Rheinisch-Westfälische Technische Hochschule Aachen

Lehrstuhl für Informatik 1 Algorithmen & Komplexität
PD Dr. Walter Unger

Bachelor Thesis

Security Aspects And Performance Of A Production Ready Encryption System with Key Generated Operation Selection

Tammo Ippen
Matrikelnr. 281214

March 2011

Supervisor: PD Dr. Walter Unger

Second Lecturer: Prof. Dr. Ulrike Meyer

Abstract

Comparing encryption algorithms at an abstract level, they all have a well designed, but fixed computation graph and they use the key and the plaintext data solely as input to this graph. This paper introduces a new idea to make the computation graph dependent from the key, in other words two different input keys lead to two different encryption algorithm or computation graphs.

Contents

1. Introduction **3**
1.1. State of the Art 4

2. Algorithms **6**
2.1. Design Space 6
2.2. Operations 7
2.2.1. Addition 7
2.2.2. XOR 7
2.2.3. Rotation 7
2.2.4. Negation 8
2.2.5. Multiplication 8
2.3. Permutation 9
2.3.1. Address Permutation 9
2.3.2. Block Permutation 9
2.4. Feistel Network 9
2.4.1. F-Box 10
2.4.2. E-Function 10
2.4.3. Multiply-With-Carry (MWC) 11
2.5. Possible Conjunctions 12
2.6. KOOS – The Final Algorithm 12
2.6.1. The Parameter 13
2.6.2. Encrypt and Decrypt 13
2.6.3. Key Extension 14

3. Analysis **18**
3.1. Statistical Test Suite 18
3.1.1. Sets of Data 18
3.1.2. STS Tests 19
3.1.3. Results 22
3.2. Performance 25

4. Conclusion and Further Researches **28**

A. Appendix **29**
A.1. Extract Parameter 29
A.2. Encryption and Decryption 30
A.3. Detailed Analysis Charts 33

1. Introduction

A new kind of symmetric encryption algorithm is developed. A symmetric encryption algorithm uses one secret key to encrypt and decrypt data or messages. They are the first developed cryptographic algorithms, starting with hieroglyphics and Atbash. Today, they are of fundamental importance when it comes to secret or private information excange in a private, business, political or military environment. They are designed to be fast and unbreakable – an attempt to break it should at least last several decades or centuries.

The current industry standard is the *Rijndael* cypher, better known as *Advanced Encryption Standard (AES)* [1]. This encryption algorithm won over 14 other candidates during the standardisation process in October 2000.

During the standardisation process, four additional encryption algorithms were announced to be finalists – *MARS, RC6, Serpent* and *Twofish.* They all succeeded in the first round of the process and are meant to be equally strong compared to *Rijndael* [10]. A short description of these cyphers follows in Section 1.1.

Comparing these encryption algorithms at an abstract level, they all have a well designed, but fixed computation graph and they use the key and the plaintext data solely as input to this graph. This paper introduces a new idea to make the computation graph dependent from the key, in other words two different input keys lead to two different encryption algorithm or computation graphs.

There exist good reasons why one should develop new, improved encryption algorithms. On the one hand, not all of the current encryption algorithms are free of suspicious behavior, such that vulnerability attacks are possible or that the algorithm even contains some kind of backdoor. For example, the former encryption standard *DES* had suspicious security holes in the form of a backdoor regarding its *S-boxes* and the current standard can be specified by a continuous fraction, which might be a hook for an efficient attack. Strengthening the confidence in the encryption algorithm is crucial, if you want the algorithm to be used.

On the other hand, computation power increased rapidly over the past years and it is very likely, that computation power increases further in the next years. The security margin of 128-bit / 192-bit / 256-bit keys might be just as secure in the future, as the 56-bit key margin of *DES* is now. So easy adoption to new security margins is essential, if the algorithm is to be used for a long time.

The ambitions for this encryption algorithm are, that it is at least as secure as the five AES finalists and that it is fast enough and easy enough to implement, to be actually used in future applications. Furthermore it should be easy and not limited, to increase the security of the algorithm, if more security is needed. To achieve these, the AES finalists get carefully examined and cryptographic strong operations are identified. From these a set of operations is selected. Their order, arrangement and their operands, i.e. the computation graph, then are determined by the key. The key length itself is not limited and should be solely deciding for the length of the computation graph, just as for the encryption strength.

During the development, many different approaches are tried and analyzed. The first attempt is made in the bachelor thesis [11] and is taken as is as a reference to the algorithm that is developed here. Many others followed, changing the set of operations and the way the parameters, like the order, the arrangement and the operands, are determined. The one, that is presented, passes the most tests from the Statistical Test Suite, but since not all tests pass all the time, further researches have to be done.

This bachelor thesis can be partitioned into three main parts. The first part gives an overview over the design space for encryption algorithms. It starts with small parts, like which operations are useful at all, and finishes with the final, fully functional encryption algorithm – *Key Orientated Operation Selection (KOOS)*.

After that, the analysis of this algorithm is presented. One the one hand with the tools used to analyse the AES candidates. The testsuite itself and the analysed sets of data are explained, too. On the other hand the performance of the algorithm is tested and analysed.

Finally, the findings and conclusions are summarised and further research has to be done.

1.1. State of the Art

The five encryption algorithms, that were announced as finalists of the AES standardisation process, can be seen as the State of the Art of symmetric encryption algorithms. All these are unbroken and frequently used in miscellaneous applications.

The *AES* winner, the *Rijndael* cypher, consists of four functions to manipulate the data, also referred to as *state*. The state is arranged as a matrix with four bytes in each row and column. The four functions are repeated several times one after another – one to substitute bytes, one to shift rows, one to mix columns and the last one to add the round key. [1]

MARS is a very symmetric cypher, consistiong of six parts: First, parts of the key are added to the plaintext, then eight rounds of unkeyed forward mixing with heavy use of S-boxes are performed. The third and fourth part are eight rounds of keyed forward and backward transformations, respectively, with the aid of the *E-function*. In the end there are eight rounds of unkeyed backwards mixing with the S-boxes again and parts of the key become substracted from the current state of the plaintext. [4]

The *RC6* cypher greatly depends on data-dependant rotations. First, some parts of the key are added to half of the plaintext, then several rounds of data-dependant rotations, additions and a permutation are performed. Finally, another part of the key is added to the other half of the plaintext. [7]

The *Serpent* cypher consists of 32 rounds and in each round first a part of the key is exclusive or'ed to the current state of the plaintext (key mixing), then reversable S-boxes change the current state and finally, a linear transformation is applied to the current state. In the last round this linear transformation is exchanged with another key mixing. [2]

The *Twofish* cypher performes 16 Feistel-like networkes and before and afterwards it exclusive or's some parts of the key to the plaintext. The used Feistel function consists of bijective, key dependant 8-by-8-bit S-boxes, a MDS matrix and a pseudo-Hadamard transformation. [9]

2. Algorithms

This chapter consists of three parts: First, the idea and the design space is explored. It shows which operations and methods are useful for encryption algorithms in general and in particular for the algorithm that is developed here. After that, all used operations are discussed in detail. As the final encryption algorithm solely works on 32-bit integer words, the description of the operations in here are limited to their 32-bit version. Finally, the developed algorithm is presented.

2.1. Design Space

The goal is to develop an encryption algorithm, whose operations, operands and their order are all dependent on the key. There are certain requirements to the operations: They have to be reversible, they should be fast and they should have a certain cryptographic strength. There are also requirements to the operands, like the possible operands should be equally often chosen and successive applications of operands and operations should not erase their effect, e.g. multiple applications of exclusiv or with the same operands may lead to small or no differences between the plaintext and the cyphertext. Given that all these parameters are extracted from the key, the algorithm needs a method that extract them equally disributed and a method to modify keys that would result in bad distributed parameters.

Looking at some other encryption algorithms, the set of possible operations becomes clear. There are simple, reversible and fast operations like XOR, addition and substraction and at least one of them is used in every encryption algorithm. Mostly, they are used to mix the key and and the plaintext, like in *AES* [1]. This is often also called *key whitening*. Then there are several kinds of circular shifts or rotations, depending on the current state of the plaintext and the key as well as static ones. *MARS* and especially *RC6* frequently use them. [5]

Permutations or *P-boxes* are another method to shuffle bits, bytes or other chunks of the current state of the plaintext.

Furthermore, *Feistel networks* are great methods to distribute changes in one word of the plaintext to one or all other words. Therefor the word is modified by an *F-function* and then, with an easily reversible function like XOR or addition, applied on the other words.

Finally, many algorithms use table-lookups, so called *S-boxes*. These S-boxes use predefined tables to replace some parts of the current state of the plaintext. It is very hard to create good S-boxes and sometimes these tables have to be reversible, too, e.g. *Serpent* uses reversible S-boxes. Though good S-boxes are cryptographic strong, their principle could lead to the suspicion, the algorithm may has a backdoor. Because of this and the fact, that the S-boxes in principle does not depend on the key, they are not that suitable for the encryption algorithm that is developed in here.

2.2. Operations

Here the operations addition, XOR, rotation, negation and multiplication are discussed. Their functionality and some examples are discribed. All operations get two unsigned integer k and p as input and return one unsigned integer c as output, where p is meant to be the plaintext and c is meant to be the cyphertext. k may be a key value or some other intermediate value.

2.2.1. Addition

This is the normal integer addition modulo 2^{32}. To encrypt a word, calculate $c = (p + k)\ mod\ 2^{32}$ and to decrypt $p = (c - k)\ mod\ 2^{32}$. Of course the modulo operation can be left out, because of the 32-bit unsigned integer. One example:

$$k = \text{0x9BD69510}, p = \text{0xED8BA100}$$
$$c = (\text{0x9BD69510} + \text{0xED8BA100})\ mod\ 2^{32} = \text{0x89623610}$$
$$p = (\text{0x89623610} - \text{0x9BD69510})\ mod\ 2^{32} = \text{0xED8BA100}$$

2.2.2. XOR

A bitwise exclusiv or operation is performed on the operands p and k. Hence XOR itself is its inverse, encryption and decryption are the same.

$$k = \text{0x9BD69510}, p = \text{0xED8BA100}$$
$$c = \text{0x9BD69510}\ XOR\ \text{0xED8BA100} = \text{0x765D3410}$$
$$p = \text{0x765D3410}\ XOR\ \text{0x9BD69510} = \text{0xED8BA100}$$

2.2.3. Rotation

A rotation of p to the left is performed. The number of rotations equals the value of the five most significant bits of the multiplication $k*(2*k+1)$. With this multiplication you make sure, that the number of rotations depent on all bits in k and not just on the five most significant [5]. This is similar to the way *RC6* performes data-dependant-rotations [7]. To decrypt such a rotation to the left, simply rotate the same number of rotations to the right.

$$k = \text{0x9BD69510}\ , p = \text{0xED8BA100}$$
$$(\text{0x9BD69510} * (2 * \text{0x9BD69510} + 1)) = \text{0x76EDD710}$$
$$\text{The five most significant bits are: } \text{0x76EDD710} \gg 27 = \text{0xE} = 14$$
$$c = \text{0xED8BA100} \lll 14 = \text{0xE8403B62}$$
$$p = \text{0xE8403B62} \ggg 14 = \text{0xED8BA100}$$

2.2.4. Negation

If the operand k is even, then it returns p with all its bits flipped, if not, it returns p as it is. This is not a very strong operation, because it uses only one bit in one of its operands and changes – in 50 % of the cases – all bits of the other one, but on a large scale and assuming the zeros and ones in the k's are equally and randomly disributed about 50 % of the bits are changed.

2.2.5. Multiplication

The operands p and k are divided into two 16 bit integer $p1$, $p2$, $k1$ and $k2$. To encrypt p each pair $k1$ and $p1$, $k2$ and $p2$ perform a multiplication modulo the prime number $2^{16}+1$. Hence zero is not a valid value for this primitive residue class and the value 2^{16} is not in the range of a 16 bit integer each of the 16 bit operands become increased by one before the multiplication takes place and afterwards the results become decreased by one.

$$(c1+1) = (k1+1) * (p1+1)\ mod\ (2^{16}+1)$$
$$(c2+1) = (k2+1) * (p2+1)\ mod\ (2^{16}+1)$$

To decrypt, the multiplicative inverse of $(k1+1)$ and $(k2+1)$ must be determined (e.g. by using Euclid's algorithm) and after that the multiplication is analog.

$$\begin{aligned}
k &= \text{0x9BD69510}, p = \text{0xED8BA100} \\
c1+1 &= (\text{0x9BD6}+1) * (\text{0xED8B}+1)\ mod\ (2^{16}+1) = (\text{0xB3F9}+1) \\
c2+1 &= (\text{0x9510}+1) * (\text{0xA100}+1)\ mod\ (2^{16}+1) = (\text{0xE851}+1) \\
&\Rightarrow c = \text{0xB3F9E851}
\end{aligned}$$

$$\begin{aligned}
p1+1 &= (0x9BD6+1)^{-1} * (0xB3F9+1)\ mod\ (2^{16}+1) \\
&= \text{0x0F49} * \text{0xB3FA}\ mod\ (2^{16}+1) = \text{0xED8B}+1 \\
p2+1 &= (\text{0x9510}+1)^{-1} * (\text{0xE851}+1)\ mod\ (2^{16}+1) \\
&= \text{0x2D32} * \text{0xE852}\ mod\ (2^{16}+1) = \text{0xA100}+1 \\
&\Rightarrow p = \text{0xED8BA100}
\end{aligned}$$

2.3. Permutation

In the algorithm two types of permutations are performed. One to shuffle the 32-bit plaintext words and one to shuffle 16-bit words between two plaintext words. The permutations in this section are all given in cycle notation, i.e. a permutation $\sigma = (2\ 4)(3\ 0\ 1)$ exchanges 2 and 4, and 3 is transposed to 0, 0 to 1 and 1 to 3.

2.3.1. Address Permutation

The plaintext is partitioned into two to eight 32-bit word. Then a corresponding permutation p of the S_2 to S_8 is extracted from the key and applied to the order of the plaintext words. E.g. (a 160 bit plaintext is partitioned into an zero based array of 32-bit words):

$$p = (2\ 4)(3\ 0\ 1)$$
$$\text{State: } (\text{0x9BD69510, 0xED8BA100, 0x12345678, 0x0, 0xFFFFFFFF})$$

Applying the permutation, the state changes as follows:

$$\text{State: } (\text{0x0, 0x9BD69510, 0xFFFFFFFF, 0xED8BA100, 0x12345678})$$

2.3.2. Block Permutation

The permutation shuffels 16-bit blocks of two 32-bit operands a and b via one out of the three independent permutations of the symmetric group S_4 with an order of four. The permutations are:

$$\begin{aligned} perm0 &= (0\ 3\ 2\ 1) & perm0^{-1} &= (1\ 2\ 3\ 0) \\ perm1 &= (0\ 2\ 1\ 3) & perm1^{-1} &= (3\ 1\ 2\ 0) \\ perm2 &= (0\ 1\ 3\ 2) & perm2^{-1} &= (2\ 3\ 1\ 0) \end{aligned}$$

Let $a = p0\ p1$ and $b = p2\ p3$ be the split-up of a and b. Performing the permutation $perm0$ the results are $a' = p3\ p0$ and $b' = p1\ p2$, in detail the 16 bits in position zero ($p0$) become the 16 bits in position three ($p3$) and so on.

This permutation can be reversed by using the inverse permutation $permX^{-1}$, respectively.

2.4. Feistel Network

A *Feistel Network* describes an arrangement, in which one word w of the plaintext effects one or all other words. Therefor the word w often is modified by an arbitrary, not necessarily invertible *F-function* and then applied to the other words via an easily invertible function, like addition or XOR. An illustration of this arrangement can be viewed in Figure 2.1. Hence the word w is carried on unchanged, a Feistel Network is always reversible.

Here three different *F-functions* are used: An function inspired by the *f-box* of the encryption algorithm *IDEA*, a variation of the *E-function* of the encryption algorithm *MARS* and a simple *multiply-with-carry* pseudo-random number generator invented by George Marsaglia.

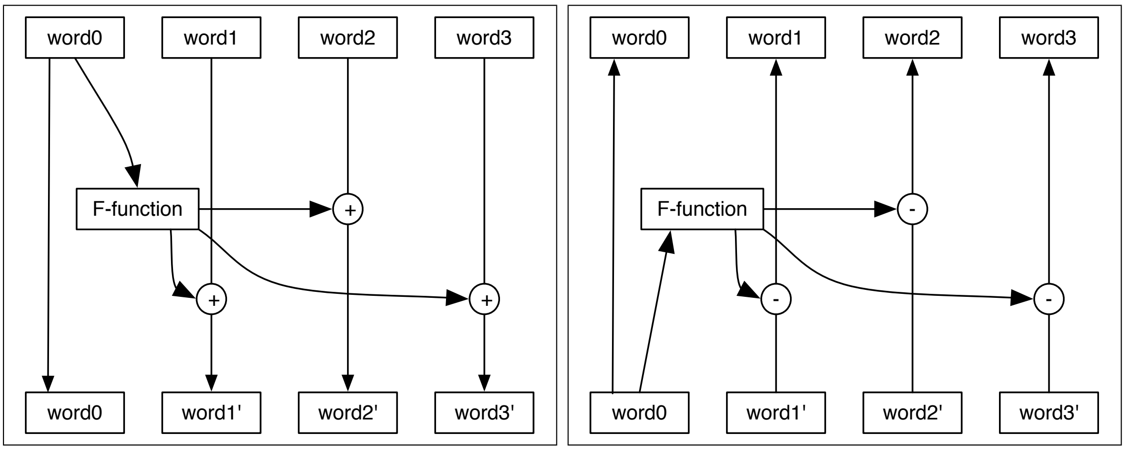

Figure 2.1.: Feistel Network: Encryption (left) and decryption (right).

2.4.1. F-Box

As can be seen in Figure 2.2, this *F-function* needs four input values, as they are $P1$, $P2$, $K1$ and $K2$, and returns two output values $C1$ and $C2$. Each of these values are 16 bit integer. In this version the *F-box* gets one 32-bit integer from the current state of the plaintext P and one 32-bit key K. Then it splits up P and K into $P1$, $P2$, $K1$ and $K2$, with $P1$ and $K1$ contain the most significant bits and $P2$ and $K2$ contain the least significant bits of P and K, respectively. The returned values $C1$ and $C2$ are concatenated to one 32-bit output value C with $C1$ as its most significant bits and $C2$ as its least significant bits. Finally, C is applied to the other words with the operation XOR.

The operation $\odot$ is a multiplication modulo the prime number $2^{16} + 1$. As described in Section 2.2.5 both operands are increased by one before the multiplication and the result is decreased by one. The operation $\boxplus$ is the regular 16 bit integer addition.

2.4.2. E-Function

The *E-function* is a well designed part of the "cryptographic core" of the *MARS* encryption algorithm. The function is illustrated in Figure 2.3. It gets one 32-bit integer *in* from the plaintext and two 32-bit keys $K1$ and $K2$ as input and returns three 32-bit outputs R, M and L. These values are added to the other plaintext words one after

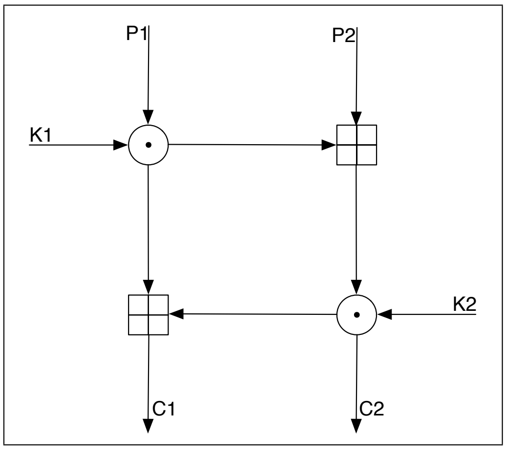

Figure 2.2.: IDEA's f-box

another: the first word plus L, the second plus M, the third plus R and the fourth word again plus L and so on.

The operations are: The regulare 32-bit multiplication $\odot$, the regular 32-bit addition $\boxplus$, XOR $\oplus$ and two kinds of rotations – "$n \lll$" is a fixed left-rotation by n and "$\lll$" is a data-dependent rotation by the value of the five least significant bits of the data. Unfortunately the original *E-function* contains an *S-box* $\boxed{S}$, but in this version the table of the *S-box* is replaced by the key itself, i.e. the value $in + K1$ is substituted by the 32-bit key at position $(in + K1)$ mod (number of 32-bit keys).

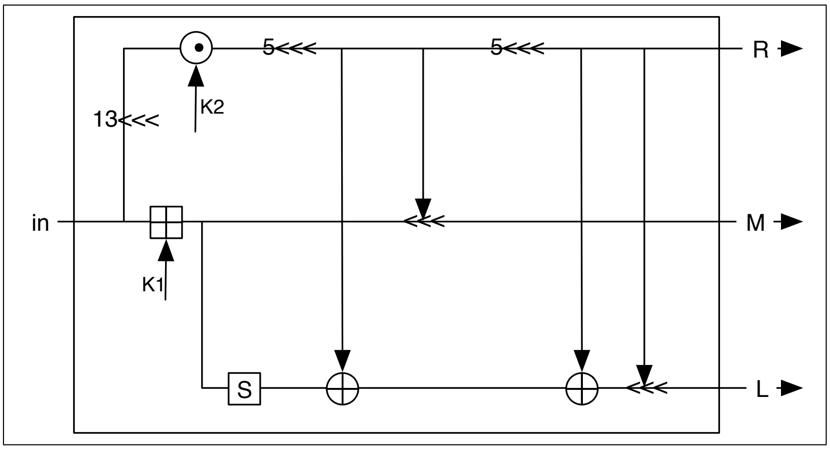

Figure 2.3.: MARS' E-function

2.4.3. Multiply-With-Carry (MWC)

A *Multiply-With-Carry* pseudo-random number generator, invented by George Marsaglia[1], is used as a *F-function*. The inputs are a 32-bit *key* and the 32-bit plaintext word *state*.

[1]The hole USENET-article can be found at `http://www.cse.yorku.ca/~oz/marsaglia-rng.html`.

It returnes a 32-bit integer calculated as shown in listing 2.1. This pseudo-random number generator is known to have a period of about 2^{60}. The result is then applied to the other plaintext words via XOR.

```
state = 36969 * (state & 65535) + (state >> 16);
key   = 18000 * (key & 65535)   + (key >> 16);
return (state << 16) + key;
```

Listing 2.1: The *MWC* function

2.5. Possible Conjunctions

In the previous sections the possible operations are presented. The subject in this section is, how to put them together. The address permutation and the feistel network work on their own on the whole plaintext and the block permutation works only on two plaintext words, so here the question solely is, in wich order to put them.

For the operations from Section 2.2 it is a different problem, as they work with pairs of plaintext words, with a plaintext and a key word or with a plaintext word and some other intermediate value.

One way is to do it similar to the algorithm presented in [11]. In this way basically one plaintext word and one operation is selected and then the operation is performed with the key as the second operand (see the left graph of Figure 2.4). Another way, is to pick one operation per plaintext word and use the key and the first plaintext word as operands for the first operation and for the following operations, one operand is the corresponding plaintext word and the other is the previous result (see right graph of Figure 2.4). There are many other possibilities to arrange operations with plaintext words and key words, and only testing them shows, which is superior.

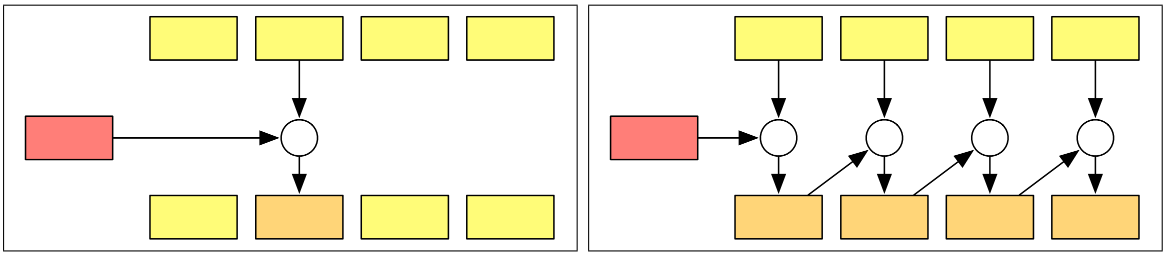

Figure 2.4.: Conjunction of operations

2.6. KOOS – The Final Algorithm

KOOS is the abbreviation for Key Orientated Operation Selection. The algorithm solely works on 32-bit integer words: It expects a key of at least one word, so all key sizes with a multiple of 32-bit are possible. Furthermore the plaintext is expected to be two to eight 32-bit integers, i.e. the block size ranges between 64 bit up to 256 bit. During the initialisation process the key becomes extended to $round * key\ size$ words and modified, so that there should be no weak keys. Considering, that the algorithm performs best with nine rounds (see Chapter 3), this is fixed to be the length of the algorithm.

The following sections explain how and which parameters are taken, how the encryption and decryption routine work and how the key is extended and modified.

2.6.1. The Parameter

From each 32-bit key word k a set of different parameters are extracted. They are a address permutation, one operation per plaintext word, the operands and the permutation for a block-wise permutation, the *F-function* for a Feistel network and finally, an order in which these operations are performed.

The used operations from Section 2.2 are XOR, addition, rotation and negation, indexed by 0 to 3, respectively. Likewise the block-wise permutations and the *F-function* are indexed. The addresses for the operands correspond to the index of the 32-bit plaintext array. The order of these operations are indexed, too. The address permutation is zero, the operations per plaintext word is one, the feistel network is two and last the block-wise permutation is three.

Each of these parameters are extracted from the key k via successive modulo operations and integer divisions. The listing A.1 shows this method in more detail.

Having this in mind, one can determine the number of bits, that are needed, to calculate each parameter. Assuming the text size is four, this algorithm divides the key in total by: $4!$ (address permutation) $*4*3$ (block permutation addresses) $*3$ (block permutation) $*4!$ (order of the operations) $*4*3*3*3$ (operations) $*3$ (Feistel function) $= 6718464$. This means $log_2(6718464) \approx 23$ bits are used, but with an text size of eight it needs $log_2(8! * 8 * 7 * 3 * 4! * 4 * 3 * 3 * 3 * 3 * 3 * 3 * 3 * 3) \approx 41$ bits of the 32-bit key. To resolve this problems, in the middle of this algorithm the key becomes rotated to the right by 7.

2.6.2. Encrypt and Decrypt

With the parameter from the previous section, each 32-bit word of the key creates a computation graph similar to the one presented in Figure 2.7. Each of these graphs can be divided into the four parts. One part is the address permutation, another a row of operations. Then for each plaintext word there is a Feistel network and finally, a block-wise permutation.

An excample graph is shown in Figure 2.7. Let this graph be created by the 32-bit word of the key at position i. The parameters, that are extracted from this key at position i, are:

Operation order:	(Address Permutation, Row of Operations, Feistel network, Block-wise permutation)
Address permutation:	(0 1)(2 3)
Operations:	$(XOR, Addition, XOR, Rotation)$
Feistel function:	$E - function$
Block-wise permutation:	$perm1 = (0\ 2\ 1\ 3)$

The address permutation and the block permutation are performed straight forward as described in Section 2.3. For the Feistel network, a plaintext word at position j and the key at position $i + j$ (modulo the number of 32-bit key words) are used as operands to the Feistel function, here the $E - function$. As the $E - function$ needs two keys, it also gets the key at position $i + j + 1$. This is done for each word of the plaintext one after another. Regarding the row of operation, the first operation is performed with the key at position i and the plaintext word at position 0, if i is even, and accordingly the word at the last position, if i is odd, as operands. The following operations use their corresponding plaintext at position j – in ascending order, if i is even or in descending order, if i is odd – as its first operand and as its second the result of the multiplication (see Section 2.2.5) of the previous result and the key at position $i + 3 * (j - 1)$.

As shown previously, each of the operations are reversable, so the decryption proceeds in reverse order. The listings A.2 and A.3 in the Appendix show these algorithms.

2.6.3. Key Extension

The key extension process is divided into three parts. First, the key becomes extended by a certain number of self-encryption rounds. Therefore, the key is encrypted by itself and the resulting cyphertext is appended to the key. In the next round this cyphertext is encrypted with the key consisting of the original key and the first cyphertext, and the resulting cyphertext is again appended to the key. So, in each round the key grows by the size of the original key, until the key reaches the size: $round * original\ key\ size$ bits. The Figure 2.5 illustrates this part further. If the key size and the plaintext size are not equal, the key is extended to the next multiple of the plaintext size, using 0xAAAAAAAA as padding.

Second, the extended key becomes hashed by a certain number of self-hashing rounds. In each round the key is partitioned into parts of the size of the original key. Each part is then encrypted using the extended key. The inverse of the resulting cyphertext is added to the lower half of this part and to the upper half of the next part – if the current part is the last part, then the next part referes to the first part. Figure 2.6 pictures this method in more detail.

Last, the extended and hashed key gets checked for 32-bit words, that do not fit certain criteria, i.e. that are supposed to perform weak during the encryption process. There

Figure 2.5.: Key Extension: First Part

are two criteria: Each word has to fulfill the frequency test (3.1.2). Therefore, the total number of ones or zeros should not fall under nine. The other criteria limits the number of consecutive 0's or 1's to nine. Thus, if a word of the key fails one of these criteria, the part containing this word becomes encrypted again and the inverse is added to the original part.

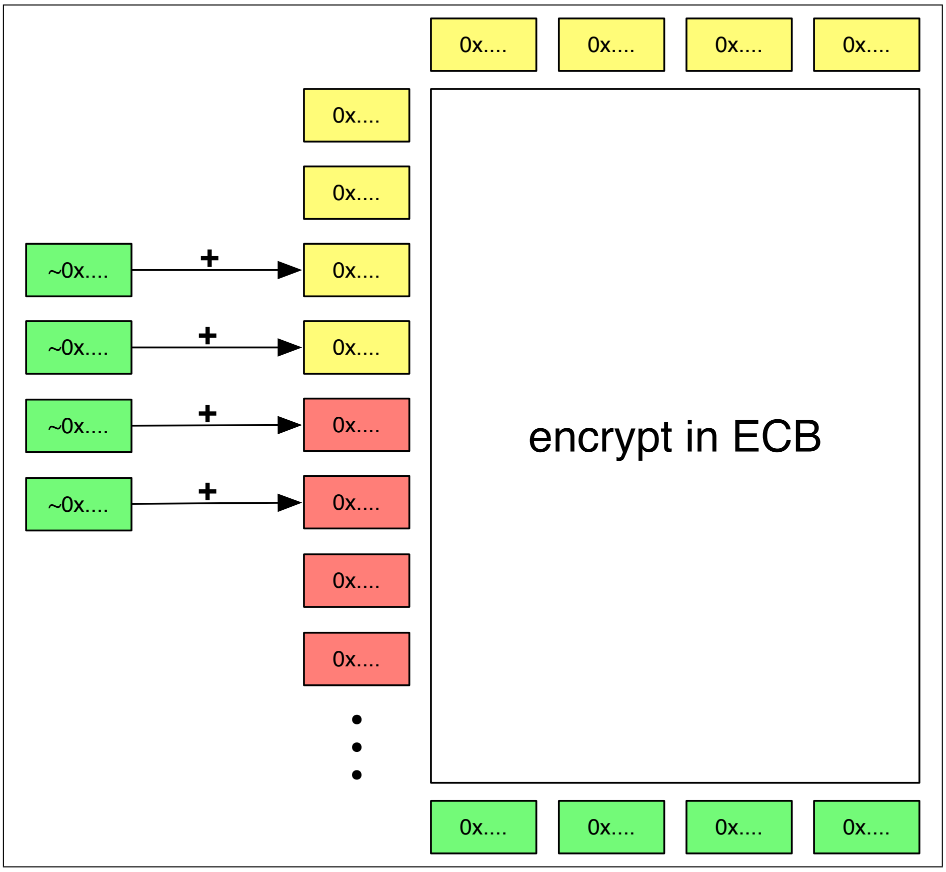

Figure 2.6.: Key Extension: Second Part

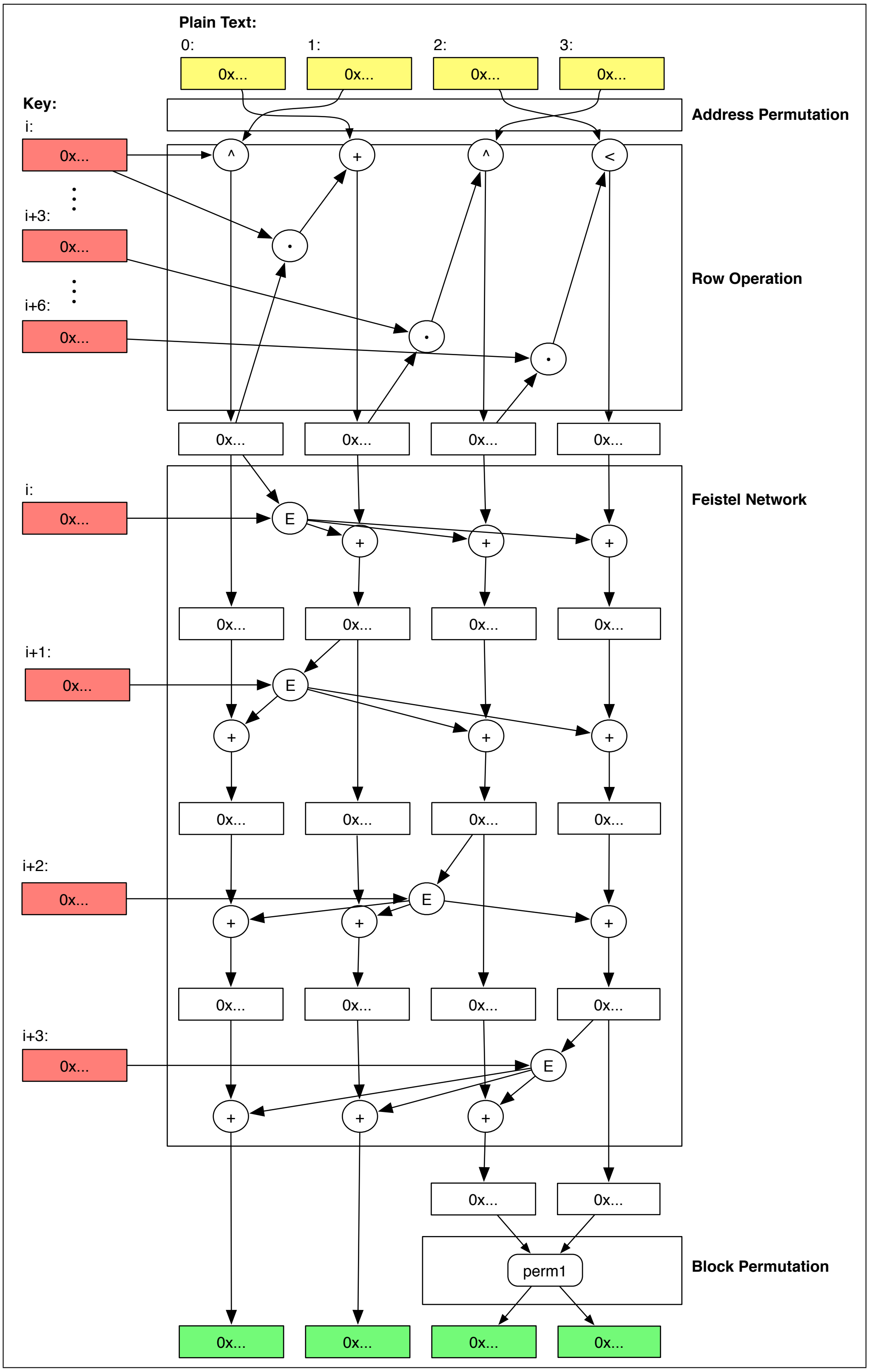

Figure 2.7.: Encryption

3. Analysis

The following chapter analyses the previously described encryption algorithm from Chapter 2. On the one hand, its security is analysed as measured by the randomness of its output. To get comparable results the Statistical Test Suite (STS) provided by the National Institute of Standards and Technology (NIST) is used. [8]

On the other hand, the performance and the used resources on different architectures are compared.

3.1. Statistical Test Suite

The standardization process of the Advanced Encryption Standard (AES) analyses the security of each candidate algorithm. Therefore the STS evaluates the randomness of several sets of data each candidate has to generate. A short description of these sets can be found in Section 3.1.1.

In order to evaluate the randomness of one set of data, the STS performs 15 different and in most cases independant tests that concentrates on one aspect of randomness. A short description of each test can be found in Section 3.1.2.

After these two sections the findings of the encryption algorithm from Chapter 2 are presented.

3.1.1. Sets of Data

These are the sets of data the NIST used to analyse the candidates for the AES [6, 10]. Each set of data should provide a good insight in how well an encryption algorithm deals with one specific situation.

Key Avalanche

The *Key Avalanche* dataset shows how well the encryption algorithm deals with small changes in the key. Therefore, a plaintext of all zero is encrypted with a random key. Then each bit in the key is flipped one after another and again a plaintext with all zero is encrypted with each of these modified keys. The changes between the cyphertext from the original key and the cyphertexts from the modified keys are the provided data.

Plaintext Avalanche

The *Plaintext Avalanche* dataset shows how well the encryption algorithm deals with small changes in the plaintext. Therefor a random plaintext is encrypted with a key of all zero. Then each bit in the plaintext is flipped one after another and each of these modified plaintexts become encrypted with a key of all zero. The changes between the cyphertext from the original plaintext and the cyphertexts from the modified plaintexts are the provided data.

Plaintext / Cyphertext Correlation

This set of data serves to analyse the correlation between the plaintext and its corresponding cyphertext. A big random plaintext becomes encrypted in electronic codebook mode [3] with a random key. The differences between plaintext and cyphertext – the correlation – are the provided data.

CBC Mode

Analysing this set of data shows whether the encryption algorithm is suitable for the cipher-block chaining mode [3]. Therefore, a big plaintext of all zero with an initialization vector of all zero becomes encrypted in CBC mode with a random key.

Random Plaintext and Key

Providing a big random plaintext and a random key this set of data serves to analyse, whether the resulting cyphertext is random too.

Low / High Density Key

This set of data serves to analyse how the encryption algorithm behaves with a low and accordingly high density key. A random plaintext is first encrypted with a key with all zero (all one), then the plaintext is encrypted with all keys having only one one (one zero) and last the plaintext is encrypted with all keys having only two ones (two zeros).

Low / High Density Plaintext

This set of data serves to analyse how the encryption algorithm behaves with a low and accordingly high density plaintext. A random key is used to first encrypte a plaintext with all zero (all one), then to encrypt all plaintexts having only one one (one zero) and last to encrypt all plaintexts having only two ones (two zeros).

3.1.2. STS Tests

The Statistical Test Suite takes a set of data and interprets it as several sequences, each in the magnitude of one million bits. One set of data consists of 128 to 300 of these sequences. Each test calculates a *P-value* for every sequence to decide whether to accept or to reject that sequence, in other words to decide whether this sequence seems to have a random distribution of zeros and ones. In the case of the AES standardization process the significance level α is 0.01, i.e. the *P-value* of a sequence has to be greater than α to be accepted. Otherwise the sequence is rejected.

There are two methods to analyse, if the whole set of data passes a test: On the one hand, there is a proportion of sequences passing the test. If this proportion is outside the interval $p \pm 3 * \sqrt{\frac{p(1-p)}{m}}$, where $p = 1 - \alpha$ and m is the number of sequences [8], then there is evidence that the set of data is not random.

On the other hand, there is a distribution of *P-values* that can be inspected: They should be equally distibuted, too. So a *P-value* of all *P-values* is provided—*P-value*$_T$. The *P-value*$_T$ should be greater or equal 0.0001 . [8]

Frequency (Monobit) Test

The proportion of zeros and ones in the entire sequence is calculated and the tests assesses the closeness of the proportion of ones to 0.5. Small *P-value* indicate, that there are too many zeros or too many ones in the sequence.

Frequency Test within a Block

This test partitions the sequence into M-bit long blocks and tests if the proportion of ones in each block is about 0.5. If the *P-value* is too small, then in at least one block there is a large deviation from equal proportion of ones and zeros.

Runs Test

The *Runs Test* calculates the number of runs of various lengths in a sequence. A run is a not interrupted series of identical bits framed by at least one bit of the opposite value. Too small *P-values* indicate, that the oscillation between ones and zeros is either too fast or too slow.

Longest Run of Ones in a Block

Again the sequence is partitioned into M-bit blocks. For each block the length of the longest run (see *Runs Test* 3.1.2) of ones is determined and checked whether this length can be expected in a random sequence. As the sequence previously has to pass the monobit test an irregularity in the length of the longest run of ones implies that there is also a irregularity in the length of the longest run of zeros. *P-values* smaller than α indicate big cluster of ones and zeros.

Matrix Rank Test

The sequence is partitioned into $M * Q$-bit blocks, where M is the number of rows and Q is the number of columns. Each block is transformed into a MxQ matrix and its binary rank is computed. If the rank distribution differ too much from the expected distribution of a random sequence, the *P-value* becomes small.

Fourier Transform Test

In this test the Discrete Fourier Transform is performed on the sequence and peak heights are analysed. The purpose is to detect repetitive patterns that are near to each other in the sequence. If so, the sequence is considered to be not random and the *P-value* is small.

Non-overlapping Template Matching Test

The focus of this test is to count the number of occurrences of predefined m-bit pattern in the sequence and to decide whether this number corresponds to the expected number of pattern in a random sequence. A m-bit window slides over the sequence and searches for a pattern. If the pattern is not found, the window moves one bit further. If the pattern is found, the window is set right after the found pattern. This test is repeated several times for different pattern.

Overlapping Template Matching Test

The focus of this test is to count the number of occurrences of predefined m-bit pattern in the sequence and to decide whether this number corresponds to the expected number of pattern in a random sequence. The differnce to the non-overlapping test is, that if the pattern is found, the window slides only one bit and resumes the search.

Maurer's "Universal Statistical" Test

The purpose of this test is to check, if the sequence is significantly compressible without loss of information. If so, the sequence is considered to be not random and the *P-value* is small.

Linear Complexity Test

The test calculates the length of a linear feedback shift register (LFSR) to determine whether the sequence is complex enough to be considered random. Random sequences are characterized by longer LFSRs, so too small *P-value* indicate too short LFSR.

Serial Test

This test determines the frequency of all overlapping m-bit pattern and assesses, if each pattern occures approximately equally often. For $m = 1$, this test is the same as the *Monobit Test*.

Approximate Entropy Test

This test compares the frequency of all overlapping m-bit and $(m+1)$-bit pattern in the sequence against the expected frequency in a random sequence.

Cumulative Sums Test

The sequence is interpreted as a random walk, where 1 is interpreted as $+1$ and 0 is interpreted as -1. The sums for increasing lengths of the random walk / of the partial sequences are calculated and the maximal excursion from zero is compared to the expected results for a random sequence. For a random sequence the excursion of the random walk should be near zero. The random walk is done twice: From the beginning to the end of the sequence and from the end to the beginning.

Random Excursion Test

Again the sequence is interpreted as a random walk and divided into cycles at positions where the random walk / the cumulative sum is zero. For the states -4, -3, -2, -1, 1, 2, 3 and 4 the number of occurences in each cycle is calulated and compared to the expected results of a random sequence.

Random Excursion Variant Test

Again the sequence is interpreted as a random walk. The total number of occurences of the states -9, -8,. . . ,-1 and 1, 2,. . . , 9 are calculated and compared to the expected results of a random sequence.

3.1.3. Results

Here the results of the Statistical Test Suite (STS) are presented. The sets of data are generated multiple times, each with another number of rounds the encryption algorithm has to run – from 1 round to 14 rounds. More rounds are possible, but there are two reasons, why I stick to a maximum of 14 rounds. Looking at the charts, there is no trend visible, that more rounds do improve the statistical randomness of the results. The other reason regards the performance of the algorithm: More rounds reduce the speed of the algorithm so far, that it is no more applicable.

To analyse the output of the STS, two types of charts are presented. In the first chart the number of different, failed tests are compared to the number of rounds, e.g. Figure 3.1. In the optimal case, no test should fail. The second chart shows, for a certain number of rounds, for every test the propotion of successful sequences (the blue rhombs) and a red line representing the minimum propotion (see Appendix A.3). This minimum propotion is calculated and dependent on the number of sequences (see Section 3.1.2) and as the tests Random Excursion and Random Excursion Variant do not use all provided sequences in all tests, a gap arises in the line. This chart can be interpreted as follows: Every rhomb above the line is a successful test and every rhomb below the line is a failed test.

All in all 188 tests per set of data are executed: 148 Non-overlapping Template Matching Test, 18 Random Excursion Variant Tests, 8 Random Excursion Tests, 2 Cumulative Sums Tests, 2 Serial Test and one for each of the rest ten kinds of tests. In the following charts, one kind of test is considered as failed, if at least one execution of this kind of test fails.

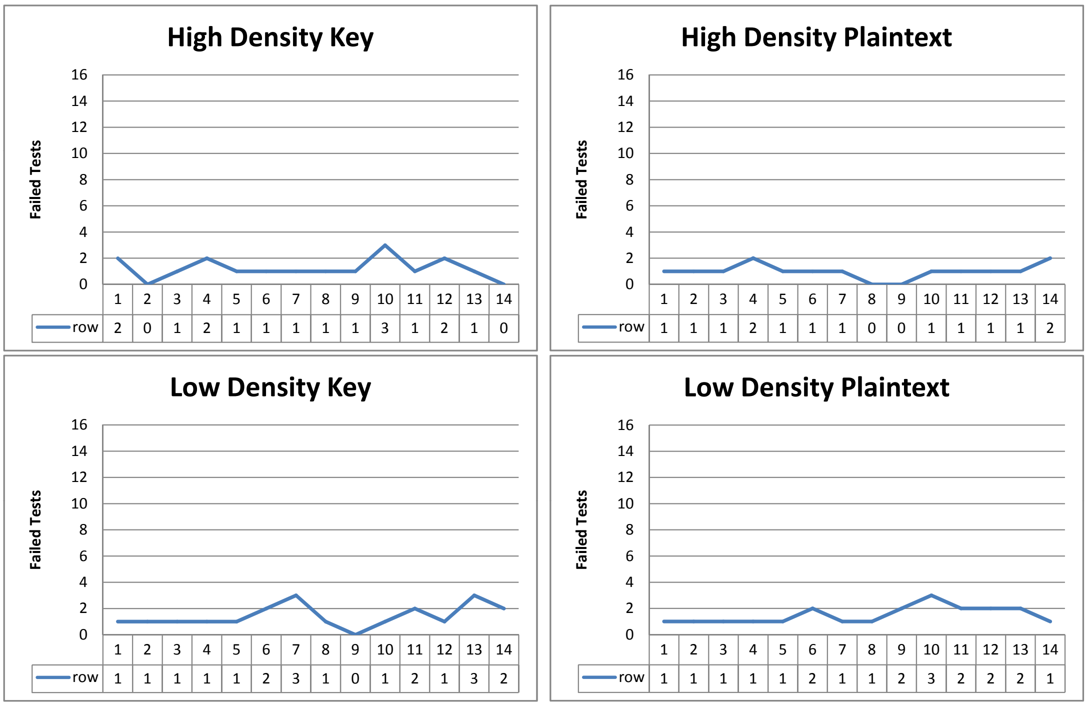

Figure 3.1.: High and Low Density Key and Plaintext

The charts in Figure 3.1 show, how well the algorithm performs with high density and low density keys and plaintext. It performes slightly better with high density keys than with low density ones. I believe this is, because of the operations being extracted from the key by modulo operations, a key with more 1's – especially in the most significant bits – can result in a better computation graph. This shows, that the current key extension method is not satisfying. It should, whatever key is given, return equally strong extended keys.

On low or high density plaintexts the encryption algorithm again performs slightly better with high desity plaintexts than with low density ones. In both cases the key is a random one, so the computation graph should be approximately equally good, but obviously plaintexts with many 1's become better encryption results.

The tests, that failed are mostly Non-overlapping Template Matching Tests – nearly in all tested datasets, if a test fails, it is most likely at least one Non-overlapping Template Matching Test – and a very few other like Random Excursion, Random Excursion Variant. This is an observation, that throughout all tested datasets can be found. Currently the encryption algorithm makes heavy use of the key as input to the computation graph: Multiple keys are used for the row of operations and in almost the same manner multiple keys are used for the Feistel network. This may be a reason for too many patterns in the resulting cyphertexts. To avoid the multiple usage of keys, the row of operations and the Feistel functions have to be re-engineered: Maybe a new conjunction of the operations in the row and less key usage in the *E-function* does the trick.

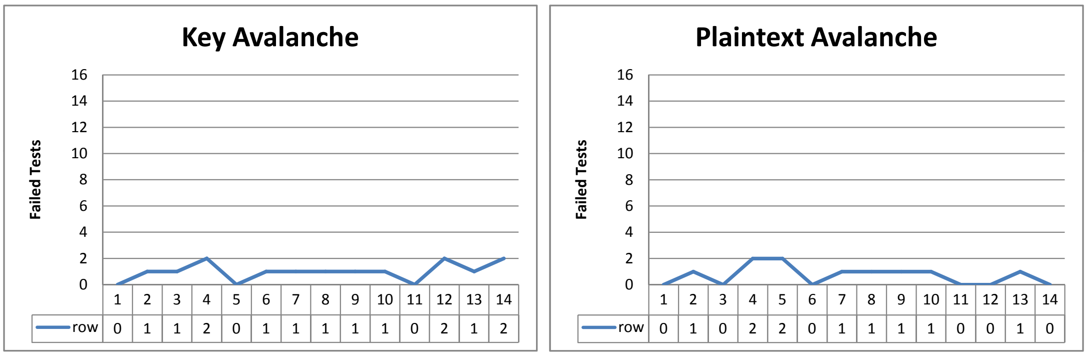

Figure 3.2.: Key and Plaintext Avalange

The results for the key avalange and plaintext avalange in Figure 3.2 show quit good performance. At the maximum two tests fail, but mostly it is the Non-overlapping Template Matching Test, again. The dataset key avalange performs a little worse than the plaintext avalange – this is once again an indicator for a not entirely satisfying key extension method.

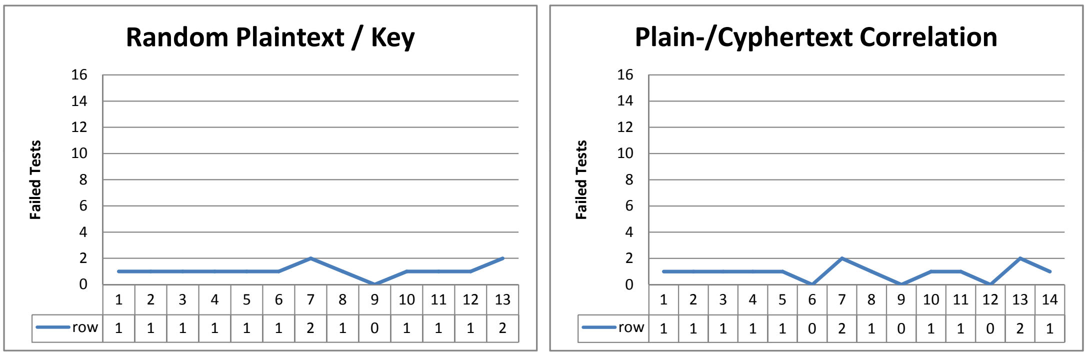

Figure 3.3.: Random Plaintext and Key and Plaintext / Cyphertext Correlation

The same goes for the results of the datasets random plaintext and key and plaintext / cyphertext correlation in Figure 3.3: At most two tests fail, but mostly it is only the Non-overlapping Template Matching Test. This is a little surprising for the dataset random plaintext and key, hence the data on its own passes all tests – the used random number generator is the Blum-Blum-Shub generator (BBS), like in the standardisation process. In other words, the encryption algorithm produces, even though the data comes from a random source, some pattern much too frequently. Looking at the charts in Figure A.8, there is no pattern in which Non-overlapping Template Matching Test fail. This, again, may signifies a too frequent use of the key.

The CBC Mode results, shown in Figure 3.4, are the best results all over the tested datasets. Most of the times all tests are passed and all the rest of the times only one test fails, with one exception at six rounds. Apparently, the special structure from the

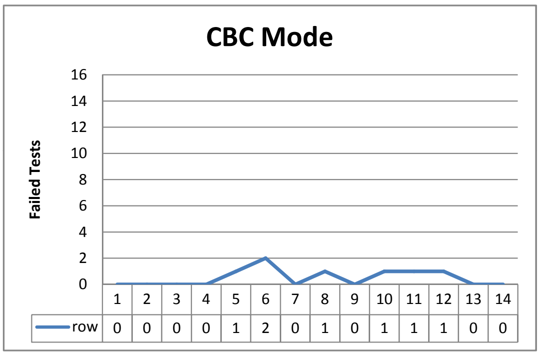

Figure 3.4.: CBC Mode

CBC Mode prevents too many template matchings in the cyphertext, but nevertheless mostly the Non-overlapping Template Matching Test fails.

The table 3.1 merges the results and differentiates in how many datasets a specific test fails at a fixed number of rounds the encryption algorithm has to run. As expected, this show that the encryption algorithm has most problems with the Non-overlapping Template Matching Test (NonOverl), followed by the Random Excursion Variant and the Random Excursion Tests and then a few other. On the other hand there are always about the same number of test failing in each round. From in total thirteen failed tests, down to five failed tests with nine rounds. This concludes, that the results are about equally stronge, i.e. equally random, whatever number of rounds the encryption algorithm runs. This conclusion corresponds to the appearance of the charts: There is always some up and down for the number of failed tests, but the trend stayes the same.

Having a closer look at the other type of charts in the Appendix A.3 you can see, although some of the tests fail, it is never a total failure. The failed tests often are just below that red line, i.e. mostly just about one to three sequence to much fail. And even if Non-overlapping Template Matching Tests fail, there are not many tests that fail – at most about five out of the 148 individual tests – and there exists no recognizable pattern among the failing tests.

All together there are two remarkable problems to this algorithm. First, to eliminate the the number of failed tests. One idea therefore, is to reduce the number of multiple key usages. Second, to enhance the key extension algorithm. After all, this is responsible for how well the computation graph is constructed. Experimenting with other operations can lead to improved performance, too.

3.2. Performance

Here the average performance of different parts of the algorithm, as described in Chapter 2, are meassured. To get a satisfying overview, the algorithm is performed on different processor architectures and with different number of rounds.

The following numbers represent a kind of lower bound for the performance of this algorithm, hence its implementation is not focused on performance, but on easy and fast adaption on new ideas (which occured frequently during the development).

MacOS X 10.6 with 2.4 GHz Intel Core i5

Rounds	Encryption Speed	Decryption Speed	Key Extension
1	$\sim$ 59 Mbit/sec	$\sim$ 59 Mbit/sec	$\sim$ 0.00006 sec/extension
5	$\sim$ 11 Mbit/sec	$\sim$ 12 Mbit/sec	$\sim$ 0.00074 sec/extension
9	$\sim$ 6 Mbit/sec	$\sim$ 6 Mbit/sec	$\sim$ 0.00226 sec/extension
10	$\sim$ 6 Mbit/sec	$\sim$ 5 Mbit/sec	$\sim$ 0.0028 sec/extension
15	$\sim$ 4 Mbit/sec	$\sim$ 4 Mbit/sec	$\sim$ 0.00604 sec/extension

Linux, CENTOS 5.5 with 2,3 GHz AMD Opteron 8356 (Barcelona)

Rounds	Encryption Speed	Decryption Speed	Key Extension
1	$\sim$ 92 Mbit/sec	$\sim$ 80 Mbit/sec	$\sim$ 0.00004 sec/extension
5	$\sim$ 18 Mbit/sec	$\sim$ 18 Mbit/sec	$\sim$ 0.0005 sec/extension
9	$\sim$ 9 Mbit/sec	$\sim$ 9 Mbit/sec	$\sim$ 0.00154 sec/extension
10	$\sim$ 8 Mbit/sec	$\sim$ 8 Mbit/sec	$\sim$ 0.00192 sec/extension
15	$\sim$ 6 Mbit/sec	$\sim$ 6 Mbit/sec	$\sim$ 0.00418 sec/extension

Linux, CENTOS 5.5 with 2,93 GHz Intel Nehalem-EP

Rounds	Encryption Speed	Decryption Speed	Key Extension
1	$\sim$ 145 Mbit/sec	$\sim$ 145 Mbit/sec	$\sim$ 0.00002 sec/extension
5	$\sim$ 31 Mbit/sec	$\sim$ 30 Mbit/sec	$\sim$ 0.0003 sec/extension
9	$\sim$ 15 Mbit/sec	$\sim$ 15 Mbit/sec	$\sim$ 0.00094 sec/extension
10	$\sim$ 13 Mbit/sec	$\sim$ 13 Mbit/sec	$\sim$ 0.00112 sec/extension
15	$\sim$ 9 Mbit/sec	$\sim$ 9 Mbit/sec	$\sim$ 0.00242 sec/extension

Rounds	NonOverl	RandExVar	RandEx	Other	Sum
1	6	1			7
2	7				7
3	6			Serial	7
4	8	4			12
5	6		2	Longest Runs	9
6	6	3	1		10
7	8	3		Approx. Entr.	12
8	8				8
9	3	1	1		5
10	8		1	Universal, Freq, CumSum, Overlapping	13
11	5	1	2	Block Freq.	9
12	7	1		Serial, Run	10
13	8	3	1	LinCompl	13
14	6	1		FFT, Longest Runs	9
Sum	92	18	8	13	131

Table 3.1.: Number of datasets, in which a specific test fails.

4. Conclusion and Further Researches

Here a new encryption algorithm is developed, whose computation graph depends on the key. The algorithm is analysed with the same methodes as the *Advanced Encryption Algorithm (AES)* during its standardisation process. Though the algorithm does not always fulfill all the tests and especially has a big problem with the Non-overlapping Template Matching Test. It could be said that it is a secure encryption algorithm, which introduces new ideas no other official encryption algorithm implements. In addition, the failing tests always came close to passing.

However, there are still tasks to do. The encryption algorithm has to be enhanced, so that it finally fulfills all tests from the Statistical Test Suite. In the first place, this means to fulfill the Non-overlapping Template Matching Tests. Currently the algorithm uses parts of the key very frequently: for the row of operations and for the Feistel network, especially with the *E-function*. This may lead to too many patterns in the resulting cyphertext. Furthermore the key extension method is not fully satisfying, as keys with a more dense filling of 1's perform slightly better than keys with a less dense filling of 1's.

When this is done, the code has to be optimised. Right now, the code is developed in a way, that allows fast adoption to new ideas: Other operations, other *F-functions*, etc. . Admittedly this leads to slow encryption / decription results with about 6 to 13 Mbit/s. I believe, this can be increased dramatically.

Finally, the algorithm has to be analysed with different key lengths. Here, solely 128-bit keys are tested, but to pass the second round of the standardisation process, key lengths of at least 192-bit and 256-bit should be tested. Partial round testings have already been performed in here and show, that the algorithm is approximately equally strong already in one round.

A. Appendix

A.1. Extract Parameter

Let *Row* be a data structure containing the parameter extracted from one 32-bit key word k. The function createPermutation() creates a permutation of the $S_{textSize}$ from the first parameter.

```
  // calculate the permutation, there are textSize! many
row.addrPerm = createPermutation(&k, textSize);

  // block-wise permutation operand 1 and 2
row.paddr1 = k % textSize;
k /= textSize;
row.paddr2 = k % (textSize-1);
k /= (textSize-1);
  // they must not be the same
if(row.paddr2 >= row.paddr1) ++row.paddr2;

  // select block-wise permutation
row.perm = k % NUM_OF_PERM;
k /= NUM_OF_PERM;

  // right rotation by 7
k = RROTATION(k,7);
  // the order of the operations
ret.opOrder = createPermutation(&x, 4);
  // select first operation
row.ops[0] = k % NUMBER_OF_OPERATION;
k /= NUMBER_OF_OPERATION;
for (i = 1; i < nTB; ++i) {
    // select other operations
  row.ops[i] = k % (NUMBER_OF_OPERATION-1);
  k /= (NUMBER_OF_OPERATION-1);
    // with none the same as its predecessor
  if(row.ops[i] >= row.ops[i-1]) ++(row.ops[i]);
}
  // F-function for feistel network
row.feistelOp = k % FEISTEL_OPS;
```

Listing A.1: Extract parameters

A.2. Encryption and Decryption

Let *Koos* be a data structure containing the key size, the text size, the number of rounds, an array with all parameters called rows (see Section A.1). The function permutateS8() permutates the second parameter with the permutation given with the first parameter. The array operators contains the function pointer to the corresponding operations. The array inverseOperators contains the function pointers to the corresponding inverse operations.

```
void encryptRow(const Koos *s, uint32_t *text)
{
  int32_t i,j,k;
  uint32_t prev, *tmp, currKey;
  Row *r;

  tmp = copyArray(text, s->textSize);
  // for each row
  for (j = 0; (uint32_t)j < s->rowSize; ++j)
  {
    // current row
    r = &s->rows[j];

    for (k = 0; k < 4; ++k) {
      switch (permutateS8(r->opOrder, k)) {
        case ADDRESS_PERMUTATION:
          for (i = 0; i < s->textSize; ++i)
            text[i] = tmp[i];
          for (i = 0; i < s->textSize; ++i)
            tmp[i] = text[permutateS8(r->addrPerm, i)];
          break;

        case ROW_OF_OPERATION:
          if((j % 2) == 0) { // even rounds
            prev = s->key[j];
            // for each textblock => operation
            for (i = 0; i < s->textSize; ++i) {
              currKey = s->key[(j+3*i)%s->keySize];
              tmp[i] = operators[r->ops[i]](prev, tmp[i]);
              prev = multOp(currKey, tmp[i]);
            }
          } else { // odd rounds
            prev = s->key[(j+3*(s->textSize-2))%s->keySize];
            // for each textblock => operation
            for (i = s->textSize - 1; i >= 0 ; --i) {
              currKey = s->key[(j+3*i)%s->keySize];
              tmp[i] = operators[r->ops[i]](prev, tmp[i]);
              prev = multOp(currKey, tmp[i]);
            }
          }
          break;

        case FEISTEL_NETWORK:
```

```
          for (i = 0; i < s->textSize; ++i)
            lawine(s, tmp, i, r->feistelOp, (j+i)%s->rowSize);
          break;

        case BLOCKWISE_PERMUTATION:
          // two byte permutation on the
          // four bytes of r->paddr1 and r->paddr2
          permutate(&tmp[r->paddr1], &tmp[r->paddr2], r->perm);
          break;

        default:
          break;
      }
    }

    // write results
    for (i = 0; i < s->textSize; ++i)
      text[i] = tmp[i];
  }
  FREE(tmp);
}
```

Listing A.2: Encryption

```
void decryptRow(const Koos *s, uint32_t *cypher)
{
  int32_t j,i, k;
  uint32_t prev, currKey, *tmp;
  Row *r;

  tmp = copyArray(cypher, s->textSize);
  // for each row from bottom to top
  for (j = s->rowSize - 1 ; j >= 0; --j)
  {
    r = &s->rows[j]; // current row

    for (k = 3; k >= 0; --k) {
      switch (permutateS8(r->opOrder, k)) {
        case BLOCKWISE_PERMUTATION:
          // inverse two byte permutation on the
          // four bytes of r->paddr1 and r->paddr2
          permutateInv(&tmp[r->paddr1], &tmp[r->paddr2], r->perm);
          break;

        case FEISTEL_NETWORK:
          for (i = s->textSize-1; i >= 0; --i)
            lawineInv(s, tmp, i, r->feistelOp, (j+i)%s->rowSize);
          break;

        case ROW_OF_OPERATION:
          if((j % 2) == 0) { // even rounds
            prev = s->key[j];
            // for each textblock => inv-operation
```

```
            // and write to inv-perm-address
            for (i = 0; i < s->textSize; ++i)
            {
              currKey = s->key[(j+3*i)%s->keySize];
              cypher[i] = inverseOperators[r->ops[i]](prev, tmp[i]);
              prev = multOp(currKey, tmp[i]);
              tmp[i] = cypher[i];
            }
          } else { // odd rounds
            prev = s->key[(j+3*(s->textSize-2))%s->keySize];
            // for each textblock => inv-operation
            // and write to inv-perm-address
            for (i = s->textSize - 1; i >= 0 ; --i)
            {
              currKey = s->key[(j+3*i)%s->keySize];
              cypher[i] = inverseOperators[r->ops[i]](prev, tmp[i]);
              prev = multOp(currKey, tmp[i]);
              tmp[i] = cypher[i];
            }
          }
          break;

        case ADDRESS_PERMUTATION:
          for (i = 0; i < s->textSize; ++i)
            cypher[permutateS8(r->addrPerm, i)] = tmp[i];
          for (i = 0; i < s->textSize; ++i)
            tmp[i] = cypher[i];
          break;
        default:
          break;
      }
    }
  }
  // write results
  for (i = 0; i < s->textSize; ++i)
    cypher[i] = tmp[i];
  FREE(tmp);
}
```

Listing A.3: Decryption

A.3. Detailed Analysis Charts

This second type of chart shows, for a certain number of rounds, for every test the propotion of successful sequences (the blue rhombs) and a red line representing the minimum propotion. This minimum propotion is calculated and dependent on the number of sequences (see Section 3.1.2) and as the tests Random Excursion and Random Excursion Variant do not use all provided sequences in all tests, a gap arises in the line. This chart can be interpreted as follows: Every rhomb above the line is a successful test and every rhomb below the line is a failed test.

The order of the rhombs matches the order of the tests in the resulting file from the STS: Frequency, Block Frequency, two times Cumulative Sums, Runs, Longest Run, Rank, FFT, 148 times Non-overlapping Template Matching, Overlapping Template Matching, Universal, Approximate Entropy, eight times Random Excursions, 18 times Random Excursions Variant, two times Serial and finally, Linear Complexity.

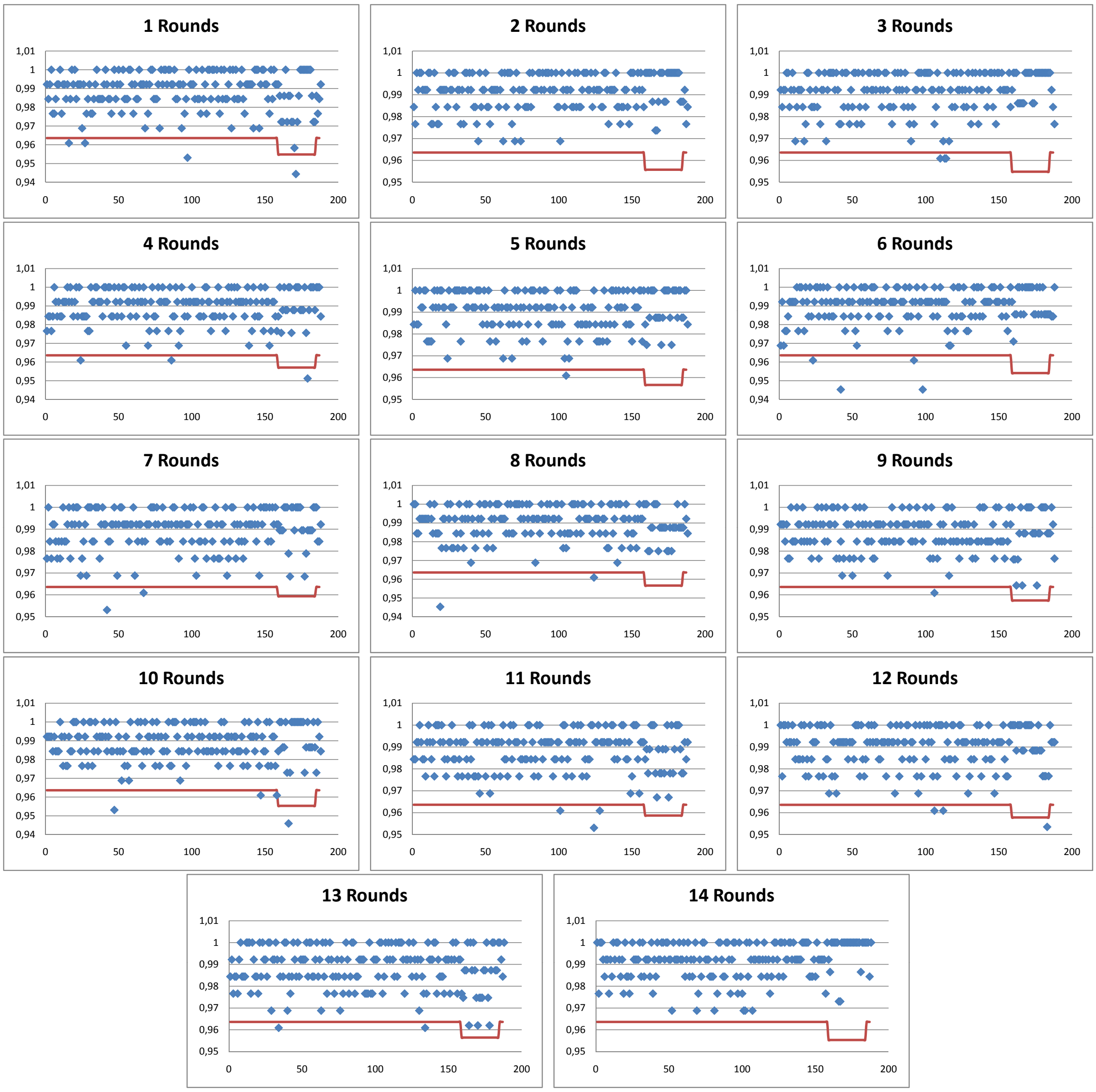

Figure A.1.: High Density Key: detailed results

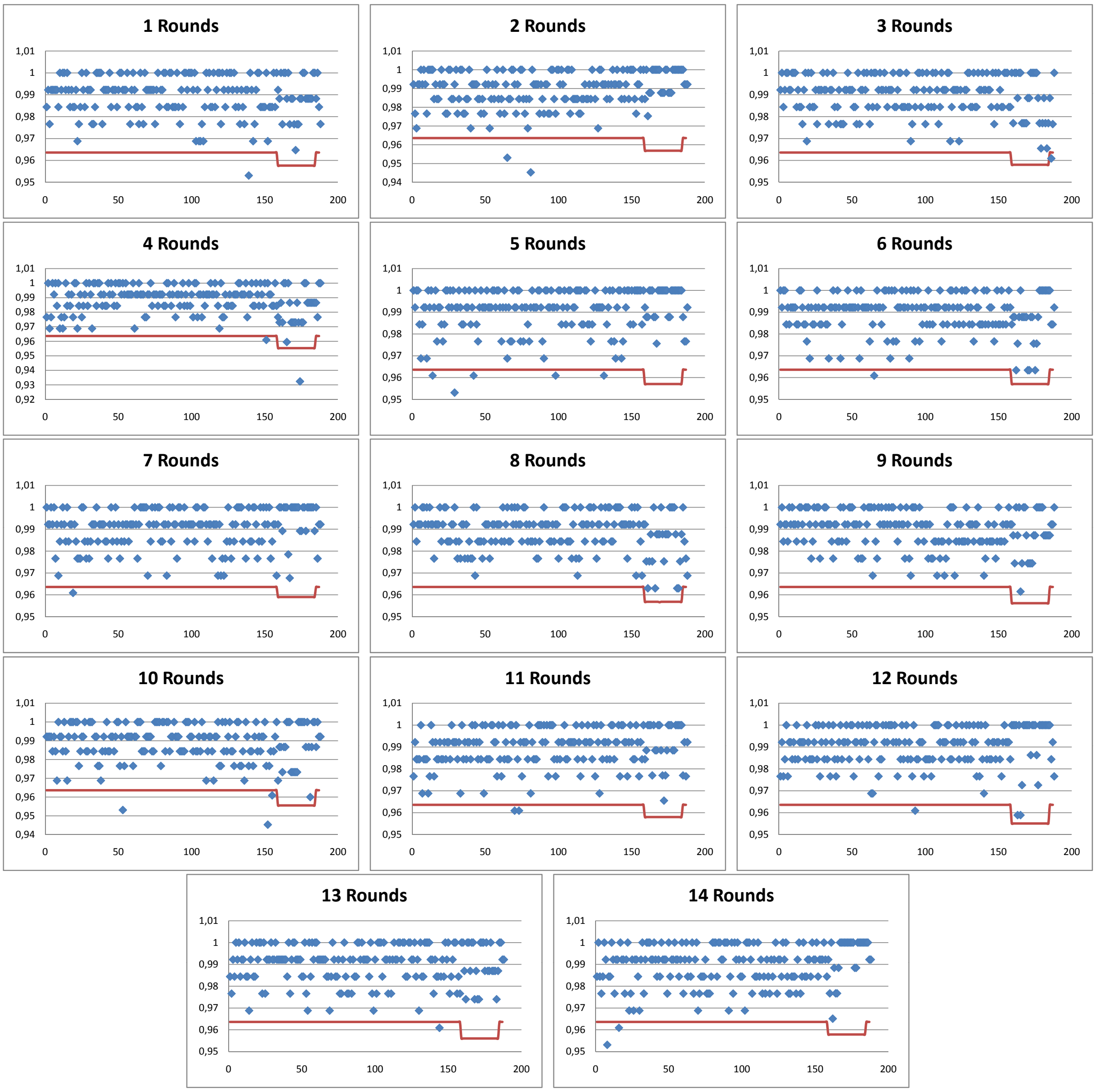

Figure A.2.: High Density Plaintext: detailed results

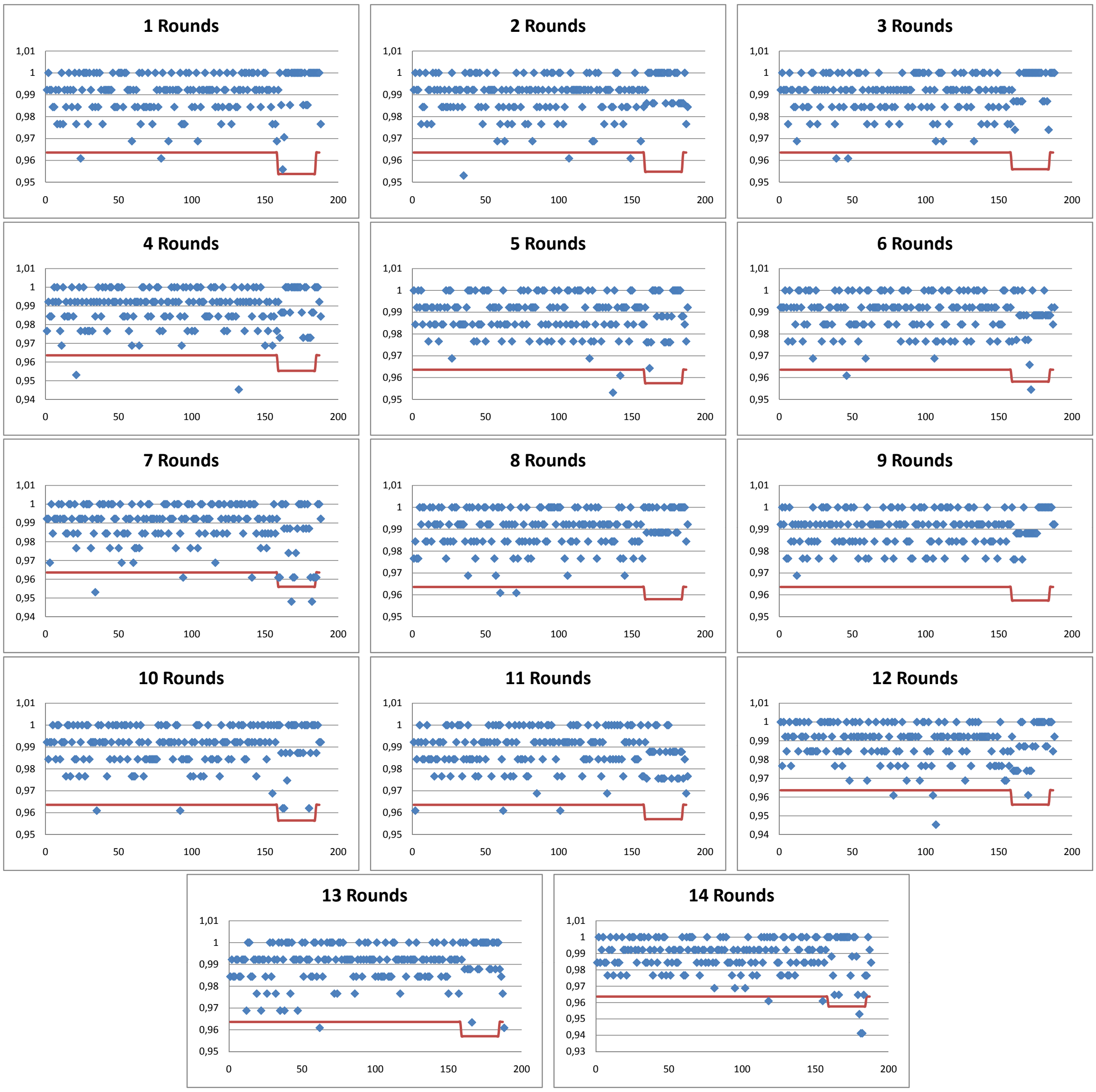

Figure A.3.: Low Density Key: detailed results

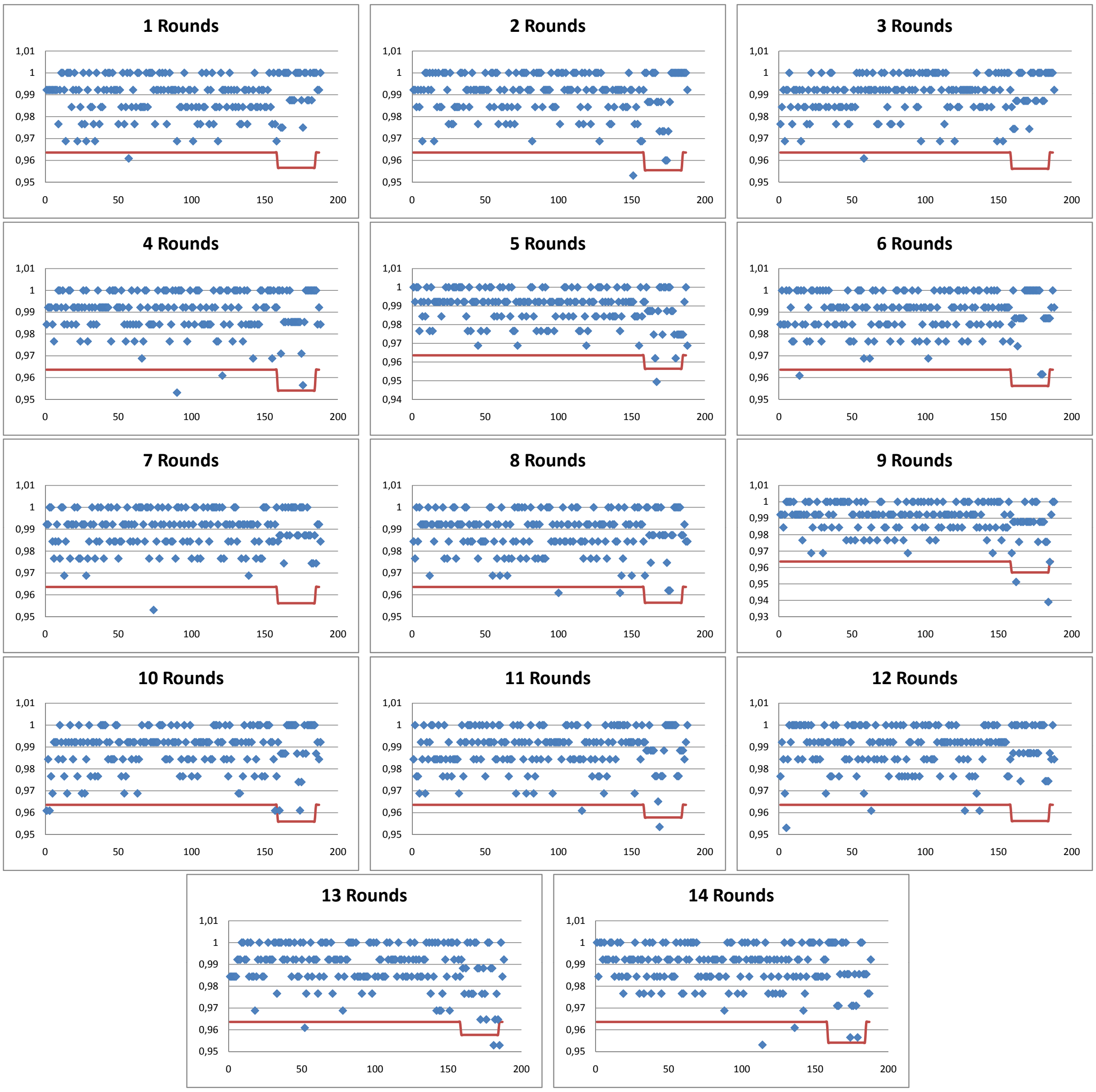

Figure A.4.: Low Density Plaintext: detailed results

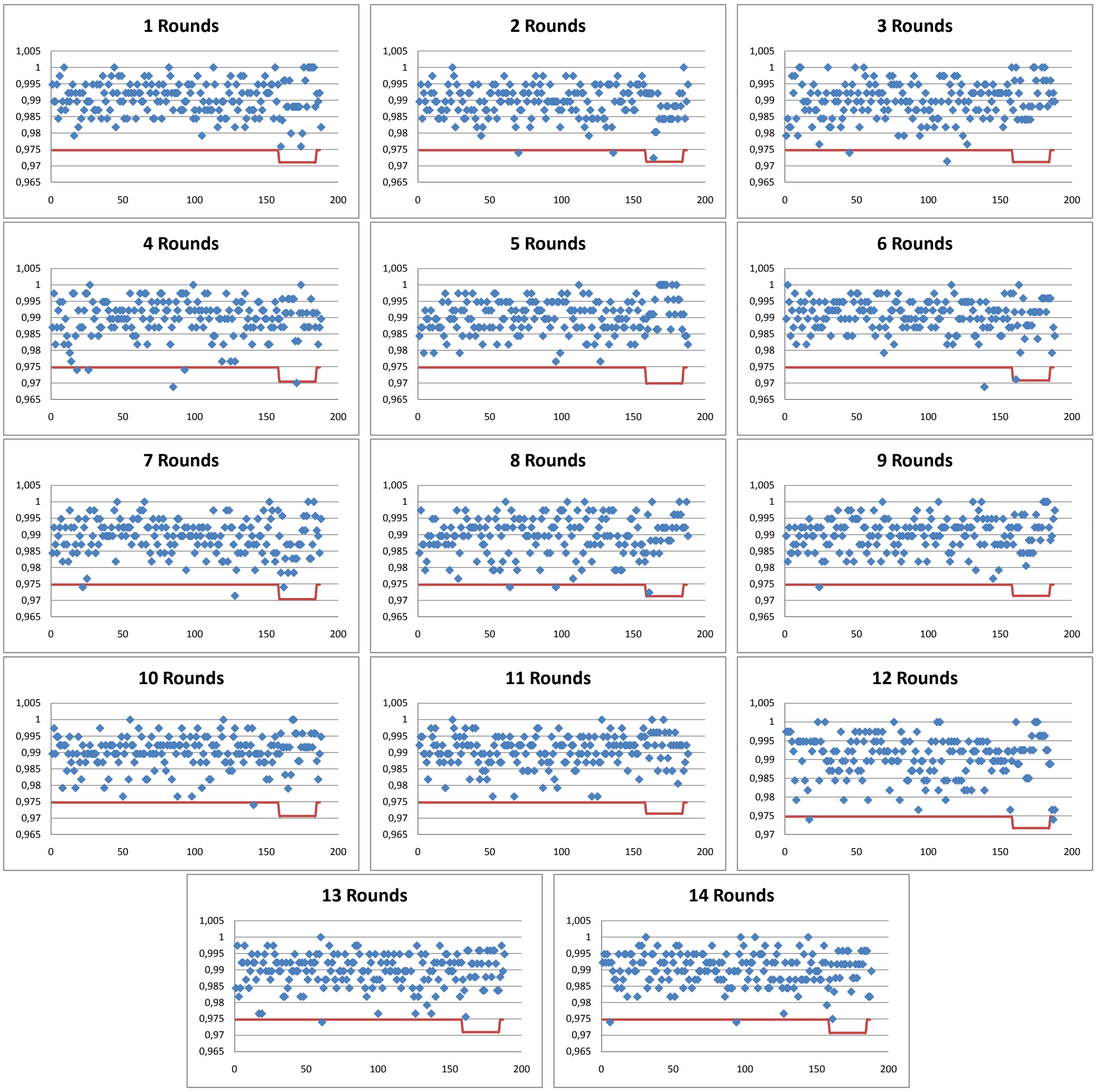

Figure A.5.: Key Avalange: detailed results

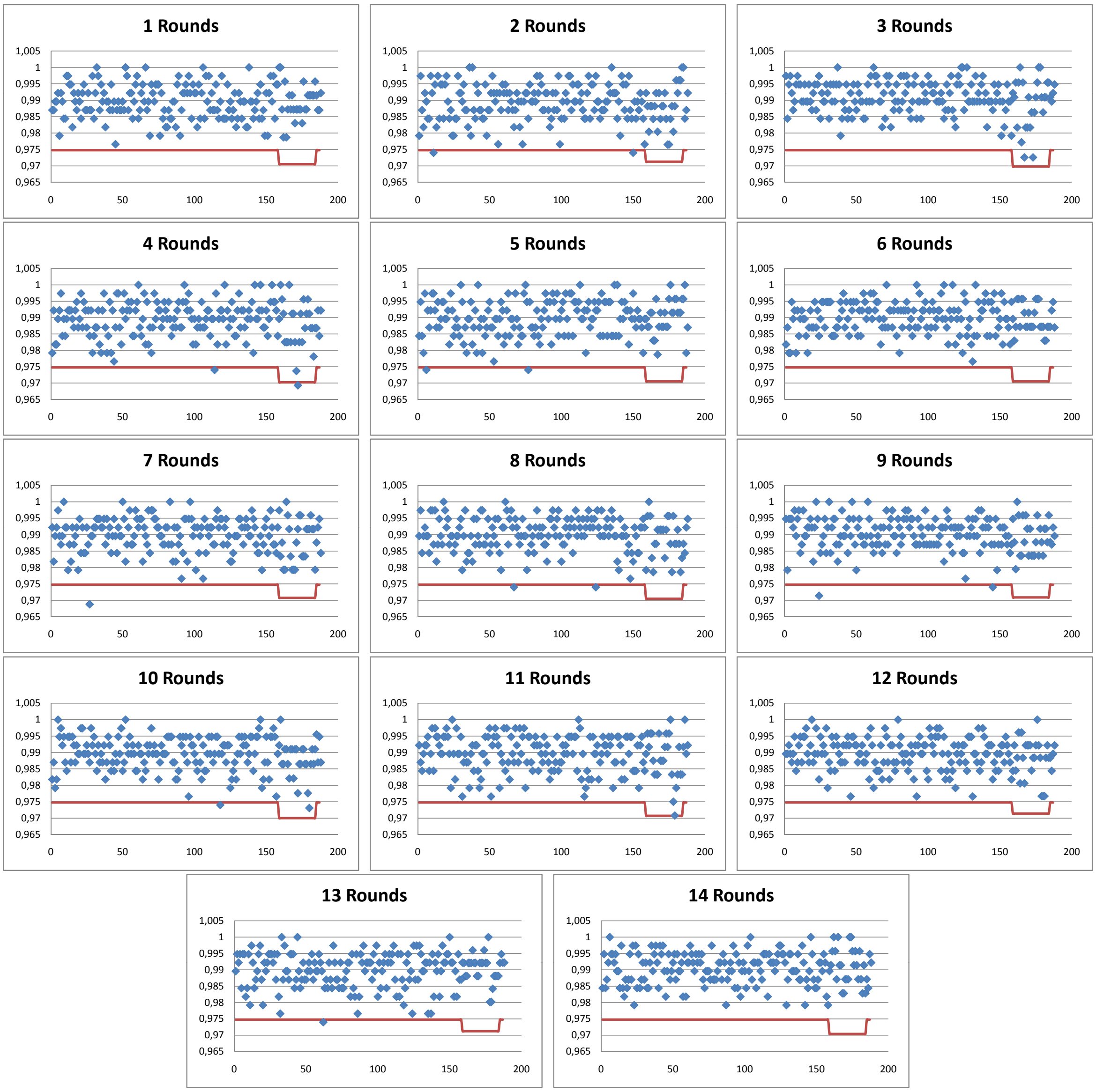

Figure A.6.: Plaintext Avalange: detailed results

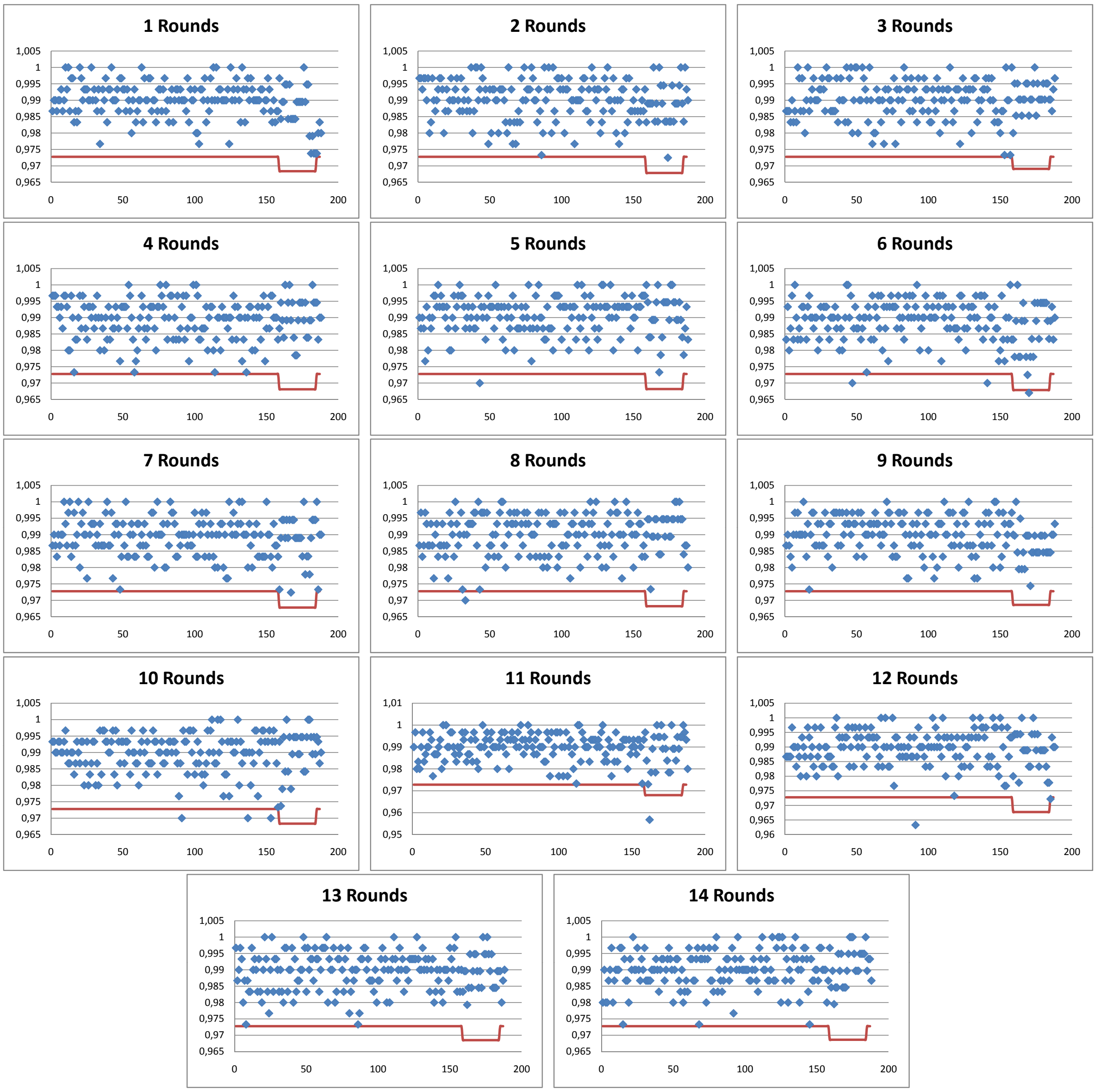

Figure A.7.: CBC Mode: detailed results

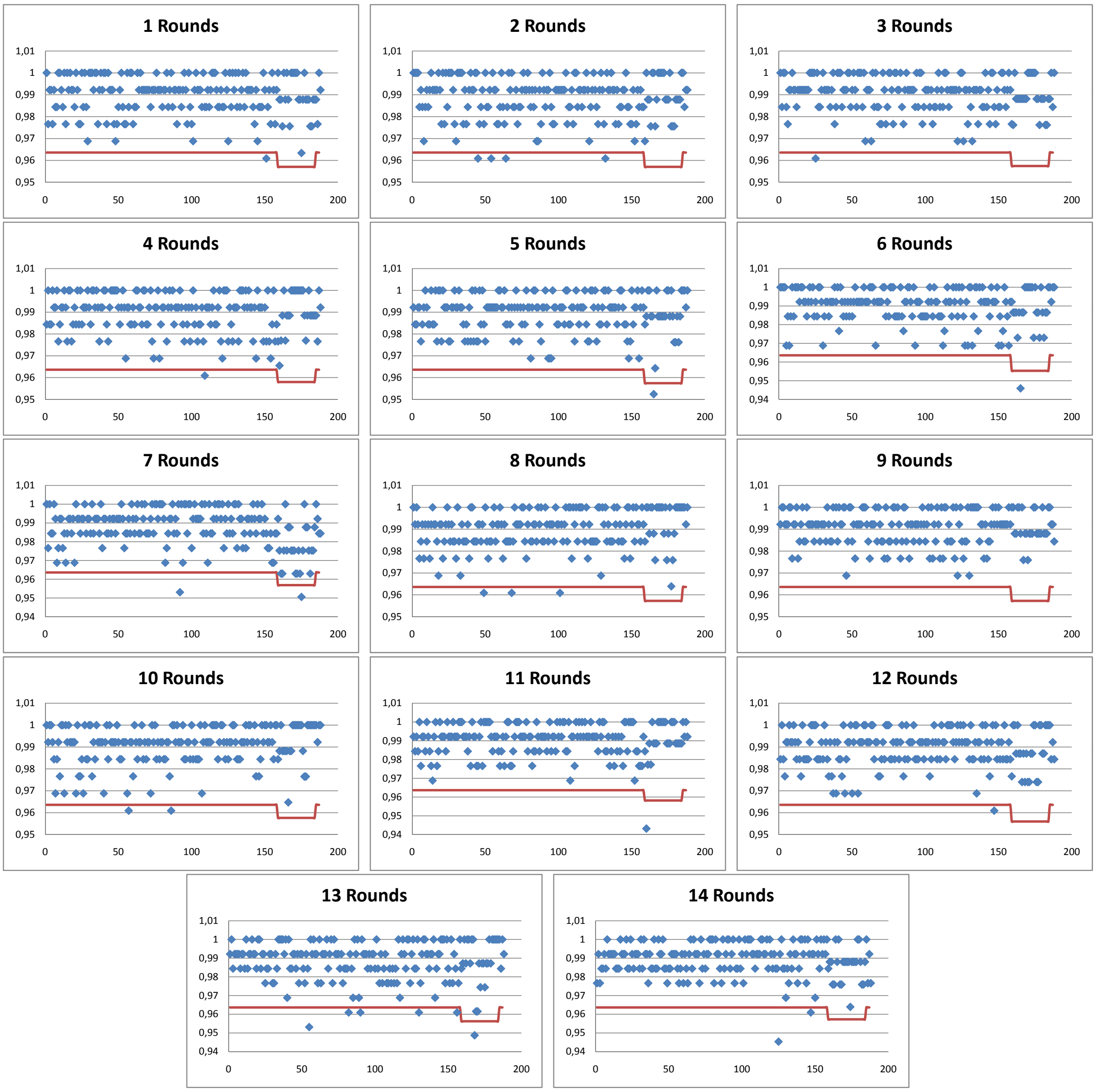

Figure A.8.: Random Plaintext / Cyphertext: detailed results

Bibliography

[1] Information Technology Laboratory (National Institute of Standards and Technology): Announcing the ADVANCED ENCRYPTION STANDARD (AES). Gaithersburg, MD 20899-8930, November 2001. – Technical Report. – Federal Information Processing Standards Publication 197

[2] Anderson, Ross ; Biham, Eli ; Knudsen, Lars: Serpent: A Flexible Block Cipher With Maximum Assurance. In: *In The First Advanced Encryption Standard Candidate Conference*, 1998

[3] Buchmann, Johannes A.: *Introduction to Cryptography.* Springer-Verlag New York, Inc., 2000

[4] Carolynn Burwick, Edward D'Avignon Rosario Gennaro Shai Halevi Charanjit Jutla Stephen M. Matyas Jr. Luke O'Connor Mohammad Peyravian David Safford Nevenko Z. Don Coppersmith C. Don Coppersmith: MARS - a candidate cipher for AES, 1999

[5] Contini, Scott ; Yin, Yiqun L.: On differential properties of data-dependent rotations and their use in MARS and RC6 (Extended Abstract). In: *Proceedings of The Second AES Candidate Conference*, S. 230–239

[6] Juan Soto, Jr.: Randomness Testing of the Advanced Encryption Standard Candidate Algorithms. In: *NIST IR 6390, National Institute of Standards and Technology.* Gaithersburg, MD 20899-8930, September 1999

[7] Rivest, Ronald L. ; Robshaw, M. J. B. ; Sidney, R. ; Yin, Y. L.: The RC6 TM Block Cipher. In: *Tn First Advanced Encryption Standard (AES) Conference*, 1998

[8] Rukhin, Andrew ; Soto, Juan ; Nechvatal, James ; Smid, Miles ; Barker, Elaine ; Leigh, Stefan ; Levenson, Mark ; Vangel, Mark ; Banks, David ; Heckert, Alan ; Dray, James ; Vo, San: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications / National Institute of Standards and Technology. Version: April 2010. `http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html`. Gaithersburg, MD 20899-8930, April 2010. – Technical Report. – Special Publication 800-22, Revision 1a

[9] Schneier, Bruce ; Kelsey, John ; Whiting, Doug ; Wagner, David ; Hall, Chris ; Ferguson, Niels: Twofish: A 128-Bit Block Cipher. In: *In First Advanced Encryption Standard (AES) Conference*, 1998

[10] Soto, Juan ; Bassham, Lawrence: Randomness Testing of the Advanced Encryption Standard Finalist Candidates. In: *NIST IR 6483, National Institute of Standards and Technology*. Gaithersburg, MD 20899-8930, March 2000

[11] Tönnis, Andreas: *Implementierung und Analyse eines Verschlüsselungsverfahrens mit schlüsselgesteuerter Operationsauswahl*, RWTH Aachen, Bachelorthesis, April 2010

Statement of Authorship

I declare that this document and the accompanying code has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. It has not been accepted in any previous application for a degree. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Aachen, March 27, 2011,

Tammo Ippen