applied sciences

MDPI

Article

Omega Network Pseudorandom Key Generation Based on DNA Cryptography

Gohar Rahman [1] and Chuah Chai Wen [2,*]

[1] Faculty of Computer Science, Information Technology, University Tun Hussein Onn Malaysia, Batu Pahat 86400, Johor, Malaysia
[2] Center for Information Security Research, Faculty of Computer Science, Information Technology, University Tun Hussein Onn Malaysia, Batu Pahat 86400, Johor, Malaysia
* Correspondence: cwchuah@uthm.edu.my

Abstract: Eliminating the risk of bugs and external decryption in cryptographic keys has always been a challenge for researchers. The current research is based on a new design that uses an Omega network-based pseudorandom DNA key generation method to produce cryptographic keys for symmetric key systems. The designed algorithm initially takes two random binary numbers as inputs to the Omega network design, generating an output of 256 symmetric keys. The Omega network uses the concept of the central dogma of molecular biology (DNA and RNA properties), including DNA replication (for DNA) and the transcription process (for RNA). The NIST test suite is applied to test the security properties of the proposed design. According to the study's findings, the suggested design is significantly suited to achieve the NIST test security properties and passes all of the NIST recommended tests.

Keywords: DNA; DNA cryptography; central dogma of molecular biology; key generation; randomness; NIST

Citation: Rahman, G.; Wen, C.C. Omega Network Pseudorandom Key Generation Based on DNA Cryptography. *Appl. Sci.* **2022**, *12*, 8141. https://doi.org/10.3390/app12168141

Academic Editor: Arcangelo Castiglione

Received: 20 June 2022
Accepted: 29 July 2022
Published: 14 August 2022

1. Introduction

Deoxyribonucleic acid (DNA) cryptography is the most recent advancement in cryptographic approaches. The natural process of DNA synthesis or production is exploited to encrypt data and later decrypt it [1,2]. DNA cryptography explains how DNA can be used as an information carrier and how the current science of biotechnology can transform plaintext to ciphertext. In addition, the primary aim of DNA cryptography is to provide greater secrecy than traditional cryptography by combining biological and computational properties [3]. DNA cryptography uses DNA computing, while DNA computing has several benefits, such as high parallelism, lower power consumption, and massive data storage. Based on these characteristics, DNA cryptography has a unique advantage in massively parallel data encryption applications, with less real-time demand, secure data storage, authentication, digital signature, and information hiding [1]. DNA cryptography transmits a message between a sender and a receiver such that an eavesdropper cannot comprehend it. A strong algorithm and strong key generation are needed to accomplish this task [4].

Numerous studies have been conducted on data security, particularly in DNA cryptography, to make data safe by encrypting it during transmission and using a specific key generation algorithm [5]. For example, the authors of [6] propose a key generation algorithm based on DNA cryptography and Linear Congruential Generator (LCG) sequences. The proposed algorithm uses unique biological characteristics and a pseudorandom generator to build a novel key generator. It was concluded that this algorithm only passed seven of the NIST tests.

The authors of [7] generated a 128-bit key by combining DNA cryptography and the Hill cipher algorithm. They claimed that the proposed technique could make the system

unbreakable and ensure a better and more secure cloud-distributed environment. The authors of [8] proposed hybrid methods combining DNA computing and mathematical division. The encryption and decryption are based on DNA computing; the key generation is based on mathematical division. Firstly, the initial key is a word with any length of characters. These characters are transformed into ASCII codes. Next, a random prime integer is triggered, greater than 255. Thirdly, each character's ASCII code is divided by the random prime number. These calculations produce the remainders. The same random prime number is divided by each character's ASCII code to obtain quotients. The final, newly generated key contains the random prime number, remainders, and quotients.

The authors of [9] proposed a cipher algorithm using biological processes and mathematical operations. The DNA sequence of nucleotides known as DNA-OTP represents the generated OTP key. Initially, the message is taken as input. The length of the message is computed, and a DNA nucleotide sequence is randomly assigned accordingly. The annealing process then produces a double helix DNA-OTP key. The transcription process converts the key to the mRNA sequence in the next step. Finally, the translation process creates a protein key from the mRNA sequence. The output is tested through the NIST test, which passes 13 out of 15 NIST tests.

Analyzing the techniques mentioned above, one may conclude that the techniques mentioned [6,9] partially fulfil the NIST random test. The research in [9] revealed a linear operation and neglected the NIST random tests. The authors of [8] used the mathematical division technique, not purely DNA-based key generation. In this context, most research work utilizes machine-learning techniques [4,10,11] and does not strictly follow DNA properties. Second, the generated keys are not random and thus fail to meet all of the NIST security properties. A researcher in DNA computing [12–14] has leveraged DNA's biological features in information security techniques [15]. However, it is well understood that the lack of randomness in key generator logical methods or mathematical functions leads to cryptanalytic breaches. As a result, a wide range of key generators with a high level of randomness is required [16]. Hence, the main motivation behind this research is to design a novel Omega network-based DNA cryptographic key generator. The design is applied to create a strong cryptographic key(s) for symmetric ciphering applications. The NSIT test suit is used to evaluate the performance of the Omega key generation algorithm in terms of randomness.

1.1. Overview of DNA

Apprehending the rudimentary principles of DNA cryptography in an emerging area of DNA computing, it is necessary to address the background details of the central dogma of molecular biology. DNA is an instruction storage molecule that carries the entire information essential for synthesizing proteins in all living organisms [10]. Every person has a unique DNA sequence, and no two individuals have an exactly matching sequence, even if they are identical twins, maternal twins, or mono-gametic twins [10,13].

DNA is a polymeric molecule of structural and functional units known as nucleotides. A genetic code or codon is a specific sequence in a properly constructed message that is triplet-coded. The nucleotides arranged in a specific sequence give specific features to each individual, which ultimately transfer to the next generation during sexual reproduction when the male gamete or sperm of the male mating partner meets with the female gamete or ovum of the female mating partner. Both sperm and ovum contain 23 chromosomes. Upon the union of sperm and ovum during the fertilization process, which produces a zygote cell, there are 46 chromosomes [11,12]. The single-cell zygote undergoes division to become a multi-cell structure and grow into a human baby. In the division process, a number of new cells are formed in the zygote, and new proteins are likely added to the cells. In new protein formation, the central dogma of molecular biology is involved. There is replication, transcription, and translation. Inside the cell center, DNA is transcribed into RNA. The RNA molecules formed inside the cell center from the DNA molecule are shifted to the cell cytoplasm for protein synthesis via translation.

1.1.1. Transcription

Transcription is a process in which a single-stranded messenger RNA (mRNA) is formed from a double-stranded DNA molecule. The process occurs inside the nucleus of a cell with the help of an enzyme called transcriptase. The mRNA acts as a transcript for the protein synthesized from that mRNA. The mRNA is transferred to the cytoplasm, where its translation is carried out [17]. In transcription, the nitrogen base of a DNA molecule named thymine is replaced with uracil in the RNA sequence, such that for a single DNA sequence AGCTTTA, the complementary mRNA strand will be TCGUUT.

1.1.2. Translation

Single-stranded mRNA formation in the cell nucleus results in protein formation inside the outer cytoplasmic region of the cell. The process of protein formation from mRNA inside the cell cytoplasm is known as Translation. The process involves the movement of mRNA from the cell nucleus to the cytoplasmic region of the cell, where mRNA uses the transcribed coded message of DNA to cause the collection of the desired protein units in the cell cytoplasm and consequently results in protein synthesis [18]. Take a DNA sequence TGC. Based on Table 1, the transcription process which produces mRNA from DNA would yield the sequence ACG. This sequence identifies the amino acid coded for by the mRNA as an ACG codon. The mRNA triplet message ACG codes for the amino acid threonine, abbreviated as Thr.

Table 1. Formation of mRNA codons and their relevant protein representations [9].

		Second Letter								
		U		**C**		**A**		**G**		
First letter	U	UUU	Phe	UCU	Ser	UAU	Tyr	UGU	Cys	U
		UUC	Phe	UCC	Ser	UAC	Tyr	UGC	Cys	C
		UUA	Leu	UCA	Ser	UAA	Stop	UGA	Stop	A
		UUG	Leu	UCG	Ser	UAG	Stop	UGG	Trp	G
	C	CUU	Leu	CCU	Pro	CAU	His	CGU	Arg	U
		CUC	Leu	CCC	Pro	CAC	His	CGC	Arg	C
		CUA	Leu	CCA	Pro	CAA	Gln	CGA	Arg	A
		CUG	Leu	CCG	Pro	CAG	Gln	CGG	Arg	G
	A	AUU	lle	ACU	Thr	AAU	Asn	AGU	Ser	U
		AUC	lle	ACC	Thr	AAC	Asn	AGC	Ser	C
		AUA	lle	ACA	Thr	AAA	Lys	AGA	Arg	A
		AUG	Met	ACG	Thr	AAG	Lys	AGG	Arg	G
	G	GUU	Val	GCU	Ala	GAU	Asp	GGU	Gly	U
		GUC	Val	GCC	Ala	GAC	Asp	GGC	Gly	C
		GUA	Val	GCA	Ala	GAA	Glu	GGA	Gly	A
		GUG	Val	GCG	Ala	GAG	Glu	GAG	Gly	G

1.1.3. Complementary Rules

DNA exists in a twisted ladder-like structure in humans, where the DNA bases A, G, C, and T are paired. These basses are adenine (A), guanine (G), cytosine (C), and thymine (T). An A in one strand will pair with a T in the other strand in two DNA helix strings, based on the complementary mapping rules in Table 2. Similarly, a G in one strand will pair with a C in the other [19]. This rule is known as Chargaff's base-pairing rule, which must be followed to keep both the strands chemically connected. This is why both the strands are always complementary to each other. According to the complementary rules, a single strand of DNA is converted to a double-stranded DNA molecule. The single strand of DNA is considered a parent strand, while the second strand is called a daughter strand.

Table 2. Complementary rules [20].

Symbol	Description	Base Represented					Complement
A	**A**denine	A					T
C	**C**ytosine		C				G
G	**G**uanine			G		1	C
T	**T**hymine				T		A
U	**U**racil				U		A
W	**W**eak	A			T		W
S	**S**trong		C	G			S
M	A**M**ino	A	C			2	K
K	**K**eto			G	T		M
R	Purine	A	G				Y
Y	P**y**rimidine		C		T		R
B	Not A (B comes after A)		C	G	T		V
D	Not C (**D** comes after C)	A		G	T	3	H
H	Not G (**H** comes after G)	A		C	T		D
V	Not T (**V** comes after T and U)	A	C	G			B
N	Any **N**ucleotide (not a gap)	A	C	G	T	4	N

2. Related Work

The authors proposed a key generation algorithm in [15] that takes an initial seed value of size 16 bits and generates a new key. This newly created key is only two bytes (16 bits) long. First, a 4×4 matrix is generated during the key generation process. The initial key number is transformed into ASCII and eight DNA bases. The initial key nucleotide sequence is placed in the first and second rows of the matrix. The remaining rows of the matrix are filled with bio-DNA addition, and bio-DNA subtraction operations used to generate eight DNA nucleotides from the initial key. Based on bio-DNA addition, the first two values of the third row and the fourth row are generated. However, the key size of the proposed algorithm is too short, only 16 bits long; a brute force attack may easily guess the key. Secondly, the natural biological steps are ignored.

The authors of [21] proposed two types of key generation for DNA encryption and decryption. The first type of key generation process is based on the ElGamal algorithm. The second type of key generation is based on DNA. The authors take random DNA sequences and generate public and private keys from the DNA codons. A random DNA sequence is generated using A, T, G, and C characters and stored in the database. However, the authors ignored the real biological steps for random key generation. Secondly, the key generation process takes a long time to encrypt and decrypt the message, as the ElGamal key generation process consists of complex calculations.

The authors of [22] designed a cryptosystem using a Malay machine and DNA cryptography. They proposed a key generation algorithm based on user attributes, such as email address, username, and personal identification number, first converting each of these attributes to ASCII characters. Secondly, the attributes are merged and obtained as a single string. This string is converted to binary numbers of size 304 bits. The last key size of 256 bits is obtained with extra binary numbers removed from the binary string's right side. The authors completely ignored the biological steps for key generation.

The authors of [23] used two dynamic encoding tables based on ASCII values. The first table is based on a 7-bit key taken from the dynamic encoding table. This table represents 96 ASCII characters as 7-bit binary sequences. The ASCII characters are randomly allocated to the 7-bit sequences, irrespective of their binary values. The second dynamic table is based on base 64 encoding rules. Each 6-bit sequence is assigned to a character from the base 64 encodings. However, character distribution is entirely random. The sender randomly chooses the 7-bit key from both tables and XORs the key with the 7-bit long plain text. Here the authors are missing the biological process, and the key size is short. The brute force attack may easily guess the real key after 27 combinations. The author improved

the key strength in [5], using the defi Hellman and genetic algorithm. This algorithm is computationally complex due to its complex operation.

The authors proposed in [24] a long 1024-bit DNA-based key using DNA computing. They used the DNA XOR operation and DNA complementary rules. In this algorithm, first, the user takes user attributes and a MAC address. For example, the user's date of birth and the names of the user's first school and first teacher. Applying some decimal and binary encoding rules, they split the binary values into two equal parts and perform the DNA XOR operation. Lastly, they use padding operations and complementary rules to obtain the final output key. In this algorithm, only one feature is included from biological DNA—its complementary rules. The remaining steps of the algorithm are entirely based on traditional calculation. Thus, it is missing the pure DNA biological steps. Secondly, this algorithm is based on linear operations and does not generate pseudorandom keys.

The authors of [25] presented a maximum-length matching algorithm where DNA strands represent the secret key of size 8 bits. The authors take the plaintext and convert the plaintext into a nucleotide sequence. Next, they select the last block of nucleotides generated from plaintext and use this as a key. They overcome the limitation of the secure key generation algorithm, namely, the "yet another encryption algorithm". The authors demonstrate that natural biological DNA can compute encryption. Moreover, the authors claimed that their proposed algorithm strongly resists frequency attacks. However, the authors of [26] claimed that the proposed algorithm in [25] requires multiple loops to execute the process, which increases the complexity sequentially.

A new S-box creation scheme based on encoding into a DNA codon, the XOR operation, and some arithmetic operations were proposed in [27]. The authors used the secret key size of 128 bits and a server name as examples for generating new S-box values. Two keys were used to strengthen the security of the created S-box. The authors claimed that their proposed method effectively passed the S-box test criteria, such as the balanced, completeness, avalanche, and strict avalanche criteria. However, the author used the block cypher S-box method, which is slow and takes time.

The authors proposed a technique in [28] for generating a random secret key based on the genetic information of a biological system. They used the process of the central dogma of molecular biology. The proposed method also uses the splicing system to generate a random key. Another method was proposed in [29], which employs the DNA hybridization process as an effective source for random key generation. However, these methods/techniques [28,29] do not pass the avalanche criterion and fail to produce pseudorandom keys with a 1-bit change. Secondly, according to the authors of [30], the pro-posed techniques [28,29] use physical processes such as thermal noise for generating random numbers.

3. Proposed Design

One Omega network is designed to generate a strong cryptographic pseudorandom DNA-based key. The design consists of biological DNA-based operations in which pseudorandom substitution and permutation operations are modified based on biological DNA, as discussed in Section 1. The modifications of pseudorandom substitution and per-mutation operations are shown in Tables 3 and 4. Table 3 presents the DNA XOR rules, which we denoted with the symbol (*). Table 4 presents the complementary mapping rules, which we denoted with the symbol (#). Table 5 presents the mRNA mapping rules, which we denoted with the symbol (μ and σ).

Our proposed design of an Omega network consists of three rows and four columns, as shown in Figure 1. We denoted it as $B_{row,\,column}$. $B_{2,3}$, $B_{2,4}$, and $B_{3,4}$ are based on DNA XOR rules. $B_{1,2}$, $B_{1,4}$, and $B_{3,3}$ are based on mRNA mapping rules, referring to the first letter followed by the second. $B_{1,1}$, $B_{2,1}$, and $B_{3,2}$ are based on mRNA mapping rules, which refer to the second letter followed by the first letter. $B_{1,3}$, $B_{2,2}$, and $B_{3,1}$ are based on complementary mapping rules. The operations are arranged randomly, and each operation is equally balanced, appearing in the design in order to produce cryptographically pseudorandom outputs.

Table 3. DNA XOR rules.

	Rule 1-T (00)				**Rule 2-C (01)**				**Rule 3-A (10)**				**Rule 4-G (11)**			
DNA XOR	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T
	10	01	11	00	10	01	11	00	10	01	11	00	10	01	11	00
A 10	A	G	C	T	G	T	A	C	C	G	T	A	T	A	C	G
C 01	G	C	T	A	T	A	C	G	G	T	A	C	A	C	G	T
G 11	C	T	A	G	A	C	G	T	T	A	C	G	C	G	T	A
T 00	C	T	A	G	C	G	T	A	A	A	C	G	G	T	A	C

Table 4. Complimentary mapping rules.

Input Binary	**Symbol**	**Description**	**Base Represented**					**Complement**	**Output Binary**
0000	A	**A**denine	A					T	0011
0001	C	**C**ytosine		C				G	0010
0010	G	**G**uanine			G		1	C	0001
0011	T	**T**hymine				T		A	0000
0100	U	**U**racil				U		A	0000
0101	W	**W**eak	A			T		W	0101
0110	S	**S**trong		C	G			S	0110
0111	M	A**M**ino	A	C			2	K	1000
1000	K	**K**eto			G	T		M	0111
1001	R	Purine	A	G				Y	1010
1010	Y	P**y**rimidine		C		T		R	1001
1011	B	Not A (**B** comes after A)		C	G	T		V	1110
1100	D	Not C (**D** comes after C)	A		G	T	3	H	1101
1101	H	Not G (**H** comes after G)	A		C	T		D	1100
1110	V	Not T (**V** comes after T and U)	A	C	G			B	1011
1111	N	Any **N**ucleotide (not a gap)	A	C	G	T	4	N	1111

Table 5. mRNA rules.

		Second Letter			
		T (00)	C (01)	A (10)	G (11)
First letter	T (00)	1110	1100	0010	0101
	C (01)	1101	0001	0100	1000
	A (10)	0000	1001	1111	1010
	G (11)	0110	0111	0011	1011

Firstly, two 128-bit binary string sperm and string ovum are generated. String sperm and string ovum is a union using XORing binary, which produces a string zygote. Three 128-bit random strings are generated, which are taken from Pi [31] and which we denote as random string *a*, random string *b*, and random string *c*. The string zygote and random strings are passed to the Omega network.

Firstly, the string zygote and random string *b* are XORed as the input $B_{1,1}$. Random string *a* and random string *c* are XORed as the input $B_{1,2}$. Random string *a* and random string *b* are XORed as the input $B_{1,3}$. String zygote and random string *c* are XORed as the input $B_{1,4}$. These have completed the first row, four columns of the Omega network.

Secondly, the output of $B_{1,1}$ and $B_{1,3}$ are XORed as the input to $B_{2,1}$. The output of $B_{1,1}$ and $B_{1,2}$ are XORed as the input to $B_{2,2}$. The output of $B_{1,2}$ and $B_{1,4}$ are XORed as the input to $B_{2,3}$. The output of $B_{1,3}$ and $B_{1,4}$ are XORed as the input to $B_{2,4}$. These have completed the second row, four columns of the Omega network.

Next, the output of $B_{2,1}$ and the output of $B_{2,4}$ are XORed as the input to $B_{3,1}$. The out-put of $B_{2,1}$ and the output of $B_{2,3}$ are XORed as the input to $B_{3,2}$. The output of $B_{2,2}$ and the output of $B_{2,3}$ are XORed as the input to $B_{3,3}$. The output of $B_{2,2}$ and the output of $B_{2,4}$ are XORed as the input to $B_{3,4}$. These have completed the third row, four columns of

the Omega network. Lastly, the output of $B_{3,1}$ is XORed with $B_{3,4}$; the length is 128 bits. The output $B_{3,2}$ is XORed with $B_{3,2}$; the length is 128 bits. Both strings are concatenated as 256 bits, yielding what is best known as a cryptographic pseudorandom DNA-based key. The complete steps are presented in Algorithm 1.

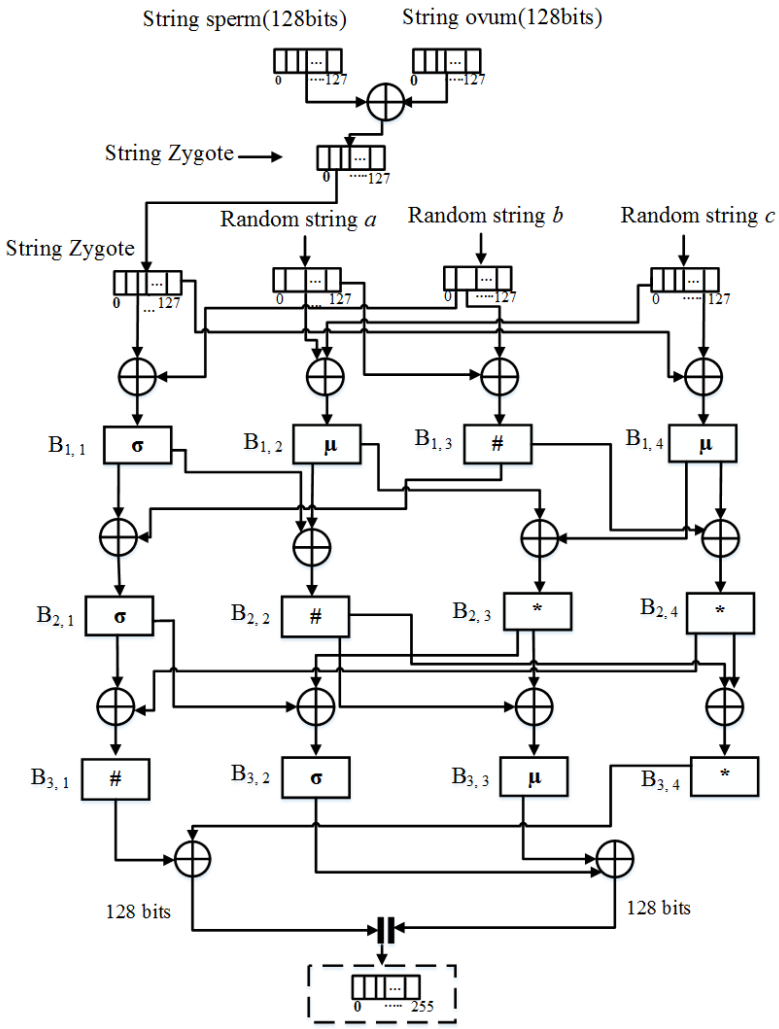

Figure 1. Omega network-based DNA key generation. Notes: *—DNA XOR rules; μ—mRNA mapping rules (first letter, second letter); σ—mRNA mapping rules (second letter, first letter); #—Complementary mapping rules.

Here, we provide a test case of the proposed Omega network design using four inputs and their corresponding outputs. The four inputs are string sperm, string ovum, and three random strings. The output is a cryptographically pseudorandom key.

- String sperm:

100010111110011110110111001110100101100011010001011010010101110100010001100100000011111000111010100000001100110010101001011100110

- String ovum:

111101011000001011010111001110100010111111100101001000001110011010010101011000010110111010111011010000001110111000010101010111011

- Random string *a*:

000000001101011011011001100011111111010011000110000010001111001010110111111100110010010001111011100100011100111111010100011111000010

- Random string *b*:

000001011001110110010000110111001001000100011100111010011001101011110100110
0001110110100101010001111010100000100111010011011101100

- Random string *c*:

110110110100000111100110000111000001011001001110011100110100100001101011
1000011101011010000100011110001100111001110010110110 0

- The final output of a 256-bit key is obtained:

111000101110110111011000110100011110110011110001110101110101010010000011 0
011111000010100111000001011010011111101010110101001010111100100001110011001000 10
100001001001011010011100000100010000001110110010000011010011111001000101101100 00
10111010001100101110 0

Table 3 shows the DNA XOR operations with different combinations of A, T, G, and C letters. The design consists of four different binary combinations: the letter A is binary 10, the letter C is binary 01, the letter G is binary 11, and the letter T is binary 00. There are four rules based on the four DNA letters. The XOR for each letter is perfectly balanced as a quarter to preserve the entropy of the input. Assume there are two strings: the first string is ACCCT and the binary is 1001010100; the second string is CCTAA and the binary is 0101001010. Note that the letter for the first string is A. Then, the operation XOR is based on rule number three. The final output for XORing these two strings is GTAGA and the binary is 1100101110. The binary mapping of these rules is given in Appendix A, Figure A4.

Table 4 represents complementary mapping letters in binary form. We have assigned four binary values for each of the nucleotide letters. Let us assume that we have a sequence of binary bits 0000 representing a base A nucleotide. The complementary output will be 0011, which represents a T base. In the second example, we have the input string 001100010010. The nucleotide sequence of this string is TCG. The complementary output will be 000000100001, which represents AGC. Further details are given in Appendix A, Figure A3.

Table 5 shows the mRNA mapping rules. Assume a string AAAGTA; the binary is 101010110010. The first letter of the string is A and the second letter is A; the output is GG. The third letter of the string is A and the fourth letter of the string is G; the output is AA. The same process is performed for the entire string. Hence, the output for the string AAAGTA is GGAATA. The details of the mRNA rules are given in Appendix A, Figures A1 and A2.

Algorithm 1. Pseudorandom DNA key generation algorithm.

Inputs: Strings: Sperm (0 . . . 127) and ovum (0 . . . 127)
Outputs: String: key (0 . . . 255)
Var: Strings: *a* (0 . . . 127), *b* (0 . . . 127), *c* (0 . . . 127), zygote(0 . . . 127)
Strings: Bi, j (0 . . . 127): i, j ∈ {1,2,3,4}

1. zygote ← sperm ⊕ ovum
2. *a* ← Rand(128)
3. *b* ← Rand(128)
4. *c* ← Rand(128)
5. $\{B_{1,1}, B_{1,2}, B_{1,3}, B_{1,4}\} \leftarrow \{$ zygote ⊕ *b*, *a* ⊕ *c*, *a* ⊕ *b*, zygote ⊕ *c* $\}$
6. $\{B_{1,1}, B_{1,2}, B_{1,3}, B_{1,4}\} \leftarrow \{ \sigma(B_{1,1}), \mu(B_{1,2}), \#(B_{1,3}), \mu(B_{1,4}) \}$
7. $\{B_{2,1}, B_{2,2}, B_{2,3}, B_{2,4}\} \leftarrow \{ B_{1,1} \oplus B_{1,3}, B_{1,1} \oplus B_{1,2}, B_{1,4} \oplus B_{1,2}, B_{1,4} \oplus B_{1,3} \}$
8. $\{B_{2,1}, B_{2,2}, B_{2,3}, B_{2,4}\} \leftarrow \{ \sigma(B_{2,1}), \#(B_{2,2}), *(B_{2,3}), *(B_{2,4})$
9. $\{B_{3,1}, B_{3,2}, B_{3,3}, B_{3,4}\} \leftarrow \{ B_{2,1} \oplus B_{2,4}, B_{2,1} \oplus B_{2,3}, B_{2,3} \oplus B_{2,2}, B_{2,4} \oplus B_{2,2} \}$
10. $\{B_{3,1}, B_{3,2}, B_{3,3}, B_{3,4}\} \leftarrow \{ \#(B_{2,1}), \sigma(B_{2,2}), \mu(B_{2,3}), *(B_{2,4})$
11. key ← $(B_{3,1} \oplus B_{3,4}).\ (B_{3,2} \oplus B_{3,3})$
12. Return key

4. Methodology

This section presents the simulation setup. Firstly, the proposed Omega network-based pseudorandom DNA key generation algorithm generates random keys. The complete execution process of the proposed DNA key generation is explained in the previous section. The randomness of the test is checked, and data sets are analyzed. The generated data sets are based on a specific function. After preparing the data set, the proposed key generation algorithm is tested using the NIST statistical test, and the results are evaluated. The overall research methodology flow diagram is shown in Figure 2.

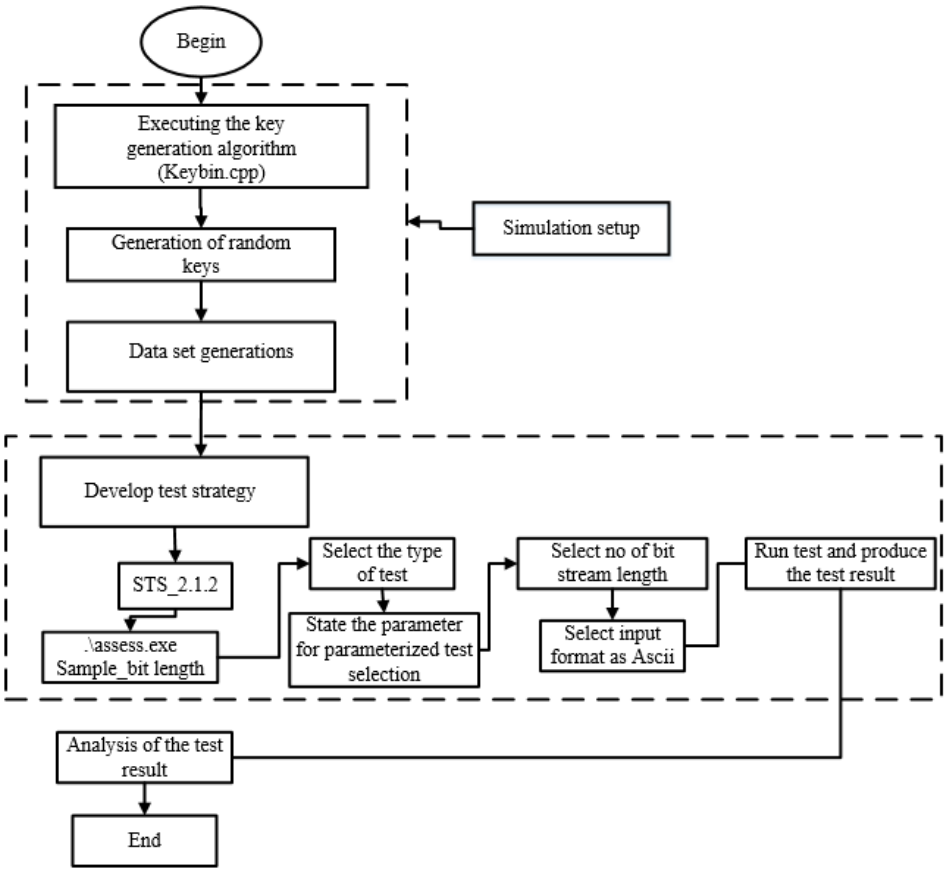

Figure 2. Research flow diagram.

4.1. Simulation Setup

The Omega network is designed to generate DNA-based keys. The source code is written in C/C++ (see Appendix A, Figures A1–A7). The design is simulated using an IntelI CITM) i7-8550U CPU 1.99 GHz, with a hard disk and 8 GB RAM. To execute the Omega network key generation algorithm, we need to generate 73,465,856 bits of DNA-based key. The string sperm and string ovum are 128 bits each.

A 256-bit DNA-based key is generated for every string sperm and string ovum. The union of string sperm and string ovum is obtained using the XOR operation; we denoted it as string zygote. To generate 73,465,856 bits, we require a 286,976-bit string zygote. Note that for each string zygote only a one-bit change as shown in Figure 3. This indicates that the changes can be made for string sperm or string ovum but subjected only to one-bit changes for each execution of 256 bits of DNA-based key. In the end, each 256 bits of DNA-based key are concatenated to obtain a total of 73,465,856 bits of DNA-based keys. The National Institute of Standards and Technology (NIST) test suite (STS 8.1.2 application) was used for pseudorandomness.

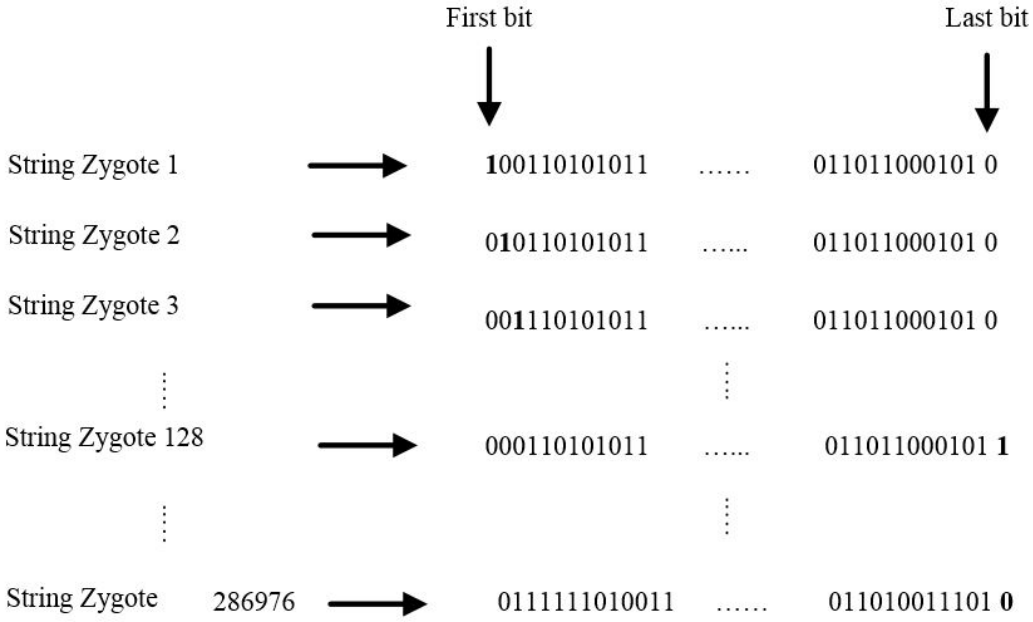

Figure 3. One-bit change example.

4.2. NIST Test Suite

The NIST Test Suite is made up of 17 different empirical tests [32] that were developed to analyze binary sequence bitstreams. The tests investigate the unpredictability of data through the use of a variety of bit statistics as well as bit block statistics. The NIST STS tests all investigate the degree to which entire bitstreams are governed by randomness. Multiple tests that can detect local nonrandomness break the bitstreams down into several generally large pieces and calculate a bit characteristic for each segment of the data. After that, the test statistic is computed by using each of these partial attributes in turn. Checking the randomness of the sequence is a part of each of the NIST STS tests, and each test is specified by one of the three types of test statistic. For example:

1. Bits: These tests examine a variety of aspects of bits, such as the proportion of bits, the frequency of bit change for example, runs, and cumulative sums;
2. M-bit blocks: These tests examine the allocation of m-bit blocks (where m is typically less than 30 bits) inside the sequence or its portions;
3. M-bit parts: These tests examine the complex properties of M-bit parts (where M is typically greater than 1000 bits) of the sequence, such as the rank of the sequence as seen in a matrix, the sequence's spectrum, or the linear complexity of the bitstreams.

Some tests that are part of the NIST STS are carried out in a greater number of variations. For instance, they may carry out a number of sub-tests and investigate additional qualities of a sequence of the same type. For example, the cumulative sum test and the serial test look at a sequence forward and backward orders.

4.3. Developing the NIST Test Strategy

NIST STS allows analysis of an input file as one block sequence or as n fixed-length sequences. We chose the parameters in the following order in the text-based user interface.

(a) File for the analysis: run the key generation algorithm and generate a 9 MB file;
(b) All the NIST recommended tests are applied;
(c) Values for the parameterized tests are set; for example, block frequency 21, overlapping template matching 9, non-overlapping template matching 9, approximate entropy 10, serial 16, linear complexity 500;
(d) Choose the number of bitstreams;
(e) File format: ASCII 0s and 1s are chosen;

(f) The sequence length, sample size, and significance level are input parameters. The sample size for 9 MB files is set to 100 binary sequences, and 0.01 is chosen as the significance level. A *p*-value is calculated for each binary sequence and sample size;
(g) A success or failure evaluation is made for each *p*-value depending on whether it exceeds or drops below the pre-determined significance level;
(h) For each statistical test, whether it passed or failed the standard passing rate requirements, a *p*-value and proportional values are recorded;
(i) Both of the measures are assessed. A sample would pass a statistical test if it passed both the proportion and uniformity criteria. If one of the two *p*-values is less than 0.01, the sample is considered unsuccessful. If this occurs, further samples are analyzed, otherwise the sample is considered to pass the randomness test.

5. Results and Discussion

The NIST test suite is a statistical package that includes 17 tests designed to evaluate the randomness of arbitrarily long binary sequences created by cryptographic random or pseudorandom number generators based on hardware or software [33]. These tests look for several sorts of non-randomness present in sequences. The probability value (*p*-value) is used in these tests to determine the randomness of the sequence. The sequence is non-random if the estimated *p*-value is less than 0.01; otherwise, it is inferred as random [34,35]. To pass the probability test, the probability value should be between 0.99 and 0.0094392. To check the randomness of the data, the minimum criterion is 100 binary sequences for 1,000,000 bits. Thus, we use 100 binary sequences of length 73,465,856 bits for each sequence evaluated and discussed. The proposed key generation algorithm results are transformed into binary because the NIST statistical test suite can consider the data only in binary form.

Secondly, according to NIST [32], the test for randomness is to be performed in two ways: first, as a test of the proportion of sequences passing a statistical test; and, second, as a test of the distribution of the *p*-values. In addition, according to the standardization of the NIST test, if the proportion value is equal to 96 sequences out of 100 passing sequences, the test will be considered a pass, otherwise as a fail. The proposed design result is presented in Table 6. The minimum pass rate for the Random Excursions (Variant) test is approximately 50 for a sample of 100 binary sequences. The minimum pass rate for the random excursion is approximately 51 for a sample of 100 binary sequences. We can see in Table 6 that the number of sequences that have managed to pass each test is greater than the minimum rate. Therefore, the proposed Omega network-based pseudorandom key generation algorithm passed all NIST statistical tests.

Table 6. NIST test results.

Test Name	Probability Test (*p*-Value)	Proportion	Result
Frequency	0.699313	98/100	Pass
Block-frequency	0.971699	99/100	Pass
Cumulative SUM (1)	0.759756	97/100	Pass
Cumulative SUM (2)	0.851383	98/100	Pass
Runs	0.534146	100/100	Pass
Longest runs	0.554420	99/100	Pass
Rank	0.275709	99/100	Pass
FFT	0.924076	100/100	Pass
Non-overlapping	0.262249	98/100	Pass
Overlapping	0.401199	100/100	Pass
Universal	0.334538	99/100	Pass
Approximate Entropy	0.494392	99/100	Pass
Random Extrusions	0.946308	51/51	Pass
Random Extrusions Variant	0.383827	50/51	Pass
Serial (1)	0.383827	98/100	Pass
Serial (2)	0.867692	98/100	Pass
Linear Complexity	0.935716	99/100	Pass

6. Comparison

This section compares the proposed Omega network-based pseudorandom DNA key generation algorithm with other DNA-based systems. The performance is evaluated in term of randomness, as shown in Table 7. In the comparison stage, the symbols "√" and "x" are used to denote whether the proposed key generation algorithm and the other related DNA-based system satisfied the respective randomness or not. Table 7 shows that the DNA-based system in [36] passes 13 NIST randomness tests, while the DNA-based system in [6] passes 3 NIST random tests, the system in [15] passes only 2 NIST tests, and the system in [9] passes 15 NIST tests. By contrast, the proposed algorithm passes all the recommended NIST randomness tests.

Table 7. Comparison of the proposed system with existing DNA-based systems in terms of randomness.

Test Name	[36]	[15]	[9]	Proposed
Frequency	√	√	√	√
Block-frequency	√	√	√	√
Cumulative SUM (1)	√	x	√	√
Cumulative SUM (2)	√	x	√	√
Runs	√	x	√	√
Longest runs	√	x	√	√
Rank	√	x	√	√
FFT	√	x	√	√
Non-overlapping	√	x	√	√
Overlapping	√	x	√	√
Universal	x	x	√	√
Approximate Entropy	x	x	√	√
Random Extrusions	x	x	x	√
Random Extrusions Variant	x	x	x	√
Serial (1)	√	x	√	√
Serial (2)	√	x	√	√
Linear Complexity	√	x	√	√

6.1. Computational Time Complexity Analysis

Computational time complexity is a way to judge how well an algorithm works by showing how quickly it gives results. It tells how many internal machine instructions a computer needs to carry out to finish an algorithmic task. The algorithm is better if it takes less time to run. This metric is utilized by a variety of researchers in order to represent the time complexity of their methods. Regarding the complexity of the proposed key generation algorithm, let us assume that a is the time required for an elementary operation (e.g., XORing two bits or generating a random bit). In the proposed algorithm, we have the number of input columns n = 4 (number of columns or functions at each row), zygote generation takes 128 XOR operations = $128 \times a$, and the random generation of a, b, and c each takes 128 random numbers = $128 \times 3 \times a$.

For each row input, we have n = 4 strings to be XORed together (with lengths of 128 bits), which gives $4 \times 128 \times a = 128 \times n \times a$. Similarly, for each row output, we have four strings to be passed to the box functions. In this case, we have $4 \times 128 \times a = 128 \times n \times a$. The key generation involves two XOR strings and one concatenation of 256 bits. So, we have $128 \times a + 128 \times a + 256 \times a = 128 \times 2 \times a + 128 \times 2 \times a = 128 \times n \times a$. Thus, the total time complexity of the proposed key generation algorithm is T(n) = $128 \times 4 \times a$ (zygote, a, b, c) + $128 \times a \times 4 \times 3$ (row inputs) + $128 \times a \times 4 \times 3$ (row outputs) = $128 \times 4 \times 7 \times a$ = $128 \times 7 \times n \times a \in (O(n))$.

6.2. Space Complexity Analysis

Space complexity measures how much space an algorithm needs to run based on input length. Initially, the proposed key generation algorithm uses n = 4 strings, which are the zygote, a, b, and c. The length of each string is 4×128 bits. After performing the

XORing operation, the length of the strings is 8 × 128 bits. If we free the space of the first four strings (zygote, *a*, *b*, *c*), then the resultant space should be S = 4 × 128. Each row of the proposed key generation algorithm contains four inputs and generates four outputs, which gives 8 × 128 bits. During the execution of the proposed key generation algorithm, four output strings are generated, and each time we can free the space of the input strings. In the end, we have two strings of 2 × 128 bits to be XORed and one resulting string key of 256 bits. The total is = 4 × 128 bits. Therefore, the complexity is the maximum space needed for all the steps, which is S_max = 8 × 128 bits (256 × n) ∈ (O(n)).

6.3. Attack Analysis

In this paper, the central dogma of molecular biology is applied to generate a pseudorandom key that provides a large key space of 256 bits. Thus, it is resistant to brute force attack. Secondly, a small change in the key value will change the resultant output of the key. It will be very difficult for the attacker to guess the key because of a one-bit change in the input. Thirdly, the proposed key generation algorithm consists of a combination of substitution and permutation operations which are based on the DNA XOR operation, mRNA, and complementary rules. The attacker cannot guess the mRNA codon and the four DNA XORing rules. Fourthly, the output of the proposed key generation algorithm is pseudorandom. If it passes the pseudorandom NIST test, meaning that the output is pseudorandom, the output is given to the attacker. The attacker will be trying to guess whether it is a pseudorandom key or a key generator that is outputted by our proposed algorithm. Therefore, in terms of pseudorandomness, it is very hard for the attacker to guess the key and break the proposed algorithm.

7. Conclusions

The proposed algorithm uses DNA properties, such as replication, complementary rules, and mRNA rules. Logical and biological operations are performed in the algorithm. The evaluation of the proposed algorithm has been carried out using the NIST test suite. One-hundred binary sequences of 73,465,856 bits were tested. The results showed that our proposed Omega network-based pseudorandom key generation algorithm is well fitted to achieve the NIST test security properties and pass all the NIST recommended tests. The time and space complexity of the proposed key generation algorithm were analyzed, (O(n)), respectively. Future work, firstly, might consist of analyzing and comparing the performances of all the DNA cryptographic techniques based on secure data transmission and computation processes. Secondly, this design can encode and decode information for all types of information and applications. Thirdly, more biological operations and computation techniques can be used to improve the proposed Omega network design.

Author Contributions: Conceptualization, G.R. and C.C.W.; methodology, G.R. and C.C.W.; software, G.R.; validation, G.R. and C.C.W.; formal analysis, G.R. and C.C.W.; investigation G.R.; resources, G.R.; data curation, G.R.; writing—original draft preparation, G.R.; writing—review and editing, G.R.; visualization, G.R.; supervision, C.C.W.; project administration, C.C.W.; funding acquisition, C.C.W. 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: This study did not report any data.

Acknowledgments: The authors would like to thank the Ministry of Higher Education of Malaysia for supporting this research under the Fundamental Research Grant Scheme, Vot No. FRGS/1/2020/ICT03/UTHM/03/5, and the Universiti Tun Hussein Onn Malaysia for partial sponsorship.

Conflicts of Interest: The authors declare that they have no conflict of interest.

Appendix A

```
#include <iostream>
#include <string>
#include <bits/stdc++.h>
#include <fstream>
 #include <time.h>
using namespace std;
string mRNA(string a,int n){
  string ans = "";
  for(int i=0;i<n;i+=4){
    if(a[i]=='0' && a[i+1]=='0' && a[i+2]=='0' && a[i+3]=='0'){
      ans+="1110";
    }
    else if(a[i]=='0' && a[i+1]=='0' && a[i+2]=='0' && a[i+3]=='1'){
      ans+="1100";
    }
    else if(a[i]=='0' && a[i+1]=='0' && a[i+2]=='1' && a[i+3]=='0'){
      ans+="0010";
    }
    else if(a[i]=='0' && a[i+1]=='0' && a[i+2]=='1' && a[i+3]=='1'){
      ans+="0101";
    }
    else if(a[i]=='0' && a[i+1]=='1' && a[i+2]=='0' && a[i+3]=='0'){
      ans+="1101";
    }
    else if(a[i]=='0' && a[i+1]=='1' && a[i+2]=='0' && a[i+3]=='1'){
      ans+="0001";
    }
    else if(a[i]=='0' && a[i+1]=='1' && a[i+2]=='1' && a[i+3]=='0'){
      ans+="0100";
    }
    else if(a[i]=='0' && a[i+1]=='1' && a[i+2]=='1' && a[i+3]=='1'){
      ans+="1000";
    }
    else if(a[i]=='1' && a[i+1]=='0' && a[i+2]=='0' && a[i+3]=='0'){
      ans+="0000";
    }
    else if(a[i]=='1' && a[i+1]=='0' && a[i+2]=='0' && a[i+3]=='1'){
      ans+="1001";
    }
    else if(a[i]=='1' && a[i+1]=='0' && a[i+2]=='1' && a[i+3]=='0'){
      ans+="1111";
    }
    else if(a[i]=='1' && a[i+1]=='0' && a[i+2]=='1' && a[i+3]=='1'){
      ans+="1010";
    }
    else if(a[i]=='1' && a[i+1]=='1' && a[i+2]=='0' && a[i+3]=='0'){
      ans+="0110";
    }
    else if(a[i]=='1' && a[i+1]=='1' && a[i+2]=='0' && a[i+3]=='1'){
      ans+="0111";
    }
    else if(a[i]=='1' && a[i+1]=='1' && a[i+2]=='1' && a[i+3]=='0'){
      ans+="0011";
    }
    else if(a[i]=='1' && a[i+1]=='1' && a[i+2]=='1' && a[i+3]=='1'){
      ans+="1011";
    }

  }
  return ans;
```

Figure A1. mRNA first letter and second letter.

```
string mRNA_REV(string a,int n){
  string ans = "";
  for(int i=0;i<n;i+=4){
    if(a[i+2]=='0' && a[i+3]=='0' && a[i]=='0' && a[i+1]=='0'){
      ans+="1110";
    }
    else if(a[i+2]=='0' && a[i+3]=='0' && a[i]=='0' && a[i+1]=='1'){
      ans+="1100";
    }
    else if(a[i+2]=='0' && a[i+3]=='0' && a[i]=='1' && a[i+1]=='0'){
      ans+="0010";
    }
    else if(a[i+2]=='0' && a[i+3]=='0' && a[i]=='1' && a[i+1]=='1'){
      ans+="0101";
    }
    else if(a[i+2]=='0' && a[i+3]=='1' && a[i]=='0' && a[i+1]=='0'){
      ans+="1101";
    }
    else if(a[i+2]=='0' && a[i+3]=='1' && a[i]=='0' && a[i+1]=='1'){
      ans+="0001";
    }
    else if(a[i+2]=='0' && a[i+3]=='1' && a[i]=='1' && a[i+1]=='0'){
      ans+="0100";
    }
    else if(a[i+2]=='0' && a[i+3]=='1' && a[i]=='1' && a[i+1]=='1'){
      ans+="1000";
    }
    else if(a[i+2]=='1' && a[i+3]=='0' && a[i]=='0' && a[i+1]=='0'){
      ans+="0000";
    }
    else if(a[i+2]=='1' && a[i+3]=='0' && a[i]=='0' && a[i+1]=='1'){
      ans+="1001";
    }
    else if(a[i+2]=='1' && a[i+3]=='0' && a[i]=='1' && a[i+1]=='0'){
      ans+="1111";
    }
    else if(a[i+2]=='1' && a[i+3]=='0' && a[i]=='1' && a[i+1]=='1'){
      ans+="1010";
    }
    else if(a[i+2]=='1' && a[i+3]=='1' && a[i]=='0' && a[i+1]=='0'){
      ans+="0110";
    }
    else if(a[i+2]=='1' && a[i+3]=='1' && a[i]=='0' && a[i+1]=='1'){
      ans+="0111";
    }
    else if(a[i+2]=='1' && a[i+3]=='1' && a[i]=='1' && a[i+1]=='0'){
      ans+="0011";
    }
    else if(a[i+2]=='1' && a[i+3]=='1' && a[i]=='1' && a[i+1]=='1'){
      ans+="1011";
    }

  }
  return ans;
```

Figure A2. mRNA second letter and first letter.

```
string complementry_rules(string a,int n){
  string ans = "";
  for(int i=0;i<n;i+=4){
    if(a[i]=='0' && a[i+1]=='0' && a[i+2]=='0' && a[i+3]=='0'){
      ans+="0011";
    }
    else if(a[i]=='0' && a[i+1]=='0' && a[i+2]=='0' && a[i+3]=='1'){
      ans+="0010";
    }
    else if(a[i]=='0' && a[i+1]=='0' && a[i+2]=='1' && a[i+3]=='0'){
      ans+="0001";
    }
    else if(a[i]=='0' && a[i+1]=='0' && a[i+2]=='1' && a[i+3]=='1'){
      ans+="0000";
    }
    else if(a[i]=='0' && a[i+1]=='1' && a[i+2]=='0' && a[i+3]=='0'){
      ans+="0000";
    }
    else if(a[i]=='0' && a[i+1]=='1' && a[i+2]=='0' && a[i+3]=='1'){
      ans+="0101";
    }
    else if(a[i]=='0' && a[i+1]=='1' && a[i+2]=='1' && a[i+3]=='0'){
      ans+="0110";
    }
    else if(a[i]=='0' && a[i+1]=='1' && a[i+2]=='1' && a[i+3]=='1'){
      ans+="1000";
    }
    else if(a[i]=='1' && a[i+1]=='0' && a[i+2]=='0' && a[i+3]=='0'){
      ans+="0111";
    }
    else if(a[i]=='1' && a[i+1]=='0' && a[i+2]=='0' && a[i+3]=='1'){
      ans+="1010";
    }
    else if(a[i]=='1' && a[i+1]=='0' && a[i+2]=='1' && a[i+3]=='0'){
      ans+="1001";
    }
   else if(a[i]=='1' && a[i+1]=='0' && a[i+2]=='1' && a[i+3]=='1'){
     ans+="1110";
   }
   else if(a[i]=='1' && a[i+1]=='1' && a[i+2]=='0' && a[i+3]=='0'){
     ans+="1101";
   }
   else if(a[i]=='1' && a[i+1]=='1' && a[i+2]=='0' && a[i+3]=='1'){
     ans+="1100";
   }
   else if(a[i]=='1' && a[i+1]=='1' && a[i+2]=='1' && a[i+3]=='0'){
     ans+="1011";
   }
   else if(a[i]=='1' && a[i+1]=='1' && a[i+2]=='1' && a[i+3]=='1'){
     ans+="1111";
   }

 }
 return ans;
```

Figure A3. Complementry mapping rules.

```
string DNA_xoring(string a,string b,int n){
  string ans = "";
  if(a[0]=='0' && a[1]=='0'){
    for(int i=0;i<n;i+=2){
      if(a[i]=='0' && a[i+1]=='0'){
        if(b[i]=='0' && b[i+1]=='0'){
          ans+="01";
        }
        if(b[i]=='0' && b[i+1]=='1'){
          ans+="10";
        }
        if(b[i]=='1' && b[i+1]=='0'){
          ans+="00";
        }
        if(b[i]=='1' && b[i+1]=='1'){
          ans+="11";
        }
      }
      if(a[i]=='0' && a[i+1]=='1'){
        if(b[i]=='0' && b[i+1]=='0'){
          ans+="10";
        }
        if(b[i]=='0' && b[i+1]=='1'){
          ans+="01";
        }
        if(b[i]=='1' && b[i+1]=='0'){
          ans+="11";
        }
        if(b[i]=='1' && b[i+1]=='1'){
          ans+="00";
        }
      }
      if(a[i]=='1' && a[i+1]=='0'){
        if(b[i]=='0' && b[i+1]=='0'){
          ans+="00";
        }
        if(b[i]=='0' && b[i+1]=='1'){
          ans+="11";
        }
        if(b[i]=='1' && b[i+1]=='0'){
          ans+="10";
        }
        if(b[i]=='1' && b[i+1]=='1'){
          ans+="01";
        }
      }
      if(a[i]=='1' && a[i+1]=='1'){
        if(b[i]=='0' && b[i+1]=='0'){
          ans+="11";
        }
        if(b[i]=='0' && b[i+1]=='1'){
          ans+="00";
        }
        if(b[i]=='1' && b[i+1]=='0'){
          ans+="01";
        }
        if(b[i]=='1' && b[i+1]=='1'){
          ans+="10";
        }
      }
    }
  }
  else if(a[0]=='0' && a[1]=='1'){
    for(int i=0;i<n;i+=2){
      if(a[i]=='0' && a[i+1]=='0'){
        if(b[i]=='0' && b[i+1]=='0'){
          ans+="10";
        }
        if(b[i]=='0' && b[i+1]=='1'){
          ans+="11";
        }
        if(b[i]=='1' && b[i+1]=='0'){
          ans+="01";
        }
        if(b[i]=='1' && b[i+1]=='1'){
          ans+="00";
        }
      }
      if(a[i]=='0' && a[i+1]=='1'){
        if(b[i]=='0' && b[i+1]=='0'){
          ans+="11";
        }
        if(b[i]=='0' && b[i+1]=='1'){
          ans+="10";
        }
        if(b[i]=='1' && b[i+1]=='0'){
          ans+="00";
        }
        if(b[i]=='1' && b[i+1]=='1'){
          ans+="01";
        }
      }
      if(a[i]=='1' && a[i+1]=='0'){
        if(b[i]=='0' && b[i+1]=='0'){
          ans+="01";
        }
        if(b[i]=='0' && b[i+1]=='1'){
          ans+="00";
        }
        if(b[i]=='1' && b[i+1]=='0'){
          ans+="11";
        }
        if(b[i]=='1' && b[i+1]=='1'){
          ans+="10";
        }
      }
      if(a[i]=='1' && a[i+1]=='1'){
        if(b[i]=='0' && b[i+1]=='0'){
          ans+="00";
        }
        if(b[i]=='0' && b[i+1]=='1'){
          ans+="01";
        }
        if(b[i]=='1' && b[i+1]=='0'){
          ans+="10";
        }
        if(b[i]=='1' && b[i+1]=='1'){
          ans+="11";
        }
      }
    }
  }
  }
  else if(a[0]=='1' && a[1]=='0'){
    for(int i=0;i<n;i+=2){
      if(a[i]=='0' && a[i+1]=='0'){
        if(b[i]=='0' && b[i+1]=='0'){
          ans+="00";
        }
        if(b[i]=='0' && b[i+1]=='1'){
          ans+="01";
        }
        if(b[i]=='1' && b[i+1]=='0'){
          ans+="10";
        }
        if(b[i]=='1' && b[i+1]=='1'){
          ans+="11";
        }
      }
      if(a[i]=='0' && a[i+1]=='1'){
        if(b[i]=='0' && b[i+1]=='0'){
          ans+="01";
        }
        if(b[i]=='0' && b[i+1]=='1'){
          ans+="00";
        }
        if(b[i]=='1' && b[i+1]=='0'){
          ans+="11";
        }
        if(b[i]=='1' && b[i+1]=='1'){
          ans+="10";
        }
      }
      if(a[i]=='1' && a[i+1]=='0'){
        if(b[i]=='0' && b[i+1]=='0'){
          ans+="10";
        }
        if(b[i]=='0' && b[i+1]=='1'){
          ans+="11";
        }
        if(b[i]=='1' && b[i+1]=='0'){
          ans+="01";
        }
        if(b[i]=='1' && b[i+1]=='1'){
          ans+="00";
        }
      }
      if(a[i]=='1' && a[i+1]=='1'){
        if(b[i]=='0' && b[i+1]=='0'){
          ans+="11";
        }
        if(b[i]=='0' && b[i+1]=='1'){
          ans+="10";
        }
        if(b[i]=='1' && b[i+1]=='0'){
          ans+="00";
        }
        if(b[i]=='1' && b[i+1]=='1'){
          ans+="01";
        }
      }
    }
  }
  else if(a[0]=='1' && a[1]=='1'){
    for(int i=0;i<n;i+=2){
      if(a[i]=='0' && a[i+1]=='0'){
        if(b[i]=='0' && b[i+1]=='0'){
          ans+="01";
        }
        if(b[i]=='0' && b[i+1]=='1'){
          ans+="00";
        }
        if(b[i]=='1' && b[i+1]=='0'){
          ans+="11";
        }
        if(b[i]=='1' && b[i+1]=='1'){
          ans+="10";
        }
      }
      if(a[i]=='0' && a[i+1]=='1'){
        if(b[i]=='0' && b[i+1]=='0'){
          ans+="00";
        }
        if(b[i]=='0' && b[i+1]=='1'){
          ans+="01";
        }
        if(b[i]=='1' && b[i+1]=='0'){
          ans+="10";
        }
        if(b[i]=='1' && b[i+1]=='1'){
          ans+="11";
        }
      }
      if(a[i]=='1' && a[i+1]=='0'){
        if(b[i]=='0' && b[i+1]=='0'){
          ans+="11";
        }
        if(b[i]=='0' && b[i+1]=='1'){
          ans+="10";
        }
        if(b[i]=='1' && b[i+1]=='0'){
          ans+="00";
        }
        if(b[i]=='1' && b[i+1]=='1'){
          ans+="01";
        }
      }
      if(a[i]=='1' && a[i+1]=='1'){
        if(b[i]=='0' && b[i+1]=='0'){
          ans+="10";
        }
        if(b[i]=='0' && b[i+1]=='1'){
          ans+="11";
        }
        if(b[i]=='1' && b[i+1]=='0'){
          ans+="01";
        }
        if(b[i]=='1' && b[i+1]=='1'){
          ans+="00";
        }
      }
    }
  }
 return ans;
```

Figure A4. DNA XOR rules.

```
int findRandom()
{ int pi=22/7;
  // Generate the random number
  int num = ((int)rand() % (pi-1));
  // Return the generated number
  return num;
}

// Function to generate a random
// binary string of length N
string non_cyclic_pi_rnd(int N)
{
  // Stores the empty string
  string S = "";

  // Iterate over the range [0, N - 1]
  for (int i = 0; i < N; i++) {

      // Store the random number
      int x = findRandom();

      // Append it to the string
      S += to_string(x);
  }

  // returning the resulting string
  return S;
}
string changing(string a,int n){
  if(n==0){
    return a;
  }
```

Figure A5. Function to generate random binary strings.

```
string changing(string a,int n){
  if(n==0){
    return a;
  }
  if(a[n-1]=='0'){
      a[n-1]='1';
  }
  else{
    a[n-1]='0';
  }
  return a;
}
```

Figure A6. Function of one-bit change.

```
string xoring(string a,string b,int n){
  string ans = "";

  // Loop to iterate over the
  // Binary Strings
  for (int i = 0; i < n; i++)
  {
      // If the Character matches
      if (a[i] == b[i])
          ans += "0";
      else
          ans += "1";
  }
  return ans;
}
```

Figure A7. Function of binary XORing.

References

1. Yung, P.Z.; Zhen, Z.W.; Zhi, W.W.; Yaseen, H.K.; Wei, D.D. A New DNA Cryptography Algorithm Based on the Biological Puzzle and DNA Chip Techniques. In Proceedings of the 2016 International Conference on Biomedical and Biological Engineering, Shanghai, China, 15–17 July 2016; pp. 360–365. [CrossRef]
2. Olga, T. Contributions to DNA cryptography: Applications to text and image secure transmission. Ph.D. Thesis, Université Nice Sophia Antipolis, Nice, France, Universitatea Tehnică Cluj-Napoca, Cluj-Napoca, Romania, 2013.
3. Anam, B.; Sakib, K.; Hosain, M.; Dhal, K. Review on the advancements of DNA cryptography. In Proceedings of the SKIMA-2010 of the 4th International Conference, Paro, Bhutan, 25–27 August 2010; pp. 177–184. [CrossRef]
4. Shurit, K.; Harleen, K.; Vactor, K. DNA Cryptography and Deep Learning using Genetic Algorithm with NW algorithm for Key Generation. *J. Med. Syst.* **2018**, *42*, 17. [CrossRef]
5. Vidhya, E.; Rathipriya, R. Key Generation for DNA Cryptography Using Genetic Operators and Diffie-Hellman Key Exchange Algorithm. *Int. J. Math. Comput. Sci.* **2020**, *15*, 1109–1115.
6. Sodhi, G.K.; Monga, H.; Gaba, G.S. DNA and LCG based security key generation algorithm. *Pertanika J. Sci. Technol.* **2017**, *25*, 1369–1380.
7. Majumdar, A.; Biswas, A.; Baishnab, K.L.; Sood, S.K. DNA based cloud storage security framework using fuzzy decision making technique. *KSII Trans. Internet Inf. Syst. (TIIS)* **2019**, *13*, 3794–3820. [CrossRef]
8. Tushar, M.; Vijay, C.A. DNA encryption technique based on matrix manipulation and secure key generation scheme. In Proceedings of the International Conference on Information Communication and Embedded Systems (ICICES), Chennai, India, 21–22 February 2013; pp. 47–52. [CrossRef]
9. Sally, S.N.; Mohmood, K.I. Cryptographic Algorithm Based on DNA and RNA Properties. *Int. J. Adv. Res. Comput. Eng. Technol. (IJARCET)* **2018**, *7*, 237–241.
10. Bin, L. BioSeq-Analysis: A platform for DNA, RNA and protein sequence analysis based on machine learning approaches. *Mol. Genet. Genom.* **2018**, *20*, 1280–1294. [CrossRef]
11. Sayantani, B.; Marimuthu, K.; Mita, N.; Anup, K.H.; Niranchana, R. Bio-inspired cryptosystem with DNA cryptography and neural networks. *J. Syst. Arch.* **2019**, *94*, 24–31. [CrossRef]
12. Suyel, N. Fast and Secure Data Accessing by using DNA Computing for the Cloud Environment. *IEEE Trans. Serv. Comput.* **2020**, *1374*, 1–12. [CrossRef]
13. Suyel, N.; Rupak, C.; Abhishek, M.; Nageswara, R.M. Securing Multimedia by Using DNA-Based Encryption in the Cloud Computing Environment. *ACM Trans. Multimed. Comput. Commun. Appl.* **2020**, *16*, 1–19. [CrossRef]
14. Dhivya, R.; Aashiq, S.; Murthy, B.K.; Vidhyadharini, B.; Sherin, F.; Rengarajan, A. An efficient medical image encryption using hybrid DNA computing and chaos in transform domain. *Med. Biol. Eng. Comput.* **2021**, *55*, 89–605. [CrossRef]
15. Dilovan, A.Z.; Habibollah, H.; Subhi, R.M.Z.; Diyar, Q.Z. Multi-Level of DNA Encryption Technique Based on DNA Arithmetic and Biological Operations. In Proceedings of the International Conference on Advanced Science and Engineering (ICOASE), Duhok, Iraq, 9–11 October 2018. [CrossRef]
16. Shakir, M.H.; Hussein, A.B. A DNA-Based Cryptographic Key Generation Algorithm. In Proceedings of the International Conference on Security and Management, Las Vegas, NV, USA, 25–28 July 2016; pp. 338–342.
17. Fursan, T.; Sharaf, T.; Sudhir, J. A new data security algorithm for the cloud computing based on genetics techniques and logical-mathematical functions. *Int. J. Intell. Netw.* **2021**, *2*, 18–33. [CrossRef]
18. Sreeja, C.S.; Misbahuddin, M.; Mohammed, N.P.H. DNA for information security: A Survey on DNA computing and a pseudo DNA method based on central dogma of molecular biology. In Proceedings of the International Conference on Computing and Communication Technologies, Hyderabad, India, 11–13 December 2014. [CrossRef]
19. Madhvi, P.; Gagandeep. DNA Cryptography: A Novel Approach for Data Security Using Flower Pollination Algorithm. In Proceedings of the International Conference on Sustainable Computing in Science, Technology and Management (SUSCOM), Amity University Rajasthan, Jaipur, India, 26–28 February 2019. [CrossRef]
20. Elmoselhy, A.; Elalfy, E.S.M. On DNA cryptography for secure data storage and transfer. In Proceedings of the IET Conference, Online, 21–23 September 2020. [CrossRef]
21. Raju, P.V.C.N.; Pritee, P. DNA encryption based dual server authentication. *Adv. Intell. Syst. Comput.* **2014**, *328*, 29–37. [CrossRef]
22. Pramod, P.; Sheena, M.; Suyel, N.; Pascal, L. A novel cryptosystem based on DNA cryptography and randomly generated mealy machine. *Comput. Secur.* **2021**, *104*, 102–160. [CrossRef]
23. Manreet, S.; Sandeep, S. BDNA-A DNA inspired symmetric key cryptographic technique to secure cloud computing. *J. King Saud Univ. Comput. Inf. Sci.* **2022**, *34*, 1417–1425. [CrossRef]
24. Suyel, N.; Sharma, S.; Ganesh, C.D.; Pascal, L. DNA computing and table based data accessing in the cloud environment. *J. Netw. Comput. Appl.* **2020**, *172*, 102835. [CrossRef]
25. Yunpeng, Z.; Xin, L.; Yongqiang, M.; Liang, C.C. An optimized DNA based encryption scheme with enforced secure key distribution. *Clust. Comput.* **2017**, *20*, 3119–3130. [CrossRef]
26. Reddy, M.I.; Kumar, A.P.S. An efficient data transmission approach using IAES-BE. *Clust. Comput.* **2020**, *23*, 1633–1645. [CrossRef]
27. Alaa, F.; Rasha, K.; Ali, R.S.; Hassan, R.Y.; Nadia, M.G.A.; Ghassan, H.A.M. A new approach to generate multi S-boxes based on RNA computing. *Int. J. Innov. Comput. Inf. Control* **2020**, *16*, 331–348.

28. Suresh, E.; Babu, C.; Naga, R.; Prasad, M.H.M.K. Inspired pseudo biotic DNA based cryptographic mechanism against adaptive cryptographic attacks. *Int. J. Netw. Secur.* **2016**, *18*, 291–303.
29. Miki, H.; Hiroki, K.; Zuhiro, O. Design of True Random One-Time Pads in DNA XOR Cryptosystem. In *Proceedings in Information and Communications Technology*; Springer: Tokyo, Japan, 2010; pp. 174–184. [CrossRef]
30. Emtious, M.S.H.; Kazi, M.R.A.; Yasihiko, M. A DNA cryptographic technique based on dynamic DNA sequence table. In Proceedings of the 19th International Conference on Computer and Information Technology (ICCIT) 2017, Dhaka, Bangladesh, 22–24 December 2017. [CrossRef]
31. Ilya, R.; Greg, H.; Jin, W. Using π digits to Generate Random Numbers: A Visual and Statistical Analysis. In Proceedings of the International Conference on Scientific Computing (CSC), Paphos, Cyprus, 3–6 December 2013.
32. Rukhin, A.; Soto, J.; Nechvatal, J.; Smid, M.; Barker, E. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*; Special Publication 800-22; National Institute of Standards and Technology (NIST): Gaithersburg, MD, USA, 2008.
33. Darshana, U.; Paryanka, S.; Sharada, V. SP 800-22 Rev. 1A: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Technical Report National Institute of Standards & Technology. 2010. Available online: https://dl.acm.org/doi/pdf/10.5555/2206233 (accessed on 19 June 2022).
34. Kordov, K.M. Modified Chebyshev map based pseudo-random bit generator. *AIP Conf. Proc.* **2014**, *1629*, 432. [CrossRef]
35. Auday, H.S.A.W.; Ramlan, M.; Zuraiti, A.Z.; Nur, I.U. Generating a new S-Box inspired by biological DNA. *Int. J. Comput. Sci. Appl.* **2015**, *4*, 32–42. [CrossRef]
36. Biswas, M.R.; Alam, K.M.R.; Tamura, S.; Morimoto, Y. A technique for DNA cryptography based on dynamic mechanisms. *J. Inf. Secur. Appl.* **2019**, *48*, 102363. [CrossRef]