sudoku oriented cryptography utilizes the colossal amount of combinations existing in a 9 x 9 (5.25×10^{27} combinations [3]) grid to obfuscate information, the Aenigmagraph is designed to maximize combinations existing within a 2D graph (estimated to be $\approx 10^{112}$ in a 9 x 9 grid) to prevent attacks.

Figure 1.5.1: A superficial example of manipulating coordinates in a 2D array (an image) using one round of the Aenigmagraph. Sudoku oriented cryptography achieves similar results.

2 Methods

2.1 Aenigmagraph Generation

The Aenigmagraph is generated using a heavily modified version of Prim's algorithm. The modified algorithm relies solely on a random function to pseudo randomly select its next points, as opposed to using weighted edges to determine the next point to traverse to. For testing, we used Java's "SecureRandom" function obtain random values. Alternatively, using a default version of Prim's algorithm, a grid may be generated with random weights on each point to achieve a similar result.

A user selects a seed, which will become the key to the system. This seed is then used by the LCG, allowing for the replication of deliberately chosen random integers. The seed can be thought of as a symmetrical key, as it is used both to store and recover information in the Aenigmagraph. Furthermore, in a trap door function, the data of the pre image will be used as a seed, and will effectively be lost after hashing.

2.2 Aenigmagraph Generation Illustration

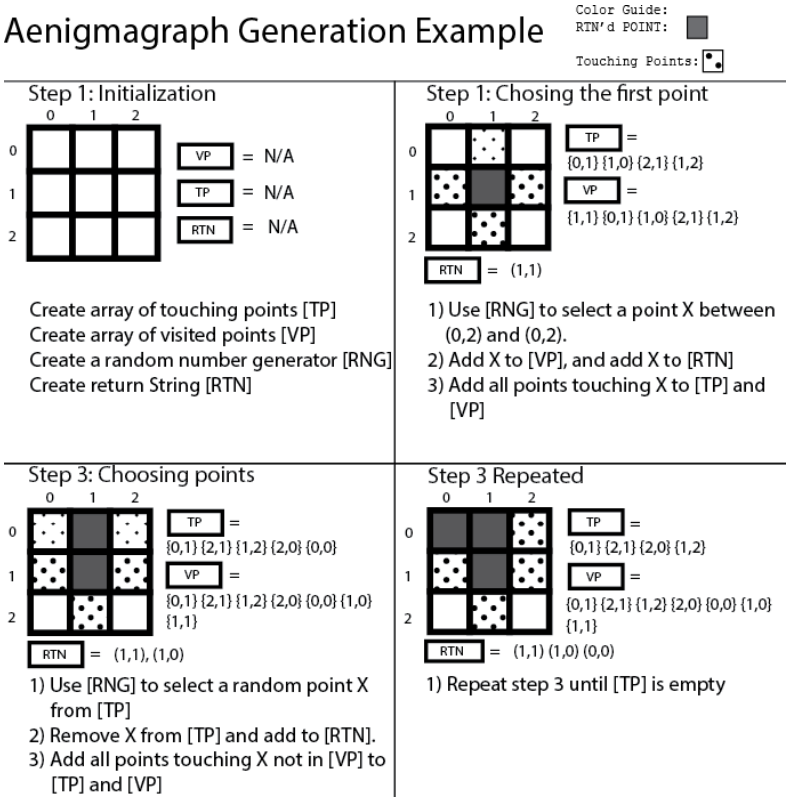


Figure 2.2.1: A visual, step by step example showcasing the steps required to generate the Aenigmagraph of 3x3 sizes (2x2 when treated as an array). The same point cannot be added to any of the data structures multiple times. The sequence is dictated by the order in which points are added to [RTN].

2.3 Proposed Code

```
Aenigmagraph(int seed, int xSize, int ySize) {
        this.seed = seed;
        this.xSize = xSize;
        this.ySize = ySize;
        this.random = new Random(seed);
        this.touchingPoints = new LinkedList<point>();
        this.visitedPoints = new HashSet<point>();
}

public point getFirstPoint() {
        int xLoc = randInt(0, xSize);
        int yLoc = randInt(0, ySize);

        updateBagOfPoints(new point(xLoc, yLoc));
        visitedPoints.add(new point(xLoc, yLoc));
        System.out.println("First cord: " + xLoc + " " + yLoc);
        return new point(xLoc, yLoc);
}

public point getAPoint() {
        if (touchingPoints.size() == 0) {
                System.out.println("There are no points left");
                return null;
        }
        point returnedPoint = touchingPoints.remove(randInt(0, touchingPoints.size() - 1));
        updateBagOfPoints(returnedPoint);
        return returnedPoint;
}
```

```
private void updateBagOfPoints(point pointRemoved) {
                int xCord = pointRemoved.getX();
                int yCord = pointRemoved.getY();

                if (yCord + 1 <= ySize)
                        if (visitedPoints.add(new point(xCord, yCord + 1)))
                                touchingPoints.add(new point(xCord, yCord + 1));

                if (yCord - 1 >= 0)
                        if (visitedPoints.add(new point(xCord, yCord - 1)))
                                touchingPoints.add(new point(xCord, yCord - 1));

                if (xCord + 1 <= xSize)
                        if (visitedPoints.add(new point(xCord + 1, yCord)))
                                touchingPoints.add(new point(xCord + 1, yCord));

                if (xCord - 1 >= 0)
                        if (visitedPoints.add(new point(xCord - 1, yCord)))
                                touchingPoints.add(new point(xCord - 1, yCord));
        }


        private int randInt(int min, int max) { // non inclusive
                return random.nextInt(max - min + 1) + min;
        }
```

This is a working code for the contemporary version of the Aenigmagraph. Although not the only way to generate the Aenigmagraph, these methods accurately represent the overall functionality of the system. From our testing, we envision this to be the most efficient way of implementing a quick and lightweight Aenigmagraph system. However, we have not discounted the potential for a more optimal solution. "Visited Points" is stored as a hash set to allow for a rapid and cost effective way of checking visited elements while simultaneously conserving memory. A linked list is used as the array needs to

grow, shrink, remove elements, and access elements in rapid succession, something other data structures struggle with in large quantities.

2.4 Encrypting Information With the Aenigmagraph

```
public array[][] generateAenigmagraph(String input, int seed)
{
     2DArray Array = 2DArray large enough to fit input;
     Aenigmagraph Aenigmagraph = new Aenigmagraph(seed, Array.size());

     point = Aenigmagraph.getFirstPoint(); //Manually adds first point
     Array[point] = input.charAt(0);       //as first point needs to be
                                           //treated as a seperate step
     for(int x = 1; x < array.length; x++)
     {
          point = Aenigmagraph.getAPoint();
          Array[point] = input.charAt(x);
     }

     return Array;
}
```

To write information into a 2D array using the Aenigmagraph, points are returned by the Aenigmagraph object, which dictates the coordinates where data will be stored in the 2D array. The first point returned by the Aenigmagraph is typically added outside of the for loop, as it must be chosen separately.

2.5 Decrypting Information with the Aenigmagraph

```
public String decrypt2DArray(2DArray, int seed)
{
     Aenigmagraph Aenigmagraph = new Aenigmagraph(seed, 2DArray.size())
     String plaintext = "";
     point = Aenigmagraph.getFirstPoint();

     plaintext += 2DArray[point]; // add first point to plaintext

     for(int x = 1; x < array.length; x++)
     {
          point = Aenigmagraph.getAPoint();
          plaintext += 2DArray[point];
     }
     return plaintext;
}
```

For decryption, the Aenigmagraph will use the seed provided by the user to generate a new graph that restores the encrypted 2D array back into the original String. The algorithm maintains a String named “plaintext”, in which it saves information in the same order it was stored. Decryption can be visualized as the opposite of encryption: rather than storing elements into a 2D Array, information is stored when accessing the 2D array.

2.6 Using the Aenigmagraph as a Trapdoor

As a trapdoor method, the seed is generated by the input, which is effectively lost after encryption. The preceding steps are identical to encryption; however, it is difficult to determine a seed without the original input.

```
int seed = input.toSeed();
Aenigmagraph = new Aenigmagraph(seed, Array.size());
...
```

2.7 Using Pre-Generated Sequences

In some cases, generating Aenigmagraph sequences proactively prior to execution may be beneficial. When pre-generating sequences prior to execution, sequences may either be used and discarded, or reused multiple times to prevent slow downs during runtime. Reusing a sequence requires storing the sequence as a fixed array, while one can also use a Linked List or a Queue as a disposable sequence. Functionality changes slightly when using a predetermined sequence to encrypt or decrypt the Aenigmagraph.

```
...encrypting using a pre-generated sequence..

LinkedList<?> input = {...information..};
//Contains cleartext stored into input//

LinkedList<point> aenigmaSequence = Aenigmagraph.generateEntireSequence();
//The above returns a linked list as a series of points. An example
//LinkedList would look like {(5,3),(5,4),(3,8)....}//

2DArray Array = new 2DArray[][];

while(!aenigmaSequence.isEmpty())
{
     2DArray[aenigmaSequence.remove(0)] = input.remove(0);
}
```

In many cases, reusing sequences allows the Aenigmagraph to coexist within many time sensitive cryptosystems, such as real time encryption and decryption, as the delay of generating points in real time may render some systems inoperable.

2.8 Calculating Total Combinations

The number of combinations increases exponentially relative to the size of the graph. However, the amount of combinations is not N factorial, as different permutations can result in the same sequence.

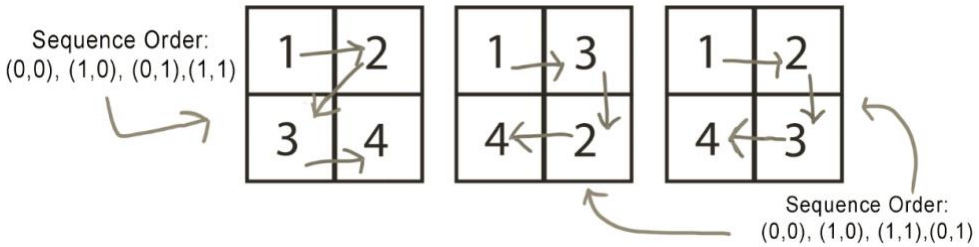


Figure 2.8.1: In a 2x2 graph, there are 4! (24) ways to uniquely order the weights 1-4. However, due to the way Prim generates, there are not 24 unique combinations of sequences. The figure to the left shows a grid labeled 1 through 4 linearly, and the directional order Prim will generate. The figures to the right show two unique permutations of Prim; however, due to the nature of Prim, unique combinations can generate the same sequence. As graph sizes increase, these overlaps decrease in size relative to the amount of combinations.

A program was made to discern the total number of unique combinations that can be generated by the Aenigmagraph of a known square size. The number of combinations increases exponentially with the size of the graph.

```
public static int count;
public GraphTheory(Aenigmagraph aenigma, LinkedList<point> points, int currentDepth, int maxDepth) {
     System.out.println(count);
     this.aenigma = new Aenigmagraph(aenigma);
          for (int a = 0; a < points.size(); a++) {
          Aenigmagraph temporyAenigma = new Aenigmagraph(aenigma);
          temporyAenigma.getPointStepOverride(points.get(a));
          if (temporyAenigma.getTouchingPoints().size() != 0 && currentDepth < maxDepth) {
               new GraphTheory(temporyAenigma, temporyAenigma.getTouchingPoints(),
                              currentDepth, maxDepth, printEvery);
          } else {
               count++;
          }
     }
}
```

Code 2.8.2: As the amount of combinations is not factorial, developing a recursive algorithm to count every possible combination up to a given depth was made. For the initial recursive call, a singular starting point is designated to count all the combinations spanning from the said point. "getPointStepOverride" manually selects the next point of the Aenigmagraph, rather than using a random number generator. To count all the combinations of points in a given grid, a "for" loop is used outside of the recursive formula to test all the starting points. The "maxDepth" variable is explained in ***Section 3.3****.*

3 Data and Results

3.1 Runtime and Memory

The runtime and memory usage of the Aenigmagraph are both linear in growth. Although exact measurements may vary from graph to graph, runtime and memory usage are designed to grow synchronously.

Memory

The proposed and working model of the Aenigmagraph stores the "touching Points" list as a linked list, which increases and decreases in size relative to the graph. Intuitively, the size of "touching Points" will never exceed the size of the graph itself. Conversely, whenever a point is added to "touching Points," it is also added to a HashSet named "visited Points" to allow for quick access of previous locations. It is worth noting that neither "visited Points" or "touching Points" will ever exceed the amount of points on the graph.

Proof

Whenever a "touching Point" is added, it is also added to "visited Points". Moreso, it is impossible for "touching Points" to ever be larger than "visited Points". As stated before, "visited Points" growth is dependent on the relative change of "touching Points". Thus the total memory used in a graph is the sum of "visited Points" and "touching Points". Because "touching Points" can never exceed the quantitative size of "visited Points", and "visited Points" can never be larger than the grid itself, it is impossible for the sum of the two at any time to exceed twice the size of the graph. Growth is at most N.

Runtime

The only time-related variable in the Aenigmagraph stems from the decision to use linked lists to choose the next point. Because "visited Points" is stored as a hash and thus access is limited to (1), "visited Points" is irrelevant when runtime growth is considered.

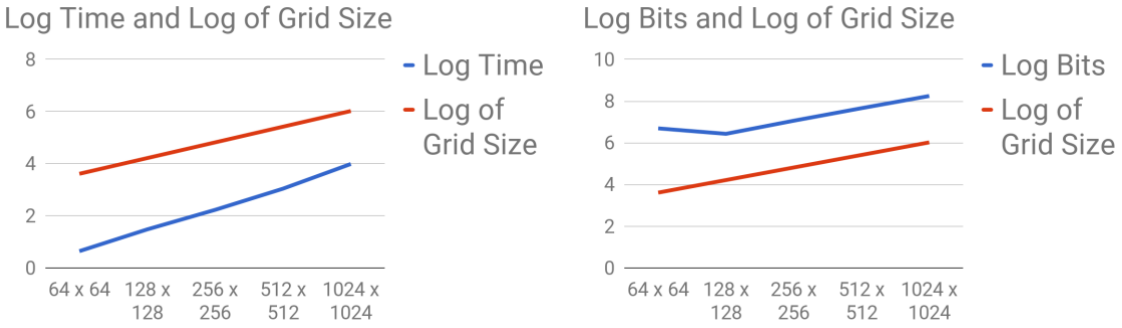
Proof

In a worst case scenario, the algorithm has to traverse to the last element of the linked list "touching Points" on every run. However, as stated before, "touching Points" will never amount to a quantity greater than a percentage of the graph. Coupled with the behaviour of random number generators, the linked list on average will only have to traverse half the linked list. With the two variables in mind, the graph per run would only have to traverse a maximum of half of a percentage of the graph. Increasing the size of a graph proportionately increases the size of these two variables, hence indicating linear growth.

3.2 Hardware Testing

Server Specifications:

Name: "Amazon T2 Micro"
Virtual CPU's: 1
Memory(GiB) = 1
Operating System = Linux, EC2



Graph 3.2.1: Showcases the linear growth in runtime and memory used to generate the Aenigmagraph sequence.

	64 x 64	128 x 128	256 x 256	512 x 512	1024 x 1024
Memory (bits)	4930250.4	2696847.6	11473187.1	45777790.0	175985205.3
Time (MS)	4.41	30.16	169.31	1098.69	9741.91

Figure 3.2.2: Showcases the average time required in MS and average memory in bits over 100 runs used to generate the Aenigmagraph.

3.3 Combinations

In the model proposed, we found that the growth of possible combinations of any given graph is exponentially related to its size. As stated in **Section 2.8,** the number of combinations of a graph is not *N* factorial, but instead follows a convoluted and peculiar pattern.

For larger recursive depths, it becomes infeasible to compute every possible graph. As such, a creative solution was found: we computed how many combinations exist at a particular depth, and derived an equation to predict the combinations found at each depth. Although the equation was never perfected, it was found that the growth of combinations per depth closely resembles the growth between factorials of integers. In order to compensate for the imperfection, a constant of .995 was multiplied to underestimate the amount of combinations at a given depth. Through a close study of various sizes, a general formula was made to estimate combinations of specific graphs at select depths.

$$\phi_x = .995 \times (log\frac{d(x-1)^2}{d(x-2)} + (\frac{log(\frac{(x-1)!\times(x-2)!)}{((x-1)!)^2})}{log(\frac{((x-2)!\times((x-3)!}{((x-2)!)^2})} \times (log(\frac{d(x-2)d(x-4)}{d(x-3)^2}))))$$

Equation 3.3.1: d(x) represents the amount of combinations existing at a particular depth. The equation is recursive, so a set amount of depths must be calculated in order to estimate future depths. The equation attempts to rationalize the growth between depths: the growth between depths follows a pattern similar to the change in factorials, and cannot be expressed with a fixed exponent. The constant of .995 is obtained from the equations behaviour that estimations were at most .995 off from the actual value, so the decision to underestimate was favoured over overestimating the amount of combinations.

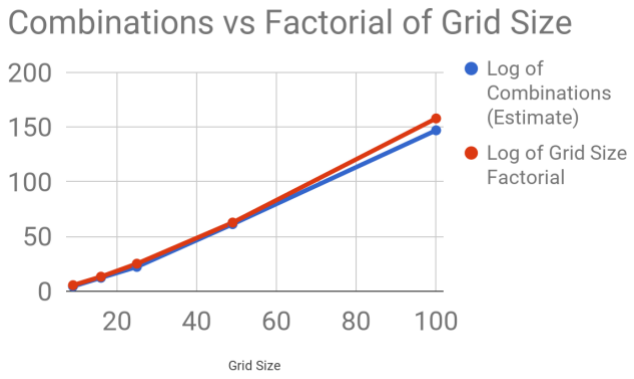


Figure 3.3.1: A diagram comparing the growth between the amount of possible combinations and the factorial of the grid size. Although incomplete, the graph gives insight into the exponential growth of the possible combinations of the Aenigmagraph. Numbers were acquired from testing various graph sizes to depth 9 and utilizing ***Equation 3.3.1****.*

For reference, the largest, fully computed graph is 4 x 4 (16). The amount of combinations found is $\approx 10^{12.4}$, while 16! is $\approx 10^{13.3}$.

4 Discussions

4.1 Limitations

As stated before, the Aenigmagraph does not edit values, making key frequency analysis relativity easy. To prevent cleartext attacks, it is paramount that either the values in the graph are eventually modified or that the graph be repeated through the same sequence multiple times to remove the oscillation-like properties (see **Figure 5.5.1**).

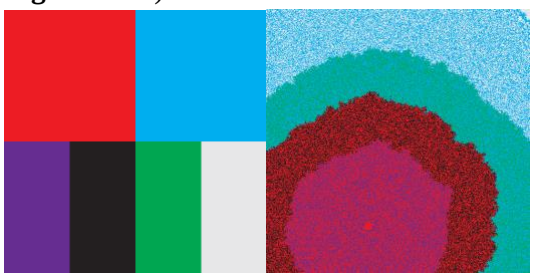


Figure 4.2.1: These images showcase the tendency of the Aenigmagraph to oscillate with a color test-card. The image to the left is the original test-card, and the image to the right is the same image with modified pixel locations. Manipulating values or repeating sequences is required to mitigate this tendency.

4.3 Usage over Selecting Random Points

Intuitively, generating a sequence of a graph solely from a random point generator without an algorithm may seem simpler and more random. Though this is true, it would come at the expense of both performance and memory. Consider, for example, the proposed solution wherein every point from 0-N has an equal chance of being chosen. Points would need to be chosen at random, and once a point is chosen it would also need to be removed. Using an arraylist would be impractical due to the rapid resizing, as the list needs to be resizable and elements need to be quickly removable. Therefore, using a fixed array would be impractical, as the array would need to continuously shrink. Using an arraylist for the proposed solution would have a $O(N^2)$ runtime due to the rapid resizing, and hence would not be feasible.

For this solution, a linked list would suffice, and each point would need to be written and saved into the linked list as an element. For every point selected, the graph on average would need to traverse $(N-x)/2$ times, where x is the number of points already chosen. In larger graphs this would prove infeasible, as the amount of elements traversed would result in exponential growth. Moreover, all N points would need to exist within memory at instantiation.

The proposed Aenigmagraph is more efficient in both time and memory. Given that not all the points are stored in memory in the Aenigmagraph at any given time, the memory used would be substantially smaller. Additionally, once a point is removed from memory, it is stored in a hash, which is faster and smaller in size than a linked list point. Coupled with the marginally smaller linked list size, the runtime would also significantly decrease. The Aenigmagraph could achieve the same result as the aforementioned algorithm; however, it would break the problem down into much smaller steps.

The difference in the number of combinations in both algorithms is marginal. However, the added speed and performance allows the Aenigmagraph to outperform the alternatives on the margin substantially.

4.4 Applied Usage: AES

There are many steps involved in AES that require writing information into a 2D grid of a given size [5].

The "shift rows" step (illustrated below) has the intention of obfuscation, which makes following the change of information more difficult. Using the Aenigmagraph as an alternative makes it difficult to track the change of information at these steps because the data is written in an unpredictable nature unique to each writing, as opposed to following a predictable change in position. The sequence can be unique to each iteration or can be reused for optimization purposes. In either case, the process is reversible.

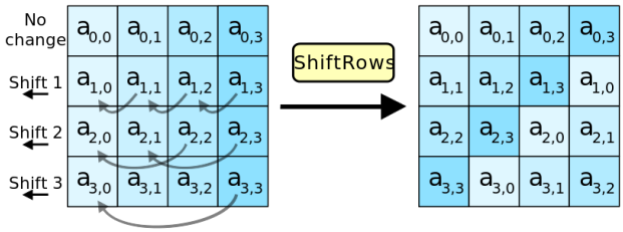


Figure 4.4.1: Figure illustrates the "shift rows" step found in AES. Diagram credit to "Matt Crypto" from Wikipedia [6].

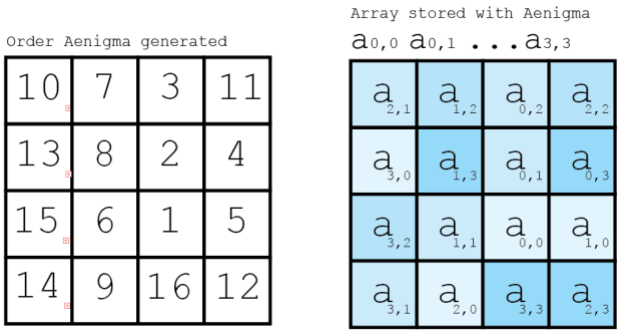


Figure 4.4.2: This figure showcases the alternative to the "shift rows" step, using the Aenigmagraph rather than shifting the rows. Colors are applied to contrast the default "shift rows" step with the Aenigmagraph.

4.5 Applied Usage: SHA 256

The Aenigmagraph also excels as a hash function, as explained in the methods section above. Implementing the Aenigmagraph to complement a cryptographic hashing function tackles two objectives in secure hashing [7]:

1) Given C in $f(m)=C$, it is harder to find a value of m.
2) Given a specific m, it is harder to find another value m' such that $f(m) = f(m')$

Because the seed is lost after every round, working in reverse adds the new layer of difficulty of determining the original order of the elements. Working in reverse is now met with the added difficulty of reversing the obfuscation of the Aenigmagraph.

When computing a specific *m*, any single error exponentially compounds upon itself, as each stage in the Aenigmagraph process acts as a seed for the next stage. An attacker would then have to attempt to compensate for the almost incoherent change in the order of the Aenigmagraph.

Because of this, the Aenigmagraph may be implemented into a hashing algorithm such as Secure Hashing Algorithm 256 (SHA-256). The initial steps of SHA-256 call for processing the message into chunks [8], in which data is manipulated to fit the size of a block. During this time, information is written into an array to be processed by the rest of the hashing system. Instead of writing information into these blocks linearly, writing information with the Aenigmagraph makes it significantly more difficult to attempt to compute specific hashes. Moreover, working in the reverse is more difficult. If in a hypothetical situation where an attacker is able to reverse the SHA-256 process up to the chunk processing stage, the attacker would then later have to reverse the Aenigmagraph and determine the original order of the chunk.

4.6 Vulnerabilities

To quote Robert Scheiner, *"Any person can invent a security system so clever that he or she can't imagine a way of breaking it."* It is difficult, but obviously not impossible, to foresee all possible ways to crack the Aenigmagraph system [9]. Considering the malleable nature of the Aenigmagraph, it is especially difficult to pinpoint individual weaknesses. However, through research into possible points of attack, we have proposed some points of concern.

The Aenigmagraph is susceptible to birthday attacks, as there are a finite number of states a graph may generate. This can be illustrated with the Pigeonhole Principle, wherein there exists an infinite number of unique seeds, but only a limited number of unique combinations. Users may be able to recreate the same Aenigmagraph sequence without knowing the original seed.

As stated before, the Aenigmagraph has a tendency to oscillate. When using the Aenigmagraph without adulterating the information, if an attacker can pinpoint the first few points, the attacker could attempt to deduce a seed from the LCG, and possibly discover the rest of the key. Repeating the 2D array through the same sequence makes it more difficult to deduce oscillations in a cleartext attack.

5 Other Proposed Uses and Extensions

Through various tests, we concluded that there are many different potential additions to the Aenigmagraph, some of which have no relation to hashing or symmetrical encryption. We hope that the following examples illustrate the malleable nature of the Aenigmagraph.

5.1 Image Steganography

Because an image is divided into pixels, treating each pixel as a point on a 2D grid allows us to use the Aenigmagraph to write information cryptically. Although for larger image sizes the Aenigmagraph takes longer to generate, common image sizes can be easily computed. Smaller, sub Aenigmagraphs may also divide up a larger image to speed up the computing process. The added benefit

of using the Aenigmagraph for image-steganography makes secure information harder to uncover, and allowing other, unorthodox features (see **Section 5.2**). Steganographic practices stand to greatly benefit from the Aenigmagraph, as it would be more difficult to extract a plaintext or ciphertext from an image.

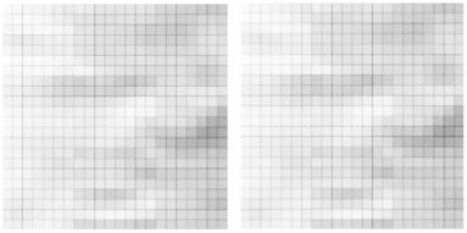


Figure 5.1.1: An example of the Aenigmagraph being used in a steganographic context. Because an image is a grid of elements, treating a pixel as a point on the Aenigmagraph allows for the writing of information with the Aenigmagraph cryptically. The picture was converted to black and white to help differentiate the subtle changes. The nature of the Aenigmagraph makes attempting to recover a ciphertext from an image very difficult due to the way the Aenigmagraph generates.

5.2 Reflexive Aenigmagraph

Aenigmagraphs are not required to contain a singular set of information, but rather may contain reflexive Aenigmagraphs coinciding within one another. This was tested when exploring the functionality in steganography of the Aenigmagraph, in which a single image contains two partitions of information. When the original information is finished writing into a 2D grid, a second Aenigmagraph could generate reflexive to the first graph without overwriting the original graphs points. The rationale behind a reflexive partition—if done properly—is that the second Aenigmagraph would be difficult to prove, allowing plausible deniability. This, however, would require non-reflexive Aenigmagraphs to write pseudo-cryptographic information into the remainder of the points to mimic real information to prevent users from attempting to prove a second partition contains meaningful data.

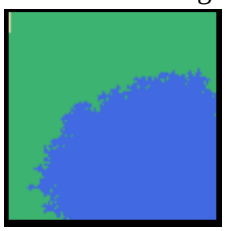


Figure 5.2.1: Two Aenigmagraphs existing in a 100x100 grid. Notice the fragmentation of the green graph: the second Aenigmagraph may disobey traditional generation rules to properly grow reflexively around another Aenigmagraph.

5.3 Multidimensional Aenigmagraphs

A multidimensional Aenigmagraph is comprised of an arbitrary amount of graphs generated in relation to one another; each graph can be generated from the same seed or from independent seeds. However, instead of writing information into a singular, two dimensional array, information is written into multiple two dimensional arrays, which are treated as a union. The added security of having information stored in a multidimensional array renders the prospect of an attacker deriving a seed from a cleartext attack increasingly unlikely. Moreover, the exponential amount of combinations the union contains is triple that of a singular graph, making it increasingly more difficult to ascertain the original

states of the graphs. In a steganographic use case, treating each of the three colors as a separate graph is a feasible way to unionize a two dimensional image.

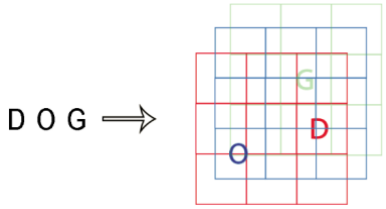


Figure 5.3.1: A visual example of "dog" being written into three separate Aenigmagraphs. Rather than storing elements into one 2D grid, it is stored into three separate 2D grids. Each grid may have its own unique seed, or, alternatively, can be entirely derived from a singular seed.

5.4 Proof-of-Work

A more abstract use of the Aenigmagraph is functioning as proof-of-work challenge. With popular cryptocurrencies such as Bitcoin using SHA-256 as a means of proof-of-work to prevent malicious votes [10], the Aenigmagraph may also function as a proof-of-work protocol to achieve similar goals. A prerequisite for a proof-of-work protocol to function is that challenges cannot be completed prior to usage; a user must use provided data to generate a specific result.

In a proposed scenario, the host decides on a 2D grid with specific information, then generates a seed to create the Aenigmagraph sequence. The host reorders the 2D grid through the sequence generated by the Aenigmagraph multiple times (to make finding a seed more difficult) and sends the original 2D grid, the obfuscated 2D grid, and the few integers in the seed to the user.

The user would then have to use the first few integers as a part of his/her solution to find the same seed the host used to generate the obfuscated 2D grid. Although collisions may occur, it requires substantially less power for the host to check if the seed is genuine when compared to the effort a user would have to expend to derive a solution. Given that Aenigmagraphs have a deterministic size, seeds will be found with a bounded probabilistic cost [11].

5.5 Image Scrambling

The Aenigmagraph can effectively scramble images by using sequences generated to reorder the position of pixels. Because of the oscillating properties of the Aenigmagraph, reordering elements an arbitrary amount of times is recommended. For **Figures 5.5.1** and **5.5.2**, the same sequence was applied to the same image three times, eliminating the need to generate a new sequence. For this protocol, information is neither added nor lost during transmission.

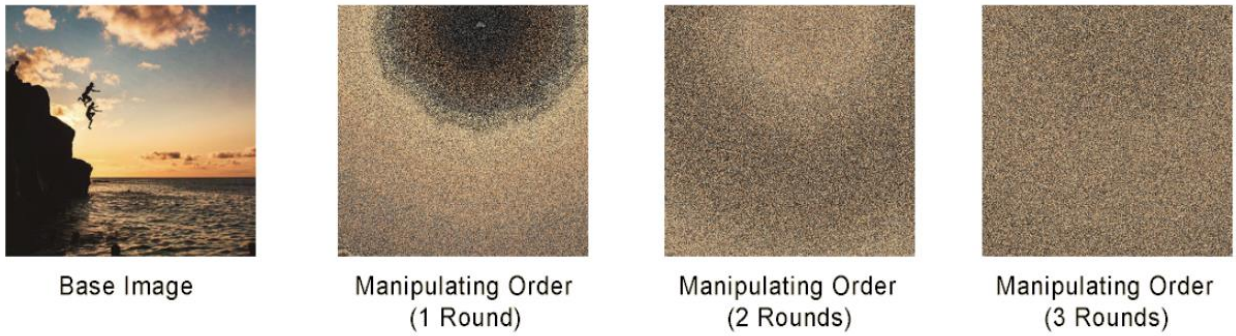


Figure 5.5.1: *Showcasing the pixel order being manipulated a sunset photo. Notice how in the first round, the image contains a wave-like pattern. However, even after one round, the original image is superficially incomprehensible.*

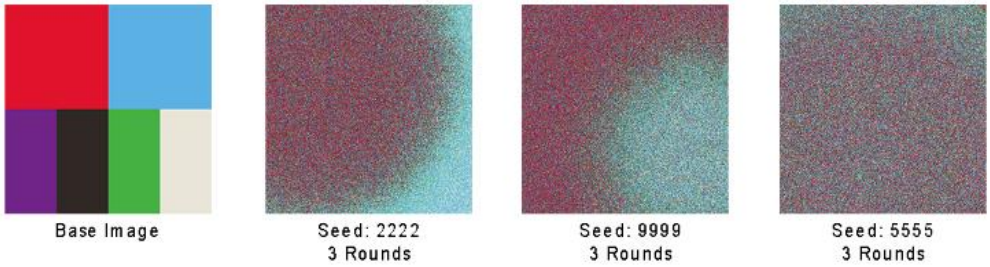


Figure 5.5.2: A color test-card is used to demonstrate the change in position pixels will undertake when rearranged by the Aenigmagraph multiple times. Some seeds will generate more obfuscated images.

Figure 5.2.3: Showing the superficial difference (or lack thereof) in scrambled images when comparing 3 rounds to 50 rounds. Oscillation-like properties become seemingly less apparent after 3 rounds.

6 Conclusion

In this paper, we proposed the Aenigmagraph as a way of improving crypto graphical systems, which alters the order information is written in order to further obfuscate sensible data and deter attackers. Rather than writing information linearly, attackers would have to deduce the original order of a 2D array to derive sensible information.

By using a heavily modified version of Prim's algorithm, we were able to cheaply generate sequences for 2D graphs to further complicate cryptographic systems. The runtime and memory usage of the Aenigmagraph is very minimal and predictable, allowing it to coexist within many crypto graphical systems without any major added cost. The immediate benefits can complement both symmetric encryption systems and track door functions, but are not necessarily limited to either.

Furthermore, we proposed various uses of the Aenigmagraph to exemplify its malleable nature. Because many computer applications involve writing information into a 2D array, the Aenigmagraph can be retrofitted into many software suites. We found that the amount of combinations of a graph grows exponentially with the amount of elements, making brute forcing inherently difficult.

The potential applications of the Aenigmagraph are at once near-limitless and yet transparently simple, while its resource-conscience optimization allows for neat and seamless integration within existing frameworks.

7 Acknowledgements

As this is a massive project to take on alone, there was a lot much needed external help. The following names, ranked by order, are people who helped me complete this paper.

Jonathan Lin - I genuinely wish to thank Jonathan for his outstanding contributions. Although no longer available in the final stages, Jonathan Lin was a very supportive cast member and helped formulate many of my thoughts into tangible ideas. Jonathan helped structure the early stages of the paper, as well as debating many of my ideas, which helped me strengthen my arguments. Jonathan also helped come up with the name “Aenigmagraph”, as well as helped contributing to my code.

Evan Hallam - I proposed the first ideas of an Aenigmagraph (then titled “szetograph”) to Evan while hunting. In the later stages of the paper, Evan gracefully helped edit and improve the final draft of the paper. Many of his edits are present in the final version, and many of his contributions proved invaluable. For this, I wish to acknowledge and thank Evan for his gracious support.

Julian Castellon - Another longtime friend of mine, Julian has always been a positive figure in my life. Not only did Julian offer friendship and support, he also provided the original materials necessary to bring this project to life. Julian also helped contribute to the editorial process, playing a pivotal role to this project's completion.

8 References

[1]Hon Wade, R., Mukund, V., Rohith, J., Rangaswamy, S., & Shetty, B.R. (2009). Steganography Using Sudoku Puzzle. 2009 International Conference on Advances in Recent Technologies in Communication and Computing, 623-626.

[2]Wu, Yue, et al. "Sudoku Associated Two Dimensional Bijections for Image Scrambling." Sudoku Associated Two Dimensional Bijections for Image Scrambling, 25 July 2012, arxiv.org/abs/1207.5856v1.

[3]Felgenhauer, B., & Jarvis, F. (2005). Enumerating possible Sudoku grids. Retrieved August 25, 2017, from http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf

[4]J. Stern, "Secret linear congruential generators are not cryptographically secure," 28th Annual Symposium on Foundations of Computer Science (sfcs 1987), Los Angeles, CA, USA, 1987, pp. 421-426.

[5]Daemen, J., & Rijmen, V. (n.d.). AES Proposal: Rijndael . Retrieved August 25, 2017, from http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf

[6] File:AES-ShiftRows.svg. (2014, February 24). Wikimedia Commons, the free media repository. Retrieved 01:28, August 26, 2017 from https://commons.wikimedia.org/w/index.php?title=File:AES-ShiftRows.svg&oldid=117233627.

[7]Schneier, B. (1996). Applied cryptography: protocols, algorithms, and source code in C (2nd ed.). Indianapolis, IN: Wiley.

[8]Descriptions of SHA-256, SHA-384, and SHA-512. (n.d.). Retrieved August 25, 2017, from http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf

[9]Schneier, B. (n.d.). Schneier on Security. Retrieved August 25, 2017, from https://www.schneier.com/blog/archives/2011/04/schneiers_law.html

[10]Nakamoto, S. (n.d.). Bitcoin: A Peer-to-Peer Electronic Cash System. Retrieved August 25, 2017, from https://bitcoin.org/bitcoin.pdf.

[11]Back, Adam. "Hashcash - A Denial of Service Counter-Measure." HashCash.org, www.hashcash.org/papers/hashcash.pdf.