Computer Science

COC251

B929107

Encryption Algorithm Implementation And Evaluation

by

Joshua A. R. Rowing

Supervisor: Dr A. M. Salagean

Department of Computer Science

Loughborough University

May/June 2022

1. Contents

2. Introduction

2.1. Project Plan

The plan is to take a cipher that is not currently in common use and, using the specification of the cipher, rebuild it from scratch in C. Once the cipher is constructed, I will test its security using cipher analysis.

2.2. Project Purpose

The purpose of this project is to better my own understanding of how ciphers are constructed and tested. This why I must construct the cipher from scratch using the specification, instead of looking at the source code. Hopefully, the process of constructing the cipher myself will give me a specific and intimate understanding of how the cipher is working, which would be much harder to obtain from simply looking at the highly optimised, unreadable, source-code.

2.3. Why C?

Encryption algorithms are traditionally coded using C or C++ as they produce and executable file and give the greatest control over the optimisation of the algorithm while still being a high-level language. As the C++ has not been covered in my degree experience so far, I chose to use C. Also, I know that if any C++ libraries are necessary, they can be included into my project file.

2.4. Why not implement a commonly used cipher?

Part of the project specification is to evaluate the and potentially find flaws or improvements in the cipher I select. Ciphers that are in common use today such as AES have already been analysed in hundreds of reports and have been proven to be secure. Therefore, I will select a cipher that does not have many corresponding analysis reports, with the hope that I may be able to discover some new ground on the security of the cipher.

2.5. Selecting a cipher

To find a cipher, I looked at the entrants for the recent CAESAR cipher competition. I chose “pi-cipher”, coded by a group of 7 computer scientists from Norwegian Universities [1]. I chose this entrant as it has a clear, well-written documentation that makes it easy to understand and implement. This includes pseudocode for the “pi algorithms” and diagrams to help visualise the cipher’s method. Also, the cipher only has one report analysing its safety, which is the proof of tag second-preimage attack on the cipher [2].

2.6. Evaluating the cipher

Once I have implemented and tested the cipher, I will use both the analysis specified in the documentation and my own analysis to try to break the cipher. I will record the record the results and use it to measure how effective the cipher is. The less information that can be gained about an encrypted message, the better the cipher.

2.6.1. Adjusting the cipher

Once I have analysed the cipher described in the specification, I will make small adjustments to the cipher algorithm and record how this affects its resistance to attacks.

If similar methods perform much worse in my tests, it will demonstrate that pi-cipher is very finely tuned, but if I get similar results with a similar altered method it will show that the cipher can be altered without affecting security of the cipher drastically. If I find a different method that performed more effectively in my tests than pi-cipher, this will show that the cipher has room for some improvements.

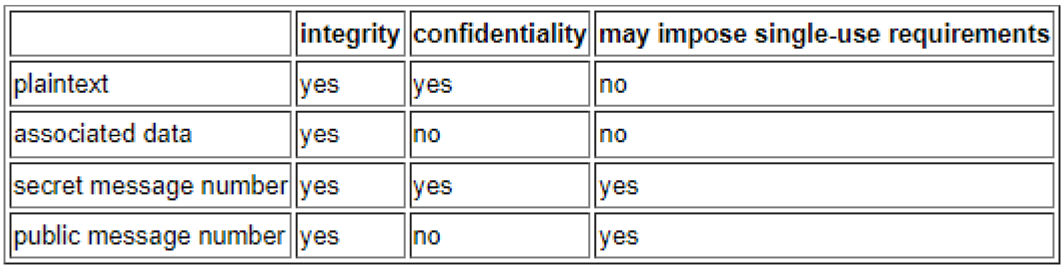
3. Literature Review

3.1. CAESAR Competition

The CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) competition was co-founded by NIST (National Institute of Standards and Technology) and Dan Bernstein with the aim of finding authenticated encryption schemes that offer advantages over AES-GCM and are suitable for widespread adoption [3]. The first round started in 2014 with 57 candidates, in the first round nine candidates' ciphers were broken and excluded from the competition, leaving 48 to compete over the next two rounds of intense analysis and comparison. The final portfolio of 6 ciphers (2 for each of the 3 use cases) was announced in February of 2019. Pi-cipher was a candidate in round two, but did not progress into round three, meaning it was unable to be broken, but when compared to the other ciphers in the competition, it contained some disadvantages. For this project, therefore I should not expect to break the cipher, however through analysis I will aim to highlight what disadvantages the cipher had.

3.1.1. Authenticated Encryption

The CAESAR requirements for confidentiality and integrity were laid out in the table below.



	integrity	confidentiality	may impose single-use requirements
plaintext	yes	yes	no
associated data	yes	no	no
secret message number	yes	yes	yes
public message number	yes	no	yes

Figure 1 Functional Requirements for submissions to CAESAR [4]

The requirements for all encrypted and associated data require data integrity, and the plaintext and secret message number also require confidentiality. These are the requirements for Authenticated encryption.6

Confidentiality in cyber security means that the data must be restricted in terms of who is allowed to view the data in a readily accessible way. [5]

Data integrity means protection from corruption or unauthorised modification, that is, it provides a tamper-proof environment for the data to be transmitted. [5]

Confidentiality and Data integrity are common principles required for any data transmitted over an insecure channel, many solutions have been suggested, but authenticated encryption is used most as it provides simple solutions for both problems. Confidentiality is provided through the encryption of the data, as it is very difficult and computationally expensive to decode without the secret key. Even with knowledge of the method and access to the cipher used, it is often very difficult to find the original message with any method faster than a brute-force key search. Data integrity can be verified by Authenticated Encryption with a MAC (Message Authentication Code), which proves the data has not been tampered with during transmission. To ensure this, at both ends a MAC is generated using some MAC algorithm. The MAC algorithm must take the encrypted data and the secret key as inputs and produce an output of length n-bits to be transmitted with the data. The idea is that only the exact data transmitted will produce the right MAC, and if any of the data is changed in transit, the computer will compare the MACs and it will know that the data has been tampered during transmission. As MACs produce a fixed length output, there is not a guarantee that two different messages won't produce the same MAC. However, MACs are usually 256 or 512 bits long, which means that, without a specific method for creating two messages with the same MAC, the chances of it occurring randomly are infinitesimally small [5].

The CAESAR competition required authenticated encryption for all its submissions as the aim of the competition was to find ciphers with advantages over AES-GCM. AES-GCM is a lightweight cipher currently in widespread use that offers authenticated encryption. The Algorithm also offers parallelisation for faster compute times and only 3% overhead (assuming a GCM packet contains 128 data blocks and one AAD (Additional Associated Data) block).This means that in addition to the message itself, The algorithm only needs to send 3% additional data to ensure its claims of data integrity and confidentiality, making it very fast and lightweight in practical use [6]. Any cipher that attempts to gain widespread adoption without a guarantee of data integrity would not be able to compete with AES-GCM no matter what advantages it has in confidentiality or compute time, as it does not fit the same use case.

3.1.2. MAC Algorithm in pi-cipher

MAC algorithms usually use a cryptographic hash function (HMAC) or cipher (CMAC) to create the MAC. In pi-cipher the pi-function is used on each block to produce an individual tag, then they are combined using the addition modulo operator $\boxplus_d$, where each tag (MAC) is a d-dimensional vector of ω-bit words and the modulus is set to the maximum value of an unsigned ω-bit integer. The values of d and ω are decided by which version of pi-cipher you choose to use or implement. The creation of the final tag is illustrated in the specification by the diagram below.

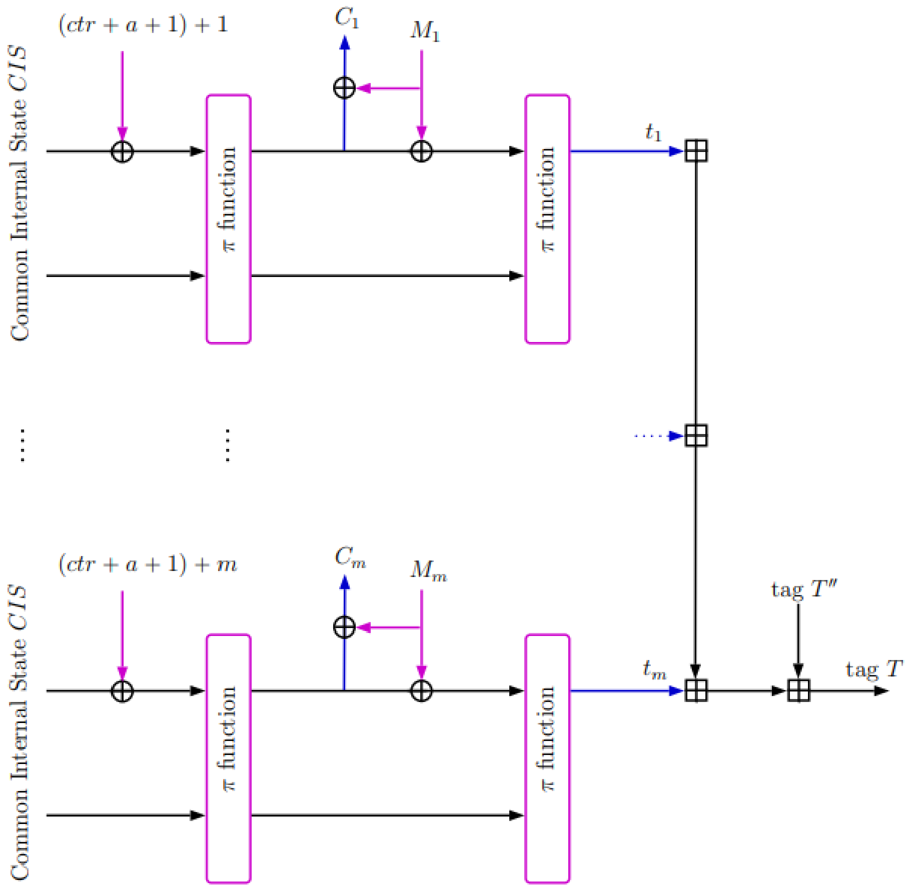


Figure 2 Processing message M with m blocks in parallel [1]

Each block of M can be processed in parallel, with each block producing an individual tag $t_1, \dots, t_m$ for each block. These tags are then combined using $\boxplus$, then T'' is added, which is obtained through the same process on the SMN (Secret Message Number) and T', (which is obtained through the same process on the Associated Data). In this way, each part of encrypted and associated data is encapsulated in a single d-dimensional vector that uniquely identifies this combination of information.

To illustrate the entire tag creation process in one image, I drew a simplified diagram as shown below.

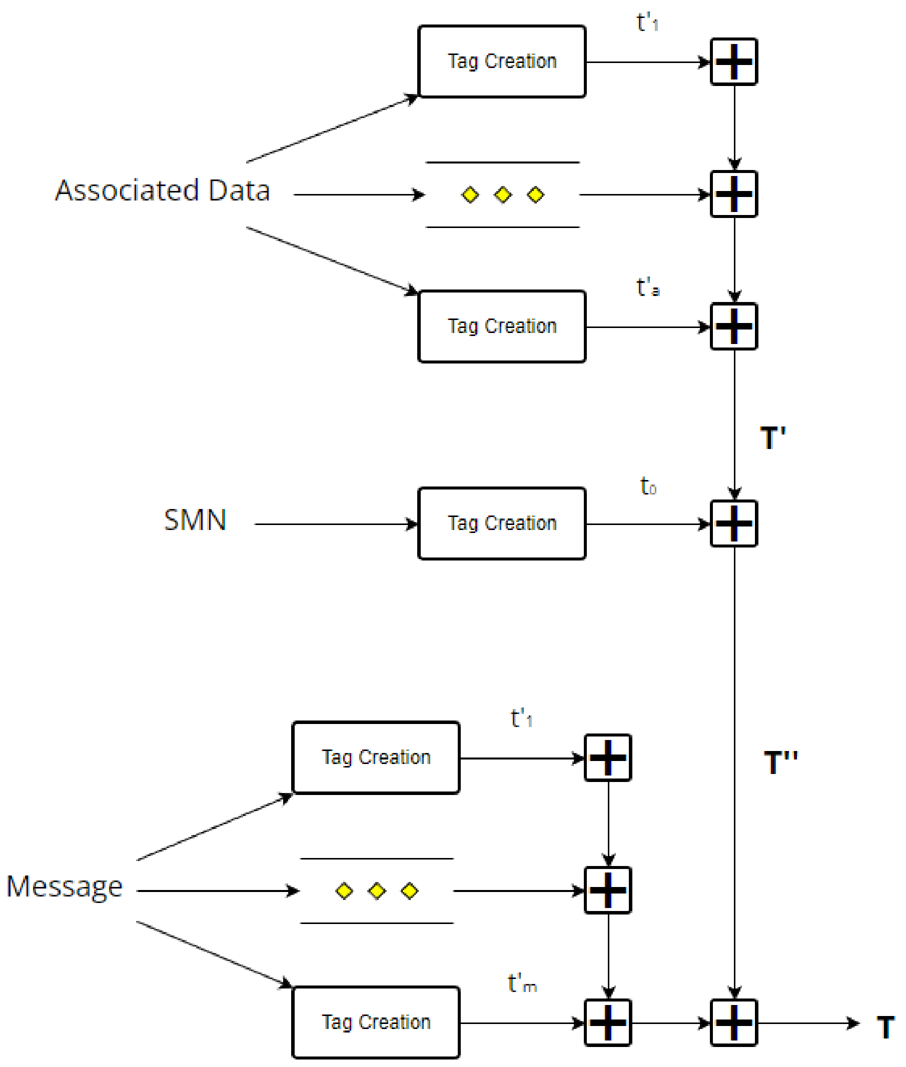


Figure 3 Tag generation in pi-cipher

It is important to note that the computation for Associated Data, SMN, and Message tags cannot be parallelised, but the data blocks within Associated Data and Message can.

3.2. Diffusion

In his 1945 Report “A Mathematical Theory of Cryptography” Claude Shannon wrote detailing the properties that make an effective cipher, and methods for measuring its security [7].

3.2.1. Definition

The method of diffusion is where the statistical structure is “dissipated” into long range statistics - i.e. into a statistical structure involving long combinations of letters in the cryptogram [7]. The effect of this is that anyone who may intercept the encrypted message will have to intercept large amounts of the message, if not all of it for it to be possible to gain any information about the original unencrypted message. This is means that it is harder or impossible for the interceptor to gain information with an incomplete ciphertext, which is even more important now than when Shannon wrote this paper. This is as communication most likely would have been sent along a single line connecting the sender and receiver, and with the structure of the internet we now have, information is broken up into small packets before being sent and they may not be sent along the same route, as they are sent according to the fastest route at the time of sending each individual packet. The additional positive effect of diffusion is that when the interceptor has sufficient material, the analytical work required is much greater since the redundancy has been diffused over many individual statistics [7]. This means that since the small amount of original information has been diffused into a large dataset, the probabilities for each combination decrease, meaning the computational power needed to decode the ciphertext will be much greater.

3.2.2. Analysis

What this means for our cipher is that we must measure the level of diffusion in the cipher to judge how difficult it would be for an analyst to find some kind of distinguishing pattern from a random permutation and use that pattern to gain some information about the original text. In this paper, Shannon defines what makes an effective statistic for analysing a cryptographic function. He states that it must be simple to measure, depend more on the key than the message, have values that can be resolved despite the “fuzziness” produced by variations in the input message, and the information gained must be simple and usable [7]. These are good principles to follow when looking at cryptanalytic methods, and I will follow them when deciding which methods to use in my paper. One method that Shannon suggests as a good example that follows these principles is “frequency count for simple substitution”. In the context of pi-cipher, that would include creating a large amount of test messages, and then making a small substitution (either a single bit or a small part of the input message) and measuring the differences that occur in the outputs of some function in the cipher. The differences would be between the original output and the output of the modified data. I could then map out how frequently each bit in the output changes, given the same difference between the inputs. This information could then be displayed in a graph to show the “shape” of the function. As I would like to test the functions with varying datasets, I will change the “frequency” metric to a “probability of changing” metric, this way the analytical values will be of the same scale no matter what the input dataset size is, making it more intuitive to understand whether a certain part of the output always changes or has a 50% chance, etc.

3.3. The Avalanche Effect

In the documentation for pi-cipher, the cryptanalysis they performed was a demonstration of the avalanche effect on their encryption algorithms * and pi. The avalanche effect is the property of a encryption algorithm, where a small change in the input of a cryptographic function produces a significant change in its output. Specifically, it means that if a single bit of the input is flipped, then each bit of the output is flipped with a probability of 0.5, on average. Half the bits being flipped on average is the goal as it represents the maximum entropy that can occur for a single bit change, as if any more bits or any less bits were flipped, then the pattern would be more predictable (for example if each bit has a 30% chance of changing, then it has a 70% chance of not changing and vice versa).

In my Analysis section, I would like to recreate the results of their analysis, verifying the results with my own implementation of the cipher. However, I would also like to make additions to their analysis, by testing over a larger sample size, and testing over multiple rounds of pi. Therefore, for those reasons I will include analysis of the avalanche effect in pi cipher in my analysis section.

3.4. More Cryptanalytic Methods and the Xi Framework

In the 2006 report "A Framework for Describing Block Cipher Cryptanalysis", Raphael Phan and Mohammed Siddiqi create a generalised framework for describing cryptanalytic attacks (Xi Framework) and provides examples of how this framework can be applied to modern cipher analyses and attacks.

3.4.1. The Xi Framework

A general cryptanalytic attack is split up into three main phases: Distinguisher search, Text collection, Key-recovery.

Distinguisher Search

The distinguisher is a property of the encryption method that is definably different from a truly random sequence produced by a random oracle [8]. A random oracle is an idea in mathematics and cryptography of a function that produces a truly random output, which is used to compare to a cryptographic function. A cryptographic function will always be different from a random oracle as if the output of a cryptographic function was truly random, then it would be impossible to decrypt it, even with the key. Therefore, we know every cipher must be in some way distinct from a truly random oracle however the difficulty comes in finding and defining that distinguishing feature. If a distinguisher is found, then it must also be proved to be useful, by proving it occurs with a non-random probability p^*, which must be higher than the probability that the same correlation would occur in a truly random sequence p. If the difference $|p^* - p|$ is greater than some negligible value ε (decided by the attacker), then the distinguisher is classified as useful [9]. This proves that the distinguisher can be used to reduce the computing power necessary to find a key by some non-negligible amount when compared to brute force key search. The successful discovery of a distinguisher leads very naturally to a successful key-recovery attack, i.e., finding and defining a useful distinguisher is the most difficult part of the attack.

Text Collection

The text collection phase is when the attacker builds up a set of plaintexts and ciphertexts significant enough to launch an attack. This phase assumes that the attacker has access to the black box oracle that applies the encryption algorithm, which is a standard assumption in cryptanalysis [9]. The data that the attacker must collect, and how flexible they can be, is defined by the type of attack that is being launched.

- If the attacker needs to choose specific inputs to the encryption oracle to measure the corresponding outputs, then it is a **Chosen-Plaintext Attack**
- If the attacker only needs to know some plaintexts correspond to some ciphertexts, then it is a **Known-Plaintext Attack**

The purpose of classifying the attacks in this way is to make clear how much information an attacker needs to launch an attack: the less information the attacker needs to launch a successful attack, the more damaging it is for the security of the cipher being tested. Chosen-plaintext attacks are generally considered to be more restricted, as the chosen plaintexts must all follow certain relationships between them [9]. An example of this would be a set of inputs $X = \{x_1, x_2, \ldots, x_i\}$ with a set of altered inputs $X' = \{x'_1, x'_2, \ldots, x'_i\}$ with some specific difference δ, from their corresponding X value.

Key Recovery

The key recovery phase is when the attacker uses the distinguisher and the texts collected in combination to recover some information of the secret key. The computer guesses the all the possible round keys for each round, working inwards from the outer rounds towards the middle rounds that contain the reduced cipher (the part of the cipher that has a distinguisher defined within it). For each Guess, the algorithm must get to the middle s rounds and then use a process to determine whether it is a reduced cipher or a random permutation that is detected [9]. If the reduced cipher is detected, the key guess is added to the list of all possible keys. This is a process called *counting* however there are two processes to obtaining a list of possible secret keys.

- *Counting* is the process of starting with an empty list, and gradually adding possible key guesses until the list is sufficiently large enough to launch an attack with a high enough probability of success (decided by the attacker).
- *Sieving* is the process of starting with a list of all possible key guesses, and gradually removing key guesses that are incorrect, until the attacker is satisfied that the list is short enough to launch the attack.

Once this list of possible secret keys has been built up or sieved down suitably, the attacker can then analyse the list of keys to find any patterns in the results. For example, if every guessed key has a 1 bit in the same place, then you know that this bit **must** be in the secret key.

3.4.2. Differential Cryptanalysis

Differential Cryptanalysis considers how a pair of inputs (x, x') with difference Δx affect the pair of output ciphertexts (y, y') with difference Δy. For this attack to work many input pairs with the same difference must be generated, and then values of Δy are compared. The attacker is looking for differences between the outputs that occur regardless of x, or occur with a high probability. As the inputs need to be chosen to have a specific difference, this is a type of chosen-plaintext attack.

This analysis is simple to measure, depends more on the key than the message, has values that can be resolved, and the information gained is simple and usable, therefore I will implement this analysis method in my report.

3.4.3. Higher-Order Differential Cryptanalysis

In his 1994 report "Higher Order Derivatives and Differential Cryptanalysis", Xuejia Lai showed that there is a simple mathematical function that can be applied to differentiate any cryptographic function to any order [10].

Let $(S, +)$, $(T, +)$ be commutative groups. For a function $f : S \rightarrow T$ The derivative at some point $a \in S$ is defined as

$$\Delta_{\alpha} f(x) = f(x + a) - f(x)$$

Since the derivative of f is itself a function from S to T, we can recursively define the i-th

Derivative

$$\Delta^{(i)}_{\alpha 1,\dots,\alpha i} f(x) = \Delta_{\alpha i}(\Delta^{(i-1)}_{\alpha 1,\dots,\alpha i-1} f(x))$$

Higher-order differential cryptanalysis therefore fits all the principles of a useful analysis tool, and I will include it in my report.

3.5. Tag Second-preimage Attack against π-cipher

3.5.1. Introduction

A second-preimage attack is a cryptographic attack that targets the preimage resistance property of a cryptographic function. It aims to find a different input that produces the same output for the given function. The tag (MAC) provides data integrity for the encryption algorithm as previously discussed provides data integrity for authenticated encryption. If an attacker was able to find a method to consistently generate a different message and AD that produces the same tag from a given ciphertext, even given the secret key (insider attack) this would compromise the data integrity of the encryption algorithm.

In 2014, Gaëtan Leurent published a paper that demonstrated a tag second-preimage attack on pi-cipher using Wagner's Generalized Birthday Attack has practical applications [2].

3.5.2. The Birthday Attack

The Birthday Attack is an attack on MAC algorithms, which takes advantage of the fact that all MACs produced come to some pre-specified length, no matter how long the input is. It is called a birthday attack as its creation was inspired by the birthday paradox, which states that it is much more likely than it first seems that two people in a group share the same birthday. For example, if there are 50 people in a room you might intuitively assume that the probability is close to 50/365 (13%). However, the probability obtained by comparing all the combinations and finding the probability that two or more of them match is approximately 97%.

This probability is calculated with the general formula:

$$P(N = n) = 1 - \frac{364}{365} \times \frac{363}{365} \times \ldots \times \frac{365 - n - 1}{365}$$

Figure 4 General formula for calculating the probability of the birthday problem [11]

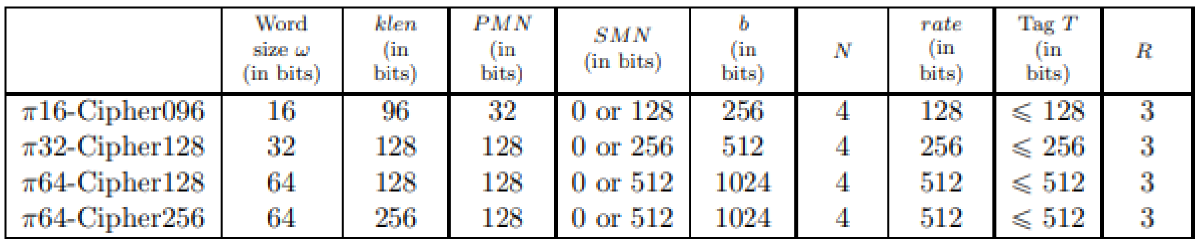
Where N is the number of people in the group, and P is the probability that 2 people share a birthday.

Applying this principle to hashing algorithms, David Wagner created a k-dimensional generalisation of the birthday problem, where the regular birthday problem is 2-dimensional and can be solved in $O(\sqrt{x})$ time, Wagner provides an example of an $O(\sqrt[3]{x})$ algortihm for k=4 lists as well as an algorithm of $O(k * 2^{n/(1+\lg k)})$ time and space complexity (sub-exponential) where k is unrestricted [12]. After I have implemented the algorithm myself I will go back to this paper and discuss their analysis of pi-cipher.

4. Implementation

4.1. Introduction

After reading the report specification, the first decision was to decide which variant of pi-cipher to implement. In the most recent version of the specification, they state that there are four versions of pi-cipher: π16-Cipher096, π32-Cipher128, π64-Cipher128 and π64-Cipher256. The number after π defines the size of ω, the size of the words inside each block i.e., the cipher is optimised for 16-bit processors, 32-bit processors and 64-bit processors with differing levels of security for each variant. The number at the end of the name defines the size of the key length as shown in the table below.



	Word size ω (in bits)	$klen$ (in bits)	PMN (in bits)	SMN (in bits)	b (in bits)	N	$rate$ (in bits)	Tag T (in bits)	R
π16-Cipher096	16	96	32	0 or 128	256	4	128	$\leqslant$ 128	3
π32-Cipher128	32	128	128	0 or 256	512	4	256	$\leqslant$ 256	3
π64-Cipher128	64	128	128	0 or 512	1024	4	512	$\leqslant$ 512	3
π64-Cipher256	64	256	128	0 or 512	1024	4	512	$\leqslant$ 512	3

Figure 5 Basic characteristics of variants of pi-cipher [1]

Choosing a Variant

My personal processor is 64-bit so I felt that I should implement the π64-Cipher128 or π64-Cipher256 variants however 64-bit variables would be too unwieldy for testing, so I decided to implement the π32-Cipher128 variant instead.

Beginning Implementation

Starting at the top of the π-function specification, I quickly realised that algorithms at the top of the design specification referenced algorithms defined lower down in the document. Therefore, I decided to start at the bottom with the most simple, atomic functions and then work back up towards the top.

4.2. ROTL 32

ROTL (Rotate On The Left) is a function that takes a 32 bit word and a rotation value as input and rotates it left, circularly, by the specified shift value. As C does not have a built-in circular shift operator I implemented it using a temporary variable and bitwise shift operators as shown below.

```
unsigned long int ROTL32(unsigned long int value, unsigned int shift) {
    //obtaining the left shifted value
    unsigned long ls = value<<shift;
    //obtaining right shift value
    unsigned long rs = value>>(32-shift);
    //obtaining XOR of right shift and left shift to get circular left shift
    return(ls^rs);
}
```

4.3. Longadd

Another atomic function that is required is the modulo addition function, that adds two numbers together and then calculates the modulo with the maximum value of an unsigned ω-bit integer. It is defined in the specification as

- Addition $+$ modulo 2^{ω};

Figure 6 modulo addition definition [1]

Rather than typing it out every time it is needed, I abstracted this to a function called “longadd”.

```
unsigned long longadd(unsigned long a, unsigned long b) {return (a+b)%((long long
int)pow(2,w));}
```

It works by using the <math.h> library to gain the pow function for mathematical powers, and storing the intermediate result in a long long int (double the size of ω) To prevent that value from overflowing. Then, I used the C modulo operator and return the final value.

4.4. ARX 32

The ARX (Addition Rotation Xor) algorithm is called * or ast in the specification and contains three main transformations: μ, ν, σ. They are done in sequence and operate over two tuples containing 4 ω-bit variables. An illustration of the structure of ARX is shown below.

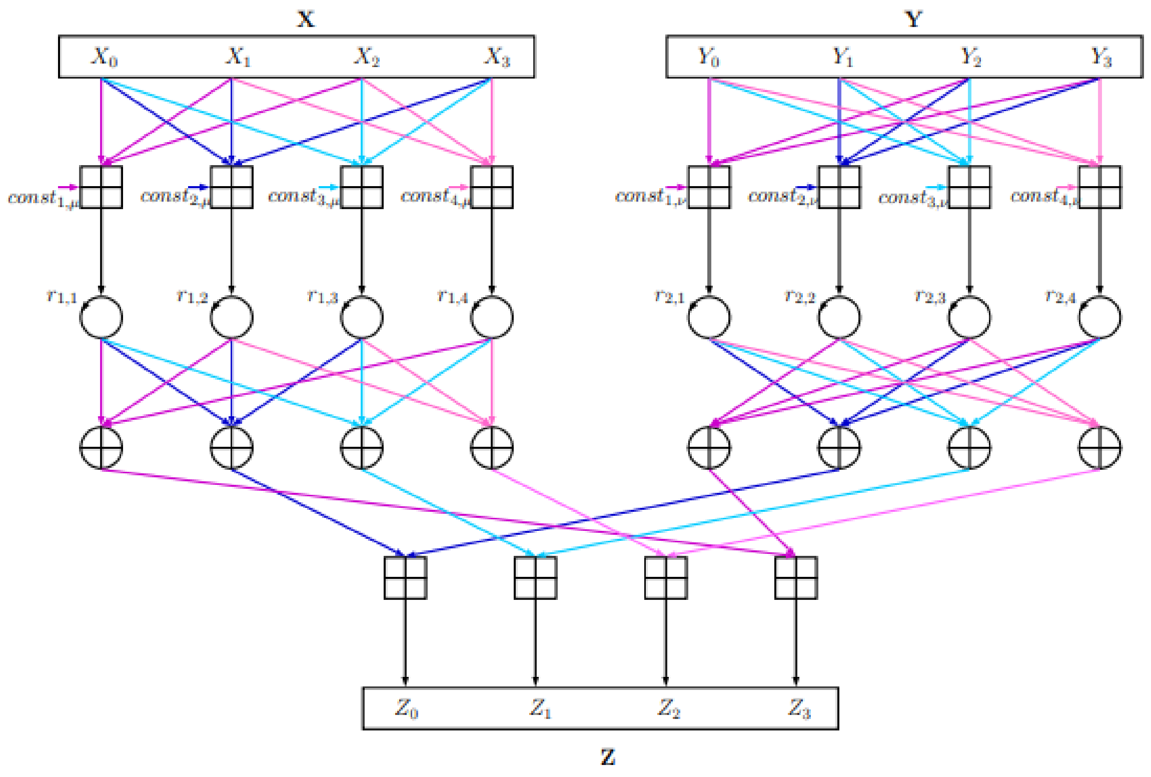
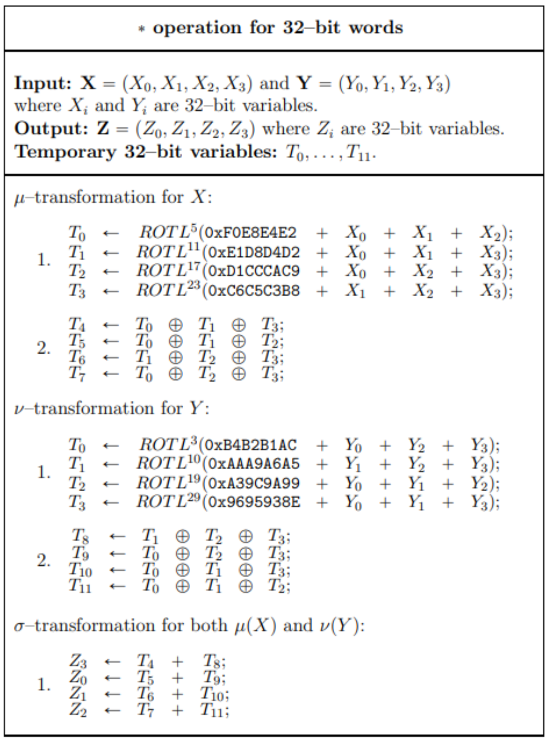


*Figure 7 Graphical representation of ARX operation * [1]*

The large ⊞ represent modulo addition and ⊕ represents Xor. The circles with $r_{1,1}, r_{1,2}, \ldots$ represent the circular rotation implemented in ROTL32. The rotation shift values come from pre-defined global vectors that are constants defined outside the function and do not change throughout pi-cipher's operation. The constants introduced in the first layer of modulo addition are also global values that do not change throughout pi-cipher.

4.4.1. Pseudocode

The function is also defined with pseudocode in the specification:



$*$ operation for 32–bit words

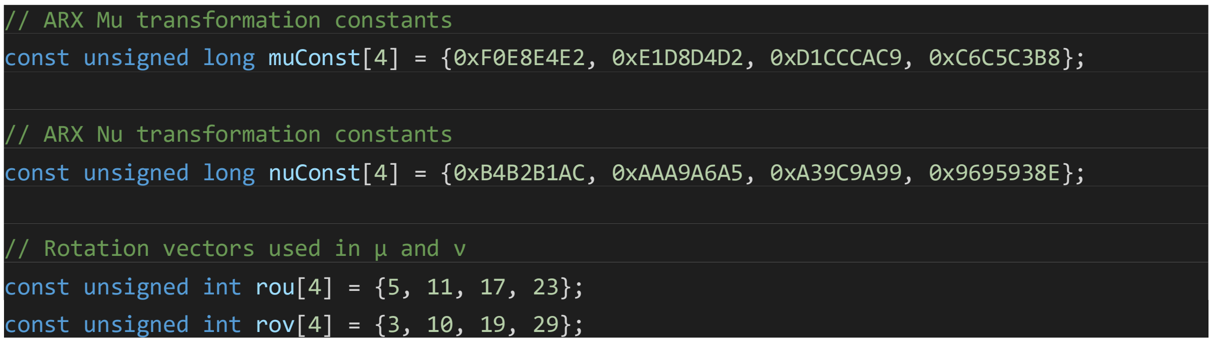
Input: $\mathbf{X} = (X_0, X_1, X_2, X_3)$ and $\mathbf{Y} = (Y_0, Y_1, Y_2, Y_3)$ where X_i and Y_i are 32–bit variables.
Output: $\mathbf{Z} = (Z_0, Z_1, Z_2, Z_3)$ where Z_i are 32–bit variables.
Temporary 32–bit variables: $T_0, \dots, T_{11}$.

```
μ–transformation for X:
    1.  $T_0 \leftarrow ROTL^5(\texttt{0xF0E8E4E2} + X_0 + X_1 + X_2);$
        $T_1 \leftarrow ROTL^{11}(\texttt{0xE1D8D4D2} + X_0 + X_1 + X_3);$
        $T_2 \leftarrow ROTL^{17}(\texttt{0xD1CCCAC9} + X_0 + X_2 + X_3);$
        $T_3 \leftarrow ROTL^{23}(\texttt{0xC6C5C3B8} + X_1 + X_2 + X_3);$
    2.  $T_4 \leftarrow T_0 \oplus T_1 \oplus T_3;$
        $T_5 \leftarrow T_0 \oplus T_1 \oplus T_2;$
        $T_6 \leftarrow T_1 \oplus T_2 \oplus T_3;$
        $T_7 \leftarrow T_0 \oplus T_2 \oplus T_3;$
ν–transformation for Y:
    1.  $T_0 \leftarrow ROTL^3(\texttt{0xB4B2B1AC} + Y_0 + Y_2 + Y_3);$
        $T_1 \leftarrow ROTL^{10}(\texttt{0xAAA9A6A5} + Y_1 + Y_2 + Y_3);$
        $T_2 \leftarrow ROTL^{19}(\texttt{0xA39C9A99} + Y_0 + Y_1 + Y_2);$
        $T_3 \leftarrow ROTL^{29}(\texttt{0x9695938E} + Y_0 + Y_1 + Y_3);$
    2.  $T_8 \leftarrow T_1 \oplus T_2 \oplus T_3;$
        $T_9 \leftarrow T_0 \oplus T_2 \oplus T_3;$
        $T_{10} \leftarrow T_0 \oplus T_1 \oplus T_3;$
        $T_{11} \leftarrow T_0 \oplus T_1 \oplus T_2;$
σ–transformation for both μ(X) and ν(Y):
    1.  $Z_3 \leftarrow T_4 + T_8;$
        $Z_0 \leftarrow T_5 + T_9;$
        $Z_1 \leftarrow T_6 + T_{10};$
        $Z_2 \leftarrow T_7 + T_{11};$
```

*Figure 8 * pseudocode definition [1]*

4.4.2. Constants

Rather than hard-coding the constants and rotation vectors, I coded them outside of the function and then referenced them inside the function, even though the values do no change and are not used anywhere else, makes the code easier to understand and change later.



```
// ARX Mu transformation constants
const unsigned long muConst[4] = {0xF0E8E4E2, 0xE1D8D4D2, 0xD1CCCAC9, 0xC6C5C3B8};

// ARX Nu transformation constants
const unsigned long nuConst[4] = {0xB4B2B1AC, 0xAAA9A6A5, 0xA39C9A99, 0x9695938E};

// Rotation vectors used in µ and v
const unsigned int rou[4] = {5, 11, 17, 23};
const unsigned int rov[4] = {3, 10, 19, 29};
```

4.4.3. ARX Function

```
unsigned long int * ARX32(unsigned long int X[4], unsigned long int Y[4]) {
    //temporary variables for transformation
    unsigned long T[12];
    unsigned long Z[4];

    //µ-transformation for X
        //addition and rotation
        T[0] = ROTL32(longadd(longadd(longadd(muConst[0], X[0]), X[1]), X[2]), rou[0]);
        T[1] = ROTL32(longadd(longadd(longadd(muConst[1], X[0]), X[1]), X[3]), rou[1]);
        T[2] = ROTL32(longadd(longadd(longadd(muConst[2], X[0]), X[2]), X[3]), rou[2]);
        T[3] = ROTL32(longadd(longadd(longadd(muConst[3], X[1]), X[2]), X[3]), rou[3]);

        //XOR
        T[4] = T[0]^T[1]^T[3];
        T[5] = T[0]^T[1]^T[2];
        T[6] = T[1]^T[2]^T[3];
        T[7] = T[0]^T[2]^T[3];

    //v-transformation for Y
        //addition and rotation
        T[0] = ROTL32(longadd(longadd(longadd(nuConst[0], Y[0]), Y[2]), Y[3]), rov[0]);
        T[1] = ROTL32(longadd(longadd(longadd(nuConst[1], Y[1]), Y[2]), Y[3]), rov[1]);
        T[2] = ROTL32(longadd(longadd(longadd(nuConst[2], Y[0]), Y[1]), Y[2]), rov[2]);
        T[3] = ROTL32(longadd(longadd(longadd(nuConst[3], Y[0]), Y[1]), Y[3]), rov[3]);

        //XOR
        T[8] = T[1]^T[2]^T[3];
        T[9] = T[0]^T[2]^T[3];
        T[10] = T[0]^T[1]^T[3];
        T[11] = T[0]^T[1]^T[2];

    //σ-transformation for both µ(X) and v(Y)
        Z[3] = longadd(T[4], T[8]);
        Z[0] = longadd(T[5], T[9]);
        Z[1] = longadd(T[6], T[10]);
        Z[2] = longadd(T[7], T[11]);

        return Z;

}
```

The function takes arrays X and Y as arguments and calculates μ, ϑ, θ transformations in sequence then returns a pointer to Z.

4.4.4. µ–transformation

The µ–transformation acts only on the X array and is defined according to the pseudocode, using longadd() operation on each value and then rotating the final value according to the rotation vectors defined earlier, then each value is stored into a temporary variable. Afterwards, the values are cross combined using the built in C Xor operation.

4.4.5. ν–transformation

The ν–transformation acts only on the Y array and acts very similarly to the µ–transformation. A different rotation vector and set of constant values are used and combined with the values of Y and are saved into a new set of temporary variables.

4.4.6. σ–transformation

The σ–transformation is an injection, combining the 8 values of X and Y into 4 output values in Z. The values are once again combined with longadd().

4.5. Pi

Below is a graphical representation of one round, which includes one iteration of E1 and one iteration of E2. In this diagram the diagonal lines are interpreted as ARX functions and the vertical lines are assignments. E1 takes N values of I as input then starts at the left first working out ARX(C1, I1) where C1 is a global constant and I1 is the first argument of E1. The output of ARX(C1,I1) is stored in J1, and then ARX(J1, I2) is calculated and so on until you reach ARX(JN-1, IN). E2 then takes the output of E1 as input and works backwards starting at ARX(JN, C2) and working to the left.

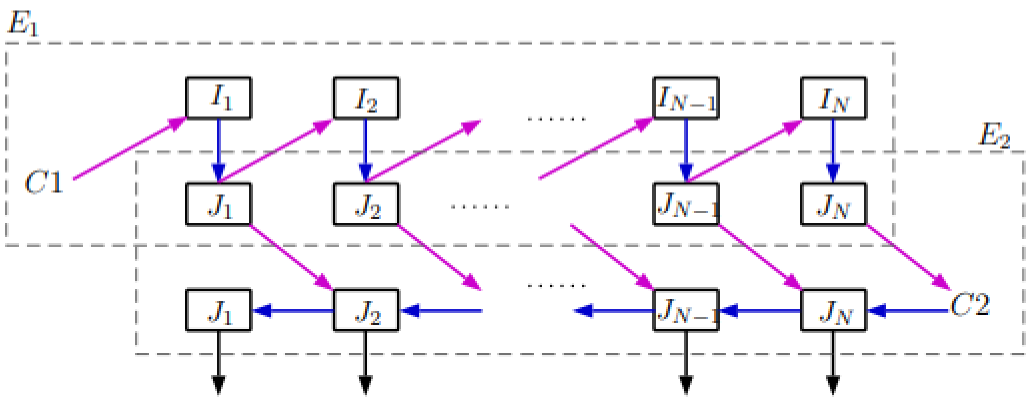
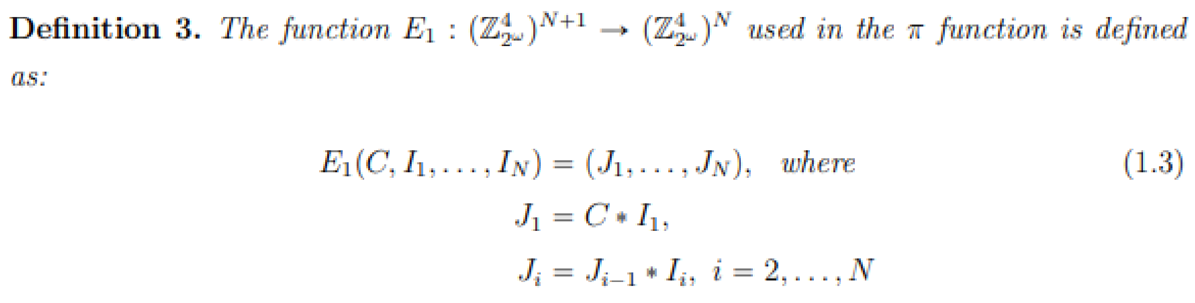


Figure 9 Graphical representation of one round of π-Cipher [1]

4.5.1. E1

Below you can see the Definition of E1 in the specification.



Definition 3. *The function* $E_1 : (\mathbb{Z}^4_{2^\omega})^{N+1} \to (\mathbb{Z}^4_{2^\omega})^N$ *used in the* π *function is defined as:*

$$E_1(C, I_1, \ldots, I_N) = (J_1, \ldots, J_N), \quad \textit{where} \tag{1.3}$$

$$J_1 = C * I_1,$$

$$J_i = J_{i-1} * I_i,\ i = 2, \ldots, N$$

Figure 10 E1 Definition [1]

Tuples

At this time, I decided to restructure my arguments. Up until now I had been using arrays as the arguments but, as they were referred to as tuples in the specification, I decided to implement a structure to recreate this in C to be more "correct" to the specification.

```
struct Tuple {
    unsigned long int a;
    unsigned long int b;
    unsigned long int c;
    unsigned long int d;
};
```

Va_args

Seeing that E1 required a variable number of inputs, I decided to learn if that was possible in C. I found that using a library <stdarg.h>, you could create a function that would take the standard inputs in the beginning and then leave an ellipsis "…" and any number of arguments could be entered afterwards and they would be added to a va_list (variable arguments list) that you could name and then take the items off the list one at a time like a queue. I also decided to create J using dynamic memory allocation, so that I could change the size depending on the value of N. Therefore, I created this first draft for the E1 function:

```c
unsigned long int * E1(struct Tuple C, int N, ...) {
    va_list args;
    va_start(args, N);
    long int *J = malloc(N*sizeof(struct Tuple));

    // for each value of i where i = 1,...,N
    for (int i=1; i<=N; i++) {

        // grab the first tuple from the N tuples inputted in the function
        struct Tuple Ii = va_arg(args, struct Tuple);
        unsigned long *Z;

        // if it's the first Tuple
        if (i == 1) {
            // * operation on Constant tuple and first tuple
            Z = ARX32(C, Ii);
            //putting values into J malloc list
            for (int j=0; j<4; j++) {
                J[j] = Z[j];
            }
        } else {
            // getting the tuple Ji-1 from malloc list J
            struct Tuple A = {J[(i-1)*4], J[(i-1)*4+1], J[(i-1)*4+2], J[(i-1)*4+3]};
            // * operation on last tuple and current tuple
            Z = ARX32(A, Ii);
            // putting values into J malloc list
            for (int j=i*4; j<(i*4)+4; j++) {
                J[j] = Z[j];
            }
        }
    }

    return J;

    free(J);
    va_end(args);
}
```

The ellipsis notifies the program that a variable number of arguments will follow, and N defines the number of inputs in that list. I then create a list called “args” that contains all the variable arguments (I tuples), and create a dynamic memory allocation for J. The reason for doing this inside the function at the time was twofold: so that I could adjust the size of J according to the size of N as previously mentioned and as creating dynamic memory allocation points to a place on the heap, it would create global pointer variable that I could then return outside of the function. I then iterated through each value of `i`, grabbing the `Ii` tuple from the queue in args. First, I check if it was the first case, if it was I calculate `ARX(C,I1),` and place the resulting `Z` tuple into `J(1)`. Otherwise, I fetch the tuple `J(i-1)` and calculate `ARX(Ji-1,Ii)`. After all values of i had been iterated, I returned the `J` pointer, freed `J`, and called the function to end the va_list. At the time I was unsure whether these functions were being called as they were after the “return”.

Re-creating E1

Having created both E1 and E2 functions with this structure, I began trying to make a pi function to combine them, but I was running into some problems:

- Constantly packing and unpacking the tuples to use their values was unnecessarily complex and inefficient (as at the end of each function it would return a pointer that would then have to be manually unpacked at the destination).
- The memory used to create dynamic memory allocations was not being freed as the pointer needed to be returned before it was freed, but calling the return operator ends execution of the function.
- Pi calls E1 and E2 multiple times with multiple different states of I and J and it becomes very complicated to try and retrieve all the values of I and J individually to enter them into each call of the function.

I realised it was very unnecessary to enter the I and J tuples individually, and it would be much easier to combined them into two big structures that can be passed into each function. I combined this with the new knowledge that N is set at the start of the algorithm and is static for the rest of execution to realise the most simple method is to create a static array for I and J according to the size of N. Array size cannot be defined with variables, but this can be circumvented by using #define to define a macro to replace all the instances of N with a number set by the user, or I could simply hardcode the value.

Tuple copy

Creating this new version of E1 I found myself running the same for-loop many times, where the four values of a tuple needed to be copied into some other array. This process created many extra lines of code and made debugging more difficult. Therefore, I decided to abstract this process into a method "tuplecopy()" as shown below.

```c
void tuplecopy(int Xi, unsigned long int *X, int Yi, unsigned long int *Y) {
    for (int i=0; i<4; i++) {
        X[Xi + i] = Y[Yi + i];
    }
}
```

This method takes two arrays X, Y and two index variables Xi, Yi. The method copies the first 4 values from Y starting at Yi and copies them into the first 4 values of X, starting at Xi. This simple function saved me a lot of time in debugging as I always knew the function would run correctly given the correct inputs.

Adjusting ARX

ARX needed to be adjusted so I changed it to take X,Y,Z as arguments and change the values of Z by reference, which gives the final version below.

```
void ARX32(unsigned long int X[4], unsigned long int Y[4], unsigned long int Z[4]) {
    //temporary variables for transformation
    unsigned long T[12];

    //μ-transformation for X
        //addition and rotation
        T[0] = ROTL32(longadd(longadd(longadd(muConst[0], X[0]), X[1]), X[2]), rou[0]);
        T[1] = ROTL32(longadd(longadd(longadd(muConst[1], X[0]), X[1]), X[3]), rou[1]);
        T[2] = ROTL32(longadd(longadd(longadd(muConst[2], X[0]), X[2]), X[3]), rou[2]);
        T[3] = ROTL32(longadd(longadd(longadd(muConst[3], X[1]), X[2]), X[3]), rou[3]);

        //XOR
        T[4] = T[0]^T[1]^T[3];
        T[5] = T[0]^T[1]^T[2];
        T[6] = T[1]^T[2]^T[3];
        T[7] = T[0]^T[2]^T[3];

    //v-transformation for Y
        //addition and rotation
        T[0] = ROTL32(longadd(longadd(longadd(nuConst[0], Y[0]), Y[2]), Y[3]), rov[0]);
        T[1] = ROTL32(longadd(longadd(longadd(nuConst[1], Y[1]), Y[2]), Y[3]), rov[1]);
        T[2] = ROTL32(longadd(longadd(longadd(nuConst[2], Y[0]), Y[1]), Y[2]), rov[2]);
        T[3] = ROTL32(longadd(longadd(longadd(nuConst[3], Y[0]), Y[1]), Y[3]), rov[3]);

        //XOR
        T[8] = T[1]^T[2]^T[3];
        T[9] = T[0]^T[2]^T[3];
        T[10] = T[0]^T[1]^T[3];
        T[11] = T[0]^T[1]^T[2];

    //σ-transformation for both μ(X) and v(Y)
        Z[3] = longadd(T[4], T[8]);
        Z[0] = longadd(T[5], T[9]);
        Z[1] = longadd(T[6], T[10]);
        Z[2] = longadd(T[7], T[11]);
}
```

Final E1 Algorithm

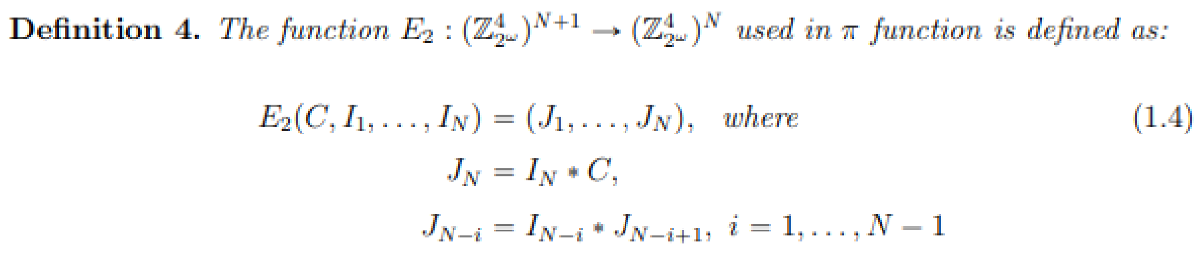
The final E1 method was implemented as shown below.

```c
// Inputs: C(Tuple), J(Array), N(No. of I), I1,...,IN(Tuples)
void E1(unsigned long int C[4], unsigned long int *I, int N, unsigned long int *J) {
    //counter variable(s)
    int i;

    // output array of ARX32 function
    unsigned long int Z[4];

    // the 4-tuples that are iterated through values of i
    unsigned long int Ii[4];
    unsigned long int Ji[4];

    // --- initial condition ---
    //fetching I1 from I
    tuplecopy(0, Ii, 0, I);

    //calculating C * I1 (store in Z)
    ARX32(C, Ii, Z);

    //store Z to J1
    tuplecopy(0, J, 0, Z);

    // --- looping for each value i = 2,...,N ---
    for (i=2; i<=N; i++) {

        //fetching Ji-1
        tuplecopy(0, Ji, (i-2)*4, J);

        //fetching Ii
        tuplecopy(0, Ii, (i-1)*4, I);

        //calculating Ji-1 * Ii
        ARX32(Ji, Ii, Z);

        //store Z to Ji
        tuplecopy((i-1)*4, J, 0, Z);
    }
}
```

It initialises all the local variables necessary (including Z tuple to store the output of ARX), then runs through the initial condition for i=1, then loops through all the values of i=2,…,N fetching the correct values of `Ii` and `Ji-1`, then storing the result in the correct position in `J` (the index is (i-1)*4 as i is indexed from 1 and J is indexed from 0).

4.5.2. E2

Once E1 was correctly implemented, E2 was linear to implement as it is similar to E1. The definition of E2 in the specification is shown below.



Definition 4. *The function* $E_2 : (\mathbb{Z}_{2^\omega}^4)^{N+1} \rightarrow (\mathbb{Z}_{2^\omega}^4)^N$ *used in* π *function is defined as:*

$$\begin{aligned} E_2(C, I_1, \ldots, I_N) &= (J_1, \ldots, J_N), \quad \textit{where} \\ J_N &= I_N * C, \\ J_{N-i} &= I_{N-i} * J_{N-i+1}, \ i = 1, \ldots, N-1 \end{aligned} \tag{1.4}$$

Figure 11 E2 Definition [1]

The function takes the same set of arguments as E1 but works in reverse order, taking `ARX(IN,C)` as the base case and working backwards towards `I1`. It does this by incrementing i and then taking it away from N to get the index for `IN-i`. The full implementation is shown on the next page.

E2 Algorithm

```c
// Inputs: C(Tuple), J(Array), N(No. of I), I1,...,IN(Tuples)
void E2(unsigned long int C[4], unsigned long int *I, int N, unsigned long int *J) {
    //counter variable(s)
    int i;

    // output array of ARX32 sub-function
    unsigned long int Z[4];

    // the 4-tuples that are iterated through values of i
    unsigned long int Ii[4];
    unsigned long int Ji[4];

    // --- initial condition ---
    //fetching IN from I
    tuplecopy(0, Ii, (N-1)*4, I);

    //calculating IN * C (store in Z)
    ARX32(Ii, C, Z);

    //store Z to JN
    tuplecopy((N-1)*4, J, 0, Z);

    // --- looping for each value i = 1,...,N-1 ---
    for (i=1; i<=N-1; i++) {

        //fetching IN-i
        tuplecopy(0, Ii, (N-1-i)*4, I);

        //fetching JN-i+1
        tuplecopy(0, Ji, (N-i)*4, J);

        //calculating IN-i * JN-i+1
        ARX32(Ii, Ji, Z);

        //store Z to JN-i
        tuplecopy((N-1-i)*4, J, 0, Z);
    }
}
```

4.5.3. Pi function

The Pi function is a combination of the E1 and E2 functions, as shown earlier in the graphical representation (Figure 9). For example, here are the definitions of one round of pi and three rounds of pi respectively.

One round of pi:

$$\pi(I_1, \dots, I_N) = E_2(C2, E_1(C1, I_1, \dots, I_N)$$

Three rounds of pi:

$$\pi(I_1, \dots, I_N) = E_2\left(C6, E_1\left(C5, E_2\left(C4, E_1\left(C3, E_2(C2, E_1(C1, I_1, \dots, I_N))\right)\right)\right)\right)$$

Constants

The constants for up to three rounds of pi are given in the specification and I implemented them as a single global array so that it could be modified easily and passed easily into the pi function.

```c
// Round constants for π32-Cipher
unsigned long int C[24] = {0x8D8B8778, 0x7472716C, 0x6A696665, 0x635C5A59,
                           0x5655534E, 0x4D4B473C, 0x3A393635, 0x332E2D2B,
                           0x271E1D1B, 0x170FF0E8, 0xE4E2E1D8, 0xD4D2D1CC,
                           0xCAC9C6C5, 0xC3B8B4B2, 0xB1ACAAA9, 0xA6A5A39C,
                           0x9A999695, 0x938E8D8B, 0x87787472, 0x716C6A69,
                           0x6665635C, 0x5A595655, 0x534E4D4B, 0x473C3A39};
```

The only justification for these values being chosen was an even distribution of bits, which means we can easily create more for more rounds of pi or modify them later.

Pi Algorithm

```c
// R is No. of rounds, C is constants, N is No. of I blocks, I is internal states, J is modified internal states
void pi(int R, unsigned long int *C, int N, unsigned long int *I, unsigned long int *J) {
    //R defines the number of rounds of pi, and therefore the number of C tuples (C=2*R)

    //counter variable(s)
    int i;

    //constant tuples
    unsigned long int Cx[4], Cy[4];

    // exception case
    if (R<1 || R>3) {
        printf("error: R value exception");
        return;
    } else {
        // for each round of pi
        for (i=0; i<R; i++) {

            //fetching constant tuples from C list
            tuplecopy(0, Cx, (i*8),  C);
            tuplecopy(0, Cy, (i*8)+4, C);

            //execute one round of pi
            E1(Cx, I, N, J);
            E2(Cy, J, N, I);
        }
    }
}
```

The method takes everything necessary as arguments, including `R` which is the number of rounds meaning it can be easily adjusted later. The pi algorithm then defines some local arrays to hold the values of the constant tuples for E1 and E2, then checks for an exception in the value of R, copies the correct tuples into the temporary arrays and calls E1 and E2 in sequence. Note that the positions of `I` and `J` are swapped in E2, this is as E1's output is E2's input. Therefore, the E2 must take E1's `J` values, and output back into the `I` "Internal State" array.

4.6. Evaluation Implementation

In this section I will go over how I implemented the tools I used to evaluate the cipher.

4.6.1. pbPlots

To evaluate the cipher, I needed a method of plotting variable data directly into a graph that looks presentable and clear. I needed this as the data I was calculating was big and recording and entering the data manually into a graph plotting software would be time consuming and wasteful.

pbPlots is an open-source plotting library available in many languages with the goals [13]:

- Have a single library for plotting that is stable across programming languages and over time.
- Is easy to build in any programming language and to include in any project.
- Is easy to use.

Having read the documentation I saw that it has been implemented in C, and I looked at the example graphs provided on the web page. I knew that pbPlots fit my requirements perfectly.

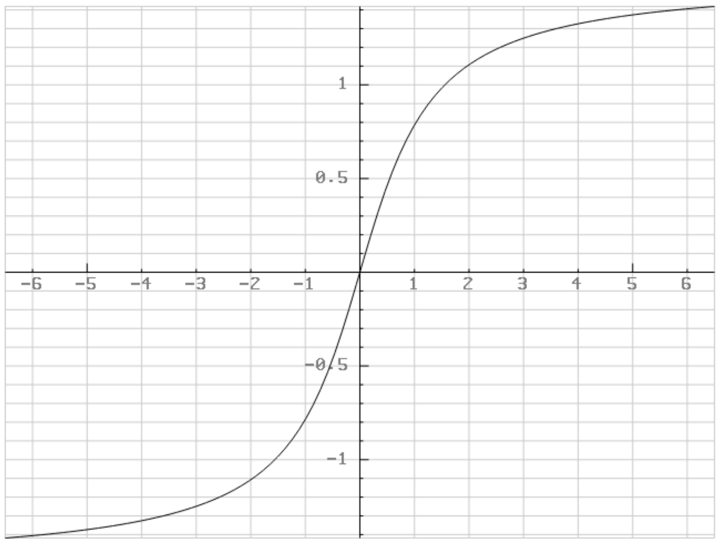


Figure 12 Example of pbPlots' capabilities [13]

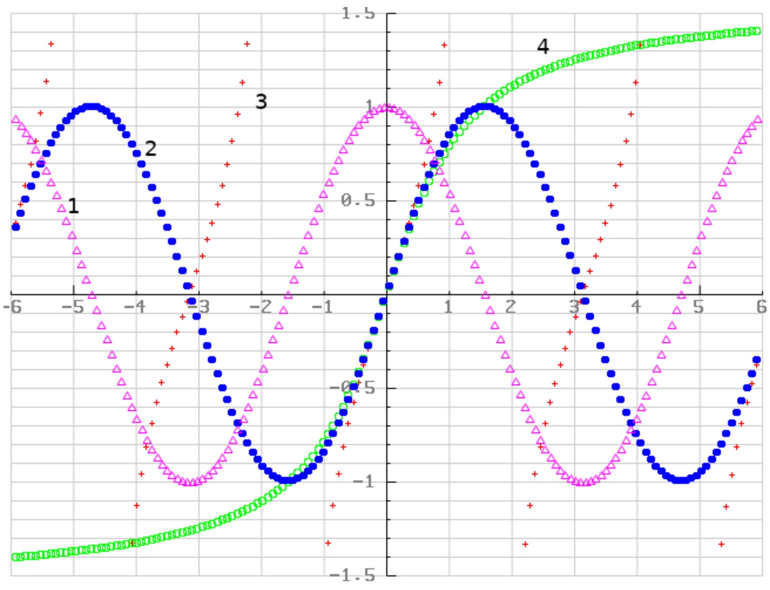


Figure 13 Example of pbPlots' capabilities [13]

4.6.2. Scatterplot Series

```
//creating an image reference
RGBABitmapImageReference *imgRef = CreateRGBABitmapImageReference();
//create error message for debugging
StringReference *errorMessage = CreateStringReference(L"", 0);

//configuring series
ScatterPlotSeries *series = GetDefaultScatterPlotSeriesSettings();
series->xs = Xaxis;
series->xsLength = 512;
series->ys = Yaxis;
series->ysLength = 512;
series->linearInterpolation = true;
series->lineType = L"solid";
series->lineTypeLength = wcslen(series->lineType);
series->lineThickness = 2;
series->color = GetGray(0.5);
```

Using the documentation, I initialised the series for a scatterplot with the default settings and started to adjust the attributes.

- xs, ys define the name of the arrays that will feed in the x and y values of the points on the scatter plot.
- xslength and yslength define the length of the arrays.
- linearInterpolation is a Boolean that defines whether to use linear interpolation curve fitting to the data points.
- lineType, linethickness, and color adjust the appearance of the line.

4.6.3. Scatterplot Settings

```
ScatterPlotSettings *settings = GetDefaultScatterPlotSettings();
        settings->width = 1000;
        settings->height = 1000;
        settings->autoBoundaries = false;
        settings->autoPadding = true;
        settings->title = L"3-bit Differential attack over pi";
        settings->titleLength = wcslen(settings->title);
        settings->xLabel = L"Position of bit in IS";
        settings->xLabelLength = wcslen(settings->xLabel);
        settings->yLabel = L"Probability of bit changing (%%)";
        settings->yLabelLength = wcslen(settings->yLabel);
        settings->xMax = 512;
        settings->yMax = 100;
        ScatterPlotSeries *s [] = {series};
        settings->scatterPlotSeries = s;
        settings->scatterPlotSeriesLength = 1;
```

Here I adjust the settings of the graph image that will be produced.

- Width and height define the size of the PNG image that will be produced in pixels.
- Autoboundaries automatically calculates the boundaries of the plot, which I disabled as some graphs I wanted to set custom minimum and maximum values to give a clearer picture of the area of interest.
- Autopadding automatically calculates the padding for the image, which I enabled.
- title, xlabel, and ylabel define the text labels for the graph.
- xMax and yMax define the maximum values on the respective axes.
- xMin and yMin define the minimum value on the respective axes.
- scatterPlotSeries passes the previously defined series into settings.

4.6.4. Drawing the Graph

```
        //drawing the plot
        DrawScatterPlotFromSettings(imgRef, settings, errorMessage);

        //converting to png and writing to file
        size_t length;
        double *pngData = ConvertToPNG(&length, imgRef -> image);
        WriteToFile(pngData, length, "DifferentialPi3.png");
```

This code takes the image reference, settings, and error message reference and draws the scatterplot, then converts it to a PNG and saves it to a file under the filename.

4.6.5. Hamming Distance Analysis

For the avalanche analysis I needed to create a series of random inputs and change each of them by one bit to save to a separate variable. Then I needed to calculate the hamming distance between the respective outputs. For more information on the methodology see Section 6.1.

Hamming Distance

```
int hammingDistance (unsigned long int x, unsigned long int y) {
        unsigned long int res = x ^ y;
        return __builtin_popcountll (res);
}
```

This function calculates the hamming distance between two unsigned long integers. It works by calling XOR on the two arguments and using a built-in function to count the number of 1s. It is very efficient and effective for my purpose.

To build this analysis on the ARX function, first I had to define the constants necessary, here is an example for an analysis of changes to the X tuple:

```
  // --- X' * function bit diffusion ---
  int main(void) {

   unsigned long int X[4], Y[4], Z[4];
   unsigned long int Xa[4];
   int i,j;
   unsigned long int O[testSize*4], Oa[testSize*4];
   double HammingDistances[testSize] = {0};
   unsigned long int bitNo, wordNo, dMask;
```

- X, Y, Z are the standard inputs and output.
- Xa is for storing the altered input.
- O and Oa are for storing the standard and altered outputs in one array.
- HammingDistances is initialised implicitly with all zeros for storing the hamming distances between the two output arrays.
- bitNo, wordNo, dMask are for changing the value of X for Xa.

Also, it is worth noting that "testSize" is a macro word replacement that is defined at the beginning of the document.

```
#define testSize    30000
```

Hamming Distances

```c
    srand(time(NULL));   // Initialization, should only be called once.
    for (i=0; i<testSize; i++) {
        for (j=0; j<4; j++) {
            X[j] = (rand() << 16) | rand();
            Y[j] = (rand() << 16) | rand();
            Xa[j] = X[j];
        }
        //calculating ARX on the generated data
        ARX32(X, Y, Z);
        //storing into list to compare
        tuplecopy(i*4, O, 0, Z);

        //iterating through each bit from 0 to 127
        bitNo = i % 128;
        //finding which word the bit affects (integer division)
        wordNo = bitNo / 32;
        //changing bitNo to reflect position within the word
        bitNo = bitNo % 32;
        //creating the delta mask
        dMask = 1 << (31-bitNo);
        //changing the original at the one bit specified using XOR
        Xa[wordNo] = Xa[wordNo] ^ dMask;
        //calculating ARX on the modified data
        ARX32(Xa, Y, Z);
        //storing into seperate list to compare
        tuplecopy(i*4, Oa, 0, Z);

        //calculating hamming distnaces and combining 4 hamming distances into one place in array
        for (j=0; j<4; j++) {
            HammingDistances[i] += hammingDistance(O[i*4+j], Oa[i*4+j]);
        }
    }
```

Rand is used here as fast to generate pseudo-random numbers are needed for the test values. The fact that it is deterministic is not important, as it is not being used for the cryptography itself. srand() is called to initialise the seed of the rand() function and, as rand() generates 16-bit random numbers I had to use it twice for each variable in combination with a left shift of 16 and OR operator to get the desired effect.

`ARX(X,Y)` is calculated with the original values and stored into O. Xa is set to the value of X as I will only need to change one bit of one of the four words for each Xa, so I only need to worry about one word to change. bitNo is set to i % 128 to constantly loop through values 0-127 throughout the main iteration loop. Integer division is used to calculate which word will change, and then the bitNo is updated to be relative to which bit inside this word changed. Then, a “delta mask” is created that will produce the desired bit to flip and then XOR is called between the two to flip that bit. `ARX(Xa,Y)` is called and stored into Oa. Finally, the Hamming distances between the 4 output words of each function is calculated and stored into the HammingDistances array.

Average

Now that the hamming distances have been calculated, the average of all the hamming distance for each bit changed must be obtained so that it can be graphed.

```
// --- getting an average ---
        unsigned long long int sum = 0;
        double avg;
        float numValues;
        double Xaxis[128], Yaxis[128];
        //assigning a list of bit numbers from 1 to 128
        for (i=0; i<128; i++) {
            Xaxis[i] = i+1;
        }
```

The sum variable is an unsigned long long int to ensure it can hold all the values it needs to without overflow. Yaxis and Xaxis are defined as arrays of double datatype, as this is the datatype necessary for the graphing library to work. Xaxis is initialised with values 0-127 as this is the range of bits that are being changed.

```
        //if an average is necessary
        if(testSize > 128) {
            for (i=0; i<128; i++) {
                sum = 0; numValues = 0;
                for (j=0; j<=testSize-128; j+=128) {
                    sum += HammingDistances[i+j];
                    numValues++;
                }
                //calculating the average
                avg = sum/numValues;
                //storing into the first place of that bit number in hamming distance array
                Yaxis[i] = avg;
            }
        } else {
            for(i=0; i<testSize; i++) {
                Yaxis[i] = HammingDistances[i];
            }
        }
```

Then, I quickly check if an average is necessary (checking if any of the bits has been visited more than once) and if so I start iterating through values of i 0-127. For each of these values of i, the algorithm checks every value 128 places apart within the length of the list. These will be all the values where the same bit is changed. For each of these points, the hamming distance value at this point is added to the sum and the tracker for the number of values is incremented. At the end a simple mean average is acquired and is assigned to its position in the Yaxis array. In the other condition, since the test size is smaller than 128, no average is necessary, and the values are simply copied into the Yaxis array.

4.6.6. Probability Distribution

The other type of analysis I implemented, was a measure of the probability of each bit changing, given some difference in the inputs. I did this by calling Xor between the two outputs and building a "bit frequency" array that tracks how many times each of the bits were changed. For more information on the methodology see section 6.2.

Constants

First, I defined the constants:

```c
// --- X' * function probability diffusion ---
int main(void) {
    unsigned long int X[4], Y[4], Z[4];
    unsigned long int Xa[4], Xb[4], Zb[4];
    int i,j,z;
    int bitFreq[128] = {0};
    float probabilities[128] = {0};
    unsigned long int bitNo = 0, wordNo = 0, dMask;
    unsigned long int result[4];
    unsigned long int currentWord;
    double Xaxis[128], Yaxis[128];
```

Outputs

Next I created the test data and obtained the two outputs. Note that this time bitNo is set to a constant value as it is making the same change each time.

```c
    srand(time(NULL));   // Initialization, should only be called once.
    for (i=0; i<testSize; i++) {
        for (j=0; j<4; j++) {
            X[j] = (rand() << 16) | rand();
            Y[j] = (rand() << 16) | rand();
            Xa[j] = X[j];
        }
        //calculating f(x)
        ARX32(X, Y, Z);

        bitNo = 1;
        wordNo = bitNo/32;
        bitNo = bitNo%32;
        dMask = 1 << (31-bitNo);
        //changing the original at the one bit specified using XOR
        Xa[wordNo] = Xa[wordNo] ^ dMask;

        for (j=0; j<4; j++) {
            Xb[j] = X[j] ^ Xa[j];
        }

        //calculating f(x Xor a)
        ARX32(Xb, Y, Zb);
```

Bit Frequency

Next, I Xor the two tuples and measure the bit frequency.

```
        //XOR the two tuples
        for (j=0; j<4; j++) {
            //creating a mask of all the bits changed
            result[j] = Z[j] ^ Zb[j];
            for (z=0; z<32; z++) {
                if ((result[j] >> z) & 1) {
                    bitFreq[j*32 + (31-z)] += 1;
                }
            }


        }
```

For each word, for each bit in the word, for loop increments the right shift of the word, then checks if it has a 1 bit as its last value. If it does, then increments the corresponding bit frequency array value accordingly.

Probabilities

Here, to work out the probability of each change occurring, you simply divide by the test size and times by 100 to get a percentage.

```
    //working out the probabilities
    for (i=0; i<128; i++) {
        Yaxis[i] = (bitFreq[i]/testSize)*100;
        Xaxis[i] = i + 1;
    }
```

5. Testing

5.1. ARX

5.1.1. ROTL

Category	Input Value	Shift Value	Expected Result	Actual Result	Passed Test
Normal	0x1cf9cfba	8	0xf9cfba1c	0xf9cfba1c	Yes
Normal	0x533cb0b6	24	0xb6533cb0	0xb6533cb0	Yes
Normal	0xcf7aa1b8	3	0x7bd50dc6	0x7bd50dc6	Yes
Extreme	0xffffffff	29	0xffffffff	0xffffffff	Yes
Extreme	0x00000000	19	0x00000000	0x0	Yes
Extreme	0xfedb3f94	40	0xdb3f94fe	0xdb3f94fe	Yes
Extreme	0x2ec4a18b	-8	0x8b2ec4a1	0x8b2ec4a1	Yes
Exception	0x52bd9454	32	0x52bd9454	0x0	No
Exception	0xfffffff1	27	Error	Error	Yes
Exception	0xae042086	4294967297	Error	Error	Yes

In response to the unexpected failure in testing, I looked at the code again and understood why this was happening: The two values are always obtained through left and right shifting and then applying XOR to the result. The problem occurs when they are the same result as XOR of the two values that are the same always returns 0, when it should be returning the original input. Therefore, I added an exception for when the shift is a multiple of 32:

```
unsigned long ROTL32(unsigned long value, unsigned int shift) {
    if(shift%32 == 0) {
        return value;
    } else {
        //obtaining the left shifted value
        unsigned long ls = value<<shift;
        //obtaining right shift for
        unsigned long rs = value>>(32-shift);
        //obtaining XOR of right shift and left shift to get circular left
shift by value specified
        return(ls^rs);
    }
}
```

Here are the updated test results including the exception:

Category	Input Value	Shift Value	Expected Result	Actual Result	Passed Test
Exception	0x52bd9454	32	0x52bd9454	0x52bd9454	Yes
Exception	0x068646cf	0	0x068646cf	0x068646cf	Yes

5.1.2. Longadd

Category	Value 1	Value 2	Expected Result	Actual Result	Passed Test
Normal	100	80	180	180	Yes
Normal	0xf6045ba0	0x52cc7655	0x48d0d1f5	0x48d0d1f5	Yes
Exception	0x21b1667d	-21	ERROR	0x21b16668	Yes

When a negative value is entered, the algorithm just treats it as an unsigned value instead of throwing an error. This is fine as it is an error with the data entry, not the function.

5.1.3. ARX32

Category	Input X	Input Y	My Result	Source Code Result	Passed Test
Normal	{0x326342ac ,0x437c38e5 ,0x4747c11 ,0x31572d8e}	{0x1c8622c2 ,0x51ed1ff5 ,0x27112628 ,0x3db8224d}	{0x55e5a035 ,0x845cbefa ,0x4380f93d ,0xd5fa05b5}	{0x55e5a035 ,0x845cbefa ,0x4380f93d ,0xd5fa05b5}	Yes
Normal	{0x61337741 ,0x2daf1bf4 ,0x1502150a ,0x4c7349a2}	{0x4a904832 ,0x6b5741a9 ,0x56aa0e06 ,0x1e840eb}	{0x52052df1 ,0x9484a027 ,0x5699b477 ,0x16dbf3c5}	{0x52052df1 ,0x9484a027 ,0x5699b477 ,0x16dbf3c5}	Yes
Extreme	{0xffffffff ,0xffffffff ,0xffffffff ,0xffffffff}	{0xffffffff ,0xffffffff ,0xffffffff ,0xffffffff}	{0x52296b41 ,0xfba6740c ,0x2a2c7589 ,0x256aa50}	{0x52296b41 ,0xfba6740c ,0x2a2c7589 ,0x256aa50}	Yes
Extreme	{0x00000000 ,0x00000000 ,0x00000000 ,0x00000000}	{0x00000000 ,0x00000000 ,0x00000000 ,0x00000000}	{0xf1b3ccb9 ,0x61340034 ,0x2cae6451 ,0xa85ea1f0}	{0xf1b3ccb9 ,0x61340034 ,0x2cae6451 ,0xa85ea1f0}	Yes

5.1.4. Tuple copy

Category	Xi	X	Yi	Y	Expected Result	Actual Result	Passed Test
Normal	0	{0,0,0,0}	0	{1,2,3,4}	{1,2,3,4}	{1,2,3,4}	Yes
Normal	4	{0,0,0,0, 0,0,0,0}	0	{1,2,3,4}	{0,0,0,0, 1,2,3,4}	{0,0,0,0,1, 2,3,4}	Yes
Normal	0	{0x61337741 ,0x2daf1bf4 ,0x1502150a ,0x4c7349a2 }	0	{0x4a904832 ,0x6b5741a9 ,0x56aa0e06 ,0x1e840eb}	{0x4a904832 ,0x6b5741a9 ,0x56aa0e06 ,0x1e840eb}	{0x4a904832 ,0x6b5741a9 ,0x56aa0e06 ,0x1e840eb}	Yes

5.2. Pi function

5.2.1. E1

Category	Internal State (IS)	Constant Tuple (C)	My Result (J)	Source Code Result	Passed Test
Normal	{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}	**(C1):** {0x8D8B8778, 0x7472716C, 0x6A696665, 0x635C5A59}	{0xda70c8a1, 0xff97e47c, 0x56e4cd98, 0x830195e4, 0x11d287e9, 0x5f018e90, 0xe5a954f1, 0x3b3a8d50, 0x81e2fdd5, 0xc85da7ad, 0x9842dd3f, 0x55a234b8, 0x473833b4, 0xf848d932, 0x4baccbf8, 0x611ef882}	{0xda70c8a1, 0xff97e47c, 0x56e4cd98, 0x830195e4, 0x11d287e9, 0x5f018e90, 0xe5a954f1, 0x3b3a8d50, 0x81e2fdd5, 0xc85da7ad, 0x9842dd3f, 0x55a234b8, 0x473833b4, 0xf848d932, 0x4baccbf8, 0x611ef882}	Yes

5.2.2. E2

Category	Internal State (J)	Constant Tuple (C)	My Result (IS)	Source Code Result	Passed Test
Normal	{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}	**(C2):** {0x5655534E, 0x4D4B473C, 0x3A393635, 0x332E2D2B}	{0xa0c1b67b, 0x89e49565, 0x8ce0fa25, 0xf8aef2fd, 0x3adac3d2, 0x477b1edb, 0xad75bc28, 0xb7788a4e, 0xd390fa5a, 0xd3fcff1a, 0x40404131, 0xffce2e86, 0x4264af11, 0x8141faa4, 0x630eaddb, 0x6f85b1bf}	{0xda70c8a1, 0xff97e47c, 0x56e4cd98, 0x830195e4, 0x11d287e9, 0x5f018e90, 0xe5a954f1, 0x3b3a8d50, 0x81e2fdd5, 0xc85da7ad, 0x9842dd3f, 0x55a234b8, 0x473833b4, 0xf848d932, 0x4baccbf8, 0x611ef882}	Yes

5.2.3. Pi

Category	Internal State (IS)	Rounds (R)	My Result	Source Code Result	Passed Test
Normal	{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}	1	{0x31ea9a67, 0x4e1b9760, 0x9ef4f9f2, 0x30e40698, 0xc1366c64, 0xafa6f0c, 0xe2ae4559, 0x41141ad5, 0xfcce8518, 0xd101e05, 0xf389410, 0x3a2ba767, 0xb5fda9c5, 0xd09e7eb0, 0x9e082b93, 0x3c61067a}	{0x31ea9a67, 0x4e1b9760, 0x9ef4f9f2, 0x30e40698, 0xc1366c64, 0xafa6f0c, 0xe2ae4559, 0x41141ad5, 0xfcce8518, 0xd101e05, 0xf389410, 0x3a2ba767, 0xb5fda9c5, 0xd09e7eb0, 0x9e082b93, 0x3c61067a}	Yes
Normal	{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}	3	{0x30168653, 0xb4d98660, 0x6ce7556b, 0x39f3b56b, 0x96cb1ebe, 0x8e85a25b, 0xbc9a6837, 0x4b3fe212, 0x689084f3, 0x58f5eb37, 0x22acb87, 0x919318ad, 0xff7a19d1, 0x3b903ba6, 0x3fc98bc5, 0x3692e2a5}	{0x30168653, 0xb4d98660, 0x6ce7556b, 0x39f3b56b, 0x96cb1ebe, 0x8e85a25b, 0xbc9a6837, 0x4b3fe212, 0x689084f3, 0x58f5eb37, 0x22acb87, 0x919318ad, 0xff7a19d1, 0x3b903ba6, 0x3fc98bc5, 0x3692e2a5}	Yes
Extreme	{0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x00000000}	3	{0xa20274db, 0xc7c4754e, 0x408da220, 0x99b9bb6d, 0x50117c24, 0x2dd776fa, 0xbb5c7070, 0x5c8ef357, 0x2ec52311, 0xcd730d65, 0xe0c17161, 0x94308a24, 0x6a3b5e3c, 0x4120c4a4, 0x94061eed, 0x82493eba}	{0xa20274db, 0xc7c4754e, 0x408da220, 0x99b9bb6d, 0x50117c24, 0x2dd776fa, 0xbb5c7070, 0x5c8ef357, 0x2ec52311, 0xcd730d65, 0xe0c17161, 0x94308a24, 0x6a3b5e3c, 0x4120c4a4, 0x94061eed, 0x82493eba}	Yes

Category	Internal State (IS)	Rounds (R)	My Result	Source Code Result	Passed Test
Exception	{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}	0	Error (all values unchanged)	{0x44dec319, 0x349fe654, 0x63d19355, 0x4a29f6f7, 0x83b51650, 0xac97b7dd, 0xd7162dd6, 0xc9c72497, 0xe665e415, 0x4428679a, 0xb335a402, 0xb1bf1ee9, 0x602fc801, 0xde5fd692, 0xe3144f18, 0x8cd3c1c2}	Yes
Exception	{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}	-1	Error (all values unchanged)	{0x3abd8551, 0xa1deadb2, 0xb3aa3370, 0xe8b87cbf, 0xd5da56d6, 0x7f62ae30, 0x8baed8f3, 0xe93eeb8f, 0x7704a654, 0xe3464bb3, 0x52b3bf01, 0x678036fa, 0x2a82dcd0, 0xe6fe5375, 0xecea0a3e, 0x64c81c75}	

6. Evaluation

6.1. Avalanche Analysis

In the specification the first function they analysed was avalanche analysis of the ARX function. They gained their results by generating 10,000 random values of X and Y to act as the input, then for each graph they made 10,000 altered values of X' and Y' respectively, changing one bit. The position of each bit was incremented each time (i.e. the first value of X' changed the first bit, the second value of X' changed the second bit and so on) looping round back to the start after they reached the 128th bit of the input X. This means that each value was tested multiple times, and an average between the output values for each bit was obtained before being plotted on the graph. The values displayed on the graph show the average hamming distance between the outputs for each bit changed.

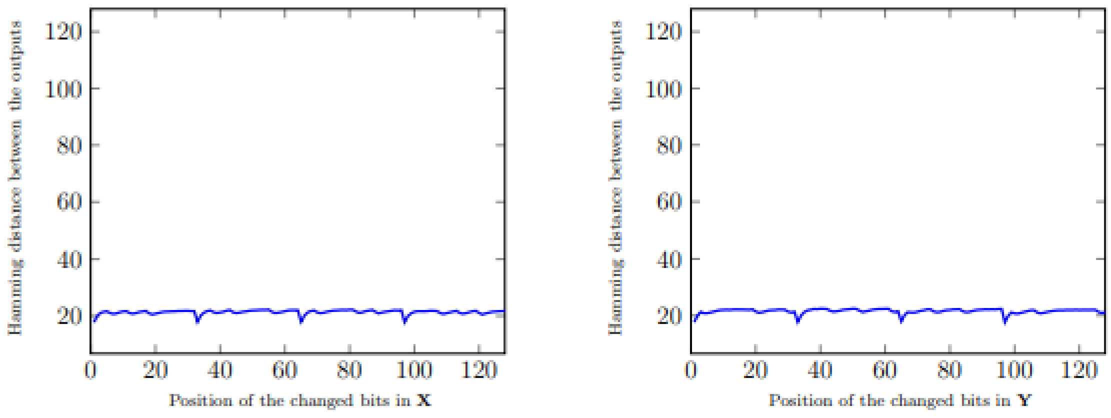


Figure 14 Avalanche effect of the ARX operation for ω=32 [2]

Next the generated 1,000 random values of the IS and respective IS' values were generated according to the same rules as before. The results also show the average hamming distance between pi(IS) and pi(IS'), over one round of pi.

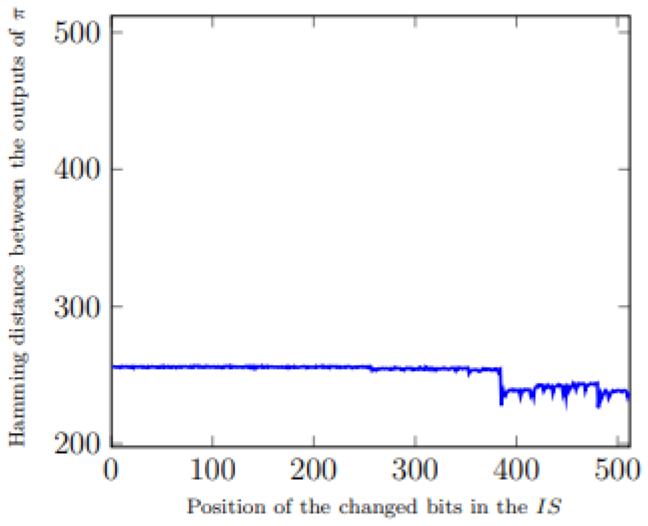


Figure 15 Avalanche effect of one round of the π function where ω=32 [2]

Now I will create my own graphs to compare to those provided in the specification.

6.1.1. X’

Here I am generating 30,000 random values of X and Y and measuring the difference between X and X’. All the values for each bit number are averaged out before being entered into the graph to show the traits of the algorithm.

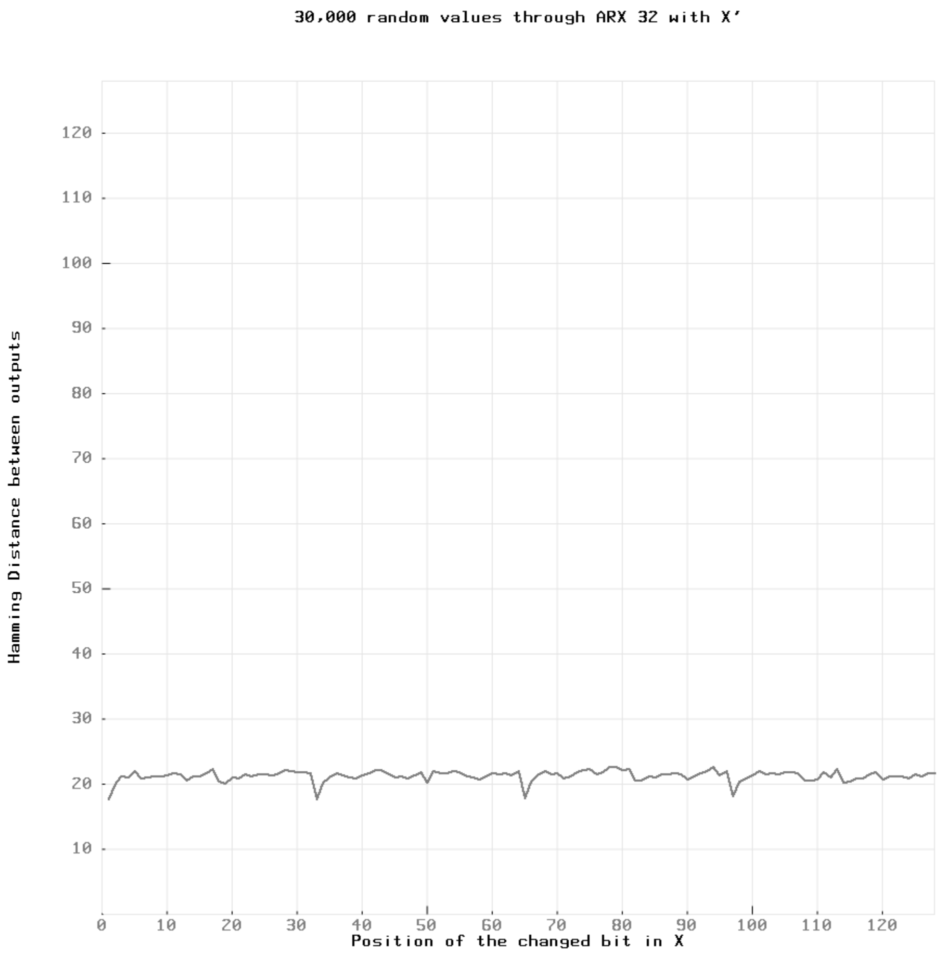


Figure 16

6.1.2. Results

The results are very similar to those in the specification. They show a steady line across entire X tuple with small dips around the 1st, 32rd, 64th, and 96th bits, which are at the bit positions at the start of each of the 4 words in the X tuple. This means it can be predicted that if you change the first bit of a word in X, it will change fewer bits in the output over the ARX algorithm.

(Min = 17.564102, **Average = 20.781250**, Max = 22.948717)

6.1.3. Y’

Here I conducted the same experiment over Y and recording the difference between Y and Y’.

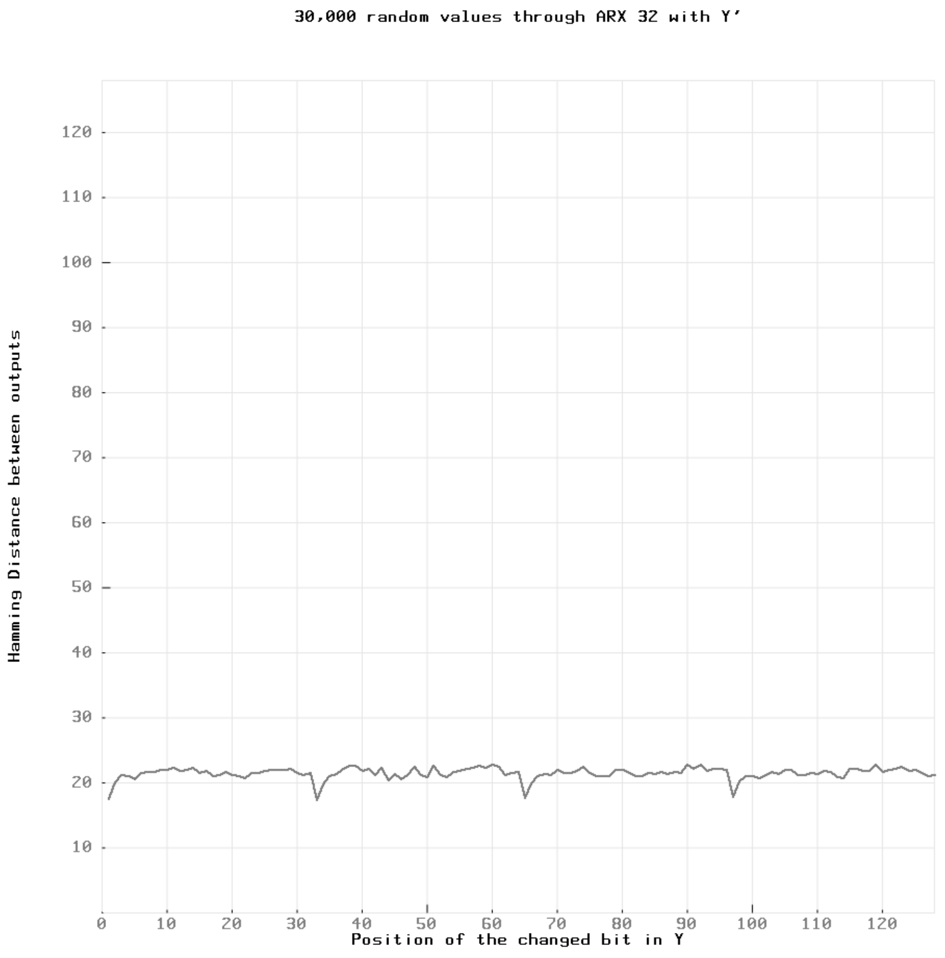


Figure 17

6.1.4. Results

The results here look very similar, showing the same dips at the start of each word, and very similar Averages, minimums, and maximums. This is good as it shows it is hard to tell whether a bit has been changed in the X tuple or the Y tuple, over the ARX algorithm, meaning they are difficult to distinguish.

(Min = 17.294872, **Average = 21.007812**, Max = 23.209402)

6.1.5. Ten Thousand values over one round of Pi

Here I generated 10,000 random Internal states and ran them through one round of Pi to measure the distribution.

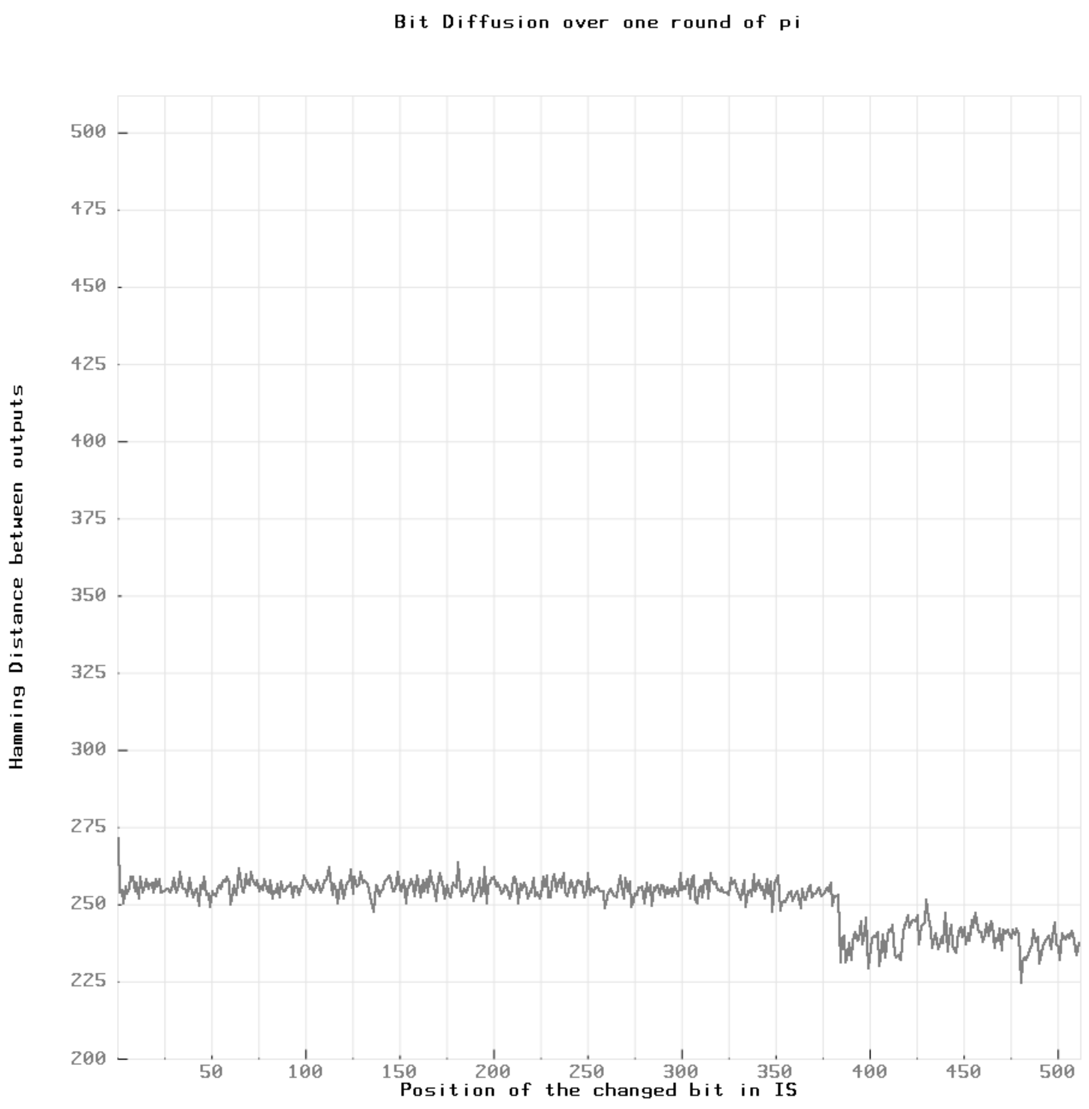


Figure 18

6.1.6. Results

The Results for this test show much higher average hamming distance at around 250 bits or half the total bits, which is an ideal level of entropy as previously mentioned. The line is also much flatter with no discernible dips. However, the exceptions are the big spike around the first bit and the last 4 words worth of bits having a consistent drop in average hamming distance, as well as having a more erratic line with greater differences between the local minima and maxima.

(Min = 224.157898, **Average = 250.871094**, Max = 266.894745)

6.1.7. Ten Thousand values over three rounds of Pi

Here I conducted the same experiment, except putting each Internal State through three rounds of pi before measuring the hamming distances between the outputs.

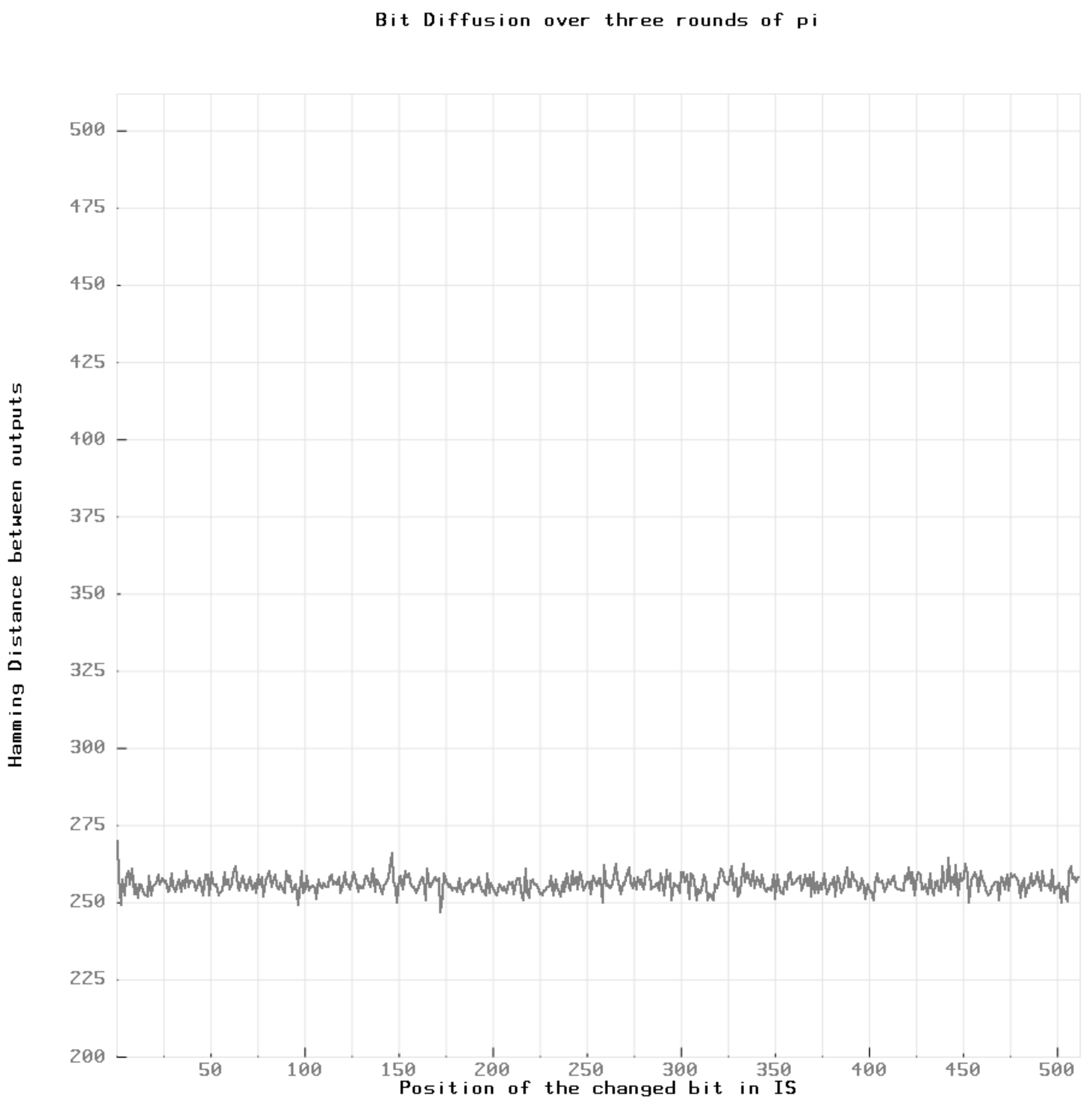


Figure 19

6.1.8. Results

Here you can see the line has become flatter and has no dips on either end of the graph, showing more consistent numbers across the bit range. However, the spike on the first bit remains evident. Also, the average has also increased to 255 bits, which is now above the ideal 250 target which was achieved before; however, I suspect this is purely because of the removal of the dip in the last 4 words which was bringing the average down.

(Min = 247.894730, **Average = 255.681641**, Max = 269.842102)

6.2. Bit Diffusion

Here I change the first bit of X over 1000 test values and measure the probability of each of the bits in the output changing, given that first bit has been changed.

6.2.1. X'

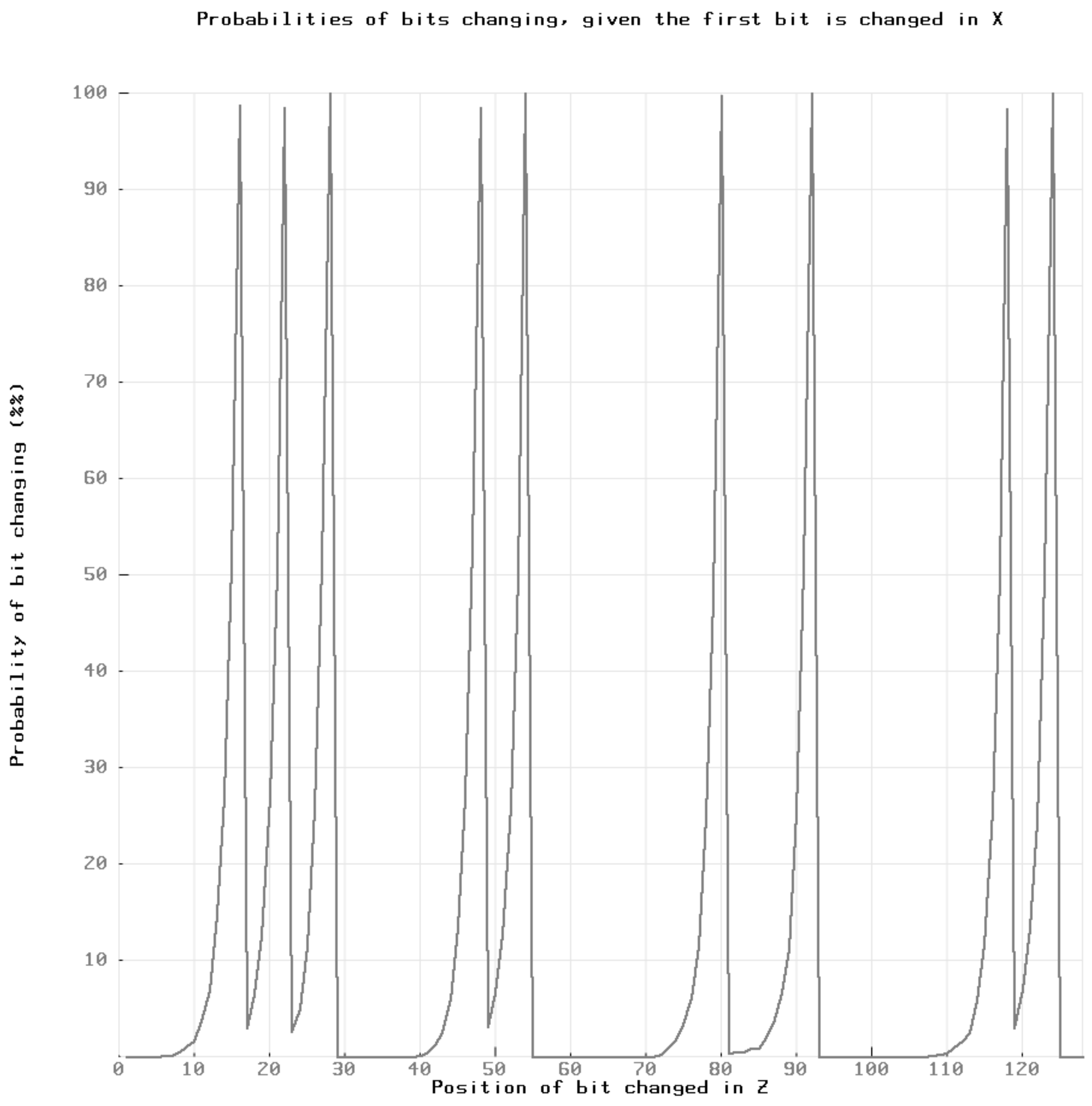


Figure 20

6.2.2. Results

The results here are very interesting, certain bits have a 100% or almost 100% chance of changing. The pattern is very distinct with each spike having a sharp decline in probability each side, but with a more gradual decline on the left side.

6.2.3. ARX Y

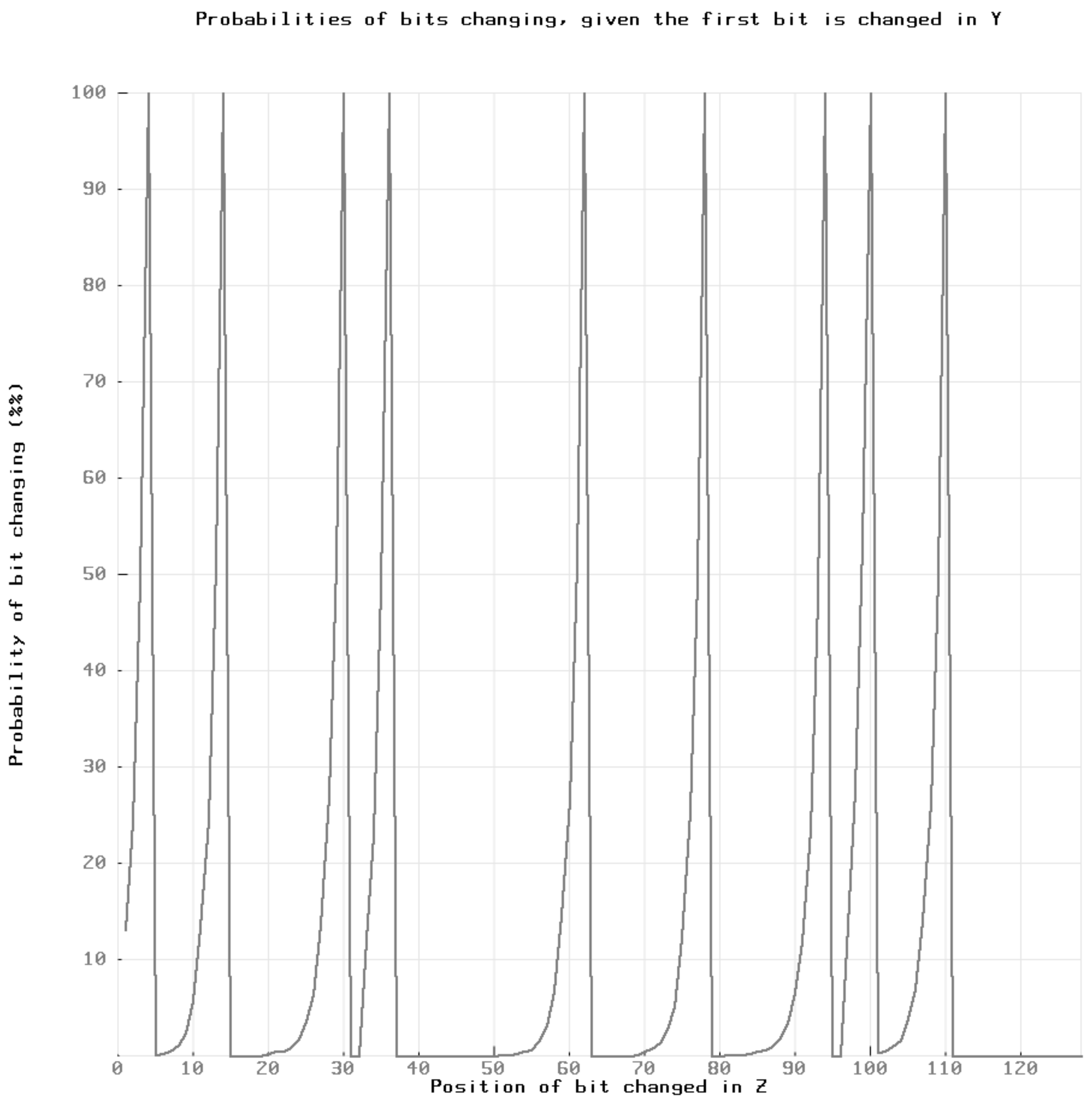


Figure 21

6.2.4. Results

Changing the first bit in Y shows a similar pattern, except with each spike hitting a 100% chance of changing. You can imagine that each bit in X and Y would produce a unique pattern like this that could potentially be used to calculate the input that matches a given output of ARX with the highest probability.

6.2.5. Pi one round

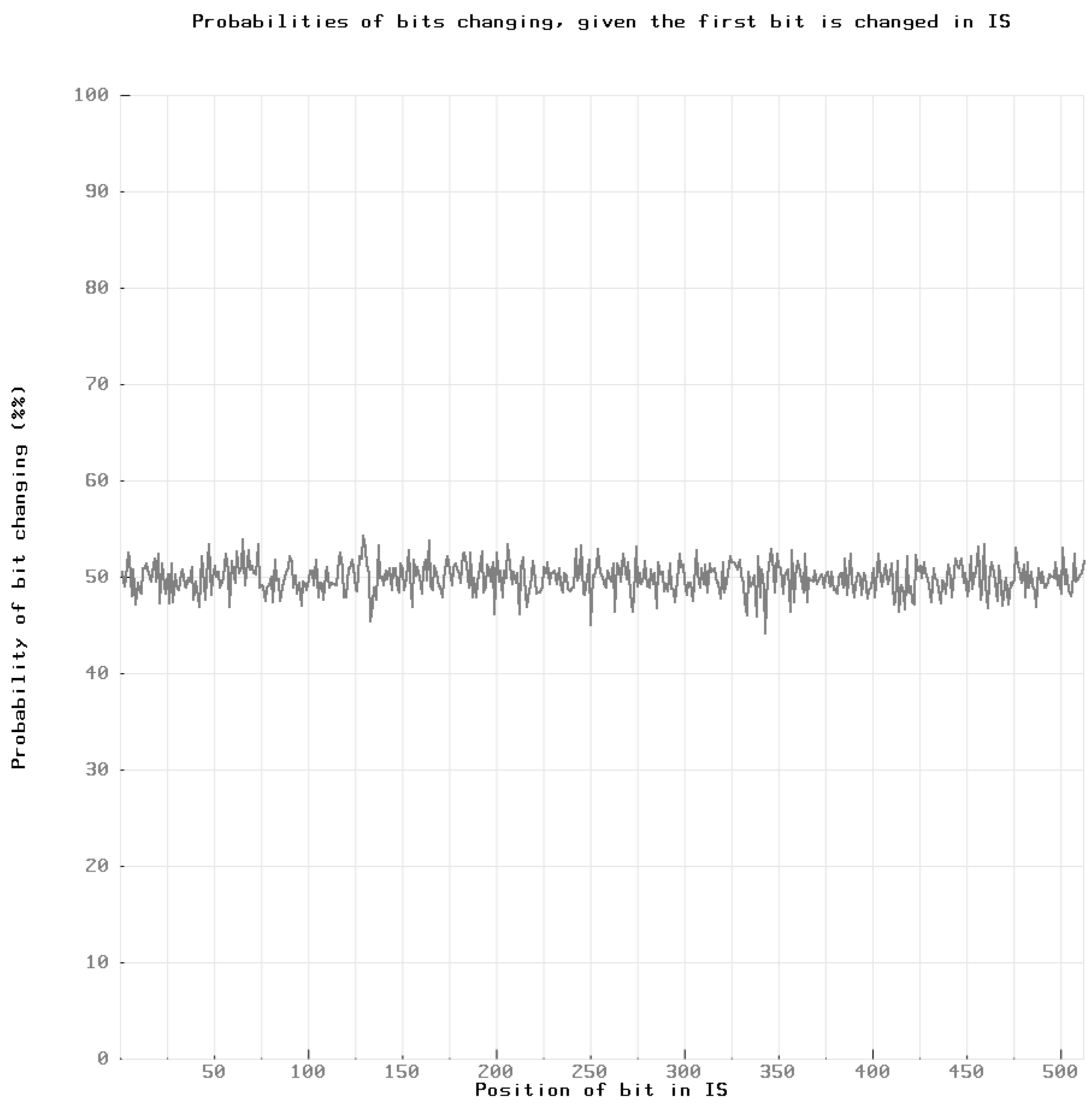


Figure 22 1 Round of Pi

6.2.6. Results

This is one round of pi with the first bit of the IS changed, measuring the probability of each bit in the IS changing, given that first bit is changed. After just one round of pi, the introduction of round constants and repetition of ARX has induced so much diffusion that it is very hard to see any distinguishing pattern in the results.

6.2.7. Pi three rounds

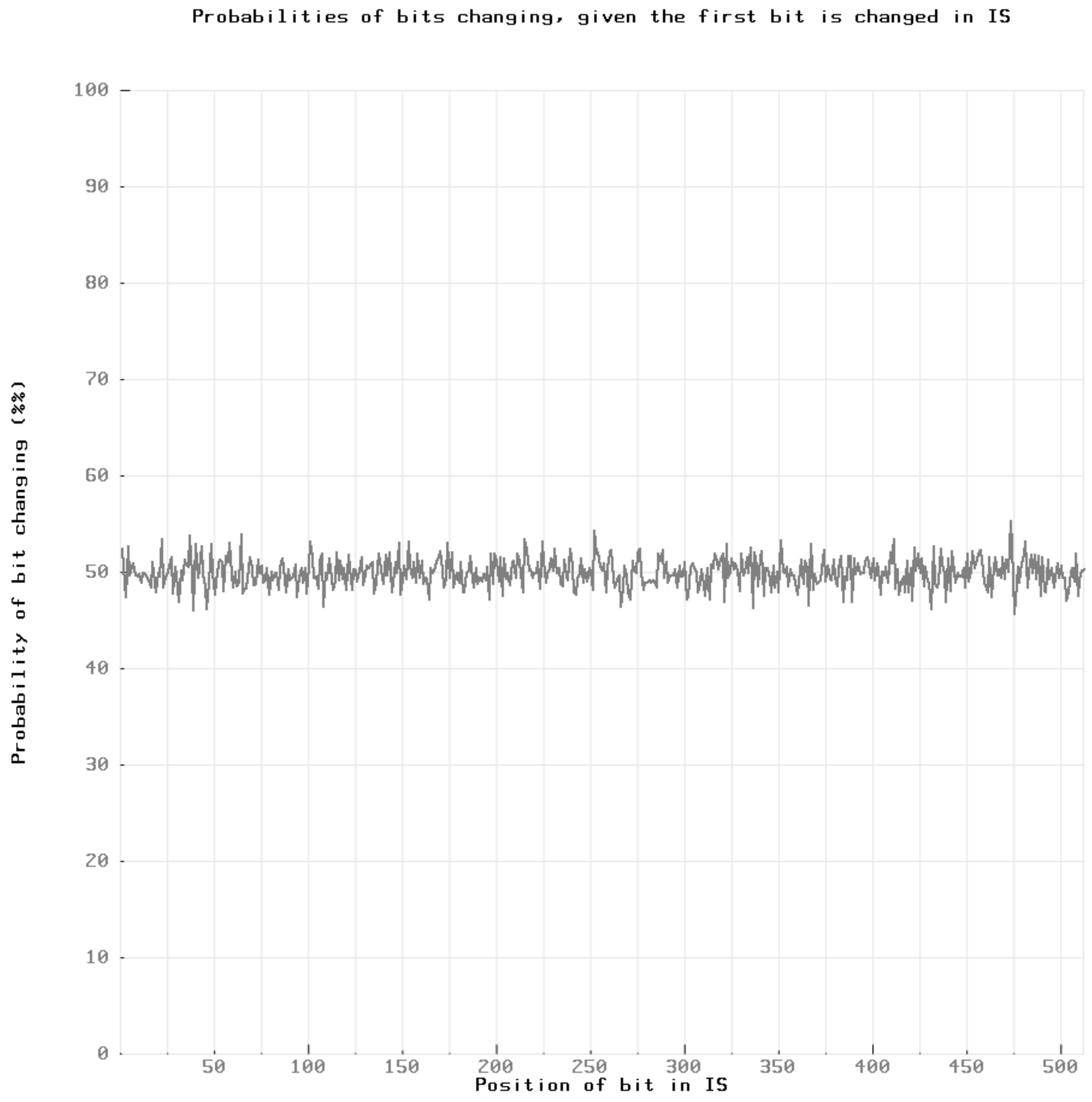


Figure 23 3 Rounds of Pi

6.2.8. Results

As expected, three rounds of the cipher have made it harder to find any distinguishing features. The spikes seem less extreme, except for the one around 475 bits. It is very difficult to tell any distinguishing features from the noise produced by variance in the inputs.

6.3. Differential Analysis

6.3.1. Differential analysis over points of interest

The “points of interest” are the dips that occurred in the hamming distance analysis of ARX, where I noticed the hamming distance between outputs was noticeably less when the first bit of each word was changed. As I have already tested changing the first bit in section 6.2.1, there’s no need to repeat it. Therefore, I have changed the first bit in each of the first two words, three words, and four words respectively to measure how it affects the probability of bits changing.

6.3.2. 2-bits

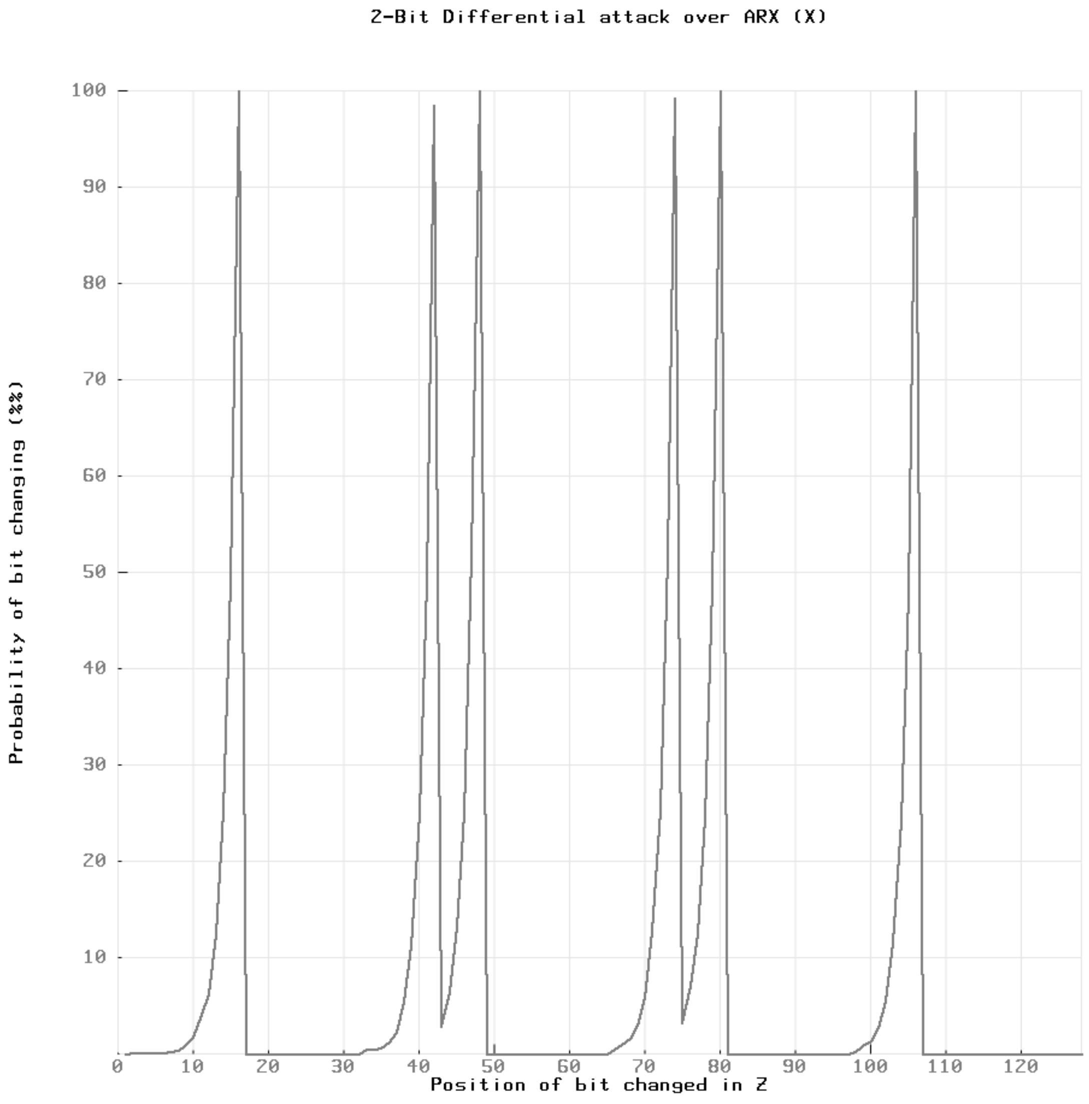


Figure 24

6.3.3. Results

The results here show a lower number of spikes than when just one bit was changed in Figure 22. When one bit was changed, the results showed 9 spikes and here there are only 6 spikes, even though more bits were changed in the input. This means that less bits were be changed in the output on average and is an interesting behaviour of the encryption algorithm that could potentially lead to a distinguisher discovery.

6.3.4. 3-bits

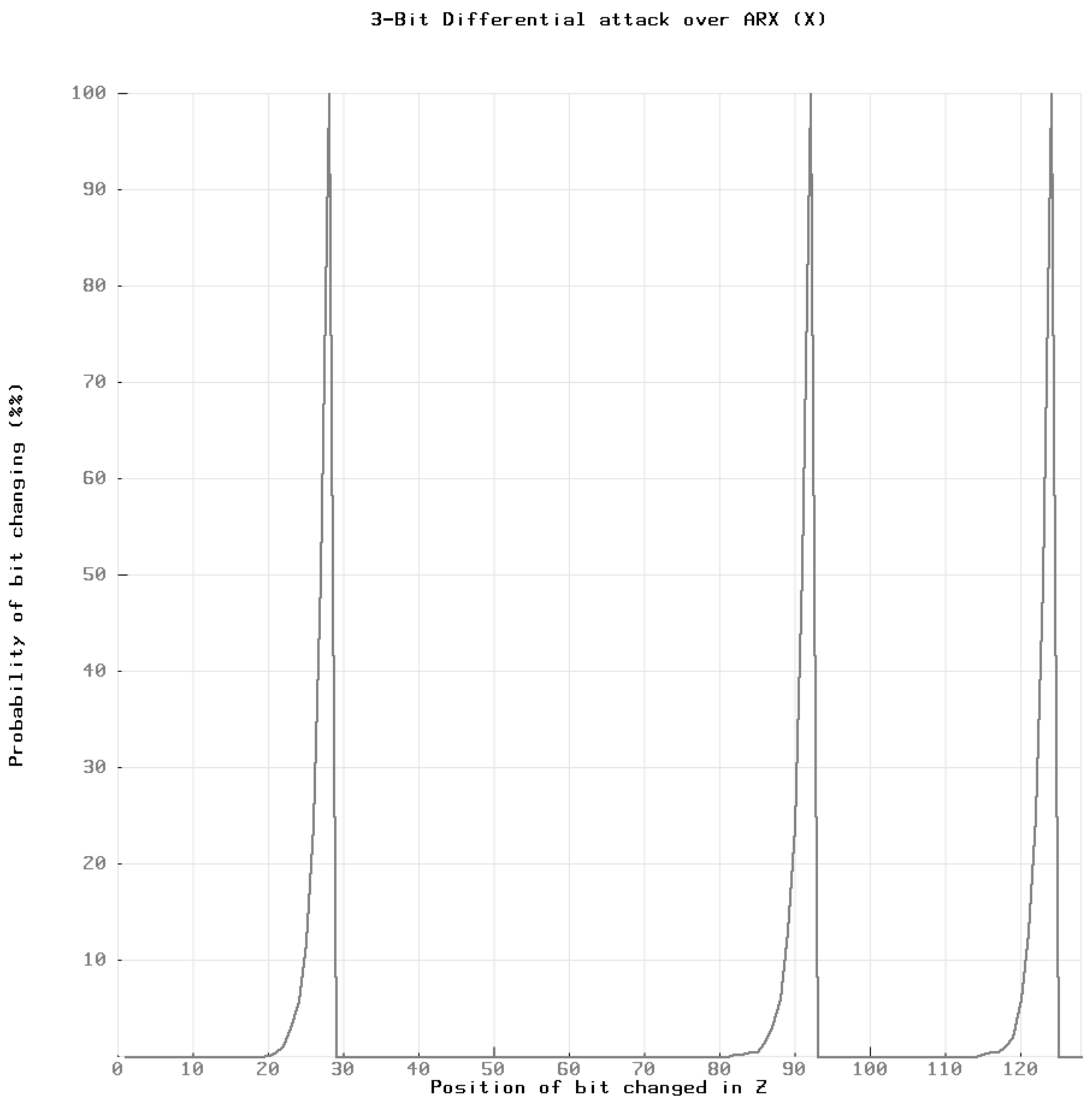


Figure 25

6.3.5. Results

The pattern continues here, where 3 bits were changed in the input, there are only 3 spikes. This shows a clear distinguishing feature of ARX that is non-random.

6.3.6. 4 bits

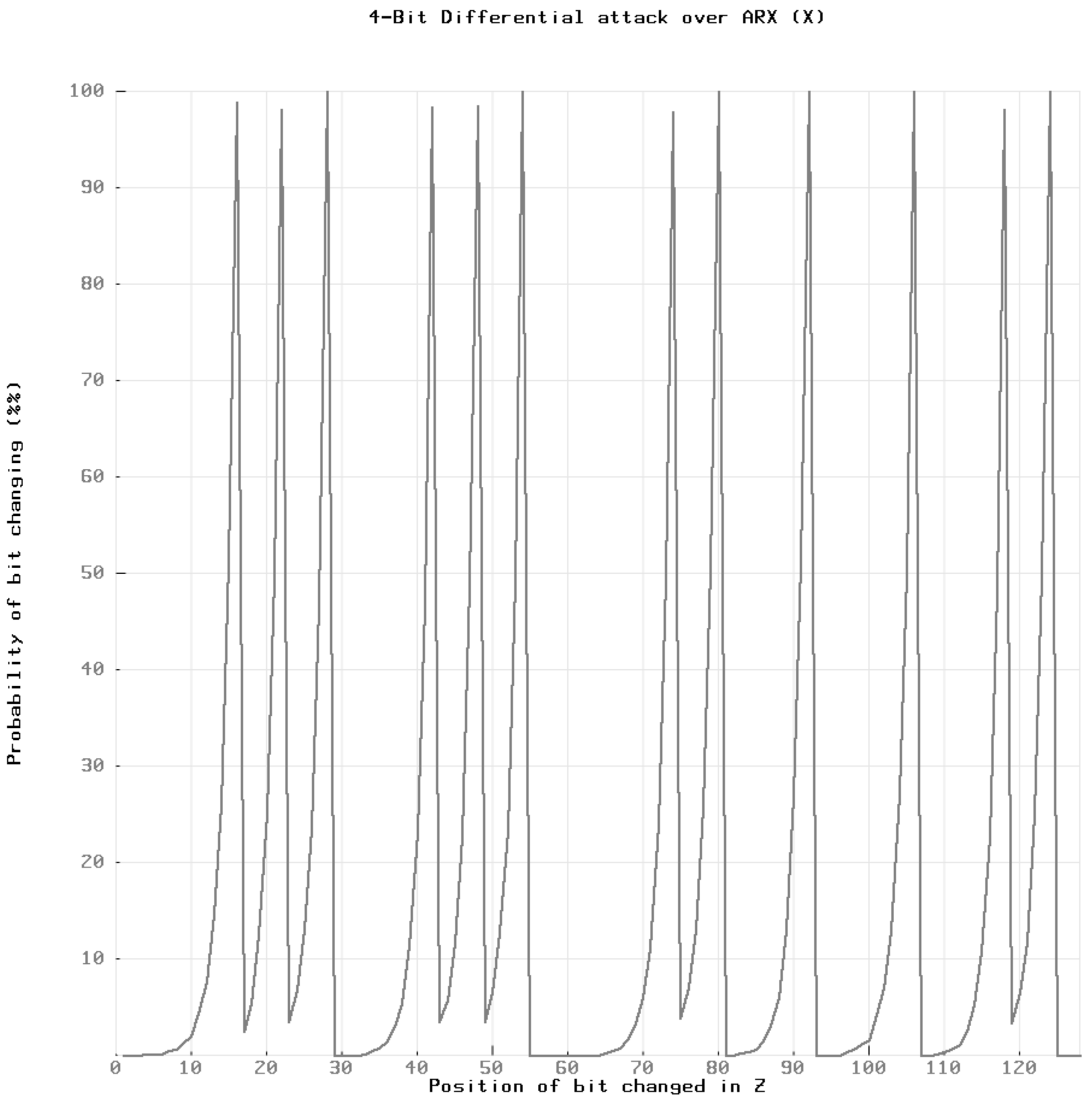


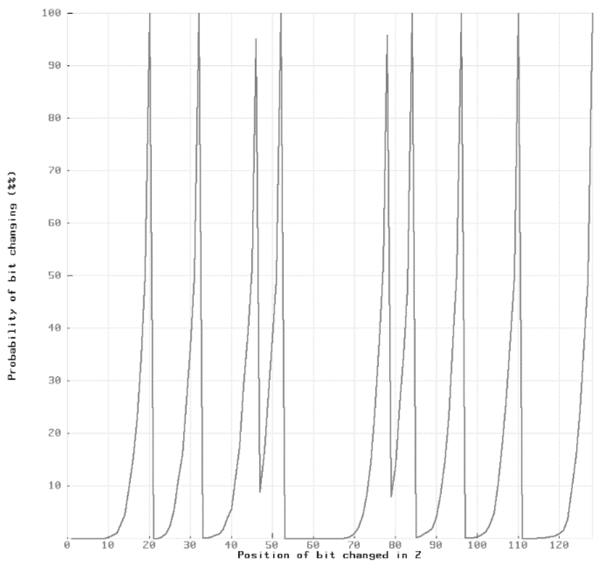
Figure 26

6.3.7. Results

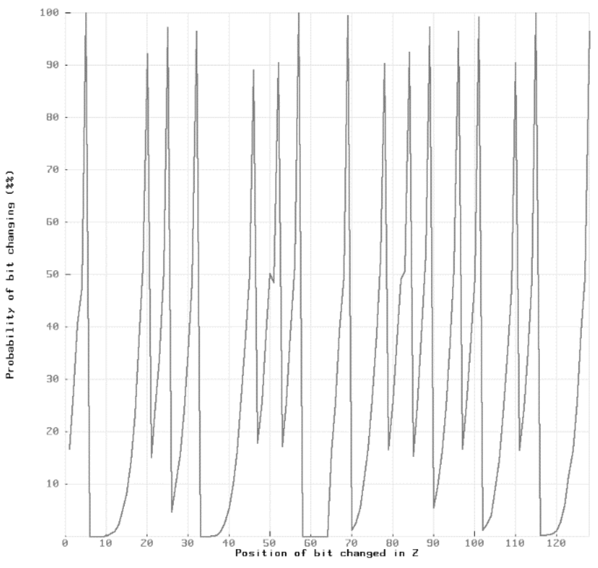
The number of spikes increased again drastically, however the spikes still all reach almost 100% probability of changing. This means that it could potentially be a distinguishing feature of ARX potentially. To contrast these results, I need to compare to a differential analysis where the bits are chosen at random.

6.3.8. Differential analysis over random values

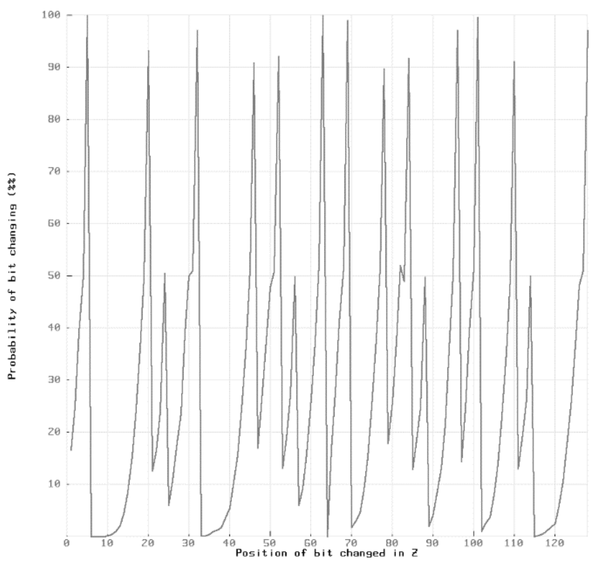
For this experiment, I wanted a control to compare the last values to. To do this I randomly generated 4 values between 0 and 127 one time; those values were: {68, 73, 103, 93}. I then used those values to decide which bits I would change as I increase the number of bits changed in a differential attack, going from left to right. I did this to create a control dataset to compare my previous values to.



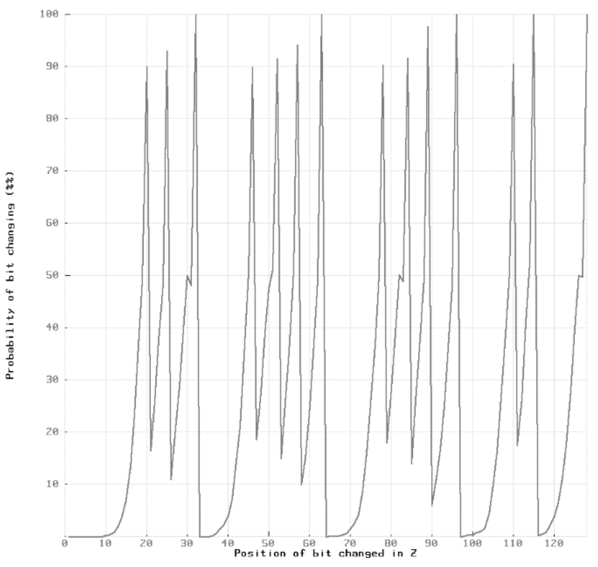
One bit



Two bits



Three bits



Four bits

6.3.9. Results

The results here are very different from the results obtained previously. Some spikes only reach 50% and the number of spikes does not decrease like it did before. This shows that the results we obtained in 6.3.2, 6.3.4, and 6.3.6 are interesting and unique.

6.3.10. Differential Analysis over pi

Here I applied the first bit of each word changes again, changing the first bit of the first two words and three words of the IS over one round of pi.

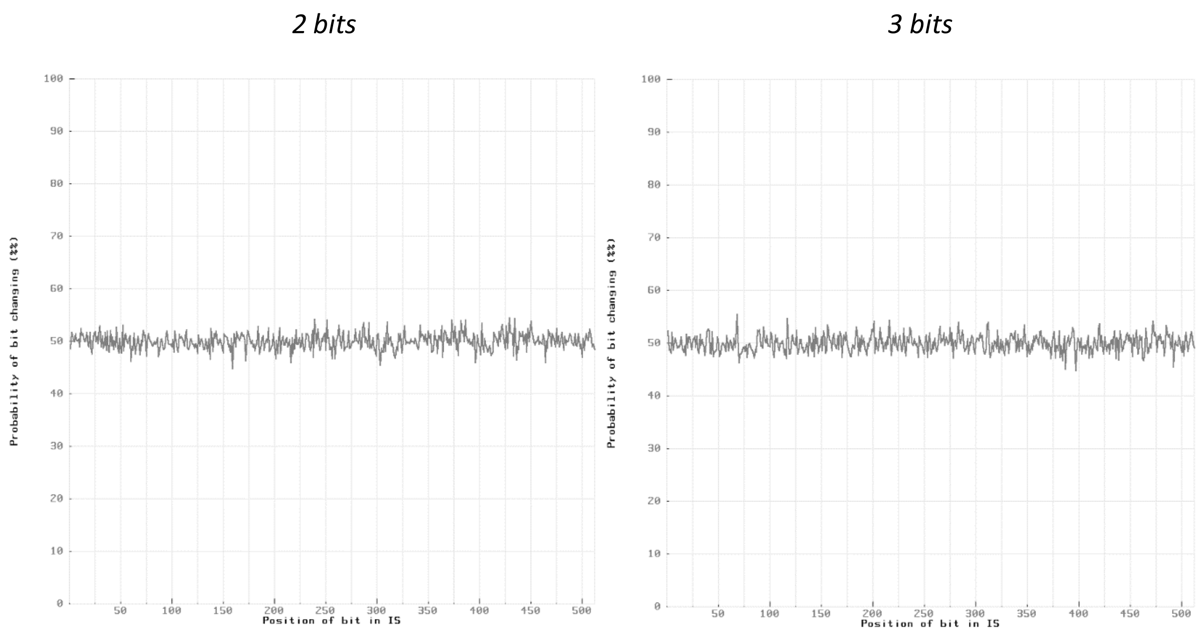


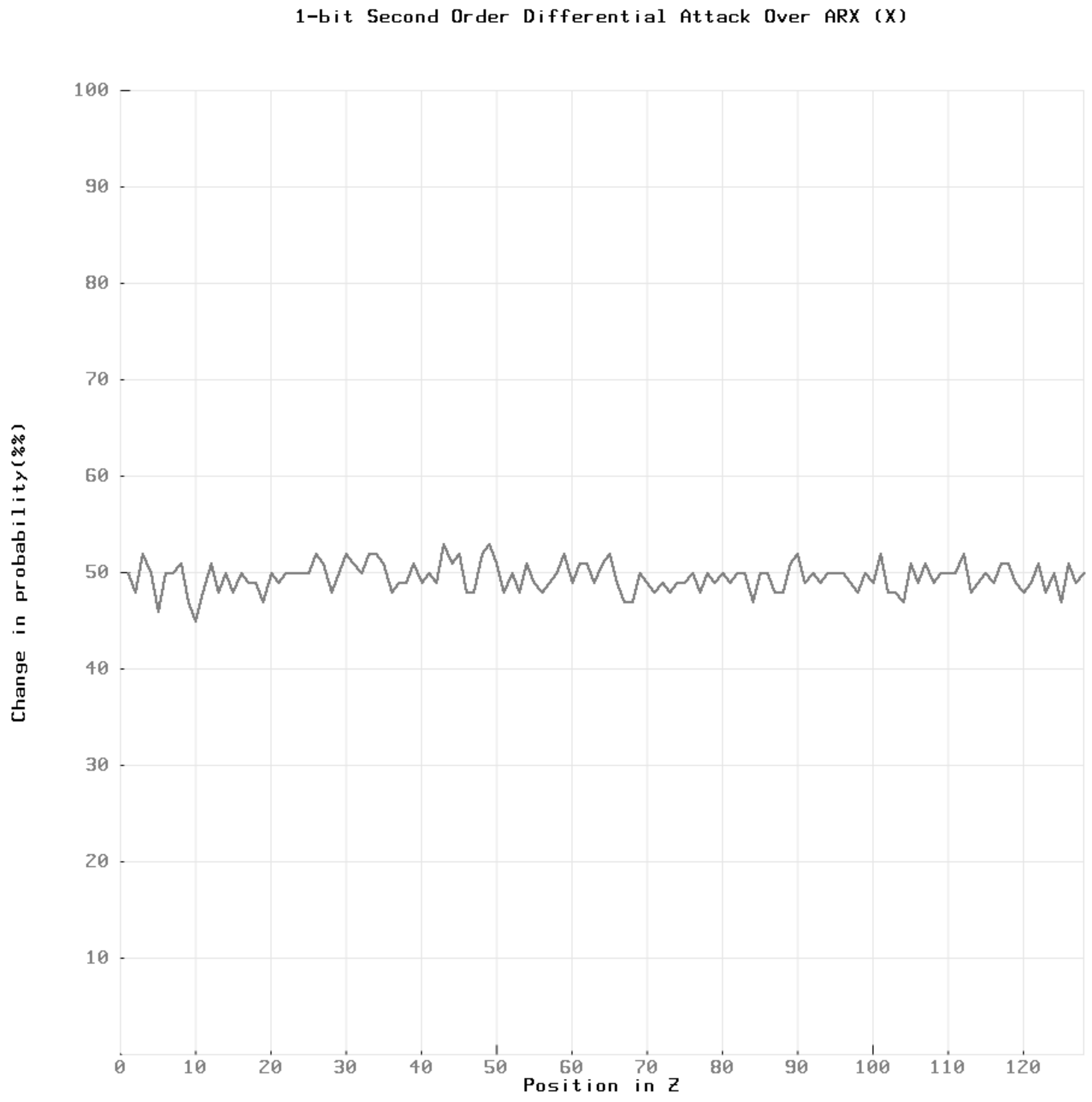
Figure 27

6.3.11. Results

Here the results show certain spikes and dips in probability that are significant, for example the spike at the 70th bit of the IS in the 3-bit analysis. However, all changes reside in the 40-60% range, meaning it's difficult to see any distinguishing feature that couldn't be put down to variance in the input dataset.

6.4. Second-Order Differential Analysis

The derivative of the function we are observing can be defined $g(x) = f(x \oplus a) \oplus f(x)$, where a is some value in the input set $s \in I$, this measures the differences between the differences. Here I changed the first bit of X and measured the change in probability using the new second-order differentiated function.



6.4.1. Results

The results do show some small spikes and dips in change in probability. However, overall, I do not see anything particularly interesting about these results.

6.5. Tag Second-Preimage Attack on Pi-Cipher

In this section I revisit the paper from section 3.5, with the knowledge I have of the cipher and its functionality I have now and analyse how the attack works and what it means for the security of pi-cipher [2].

The attack starts with m lists $L_1, L_2, \dots, L_m$ of n-bit words, where m is the number of blocks in the tag and n is the number of bits in each word. Then each pair of lists is compared and joined using the join operator:

$$L \bowtie_{\tau} L' = \left\{ \left(l \boxplus_8 l', (a, a') \right) \,\middle|\, (l, a) \in L, (l', a') \in L', \mathsf{low}_{\tau}(l \boxplus_8 l') = 0 \right\}.$$

Figure 28 Join operator for birthday attack on pi-cipher [2]

The join operator works by sorting the two lists L and L' by the τ least significant bits and stepping through the lists simultaneously to find values that match their τ lowest bits. The operation $\boxplus_8$ is an adaption from the generalised birthday attack to fit pi-cipher, in the original paper the lists are compared with $\oplus$. The aim is to find values $l_1 \epsilon L_1, \dots, l_m \epsilon L_m$ such that $\oplus_{i=1}^{m} l_m = 0$, but Leurent needed to solve the m-sum problem for the word-wise modulo addition $\boxplus_8$. However, Wagner showed how to solve the generalized birthday problem with modulo addition, and his trick also works for the word-wise modular addition [2]. Leurent also proved that the adapted join ⋈ could be computed efficiently by negating the list L to define $-L = \{(-l, a) | (l, a) \in L\}$, where $-l$ is the additive inverse regarding the word-wise addition i.e.,$-l \boxplus_8 l = 0$. He then would step through -L and L' to find elements where they agree on their τ lowest bits; the corresponding sum of these elements would have the low bits set to 0. Therefore, making this variant of Wagner's algorithm suitable for an attack on pi-cipher.

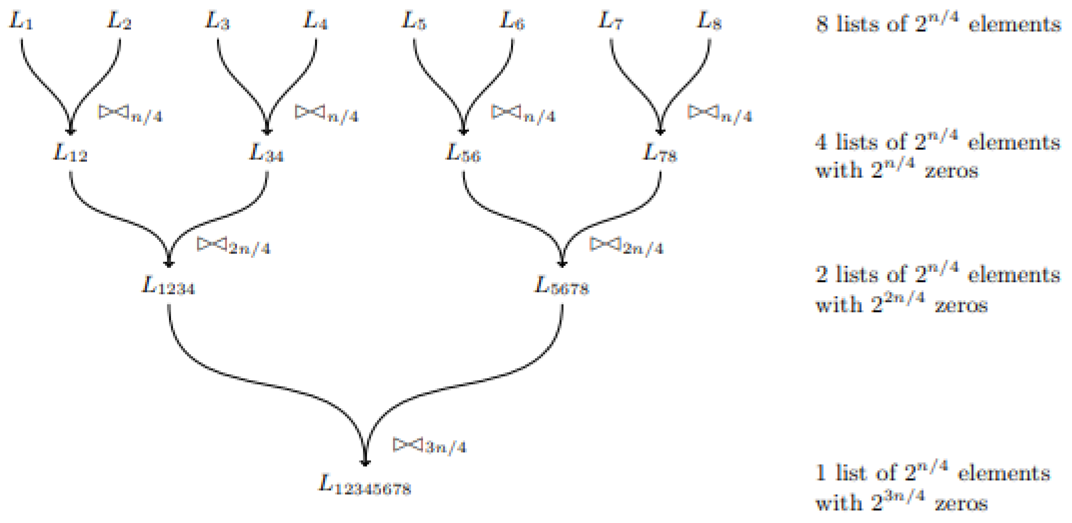
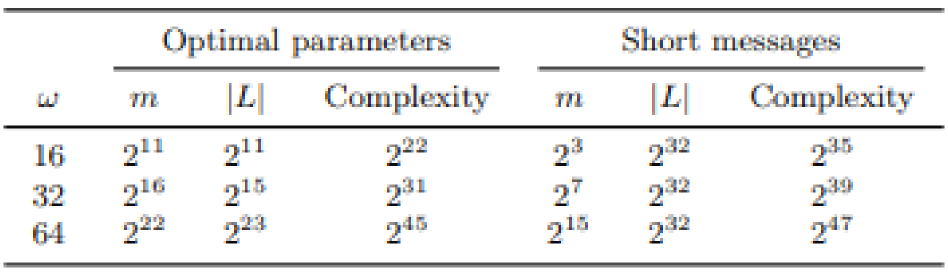

L_1
L_2
L_3
L_4
L_5
L_6
L_7
L_8
$\bowtie_{n/4}$
L_{12}
L_{34}
L_{56}
L_{78}
$\bowtie_{2n/4}$
L_{1234}
L_{5678}
$\bowtie_{3n/4}$
$L_{12345678}$
8 lists of $2^{n/4}$ elements
4 lists of $2^{n/4}$ elements with $2^{n/4}$ zeros
2 lists of $2^{n/4}$ elements with $2^{2n/4}$ zeros
1 list of $2^{n/4}$ elements with $2^{3n/4}$ zeros

Figure 29 An illustration of attack on pi-cipher where ω = 16 [2]

6.5.1. Analysis

To launch an attack against the version of pi-cipher with the smallest word length (ω=16), 8 list of size 2^{32} are needed, giving a total space complexity of 2^{35} bits (approximately 4 Gigabytes), which is feasible for an attacker to have or obtain. It is commonplace that computers have more than 4GB of RAM currently, which means that an attack on this version of the cipher is feasible. As you can see from the figure below, attacks on the encryption other versions of pi-cipher with larger word sizes do become unfeasible due to too large space and time complexity.



	Optimal parameters			Short messages		
ω	m	$\lvert L\rvert$	Complexity	m	$\lvert L\rvert$	Complexity
16	2^{11}	2^{11}	2^{22}	2^{3}	2^{32}	2^{35}
32	2^{16}	2^{15}	2^{31}	2^{7}	2^{32}	2^{39}
64	2^{22}	2^{23}	2^{45}	2^{15}	2^{32}	2^{47}

Figure 30 Space complexity of attack for different values of ω [2]

6.5.2. Response

In the latest version of pi-cipher specification, the creators emphasised that the variant of pi-cipher with 80 bits of security (which listed as one of the options in previous specifications) is considered insecure due to the reported practical speed of many different computing systems (GPUs, super-computers, FPGAs), and therefore they abandoned it in this version of the specification and for any future cryptographic designs [1]. This was directly in response to the tag second-preimage attack that Leurent proved was practical against the lowest security version of the cipher.

7. Altered Analysis

Here I will be adjusting the algorithm and measuring how they affect the security of the cipher using hamming distance analysis.

7.1. Constants

The constants in pi-cipher are given specifically, and they were chosen to have an even distribution of "on" and "off" bits throughout the bit string [1]. Here I will test both random and extreme constant values against the values already obtained in the previous section. The constant values are incorporated through the E1 and E2 functions (which are part of the Pi function) and are not included directly in the ARX function, therefore I will be only testing the pi function with these changes.

7.2. Random Constants

Using the built in "Rand" function I generated 24 values to replace the Constants given in the spec and once again generated 10,000 random Internals States to measure the bit diffusion. Here are the results I obtained below.

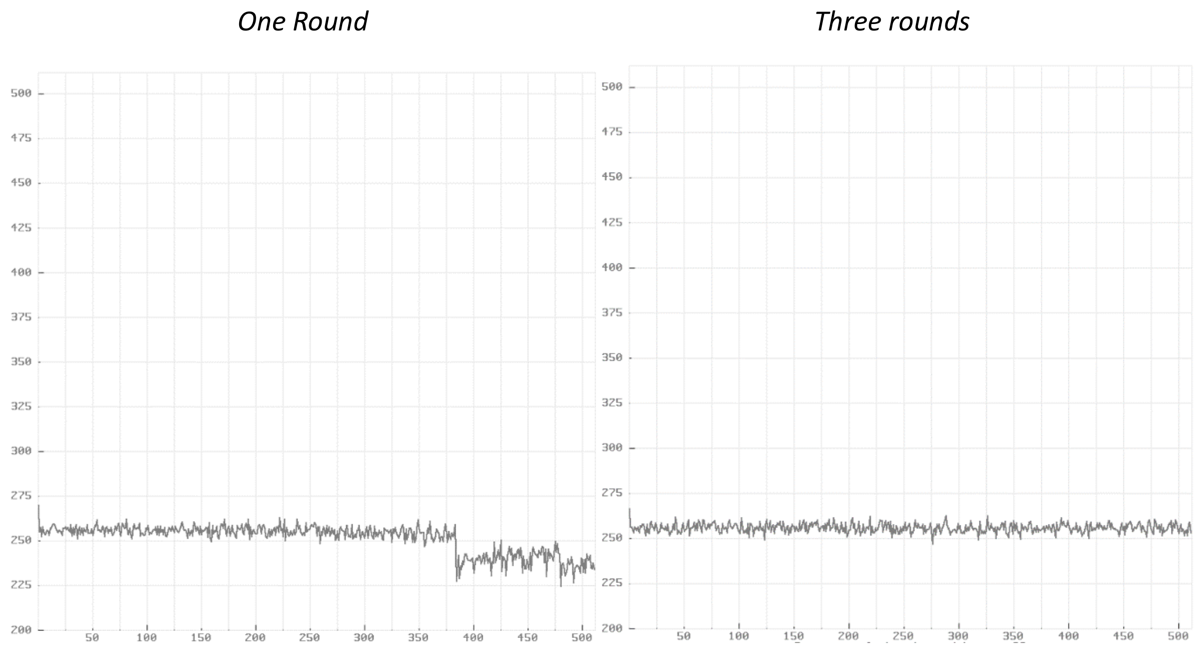


Figure 31 Random constants over 1 round of pi and 3 rounds of pi

Figure 11: (Min = 225.631577, **Average = 251.029297**, Max = 268.736847)

Figure 12: (Min = 248.105270, **Average = 255.628906**, Max = 270.631592)

7.2.1. Results

As you can see here the results are almost indistinguishable from the original, and any distinction that might be made could be put down to the random nature of the test values. Since the values were randomly generated, you might expect that the results to look similar since random numbers will roughly have an even number of on and off bits. What can be learnt from this date, however, is that the algorithm is not as sensitive to change as you might have previously thought.

7.3. Maximised Constants

Here I implement the same test as before but instead of using random values, I use extreme values (0xffffffff) as the constants.

One Round *Three rounds*

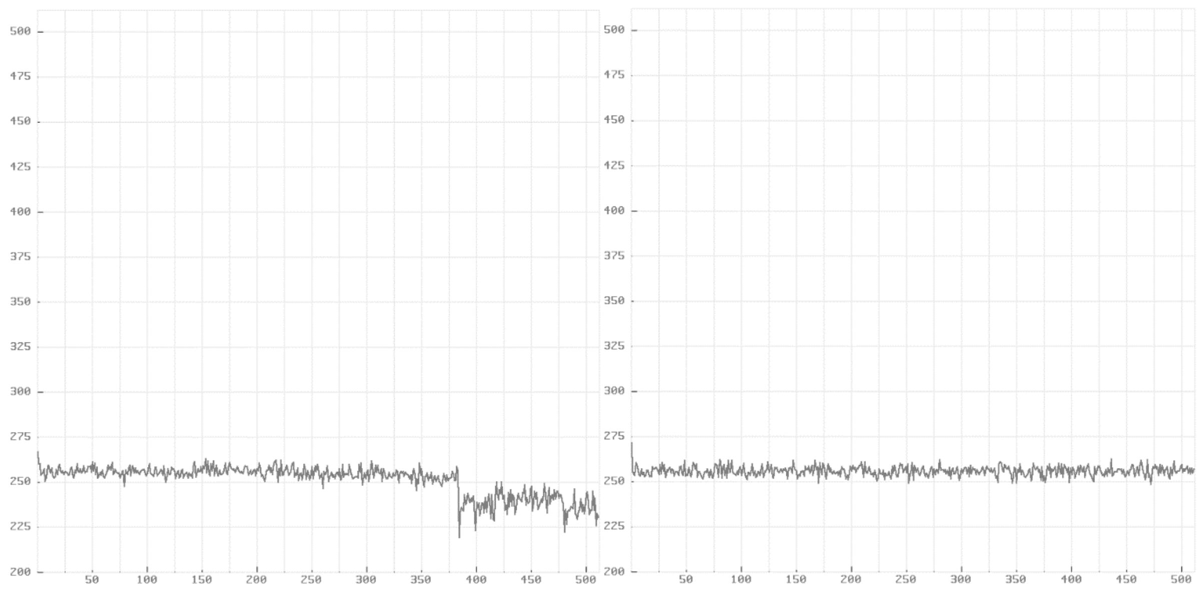


Figure 32 Maximised constants over 1 round of pi and 3 rounds of pi

Figure 13: (Min = 219.157898, **Average = 251.044922**, Max = 271.736847)

Figure 14: (Min = 246.105270, **Average = 255.585938**, Max = 269.052643)

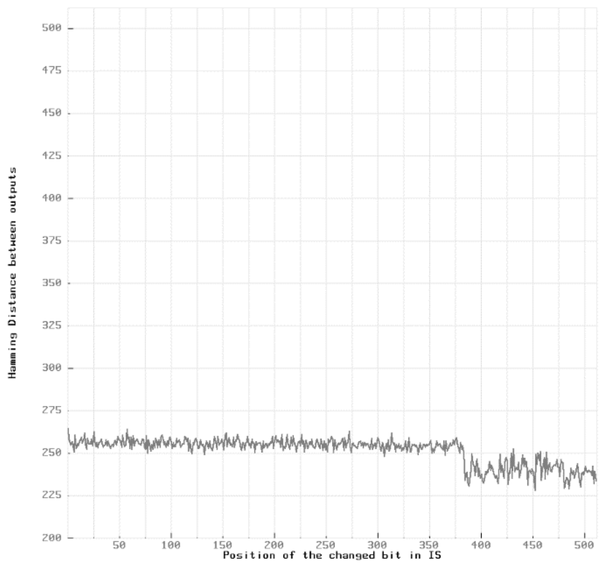
7.3.1. Results

The graphs here show a slight difference, as the maxima and minima are slightly more extreme, and this might make it slightly easier to identify certain non-random patterns in the algorithm. However, it is interesting to see how similar the results are using constants that are all the same maximised value as opposed to values that were specifically chosen to have an even distribution of bits.

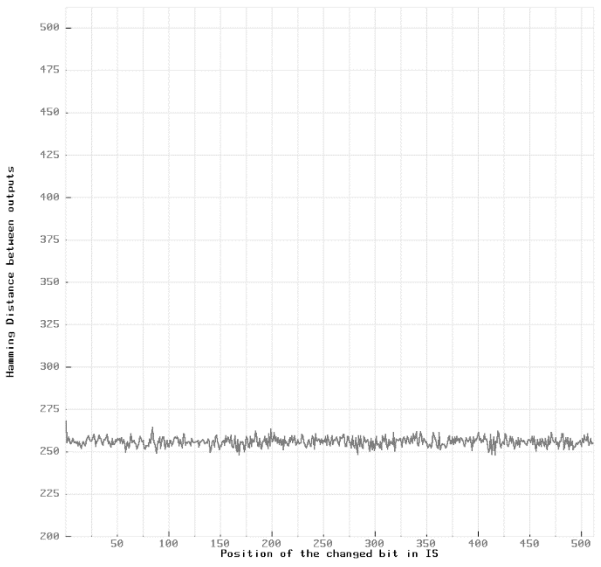
7.4. Rounds of cipher

Here I analysed the change in hamming distances over the different rounds of pi cipher. 1 round and 3 rounds have already been analysed but here I extend that analysis to 2 rounds and 4 rounds (creating my own round constants) to measure how many rounds is most secure.

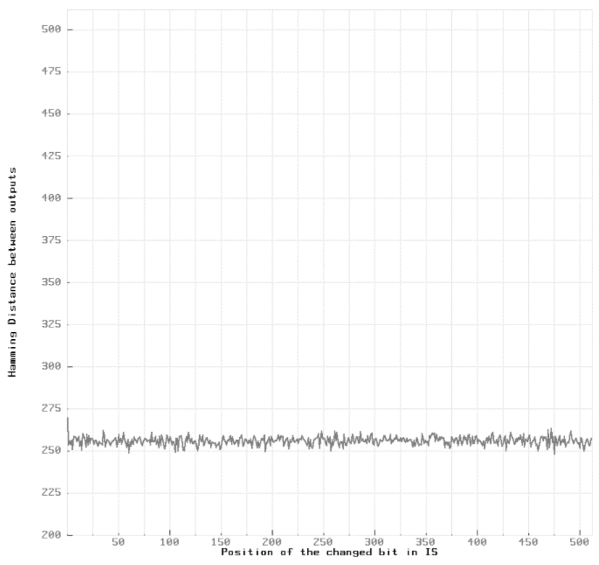
One round



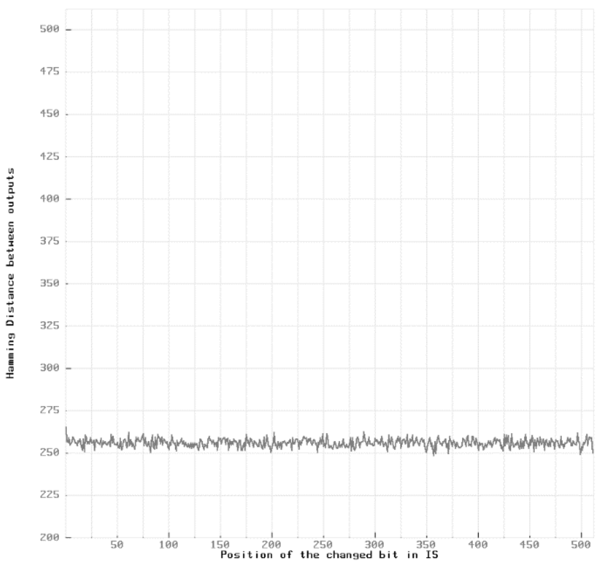
Two rounds



Three rounds



Four Rounds



7.4.1. Results

One round has an obvious flaw in that the last 128 bits have a significantly lower hamming distance, therefore is not suitable to be the standard for the cipher. When two rounds of pi and three rounds of pi are compared, the three round cipher produces a smooth line with less noticeable spikes in hamming distance, meaning it is less distinguishable. Finally, four rounds produce one noticeable advantage over three rounds, which is a smaller spike on the first bit. However, this only affects one bit, and the compute time was noticeably slower even if only by half a second, over a large data encryption it would be significant.. Therefore, I believe 3 rounds of pi to be the optimal number of rounds to keep the cipher lightweight, however adding more rounds of encryption can be done and does not harm the security of the cipher.

8. Conclusion

In conclusion, this variant of pi-cipher provides a high level of confidentiality for the encrypted data. It is difficult to define any distinguishing features of the cipher that have practical applications for launching an attack on the cipher. As the round constants are defined as concrete values in the specification, an insider attack that has knowledge of these constants and the pi methodology could launch an effective attack, however this is outside the requirements of the cipher. The cipher provides some level of tag second-preimage resistance, however the reduced data integrity in some of the variants restricts it from being viable for widespread adoption over alternatives like AES-GCM.

9. References

[1] D. Gligoroski, H. Mihajloska, S. Samardjiska, H. Jacobsen, M. El-Hadedy, R. Erlend Jensen and O. Daniel, “Pi-Cipher,” 12 10 2015. [Online]. Available: http://pi-cipher.org. [Accessed 9 2022].

[2] G. Leurent, “Tag Second-preimage Attack against π-cipher,” HAL, 2014.

[3] C. F. S. L. Farzaneh Abed, “General classification of the authenticated encryption schemes for the CAESAR competition,” *Computer Science Review,* pp. 13-26, November 2016.

[4] CAESAR, “Cryptographic Competitions,” CAESAR, 27 January 2014. [Online]. Available: https://competitions.cr.yp.to/caesar-call.html. [Accessed October 2022].

[5] Z. W. Y. T. Z. L. Fangyong Hou, “Protecting integrity and confidentiality for data communication,” IEEE, 2004.

[6] J. W. N. F. M. B. Stefan Lemsitzer, “Multi-gigabit GCM-AES Architecture Optimized for FPGAs,” *International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 227-238, 2007.

[7] C. E. Shannon, “A Mathematical Theory of Cryptography,” 1945.

[8] S. K. a. W. Meier, “Distinguishing Attack on MAG,” 2005.

[9] M. I. a. M. U. S. Raphael C.-W. Phan, “A Framework for Describing Block Cipher Cryptanalysis,” IEEE, 2006.

[10] X. Lai, “Higher Order Derivatives and Differential Cryptanalysis,” *Communications and Cryptography,* p. 227–233, 1994.

[11] dCode, “Birthday Probabilities,” [Online]. Available: https://www.dcode.fr/birthday-problem. [Accessed January 2023].

[12] D. Wagner, “A Generalized Birthday Problem,” *Annual International Cryptology Conference,* pp. 288-304, 2002.

[13] M. Johansen, “pbPlots repository,” progsbase, 30 May 2022. [Online]. Available: https://github.com/InductiveComputerScience/pbPlots. [Accessed March 2023].