cryptography

MDPI

Article

Matrix Encryption Walks for Lightweight Cryptography

Aeryn Dunmore [1,*], Juliet Samandari [2] and Julian Jang-Jaccard [1]

[1] Cybersecurity Lab, Massey University, SH17, Albany, Auckland 0632, New Zealand
[2] Department of Computer Science and Software Engineering, University of Canterbury, Corner Science and Engineering Roads, Upper Riccarton, Christchurch 8041, New Zealand
* Correspondence: a.dunmore@massey.ac.nz

Abstract: In this paper, we propose a new symmetric stream cipher encryption algorithm based on Graph Walks and 2-dimensional matrices, called Matrix Encryption Walks (MEW). We offer example Key Matrices and show the efficiency of the proposed method, which operates in linear complexity with an extremely large key space and low-resource requirements. We also provide the Proof of Concept code for the encryption algorithm and a detailed analysis of the security of our proposed MEW. The MEW algorithm is designed for low-resource environments such as IoT or smart devices and is therefore intended to be simple in operation. The encryption, decryption, and key generation time, along with the bytes required to store the key, are all discussed, and similar proposed algorithms are examined and compared. We further discuss the avalanche effect, key space, frequency analysis, Shannon entropy, and chosen/known plaintext-ciphertext attacks, and how MEW remains robust against these attacks. We have also discussed the potential for future research into algorithms such as MEW, which make use of alternative structures and graphic methods for improving encryption models.

Keywords: cryptography; matrix encryption; lightweight encryption; lattice encryption; IoT encryption

Citation: Dunmore, A.; Samandari, J.; Jang-Jaccard, J. Matrix Encryption Walks for Lightweight Cryptography. *Cryptography* **2023**, *7*, 41. https://doi.org/10.3390/cryptography7030041

Academic Editors: Josef Pieprzyk, Leonie Ruth Simpson and Mir Ali Rezazadeh Baee

Received: 19 July 2023
Revised: 9 August 2023
Accepted: 14 August 2023
Published: 16 August 2023

1. Introduction

Security is now necessary for any and all devices, and the encryption on lightweight or Internet of Things (IoT) devices is of specific concern. This is because these devices are limited in computational power and memory. As such, many schemes have been presented over the years to address this issue, such as GRAIN-128 [1]. Most of these ciphers, including GRAIN-128, still use traditional methods of encryption, namely, Feistel rounds, which are computationally expensive on lightweight devices. As such, there is a need for symmetric encryption schemes, which make use of non-traditional methods that are computationally less expensive. Most block ciphers are too computationally expensive for low-resource devices, though many have been adapted to try and address this—including DES-L or DES-Light. When considering architecture such as RFID or smart cards, the resource and computational requirements of the algorithm are more critical than ever [2]. To this end, we have developed a new scheme, named Matrix Encryption Walks (MEW), specifically for low-resource environments such as IoT devices or smart cards. Our method makes use of the coding structures referred to as matrices, also known as two-dimensional lattices. This scheme employs matrices to encrypt data using "graph walks" along the key, in the manner of a stream cipher. For our purposes, a graph walk is the path taken through the vertices of a graph structure. In this case, we utilize matrices as highly connected graphs, and as we use two 2D matrices, this effectively creates one 3-dimensional graph-like structure. The graph walks in this paper are therefore paths through the two Key Matrices, in which the algorithm passes back and forth from one to the other, creating a ciphertext encoding as it traverses the two keys. This can be seen a simple example in Figure 1. The path that weaves through the two Key Matrices provides the values for a simple Exclusive-OR (XOR) operation. Each byte of the plaintext is XOR'd first with the byte of the current coordinate

in the first Key Matrix and then XOR'd with the byte from the next coordinate (obtained using the first XOR'd byte for movement and direction) in the second Key Matrix. The first XOR'd byte of the plaintext determines where in the matrices the algorithm moves next. This is explained in more detail in Section 3.1, and a full example of the encryption of a 16-byte string is shown in Section 3.3. For a briefer view of the overall behavior of the algorithm, Figure 1 shows the path from coordinate to coordinate for each byte.

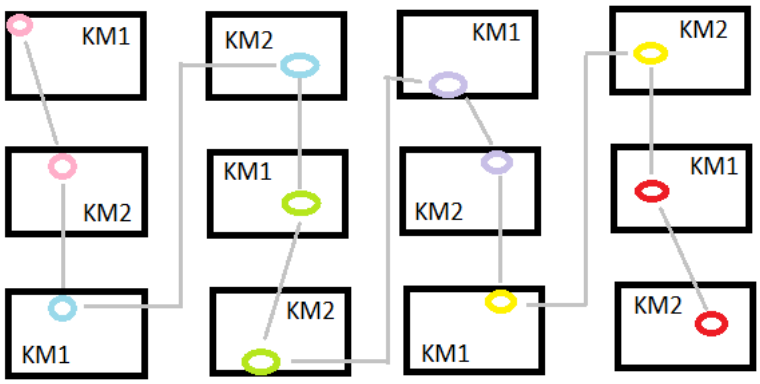

Figure 1. A simplistic view showing how the MEW algorithm weaves between the two Key Matrices byte by byte. Each different color represents a different plaintext byte.

In the Proof of Concept code, the string of plaintext for input is generated using Python's inbuilt Random library, and the last two binary digits of every byte provide the details for the direction, or where to move to obtain the next key byte, while the remaining bits provide the distance, or movement, for the direction. The Proof of Concept also uses the Random library to generate the contents of each Key Matrix. The scheme offers a lightweight symmetric encryption scheme with high levels of security. We demonstrate that the key space for the algorithm is significant and offers high levels of resistance to cryptanalysis, while the operations performed for encryption and decryption are linear in computational complexity, making the algorithm ideal for scenarios in which robust but lightweight encryption is required. Lightweight encryption is an area that has become increasingly important as the number of connected Internet of Things (IoT) and other resource-constrained devices grow. There are approximately 15 billion connected IoT devices, and this number is expected to nearly double by 2030. The number of IoT devices also currently outnumbers non-IoT devices, and this gap is only expected to increase over time [3]. These devices are often small microcontrollers with limited RAM, flash, and significantly slower speeds. However, these devices are being used for infrastructure and other sensitive tasks, so adequate security is imperative [4]. IoT devices have many varying applications and are expected to be widely deployed. However, one of the main concerns around the use of these devices is around security [5]. The prevalence of these constrained devices makes it imperative that encryption options are designed to be suitable for them. The simplicity of the operations in our proposed algorithm—namely, moving along the coordinates and using an XOR operation—are computationally inexpensive, making them ideal for lightweight encryption. Our unique contributions are as follows:

- We have built on prior research in the using of both lattices and matrices for alternative encryption schemes that eschew traditional Feistel cipher rounds.
- We have developed a new model for encrypting data on low-resource and IoT devices.
- We have provided extended theoretical examination of the model's potential security benefits.
- We have provided a Proof of Concept algorithm, which is available for further examination in Python at a GitHub repository, with the link in the text, and a full step-by-step example of the encryption process in two different ways (see Sections 3.2 and 3.3).
- We have provided an overview of the resource requirements and the execution of the algorithm in practice using said Proof of Concept algorithm.

- We have discussed these results and highlighted important directions for future research in this area, particularly noting potential examinations for cryptanalysis of the algorithm.

This paper is structured as follows: In Section 2, we review the related work in the fields of matrix encryption, graph walks, and lightweight cryptography design. In Section 3, we describe the proposed algorithm in detail, along with the experimental setup and results. In Section 4, we run our algorithm through a barrage of different methods for cryptanalysis and provide several explanations of the security generated by MEW. In Section 5, we critically examine the algorithm and discuss potential uses and future research directions for the proposed scheme. Finally, in Section 6, we provided a final summary of the research undertaken in this paper.

2. Related Work

2.1. Matrix Encryption Schemes

In prior research leading to this paper, in [6,7], the authors utilize the matrix construct as a key for an encryption system, named Coordinate Matrix Encryption (CME). This system is run against standards such as AES and offers a high-performance, high security opportunity. In [7], the Matrix Key contains multiple instances of all possible permutations of bytes, as well as a significant number of "padding" cells, or empty coordinates. This is used to provide extra security by inserting coordinates into the ciphertext that correspond to null values and that are discarded on decryption. An example of an 8×8 CME Key Matrix is shown in Figure 2, as compared to a Key Matrix for our Matrix Encryption Walks scheme, also 8×8, shown in Figure 3. The coordinates of the values themselves provide the ciphertext, unlike in MEW, where the values within the matrix are used to create the plaintext. The Coordinate Matrix Encryption scheme in [7] is also focused on non-singular mappings, in which one symmetric key and one plaintext can result in multiple different ciphertexts. In MEW, given the same plaintext and key, the returned ciphertext will be consistent.

```
Total strings: 8
Number of occupied spaces: 32
Number of blank spaces: 32
Total matrix size: [8,8]
[---][---][101][---][---][---][---][---]
[---][010][011][---][---][---][000][---]
[000][111][---][001][---][101][---][---]
[110][000][001][---][---][---][---][111]
[---][---][---][010][111][100][100][---]
[010][100][001][---][---][100][101][011]
[---][---][101][010][---][110][---][---]
[110][---][011][011][111][000][001][110]
Set up complete, time taken: 19 ms.
Total memory used: 0.43109130859375 MB
```

Figure 2. An example of a randomly generated 8×8 Key Matrix for Coordinate Matrix Encryption (CME) as in [6,7] (Reprinted with permission from the author).

206	11	57	19	106	98	7	75
135	172	68	176	156	177	249	208
215	141	115	92	128	220	238	196
7	29	53	9	27	36	33	8
12	255	251	123	125	70	185	94
211	31	214	158	65	65	39	15
68	229	153	133	112	176	198	204
252	172	26	98	150	235	25	195

Figure 3. An example 8 by 8 secret key for use in the Matrix Encryption Walks (MEW) scheme.

There exists a multitude of other cryptosystems that take advantage of a matrix structure. Using matrices as keys has been proposed in many different works, from Elliptic Curve Cryptography [8] to secure text messaging [9]. Matrix-based encryption has also been explored for securely encrypting images, such as in [10,11]. These systems, however, still often make use of traditional cipher rounds and procedures, meaning that for our purposes of lightweight encryption, they do not offer the low-resource encryption needed.

With regards to using matrices as keys, the MEW scheme provided for encryption in this paper bears a small resemblance to MASK, or Matrix Array Symmetric Key, in [11] and is used to generate a symmetric key for encryption of images. However, MASK makes use of traditional cryptographic functions, including using 16 block rounds for encryption and decryption with a key strength of 256.

The use of matrix multiplication to encrypt plaintext has been established in several papers. The use of "golden" matrices, built using Fibonacci sequences, as in [12], has been suggested to create encryption algorithms to turn incoming sequences into continuous ciphertexts. Unlike in MEW, this means there are significant restrictions and requirements for the creation of each matrix. MEW does not have this limitation—any matrix containing randomly generated bytes is acceptable as a key.

The McEliece cryptosystem is a public key encryption scheme based on matrices, suggested as an option for post-quantum encryption. It was introduced in 1978 by R. J. McEliece [13]. The McEliece system involves the use of a generator matrix, G, a random non-singular matrix, S, and a random permutation matrix, P [14]. The public key is the multiplication of these three matrices, as follows:

$$G' = S \cdot G \cdot P \tag{1}$$

While the McEliece system has gained ground in recent years as an option for post-quantum cryptography, the algorithm involves a number of matrix operations that increase computational complexity.

2.2. Graph Walks and Lattices

Implementation of graphs for encryption through the use of walks along nodes and vertices has been established in several papers, such as [15–17]. In particular, the use of paths in Ramanujan graphs in order to create secure hash functions, as in [16], involves the input of the function being used as directions for the walk along the vertices. This is the theory that underpins the encryption algorithm proposed in this paper. If we consider our Key Matrices to be highly connected graphs, and the values within to be vertices, MEW operates by walking along those vertices according to the first XOR'd byte, and the combination of plaintext with bytes from the vertices creates the eventual ciphertext. This, in concert with Coordinate Matrix Encryption, was the idea behind the development of MEW.

The use of graph families of large girth and unbounded degree, particularly Cayley graphs, for cryptography has been examined in papers such as [18]. Using large undirected graphs to encrypt data involves looking at the set of vertices as the space for the plaintext and the path within the graph as the secret key or password. If one chooses different

starting points within the graph, a single graph and plaintext can result in multiple different ciphertexts. This provides non-singular mappings of plaintext to ciphertext, such as those in [6,7]. The security of a graph walk-based cryptosystem increases with the girth of the graph. For a graph of girth g, a fixed prime number k, and paths of length s, the security of a graph walk scheme can be calculated as follows:

$$s \leq (g-1)/2 \tag{2}$$

$$x = k(k-1)^{s-1} \tag{3}$$

where x is the possible keyspace.

Further graph-based systems have been proposed that utilize matrices to store data, such as [19], in which Euler graphs and Hamiltonian circuits are employed to generate encrypted ciphertexts. The authors use an incidence matrix to calculate the possible paths and circuits through the plaintext and then use the adjacency matrix as the ciphertext.

2.3. Lightweight Encryption Methods

Many lightweight encryption algorithms have been proposed to provide security for Internet of Things devices and new smart technologies. However, many of these algorithms rely on the same traditional encryption rounds and techniques as are used in standard encryption such as the Advanced Encryption Standard (AES). Papers such as [20] provide encryption schemes for lightweight architectures but rely on block rounds, with substitution, permutation, and a 64-bit key, which is no longer considered to provide a significantly secure keyspace. Versions of the now-defunct Data Encryption Standard (DES), such as DESL or DES Light, from [21], have been suggested as low-computational resource encryption methods. However, these often significantly sacrifice security strength in order to limit the required resources. In [2], the authors surveyed a number of lightweight encryption methods designed for low resource systems and found a general focus on traditional methods with smaller block sizes, rounds, and keys. This differs greatly from the type of algorithm proposed in this paper. Similarly, ref. [22] uses substitution–permutation and Feistel rounds to provide an IoT-specific encryption scheme called LRBC. The classic Feistel cipher can prove to be quite resource-intensive and time-consuming. In [23], they survey the lightweight cryptosystems proposed for use by IoT devices. They compare the key sizes and constructions of these ciphers and find that about half of the options use block ciphers and only three are stream ciphers. They also find that there necessarily may be trade-offs between the security of the scheme and the computation required.

The National Institute of Standards and Technology (NIST) is a US government institute responsible for providing standards. In 2016, they identified cryptography for constrained devices as an important security area and hence announced a competition to find a suite of lightweight cryptography algorithms for use by IoT devices. They requested proposals for schemes that provide Authenticated Encryption with Associated Data (AEAD), with additional hashing functionalities being optional [24]. In February 2023, they selected a suite called ASCON, which includes authenticated encryption and hashing algorithms. ASCON is permutation-based and varies the number of rounds of the constants used depending on the variant being used. ASCON's encryption construction requires the nonce to be unique in order for security to be assured [25].

3. Proof of Concept and Use

3.1. Methodology

The encryption algorithm used for MEW was based partially on the work in [6,7] and on [16], using matrices and graph walks for encryption purposes. The CME method developed by [6,7] used matrices, which were half full and had many repetitions of the same byte within, and the ciphertext consisted of matrix "addresses" for each of the plaintext bytes, while also adding a chaotic element through a randomized binary choice to decide whether an empty coordinate should be inserted into the ciphertext at the next position.

In this way, the scheme achieved high security and a non-singular mapping of plaintext to ciphertext. Thus, the proposed Coordinate Matrix Encryption scheme resembled the encryption version of the popular board game Battleship. One noticeable negative outcome of the CME system is that the ciphertext is, on average, approximately three times the length of the plaintext. This is shown in Figure 4.

The use in CME of matrices and coordinates to encrypt data is what led to our proposal for MEW. As MEW operates, the characters of the plaintext dictate the walk along the Key Matrix, and unlike in [7], there are no padding characters, and each mapping using the same key/plaintext pair results in the same ciphertext.

```
Plaintext:
0101000110111011
Ciphertext:
1110011001001011101101001100010001100111101100111010010000000010
```

Figure 4. The example input and output of a binary CME scheme. The extra length of the ciphertext can be a detractor when looking at resource-constrained environments [7]. (Reprinted with permission from the author.)

3.2. MEW Algorithm Process

The Proof of Concept code for MEW creates two random matrices of bytes, tested at sizes 8×8, 16×16, 32×32, 64×64, 128×128, and 256×256, which are used as the Key Matrices for the encryption (the Proof of Concept code can be found at GitHub https://github.com/aerynsfyre/matrix-walk-encryption (updated on 29 July 2023)). The plaintext is translated first to a byte array, and then each byte is translated to a binary string as it is encrypted. The output ciphertext is an array of bytes. The process of encrypting and decrypting a single plaintext can be seen in Figure 5 or in more detail in Section 3.3. Each byte of the plaintext is encrypted one by one, as in most stream ciphers—though, MEW doubles back on itself, meaning it does not quite qualify as a stream cipher. First, the byte of the plaintext at the current index is XOR'd with the byte from the current (unchanged) coordinate of the first Key Matrix, starting at coordinates (0,0). This result is translated to a binary string, and the last two characters of this binary string determine the direction of the next step of the walk in the Key Matrix, as shown in Table 1. The remaining bits of the binary string, (those excluding the final two), are used to determine how far the walk will move in the given direction, referred to as the movement. The coordinates for (x, y) are then updated by adding the direction and movement (using the modulus operator so as to wrap around the table). The location in the second Key Matrix is then obtained using the new coordinates. The result of the first XOR is then XOR'd with the byte in the location on the second Key Matrix. The (x, y) obtained in this byte's processing is then the coordinate for the first Key Matrix. This continues until the entire plaintext is encrypted. The final two bytes of the ciphertext are the ending coordinates of the Key Matrix. In order to provide a significant avalanche effect and higher levels of security, this first ciphertext string is then reversed and encrypted again using the same method.

Decryption is conducted in two stages. First, the ending two bytes of the ciphertext are used to determine the start position for the decryption. The decryption itself starts on the byte immediately preceding these two location bytes. The byte is XOR'd with the second Key Matrix, and then the result is turned into a binary string in order to obtain direction and movement, with the decryption direction as per Table 1. Once the movement and direction is recorded, the new location is found in the first Key Matrix, and the current XOR'd byte is XOR'd once more, with the byte in the new location of the first Key Matrix. The resulting byte is recorded in an array, using the index location of the position in the ciphertext, and then the movement and direction are used to find the next location in the Key Matrices. This process is repeated until the algorithm has reached the first byte of the ciphertext. Then, the resulting semi-decrypted plaintext is reversed or flipped, and decryption is performed

again, using the same method. This then gives the final plaintext. The process of encrypting and decrypting a plaintext string can be seen in Figure 5.

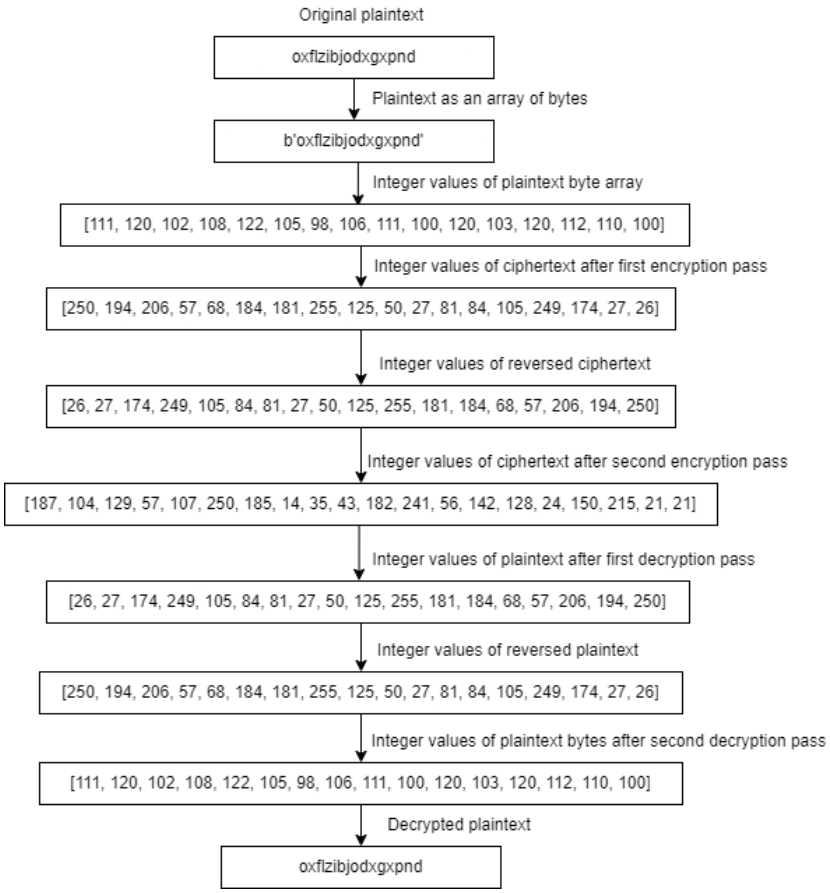

Figure 5. The process a single plaintext string goes through for encryption and decryption using the Matrix Graph Walk scheme. Example uses a 16-byte plaintext and size 32 Key Matrices.

Table 1. Encryption directions based on the final two bits of the result of an XOR with the plaintext and the key byte.

Bits	Direction	
	Encryption	Decryption
00	Down	Up
11	Up	Down
01	Right	Left
10	Left	Right

The first XOR between the plaintext and the first Matrix Key is used to obfuscate the directionality of the walk. If just the plaintext byte was used, this would be vulnerable to cryptanalysis because a known plaintext/ciphertext pair would provide the path taken for the matrix walk, and the second XOR with the second key matrix prevents a plaintext/ciphertext pair from being used to obtain the values of the first Matrix Key. Reversing the ciphertext and encrypting it a second time significantly improves the avalanche effect, as well as the security of the ciphertext. If the random bit changed is the last bit of the plaintext, then the first encryption pass will be almost identical for both of the original and altered plaintexts. Reversing the ciphertext and encrypting it again allows the final bit to impact on the overall result in a manner that increases the avalanche effect. This security is discussed further in Section 4. The interaction between the two Matrix Keys can be seen in Figure 6. The walk moves along the first key and goes back and forth into the second key.

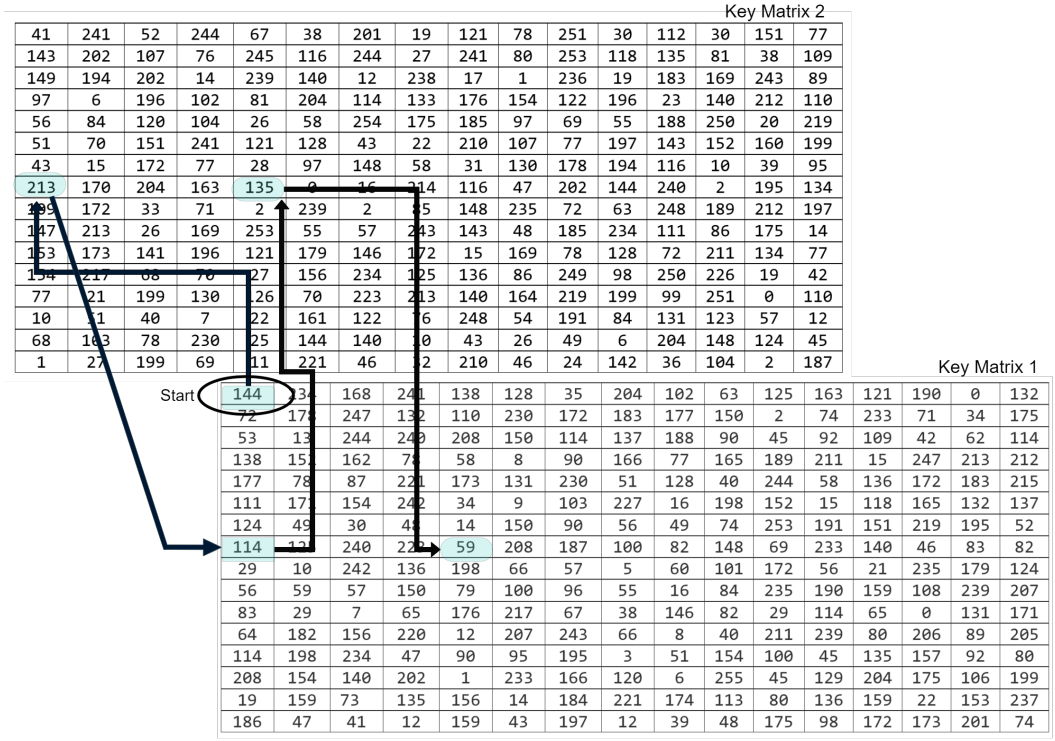

Figure 6. An example of interaction between two 16 × 16 Matrix Keys for the purpose of encryption. The byte in the location in the first key is used, then after direction and movement are obtained and used, the byte from the matching location in the second key is used, before moving to the corresponding current location in the first key.

3.3. Example Outputs

As part of the research into the prototype algorithm, we created random length inputs of characters and encrypted them using Matrix Keys such as the 8 × 8 example in Figure 3. We also encrypted English language text for the purposes of measuring frequency analysis. Since there is no correlation between the bytes of the key used in the XOR and the bytes of the plaintext, frequency analysis does not lend any clarity to cryptanalysis of the algorithm.

For the purpose of completeness, we have provided an output for the full process of the algorithm, tracing every step and change through the encryption and decryption stages. This can be seen in Table 2, which shows the two stages for encryption as they move byte by byte along the plaintext, reverse it, and move byte by byte along the reversed string, and then perform the inverse operations to decrypt it. The importance of the extra bytes added at the end of the final stage of encryption is done so the algorithm knows where the path of the encryption ended, giving the decryption process their starting coordinates. In Table 2, the column Position KM1 gives the coordinates at which the encryption of that byte of the string are being encrypted in the first Key Matrix. After applying both direction and distance, the Position KM2 column gives the coordinates in the second Key Matrix where the byte will be encrypted. Position KM2 becomes Position KM1 in the next byte, showing the path weaving between the two matrices as in Figure 6.

Table 2. The full process of encrypting and decryption of a 16-byte string using a set of 32 × 32 Key Matrices.

Original Plaintext: kztrspodbxxsxwgv **Plaintext as Byte Array:** [107, 122, 116, 114, 115, 112, 111, 100, 98, 120, 120, 115, 120, 119, 103, 118]						
Encryption Pass 1:						
Index	**Position KM1**	**Current Byte**	**Direction**	**Number of Spaces**	**Position KM2**	**Ciphertext Byte**
0	[0,0]	107	10	26	[6,0]	244
1	[6,0]	122	10	25	[13,0]	237
2	[13,0]	116	1	2	[15,0]	217
3	[15,0]	114	1	2	[17,0]	108
4	[17,0]	115	11	28	[17,4]	88
5	[17,4]	112	0	10	[17,14]	195
6	[17,14]	111	1	18	[3,14]	149
7	[3,14]	100	1	22	[25,14]	142
8	[25,14]	98	10	12	[13,14]	40
9	[13,14]	120	1	29	[10,14]	227
10	[10,14]	120	10	1	[9,14]	201
11	[9,14]	115	10	7	[2,14]	176
12	[2,14]	120	10	8	[26,14]	238
13	[26,14]	119	11	4	[26,10]	139
14	[26,10]	103	11	11	[26,31]	92
15	[26,31]	118	0	13	[26,12]	186
Encryption Result Stage 1: [244, 237, 217, 108, 88, 195, 149, 142, 40, 227, 201, 176, 238, 139, 92, 186, 26, 12] **Reversed:** [12, 26, 186, 92, 139, 238, 176, 201, 227, 40, 142, 149, 195, 88, 108, 217, 237, 244]						
Encryption Pass 2 (Final):						
Index	**Position KM1**	**Current Byte**	**Direction**	**Number of Spaces**	**Position KM2**	**Ciphertext Byte**
0	[0,0]	12	1	3	[3,0]	9
1	[3,0]	26	0	13	[3,13]	39
2	[3,13]	186	1	15	[18,13]	53
3	[18,13]	92	1	29	[15,13]	117
4	[15,13]	139	11	1	[15,12]	248
5	[15,12]	238	10	14	[1,12]	98
6	[1,12]	176	10	26	[7,12]	11
7	[7,12]	201	1	22	[29,12]	77
8	[29,12]	227	1	2	[31,12]	188
9	[31,12]	40	10	17	[14,12]	98
10	[14,12]	142	0	26	[14,6]	231
11	[14,6]	149	1	21	[3,6]	145
12	[3,6]	195	0	4	[3,10]	136
13	[3,10]	88	1	7	[10,10]	71
14	[10,10]	108	10	7	[3,10]	6
15	[3,10]	217	0	7	[3,17]	19
16	[3,17]	237	0	12	[3,29]	187
17	[3,29]	244	1	0	[3,29]	138

Table 2. *Cont.*

Encrypted Result Stage 2 (Final) / Output Ciphertext: [9, 39, 53, 117, 248, 98, 11, 77, 188, 98, 231, 145, 136, 71, 6, 19, 187, 138, 3, 29]						
Decryption Pass 1:						
Index	**Position KM1**	**Current Byte**	**Direction**	**Number of Spaces**	**Position KM2**	**Ciphertext Byte**
17	[3,29]	0	1	0	[3,29]	129
16	[3,29]	0	0	12	[3,17]	176
15	[3,17]	0	0	7	[3,10]	28
14	[3,10]	0	10	7	[10,10]	158
13	[10,10]	0	1	7	[3,10]	157
12	[3,10]	0	0	4	[3,6]	16
11	[3,6]	0	1	21	[14,6]	213
10	[14,6]	0	0	26	[14,12]	104
9	[14,12]	0	10	17	[31,12]	70
8	[31,12]	0	1	2	[29,12]	137
7	[29,12]	0	1	22	[7,12]	217
6	[7,12]	0	10	26	[1,12]	106
5	[1,12]	0	10	14	[15,12]	186
4	[15,12]	0	11	1	[15,13]	7
3	[15,13]	0	1	29	[18,13]	245
2	[18,13]	0	1	15	[3,13]	61
1	[3,13]	0	0	13	[3,0]	180
0	[3,0]	0	1	3	[0,0]	13
Decryption Result Stage 1: [12, 26, 186, 92, 139, 238, 176, 201, 227, 40, 142, 149, 195, 88, 108, 217, 237, 244] **Reversed:** [244, 237, 217, 108, 88, 195, 149, 142, 40, 227, 201, 176, 238, 139, 92, 186, 26, 12]						
Decryption Pass 2 (Final):						
Index	**Position KM1**	**Current Byte**	**Direction**	**Number of Spaces**	**Position KM2**	**Ciphertext Byte**
15	[26,12]	0	0	13	[26,31]	180
14	[26,31]	0	11	11	[26,10]	47
13	[26,10]	0	11	4	[26,14]	147
12	[26,14]	0	10	8	[2,14]	162
11	[2,14]	0	10	7	[9,14]	30
10	[9,14]	0	10	1	[10,14]	134
9	[10,14]	0	1	29	[13,14]	117
8	[13,14]	0	10	12	[25,14]	178
7	[25,14]	0	1	22	[3,14]	217
6	[3,14]	0	1	18	[17,14]	73
5	[17,14]	0	0	10	[17,4]	40
4	[17,4]	0	11	28	[17,0]	243
3	[17,0]	0	1	2	[15,0]	137
2	[15,0]	0	1	2	[13,0]	137
1	[13,0]	0	10	25	[6,0]	230
0	[6,0]	0	10	26	[0,0]	106
Decryption Result Stage 2 (Final) / Original Plaintext: [107, 122, 116, 114, 115, 112, 111, 100, 98, 120, 120, 115, 120, 119, 103, 118] **Success**						

3.4. Execution Time

All experiments were conducted on a PC with a 12th Gen Intel Core i5-1235U, 2.5 GHz processor, a 64-bit operating system, and 16 GB of RAM. The implementation of the Proof of Concept code utilized Python 3 and converts plaintext characters into bytes for encryption, as well as translating them into binary strings in order to check the last two bits of each character for the direction of the walk.

Table 3 shows the execution time of the Proof of Concept algorithm for an 8 × 8 Key Matrix. Execution time for the 16 × 16 Key Matrix is shown in Table 4, with results for the 32 × 32 Key Matrix shown in Table 5. As is clear from the tables, the key size does not impact the time taken to encrypt or decrypt the text. The execution time was recorded over 1000 iterations for each data size, and then the mean value was calculated.

Table 3. Execution time for the Proof of Concept algorithm with Key Matrix of size 8 × 8 in milliseconds.

Data Length	Encryption	Decryption
1024	0.578125	0.53125
2048	0.28125	0.484375
4096	1.890625	2.125
8192	3.140625	3.140625
16,384	6.015625	5.109375

Table 4. Execution time for the Proof of Concept algorithm with Key Matrix of size 16 × 16 in milliseconds.

Data Length	Encryption	Decryption
1024	0.265625	0.15625
2048	1.09375	0.59375
4096	0.578125	0.359375
8192	3.1875	3.4375
16,384	7.453125	6.953125

Table 5. Execution time for the Proof of Concept algorithm with Key Matrix of size 32 × 32 in milliseconds.

Data Length	Encryption	Decryption
1024	0.546875	0.5
2048	0.90625	0.75
4096	1.984375	1.796875
8192	3.609375	3.046875
16,384	5.890625	6.25

The fact that increasing the key size does not seem to significantly impact the performance of the algorithm suggests that distinct security benefits could be offered by this type of lightweight, symmetric encryption. The 32 × 32 Key Matrix offers a very high level of protection, and the encryption and decryption time of the longest lengths of data do not differ in any significant way from that of the smallest used key size of 8 × 8. The plots in Figures 7 and 8 show how the size of the key does not appear to have a significant impact on the execution time for encryption or decryption, with the plots showing key sizes of 8, 16, 32, 64, 128, and 256, with plaintext lengths from 1024 to 16,382. The generation of the Key Matrices is affected by the size, and appears to increase in proportion with the increase in the key size. This is shown in Table 6.

Table 6. Average time taken to generate the secure random Key Matrix by size over the course of 5000 iterations (measured in milliseconds).

Key Length	Key Generation Time
8	0.021875
16	0.05
32	0.1625
64	0.621875
128	2.41875
256	8.040625

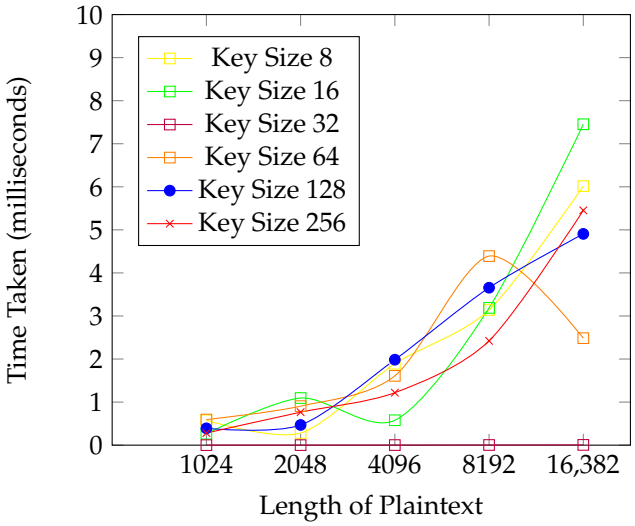

Figure 7. Execution time for encryption over different key sizes, averaged through 1000 iterations.

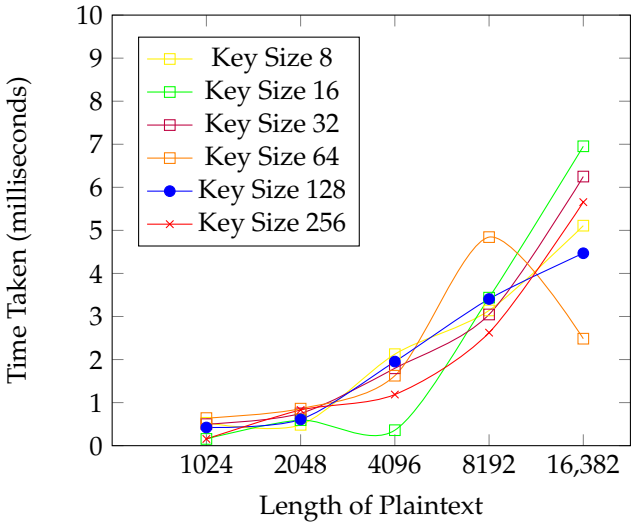

Figure 8. Execution time for decryption over different key sizes, averaged through 1000 iterations.

Given the importance of resource consumption in lightweight encryption, we were careful to view the memory required to store the two Key Matrices. We have shown the allocation in Figure 9, though it must be taken with the caution that this is one of the areas in which the construction of the implementation in our Proof of Concept code can significantly alter the results.

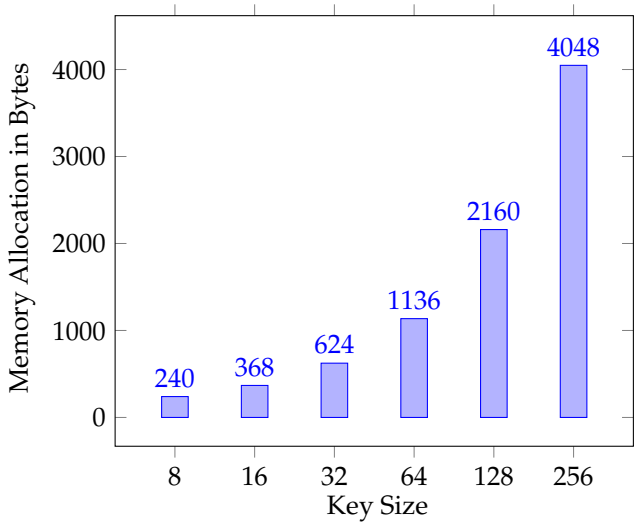

Figure 9. Memory requirements for the Matrix Walk keys by size, in bytes.

4. Security of the Algorithm

4.1. Avalanche Effect

In order to achieve the most significant avalanche effect, larger key sizes are required. A key size of 128 × 128 over a plaintext of length 4096 provides an average avalanche effect with the alteration of a single bit at 91.75%. We used key sizes of 128 and 256 to encrypt long plaintexts. Small key sizes do not provide adequate differences in the ciphertext for the avalanche effect to meet the requirement of 50%, because they wrap around a small table too many times. We measured the avalanche effect over 1000 iterations of encryption and decryption for each size of the plaintext and each key size of 128 and 256. The results can be seen in Table 7. The larger key sizes result in more significant avalanche effects, though even a key size of 64 with a plaintext length of 2048 resulted in an avalanche effect of 86.5%. The smaller key sizes are not appropriate for large amounts of data due to the lowered avalanche effect in these cases. A comparison of avalanche effect over different key sizes and plaintext lengths can be seen in Figure 10. As an example, a key size of 64 is not large enough to provide the necessary avalanche effect for a plaintext length of 16,382 bytes. However, keys of sizes 128 and 256 provide extremely high levels of alteration after the change of only one bit, even on plaintext data as long as 32,764 bytes.

An example of the way the algorithm moves through the Key Matrix can be seen in Figure 10.

Table 7. The overall avalanche effect by key size and plaintext length over 1000 iterations.

Key Size	Plaintext Length	Avalanche Effect
128	256	99.13%
256	256	98.88%
128	512	98.20%
256	512	99.29%
128	1024	97.73%
256	1024	98.77%
128	2048	95.53%
256	2048	98.60%
128	4096	91.76%
256	4096	97.31%

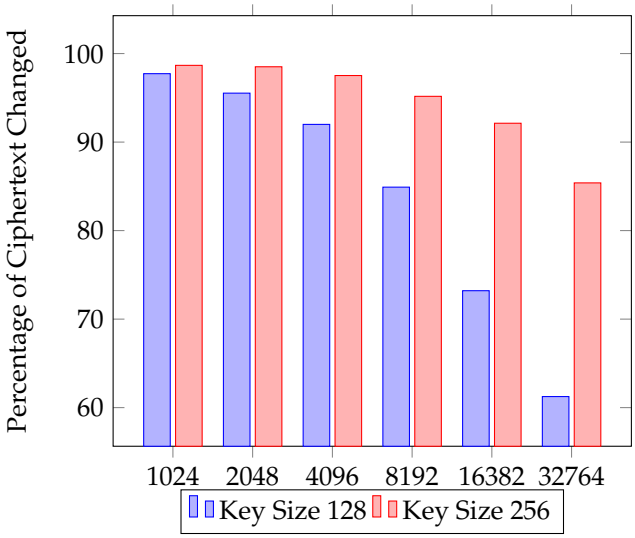

Figure 10. The avalanche effect over larger series of plaintexts and different key sizes.

4.2. Key Space

The security of the algorithm based on key strength can be computed using the possible key space for a single Key Matrix:

$$p_{ossiblekeys} = (n^2)^{256} \tag{4}$$

where n is the size of the Key Matrix. Each coordinate in the key is a randomly selected byte of value 0–255, meaning that a Key Matrix of size 8 has 16^{256} possible values, and a Key Matrix of size 16 has 256^{256} possible values. The largest Key Matrix size in the Proof of Concept algorithm, size $n = 256$, has a possible key space of 256^{512}. As two Key Matrices are used for the encryption and decryption process, this keyspace is doubled in practice. Even when taking into account the quantum algorithms for computing potential symmetric keys, such as Grover's Algorithm [26] (among others [27,28]), this still presents a strong front against brute force attacks and popular cryptanalysis. The high avalanche effect achieved with the larger key sizes and the use of two Matrix Keys offers a robust encryption algorithm with a fairly simple implementation.

While in this paper and the proof of concept algorithm we have used 2-dimensional matrices for the keys, it will not be a difficult proposition to extend these keys into more dimensions and simply use more bits of each byte to determine the direction in which to walk along the matrix. This would provide a significant increase in security, with only a minor increase in computational complexity for encryption/decryption operations, keeping the complexity in scalar linear time.

4.3. Frequency Analysis

In order to examine the possibility of cryptanalysis through frequency analysis, we made use of several online tools provided by dCode, an online "code cracker" software [29]. In order to perform true frequency analysis, we needed to use an English language string. We chose the first line of Pride & Prejudice, by Jane Austen [30].

> "It is a truth universally acknowledged that a single man in possession of a good fortune must be in want of a wife."
>
> The resulting ciphertext was:
>
> 42 183 90 0 213 209 246 49 101 66 200 155 46 81 50 48 122 233 133 222 119 129 176 47 45 196 152 143 207 123 150 239 215 62 121 153 92 232 199 67 40 13 191 48 91 161 26 49 209 85 37 85 174 255 252 49 12 53 247 67 50 20 193 184 164 120 188 196 144 212 55 186 184 130 150 177 129 124 88 208 60 110 147 194 111 105 9 20 54 33 220 171

230 165 85 151 13 194 255 140 156 227 159 50 92 94 103 97 163 103 195 77 115 41 128 39 205 21 19

When changed back into characters for the purpose of analysis, the cipher text becomes:

¿0[¡1ÑU%U®ÿü1zéÞw°/-ÄÏï×>yÇC(
Âÿã2ĝa£gÃMs)′ÍÐ<nÂoi 6!Ü«æ¥U

This was encrypted with a set of 32 × 32 Key Matrices, both generated using Python's inbuilt Random function. The ciphertext for the quote was then run through dCode's Frequency Analysis software. Table 8 below shows the frequency of the ciphertext characters compared to the frequency of the English language characters. For ease of comparison, we only included 26 of the 51 characters in the ciphertext, in order of most frequent to least frequent. As shown, there is no discernible pattern to the numbers in the ciphertext. This is because the bytes used to encrypt the plaintext have no correspondence to the bytes in the plaintext. The plaintext only indicates where to go within the Key Matrix once XOR'd with a byte from the first Key Matrix and is XOR'd with a second byte from the other Key Matrix. This makes it well-defended against frequency analysis attacks.

Table 8. The frequency of characters in the ciphertext compared to the frequency of letters in the English language according to [29].

Ciphertext Frequencies		**English Frequencies**	
Character	**Frequency**	**Letter**	**Frequency**
·	12.16%	E	12.70%
U	4.05%	T	9.10%
1	2.7%	A	8.20%
Ÿ	2.7%	O	7.50%
Ü	2.7%	I	7.00%
Ï	2.7%	N	6.70%
\	2.7%	S	6.30%
Â	2.7%	H	6.10%
Ã	2.7%	R	6.00%
G	2.7%	L	4.00%
¿	1.35%	D	4.30%
0	1.35%	C	2.80%
[	1.35%	U	2.80%
¡	1.35%	M	2.40%
∑	1.35%	W	2.40%
Ñ	1.35%	F	2.20%
%	1.35%	G	2.00%
®	1.35%	Y	2.00%
Z	1.35%	P	1.90%
É	1.35%	B	1.50%
Þ	1.35%	V	1.00%
W	1.35%	K	0.80%
°	1.35%	J	0.20%
/	1.35%	X	0.20%
-	1.35%	Q	0.10%
Ä	1.35%	Z	0.10%

4.4. Shannon Entropy

As part of our analysis of the algorithm's security, we used the ciphertext generated in Section 4.3 to calculate the Shannon entropy of the ciphertext. The calculation of the actual and ideal Shannon entropy was conducted using a python script (also available in the GitHub files). The entropy was found as in Table 9. We not only examined the English language string, but also two other strings randomly generated as plaintext. Since the English language string contains 23 words and 115 characters—corresponding to 119 bytes once encrypted—we used plaintexts of length 115 for comparative purposes. The closeness of the entropy values between the actual English language quote and the randomly chosen characters demonstrates how the algorithm produces near ideal results in this area.

Table 9. The calculated Shannon entropy for the English language string introduced in Section 4.3.

Text Type	Shannon Frequency	
	Actual	Ideal
English Language Quote	6.556459254850041	6.894817763307944
Random	6.440283306064294	6.94251450533924
Random	6.462680765431477	6.94251450533924

4.5. Chosen and Known Plaintext/Ciphertext Attacks

The security of the XOR of two strings is well known. The singular XOR operation of one plaintext and one key is trivial to break in known plaintext/ciphertext attacks. However, in our algorithm, each byte of the plaintext is XOR'd with two bytes of the keys, and then double XOR-d again with a different set of two bytes. Thus, using a known plaintext attack by removing the plaintext from the ciphertext provides only the product of XOR'ing four separate bytes of the key. Knowing the path the plaintext will take through the key matrix would be possible if the directionality and movement relied solely on the unencrypted plaintext. However, because the byte has been XOR'd with the first Key Matrix to obtain the next location, knowing the plaintext does not reveal any information about the path, even supposing that the attacker knows the size of the Key Matrices. Furthermore, once the first pass is finished, the half encrypted plaintext (already combined via XOR with a byte from each of the Key Matrices) is reversed, and then walks a new path through the keys based on coordinates generated by the bytes of the half-way encryption. At the end of this second path, the knowledge of the plaintext gives no real advantage, as subtracting it from the ciphertext only provides a string created such that it contains an XOR of 4 bytes of the two keys and one byte of the half-encrypted plaintext in each byte. Due to the reverse and second encryption, flipping a single bit of the plaintext in a known plaintext attack would also fail to provide any useful information. Regardless of where the changed bit is, it will change at least half of the path the plaintext takes. For example, if one flips only the last bit of the plaintext, the first encryption pass will be very similar to the original plaintext. However, the reversal and second encryption will start in a different place (as flipping the final bit will send the last byte of the plaintext to a different coordinate), and thus, the entirety of the second encryption will be likely very different from the ciphertext from the original plaintext, as is discussed further in Section 4.1.

5. Discussion and Future Research

The proposed model offers extremely lightweight symmetric encryption with a run complexity of $\Theta(2n)$, as it loops through the full plaintext twice. Since the size of the key does not effect the time taken to encrypt or decrypt the data, there is potential for applications in lightweight development environments such as IoT devices or smart cards, particularly if key generation is not performed on the device itself. When the space required to store the keys of size 32×32 is measured, it requires only 624 bytes, while a key size of 16×16 requires 368 bytes, and a key size of 8×8 requires 240 bytes. The full list of memory requirements for the different key sizes can be seen in Figure 9. This algorithm has overall a relatively low storage requirement, which could be met by low-resource devices.

The generation of Key Matrices for the scheme requires secure random number generation for seeds, which is an open research problem.

In our testing, we were unable (due to limits in computational power and time constraints) to create a thorough comparison of this proposed algorithm against other current state-of-the-art lightweight encryption algorithms. We were able to compare security with regards to keyspace, along with avalanche effect, frequency analysis, Shannon entropy, and chosen or known plaintext attacks. Future research should examine the security of the algorithm, for example using the OWASP framework presented in [31].

The Proof of Concept algorithm is slowed by the translation of each byte into binary to check for the directionality. In a more finalized or polished implementation, a dictionary of bytes with the directions and movement already calculated could be put into use in order to change this operation into a simple lookup, speeding up the algorithm.

The current implementation uses two 2D arrays of bytes to store the keys for the algorithm. The memory requirements for this particular implementation can be seen in Figure 9. Memory allocation is heavily implementation and device-dependent, and future research could examine the possibilities for decreasing the memory requirements so as to use the stronger keys on lightweight architectures.

Implementations that take the opportunity to extend the Key Matrix into more dimensions would also offer increased security, and would require a new split—as opposed to the two final bits for direction and the other six for movement—of each byte in order to have the creation of two direction and movement coordinates. This would provide an even more robust encryption of the plaintext and a significant increase in key space.

Since the proposed algorithm operates byte by byte, rather than in block rounds, it can be classified as a type of stream cipher. It eschews traditional Feistel rounds and has more in common with low-resource stream ciphers such as GRAIN (proposed in [1,32]), which operates using either an 80 or 128 bit key. In comparison, the key strength of the proposed Matrix Graph Walk scheme is several orders of magnitude greater than this lightweight stream cipher. However, due to the way it doubles back, it may not truly count as a stream cipher.

The security of the algorithm, as demonstrated in Section 4, displays the robustness of the underlying theory. Even in a basic implementation, the encryption provided is strong against different methods of cryptanalysis. One area we would have ideally been able to test was the potential for side-channel attacks. Unfortunately, due to computational limitations and lack of equipment, this remains as a subject for future research. A full security analysis and comparison, as suggested for stream ciphers, would also be an excellent opportunity for further research.

Future research should explore different methods of implementing the algorithm, including the addition of a byte dictionary to perform lookup operations and the possibility of expanding the Key Matrices into more dimensions. Potential block cipher implementations, in which the encryption and decryption are performed on blocks of plaintext of a fixed size, should also be explored for their possible use in encrypting larger sizes of data. The implementation of this algorithm on lightweight architecture such as IoT devices or smartcards also presents an opportunity to test how well this simple encryption algorithm would perform for architectures that need this type of minimalism from encryption schemes.

6. Conclusions

This paper has proposed a lightweight encryption algorithm called Matrix Encryption Walks, or MEW, based on existing cryptographic research into graph walks and literature regarding the use of matrices as encryption keys. We have provided a thorough grounding of the theory behind MEW, a detailed description of the way MEW operates, experimental results showing the speed and strength of this proposed method, as well as examples of Key Matrices and how the algorithm walks through these keys to produce the ciphertext. We have given a serious breakdown of the security of MEW as currently applied, using the key space, avalanche effect, frequency analysis, Shannon entropy, and the potential

for known or chosen plaintext/ciphertext attacks. Given this assessment of the security, and efficiency of MEW, we believe it seems to be a viable option for lightweight security through stream ciphers. Further research is needed into potential cryptanalysis and other potential implementations for performance enhancement and resource reduction.

Author Contributions: Conceptualization, A.D.; methodology, A.D.; software, A.D.; validation, A.D. and J.S.; formal analysis, A.D.; investigation, A.D.; resources, J.J.-J.; data curation, A.D.; writing—original draft preparation, A.D.; writing—review and editing, A.D. and J.S.; visualization, A.D.; supervision, J.J.-J.; project administration, J.J.-J. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All files for this project can be found online at GitHub (https://github.com/aerynsfyre/matrix-walk-encryption, updated on 29 July 2023).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AES	Advanced Encryption Standard
CME	Coordinate Matrixx Encryption
DES	Data Encryption Standard
DESL	Data Encryption Standard Light
ECC	Elliptic Curve Cryptography
IoT	Internet of Things
MASK	Katrix Array Symmetric Key
MDPI	Multidisciplinary Digital Publishing Institute
MEW	Matrix Encryption Walks
NIST	National Institute of Standards
PC	Personal Computer
RAM	Random Access Memory
XOR	Exclusive OR operation

References

1. Hell, M.; Johansson, T.; Maximov, A.; Meier, W. A Stream Cipher Proposal: Grain-128. In Proceedings of the 2006 IEEE International Symposium on Information Theory, Seattle, WA, USA, 9–14 July 2006; pp. 1614–1618. [CrossRef]
2. Singh, S.; Sharma, P.K.; Moon, S.Y.; Park, J.H. Advanced lightweight encryption algorithms for IoT devices: Survey, challenges and solutions. *J. Ambient. Intell. Humaniz. Comput.* **2017**, 1–18. [CrossRef]
3. Duarte, F. Number of IOT Devices (2023–2030). Exploding Topics. 2023. Available online: https://explodingtopics.com/blog/number-of-iot-devices (accessed on 29 July 2023).
4. Atkins, D. Requirements for post-quantum cryptography on embedded devices in the IoT. In Proceedings of the Third PQC Standardization Conference, Virtual, 7–9 June 2021.
5. Kumar, S.; Tiwari, P.; Zymbler, M. Internet of Things is a revolutionary approach for future technology enhancement: A review. *J. Big Data* **2019**, *6*, 111.
6. Cusack, B.; Chapman, E. Using graphic methods to challenge cryptographic performance. In Proceedings of the 14th Australian Information Security Management Conference, Edith Cowan University, Perth, Australia, 5–6 December 2016; pp. 30–36.
7. Chapman, E. Using Graphic Based Systems to Improve Cryptographic Algorithms. Ph.D. Thesis, Auckland University of Technology, Auckland, New Zealand, 2016.
8. Kinani, E.H.E. Fast Mapping Method based on Matrix Approach For Elliptic Curve Cryptography. *Int. J. Inf. Netw. Secur. (IJINS)* **2012**, *1*, 54–59. [CrossRef]
9. Balamurugan, R.; Kamalakannan, V.; Rahul, G.D.; Tamilselvan, S. Enhancing Security in Text Messages Using Matrix Based Mapping and ElGamal Method in Elliptic Curve Cryptography. In Proceedings of the 2014 International Conference on Contemporary Computing and Informatics (IC3I), Mysuru, India, 27–29 November 2014; pp. 103–106. [CrossRef]

10. Abu-Faraj, M.; Al-Hyari, A.; Alqadi, Z. A Complex Matrix Private Key to Enhance the Security Level of Image Cryptography. *Symmetry* **2022**, *14*, 664. [CrossRef]
11. Kumar, T.; Chauhan, S. Image Cryptography with Matrix Array Symmetric Key using Chaos based Approach. *Int. J. Comput. Netw. Inf. Secur.* **2018**, *10*, 60–66. [CrossRef]
12. Stakhov, A. The "golden" matrices and a new kind of cryptography. *Chaos Solitons Fractals* **2007**, *32*, 1138–1146. [CrossRef]
13. McEliece, R.J. A Public-Key Cryptosystem Based On Algebraic Coding Theory. In *Coding Thv*; Technical Report 42–44; National Aeronautics and Space Administration, Jet Propulsion Laboratory, California Institute of Technology: Pasadena, CA, USA, 1978.
14. Repka, M.; Zajac, P. Overview of the Mceliece Cryptosystem and its Security. *Tatra Mt. Math. Publ.* **2014**, *60*, 57–83. [CrossRef]
15. Ustimenko, V. On Graph-Based Cryptography and Symbolic Computations. *Serdica J. Comput.* **2007**, *1*, 131–156. [CrossRef]
16. Costache, A.; Feigon, B.; Lauter, K.; Massierer, M.; Puskás, A. Ramanujan graphs in cryptography. *arXiv* **2018**, arXiv: 1806.05709. https://doi.org/10.48550/arxiv.1806.05709.
17. Ustimenko, V. On semigroups of multiplicative Cremona transformations and new solutions of Post Quantum Cryptography. *Cryptol. Eprint Arch.* **2019**. Available online: https://eprint.iacr.org/2019/133 (accessed on 29 July 2023).
18. Ustimenko, V.A. On linguistic dynamical systems, families of graphs of large girth, and cryptography. *J. Math. Sci.* **2007**, *140*, 461–471. [CrossRef]
19. Nandhini, R.; Maheswari, V.; Balaji, V. A Graph Theory Approach on Cryptography. *J. Comput. Math.* **2018**, *2*, 97–104. [CrossRef]
20. Usman, M.; Ahmed, I.; Aslam, M.I.; Khan, S.; Shah, U.A. SIT: A Lightweight Encryption Algorithm for Secure Internet of Things. *arXiv* **2017**, arXiv: 1704.08688. https://doi.org/10.48550/arxiv.1704.08688.
21. Leander, G.; Paar, C.; Poschmann, A.; Schramm, K. New lightweight DES variants. In Proceedings of the Fast Software Encryption: 14th International Workshop, FSE 2007, Luxembourg, 26–28 March 2007; Revised Selected Papers 14; Springer: Berlin/Heidelberg, Germany, 2007; pp. 196–210.
22. Biswas, A.; Majumdar, A.; Nath, S.; Dutta, A.; Baishnab, K.L. LRBC: A lightweight block cipher design for resource constrained IoT devices. *J. Ambient. Intell. Humaniz. Comput.* **2020**, *14*, 5773–5787. [CrossRef]
23. Rana, M.; Mamun, Q.; Islam, R. Lightweight cryptography in IoT networks: A survey. *Future Gener. Comput. Syst.* **2022**, *129*, 77–89. [CrossRef]
24. Turan, M.S.; McKay, K.; Chang, D.; Kang, J.; Waller, N.; Kelsey, J.M.; Bassham, L.E.; Hong, D. Status Report on the Final Round of the NIST Lightweight Cryptography Standardization Process. 2023. Available online: https://nvlpubs.nist.gov/nistpubs/ir/2023/NIST.IR.8454.pdf (accessed on 29 July 2023).
25. Dobraunig, C.; Eichlseder, M.; Mendel, F.; Schläffer, M. Lightweight Authenticated Encryption & Hashing. Available online: https://ascon.iaik.tugraz.at/ (accessed on 29 July 2023).
26. Grover, L.K. A fast quantum mechanical algorithm for database search. In Proceedings of the twenty-eighth annual ACM symposium on Theory of computing—STOC '96, Philadelphia, PA, USA, 22–24 May 1996; pp. 212–219. [CrossRef]
27. Malviya, A.K.; Tiwari, N.; Chawla, M. Quantum cryptanalytic attacks of symmetric ciphers: A review. *Comput. Electr. Eng.* **2022**, *101*, 108122. [CrossRef]
28. Jozsa, R. Searching in Grover's Algorithm. *arXiv* **1999**, arXiv:quant-ph/9901021. [CrossRef]
29. dCode. Frequency Analysis on dCode.fr. Available online: https://www.dcode.fr/frequency-analysis (accessed on 8 August 2023).
30. Austen, J. *Pride and Prejudice*; Broadview Press: Peterborough, ON, Canada, 2001.
31. Fredj, O.B.; Cheikhrouhou, O.; Krichen, M.; Hamam, H.; Derhab, A. An OWASP Top Ten Driven Survey on Web Application Protection Methods. In Proceedings of the Risks and Security of Internet and Systems, Ames, IA, USA, 12–13 November 2021; Garcia-Alfaro, J., Leneutre, J., Cuppens, N., Yaich, R., Eds.; Cham, Switzerland, 2021; pp. 235–252.
32. Hell, M.; Johansson, T.; Meier, W. Grain: A stream cipher for constrained environments. *Int. J. Wirel. Mob. Comput.* **2007**, *2*, 86. [CrossRef]