Degree project

Algorithm and software development for security estimation of SPN-based block cipher against related-key attacks

Author: Dmytro Kaidalov  
Supervisor: Mathias Hedenborg  
External Supervisor: Roman Oliynykov, Kharkiv National University of Radioelectronics

Date: 2014-06-05

Course Code: 5DV00E, 30 credits  
Level: Master

Department of Computer Science
Acknowledgements

I would like to express gratitude to my supervisor Ph.D. Roman Oliynykov, who was really interested in my work, encouraged me, stimulated and helped me with this thesis. I also thank to my friends for supporting me during hard working days.
Abstract

Symmetric block ciphers are among the most widely used cryptographic primitives. In addition to providing privacy via encryption, block ciphers are used as basic components in the construction of hash functions, message authentication codes, pseudorandom number generator, as part of various cryptographic protocols and etc. One of the most popular block ciphers nowadays is AES (Advanced Encryption Standard), which has been used as a standard of encryption in many countries of the world. In spite of popularity of this cipher a huge attack was found on its key-expansion algorithm some years ago. That is why it is important to analyze carefully this component and understand what weak points admit attacks. Since we know that we can improve existing algorithm to protect cipher from attacks or build up a new algorithm taking into account founded weaknesses so there will be no chance to break it with existing knowledge.

The goal of this project is to create some method which can estimate security of encryption algorithm against related-key attacks. For this reason the perspective block cipher is introduced. This cipher is a candidate to the public standard of encryption in Ukraine so that is why this research is very important. Actually the introduced method of estimation is created especially for this cipher but also can be used for other ciphers based on the substitution-permutation network. The developed method was applied to the cipher and results are represented in the report. Also the complexity estimation of this algorithm is expressed. The software implementation is described in the last chapter of report.

Key words: cryptography, cryptanalysis, block cipher, encryption, related-key attack, key-expansion algorithm, substitution-permutation network
### Content

1 Introduction ........................................................................................................................................... 1  
  1.1 Problems and Motivation ................................................................................................................. 1  
  1.2 Goals and Criteria ............................................................................................................................. 1  
  1.3 Structure of Report ............................................................................................................................ 2  
2 Cryptographic algorithms in modern information security systems .............................................. 3  
  2.1 Using cryptography in modern information security systems ......................................................... 3  
  2.2 The role of block ciphers in modern cryptography ............................................................................ 3  
  2.3 The actuality of developing the new standard for block symmetric encryption ............................ 5  
3 A proposal for the new standard of block encryption ....................................................................... 6  
  3.1 Cipher description ............................................................................................................................ 6  
    3.1.1 General parameters ..................................................................................................................... 6  
    3.1.2 Basic transformations ................................................................................................................. 7  
      3.1.2.1 Key addition ......................................................................................................................... 7  
      3.1.2.2 Bytes substitution .................................................................................................................. 8  
      3.1.2.3 Shift rows ............................................................................................................................. 8  
      3.1.2.4 Mix columns ......................................................................................................................... 9  
    3.1.3 Key expansion scheme ............................................................................................................... 9  
    3.1.4 Encryption transformation ........................................................................................................ 11  
  3.2 Formulation of the problem .............................................................................................................. 12  
4 The algorithm for estimation cipher security against related-key attacks .................................... 13  
  4.1 Description of related-key attack .................................................................................................... 13  
  4.2 Proposed method for cipher security estimation ........................................................................... 14  
  4.3 The algorithm for counting active bytes ....................................................................................... 15  
    4.3.1 General description .................................................................................................................... 16  
    4.3.2 Key addition ............................................................................................................................. 17  
    4.3.3 Bytes substitution ...................................................................................................................... 17  
    4.3.4 Shift rows ................................................................................................................................. 17  
    4.3.5 Mix columns ............................................................................................................................ 18  
  4.4 Description of results ....................................................................................................................... 19  
  4.5 Complexity of algorithm ............................................................................................................... 21  
5 Description of program implementation .......................................................................................... 24  
6 Conclusions and future work ........................................................................................................... 28  
References ................................................................................................................................................ 29  
Appendix A – Source code in C++ ....................................................................................................... 30
Figures List
Figure 2.1 – General model of protected information and telecommunication system.... 4
Figure 3.1 – Representation of bytes in 128-bits version of cipher.......................... 6
Figure 3.2 – Representation of bytes in 256-bits version of cipher.......................... 6
Figure 3.3 – Representation of bytes in 512-bits version of cipher.......................... 7
Figure 3.4 – Bytes substitution............................................................................. 8
Figure 3.5 – Shifts in 128-bits block.................................................................... 9
Figure 3.6 – Shifts in 256-bits block.................................................................... 9
Figure 3.7 – Shifts in 512-bits block.................................................................... 9
Figure 3.8 – Computing of intermediate value $K_t$............................................. 10
Figure 3.9 – Computation of round keys............................................................... 11
Figure 3.10 – Main encryption loop................................................................. 12
Figure 4.1 – Local collisions in AES[5]................................................................ 13
Figure 4.2 – AES-256 key schedule codeword (4.5 key-schedule rounds)[5]......... 14
Figure 4.3 – Differential characteristic for $K_t$ computation............................... 15
Figure 4.4 – Computation of active bytes........................................................... 16
Figure 4.5 – Key addition..................................................................................... 17
Figure 4.6 – Shift rows........................................................................................ 17
Figure 4.7 – Shift rows....................................................................................... 18
Figure 4.8 – Mix columns.................................................................................... 19
Figure 4.9 – $K_t$ computation............................................................................. 20
Figure 4.10 – Round key computation.............................................................. 20
Figure 4.11 – Main encryption loop................................................................. 21
Figure 5.1 – Diagram of classes.......................................................................... 24
Figure 5.2 – Program output............................................................................. 27

Tables List
Table 3.1 – Acceptable combinations of blocks and keys .................................... 7
Table 3.2 – Amount of encryption rounds for different versions of cipher............ 7
Table 3.3 – Amount of bytes for shifting depends on the block size...................... 8
Table 4.1 – Amount of branches depends on amount of non-zero bytes in column ... 18
Table 4.2 – Description of best differential characteristic...................................... 19
Table 4.3 – Amount of branches depends on the amount of active bytes in column ... 22
1 Introduction

Information technologies at this stage play a crucial role in the human life. Computers have been spread everywhere from large corporations to individual home users. Obviously computer provides us with many useful services however network communication between stations is more important nowadays. In the last few decades the worldwide Internet network has received wide popularity. It allows to unite computers which are located in the different parts in the world. Nevertheless, Internet is an open network where data is transmitted over unsecure environment and the question about data protection is arising immediately. In particular, it is required to provide the following basic services for data protection: confidentiality, integrity, authenticity and etc. These services can be provided by cryptographic protection of information.

One of the main cryptographic primitives is a symmetric block cipher which is historically came out first and nowadays is the most common used cryptographic transformation along with asymmetric algorithms [1]. With help of block ciphers next problems can be solved: data encryption (guarantee of confidentiality), computation of hashes (guarantee of integrity), generation of pseudorandom numbers and others. Definitely the main advantage of block ciphers is the speed of operations that makes them indispensable in the organization of high-speed secure data channels [2].

The one of the most important thing when designing a block symmetric cipher is an assessment of its safety against known attacks. Modern cipher must have a sufficient level of safety so it is not possible to be attacked with the use of modern computational capabilities.

In the last years the SPN-based ciphers became especially popular. It is such construction in base of which the substitution-permutation network is used. The most popular algorithm of this class is an AES (Advanced encryption standard) [3], which is widely used in the modern world.

1.1 Problems and Motivation

From [5,6,7] it is known that related-key attacks were found for algorithms AES-192 and AES-256 which significantly reduced the security level of these algorithms. That is why the question about new SPN-based block cipher which would be secure from attacks which is present in AES is arisen.

In this report the new SPN-like cipher is presented. Authors propose this cipher as a perspective algorithm [4] which is based on the AES but without weaknesses that are present in it. As it is known from [5,6,7] related-key attacks were found for algorithms AES-192 and AES-256 that significantly reduced the security level of these algorithms. New algorithm of encryption was proposed to the public competition of block cipher selection to be a prototype during development of Ukrainian National Standard [4]. That is the reason of researching this cipher.

But immediately the next question arose: how to prove the security of this algorithm against related-key attacks which is applicable to AES cipher? This is the main problem. This is a challenging problem to solve since for now there are no efficient algorithms for estimating cipher security against related-key attacks.

1.2 Goals and Criteria

The following goals and criteria are defined for solving above problem:

1. The first goal is to understand the architecture of proposed algorithm. Find the difference between AES and new algorithm. This goal is reached when all blocks of new cipher are described in details and the main properties and differences from AES are defined.
2. For the next step we need to research the related-key attack. How this attack works for AES algorithm. This goal is reached when attack is described and examples are provided.

3. The third goal is to develop the algorithm for estimating security of cipher against related-key attack. This goal is reached when the developed algorithm is described in details. This algorithm should give some numerical value with help of which the estimation of cipher can be done.

4. The fourth goal is to implement mentioned algorithm and then apply it to the described cipher. This goal is reached when the developed algorithm gives the numerical value which estimates the cipher. We should find out if the suggested cipher is secure or not.

5. The fifth goal is to calculate the complexity of developed algorithm. This goal is reached when formulas to calculate complexity and concrete numerical values of complexity are represented.

1.3 Structure of Report
Section 2 is the introduction section. It describes how cryptographic algorithms are used in the modern security systems. Most of the information in this section is related to block ciphers.

Section 3 describes the proposed algorithm of encryption. All part of the cipher is described in details.

Section 4 is about developed algorithm for estimation security of cipher against related-key attacks. It starts from general description of related-key attack with examples for AES. And then developed algorithm is described in details. Also final results and calculation of complexity is represented in this section.

Section 5 represents overview of program implementation. It does not describe in details all code sources but gives general overview how the algorithm is implemented.

Section 6 represents conclusion and future work.
# 2 Cryptographic algorithms in modern information security systems

Nowadays one of the most important components of any country’s development is its level of informatization. This is the reason that technologies of processing information are widely spread in most of the countries all over the world. Information and telecommunication systems are being integrated in almost all spheres of society’s life. Information and telecommunication systems play considerably important role in such areas as government administration, economics, education, military and others.

## 2.1 Using cryptography in modern information security systems

During functioning of information & telecommunication system information processing is performed. Processing of information implies the following operations: collection, storage, modification, transmission, destruction and a number of other operations [8]. For data transmission different telecommunication systems can be used. For example, it can be internal communication lines, as well as external ones with connection to global international systems.

Nowadays the use of electronic documents is becoming very popular. Electronic processing of documents is widely used in people’s everyday life. It is very convenient to use electronic documents for performing financial operations in telecommunication systems. The number of financial transactions which are performed in electronic way is increasing every year.

However, the spread of electronic documents causes development of new types of fraud connected with this activity. In recent years the number of fraud cases concerning credit cards, electronic securities, etc. are constantly increasing. Banking industry suffers from this type of attack most of all. These matters are becoming more and more topical all over the world because of rising popularity of electronic payments among the people.

Open communicational channels are usually used to perform different operations with electronic documents. That is why it is mandatory to ensure that the data which is transmitted is properly protected. The basic security services should be provided for the following types of operation: confidentiality, integrity, authenticity and others [8]. For this reason special systems of complex information protection are created. One of the most important components of complex protection systems is the subsystem of cryptographic protection of information [8]. With help of such complex systems it is possible to implement encryption of information (for providing confidentiality), electronic digital signature for messages (for providing integrity and authenticity), etc.

## 2.2 The role of block ciphers in modern cryptography

Special role among the tools of information protection belongs to block symmetric ciphers which are widely used in cryptography nowadays. Block ciphers can be used not only for encryption of information but also for computation of hash sums, generating pseudorandom sequences of digits, etc. [1].

First block ciphers have appeared in ancient times. Nowadays there are a lot of different algorithms of block symmetric encryption. All leading countries have special standards which manage using of cryptographic algorithms for data protection. For example, in the USA there is such standard of block symmetric encryption as FIPS-197 [3] (also known as Rijndael), in Russia GOST 28147-89 algorithm [9] developed in Soviet Union but still is in use.
We have to consider a general model of cryptographic protection of information in information and telecommunication system for evaluating the role of block cipher in it. General model can be shown with the help of the next scheme (Figure 2.1) [8].

![General model of protected information and telecommunication system](image)

In figure 2.1 $MS_1$ and $MS_2$ are the message sources which generate messages $M_i$ for transmitting over the communication channel or for storing in the database $DB$. From the source message $M_i$ is transmitted to the authentication service ($AS_1$ and $AS_2$), where the integrity and authenticity of the message is checked with help of the key which is taken from key service $KS$. The outcome of the authentication service is protected message which goes to the encryption service ($E_1$ and $E_2$). Encrypted message is transmitted over the communication channel ($TS$ — telecommunication system) with help of the transmitter $T$ or stored in the database $DB$. In the telecommunication system, which operates over the open communication channel, information can be accessed by the cryptanalyst $KA$ which can try to compromise transmitted message or key information.

At the receiving side all operations are repeated in the reversed order to get the original message.

Let us consider the role of block cipher in such scheme. With the help of the authentication service integrity and authenticity of messages are provided. For this reason the electronic digital signature is used. Electronic digital signature is a set of some data which is encrypted with the help of the asymmetric cryptographic algorithm [1]. A part of this data is a hash sum of signed message which helps to check integrity of the message. For getting hash sum hash function is used which often contains the block cipher in its base.

Next step is message encryption which is always performed with symmetric cipher. Block symmetric cipher or stream cipher can be used for this purpose. Symmetric algorithm is used in this stage because of performance requirements. It is a known fact that symmetric algorithms much faster than asymmetric ones [8].

The other area of block symmetric cipher usage in this scheme is to generate the pseudorandom sequences of numbers which is needed for getting keys of encryption and authentication. It is a known fact that cryptograms which are taken with modern block ciphers, have properties of random sequences. That is why block ciphers are often used as generators of pseudo random numbers [1].

Thus, as it is shown in scheme 2.1, block ciphers are used for different purposes. They are important components of the complex system of information security.

Let us consider threats which can be implemented in the described model of telecommunication system:
1. Violation of the confidentiality of information and keys. The result of this threat is an access to the critical information and keys which become available for unauthorized users, hackers and/or cryptanalyst.

2. Formation of the fake cryptograms and their transmission over the telecommunication system by a hacker or a cryptanalyst in order to harm.

3. Modification of cryptograms and messages with the aim of their damaging and harming end users.

4. Violation of system performance including inability to access the system resources by sending modified commands and signals.

5. Threats which are connected with the violation of observability of information processing and storage.

Thus, after analyzing described threats the next conclusion can be done: the cryptographic transformations which are used in the information and telecommunication system which is shown above provide protection from the first three threats (violation of confidentiality, integrity and authenticity). As it was mentioned above, the block ciphers are used for both cryptographic transformations from the model (encryption and digital signature) so it is possible to say that block ciphers are used to provide confidentiality, integrity, authenticity and other services. This fact confirms the important role which symmetric block ciphers play in the modern information and telecommunication systems.

2.3 The actuality of developing the new standard for block symmetric encryption

Nowadays American standard AES (Advanced Encryption Standard) is the most popular block symmetric cipher in the world. This cipher has three possible modifications: AES-128, AES-192 and AES-256 [3]. However, in spite of such popularity, some cryptographic attacks on this cipher have been detected recently [5,6,7]. These attacks considerably decrease the complexity of breaking the AES-192 and AES-256 versions of cipher. Of course these attacks are theoretical because practically the complexity of breaking is still very big. But this vulnerabilities should be explored enough to prevent its appearing in the new algorithm which can be developed in the future.

All this facts say about necessity of developing a new cipher which will be without described vulnerabilities. In this report perspective cipher is described [4] which is based on the AES and has all its advantages such as speed, cryptographic strength, simple implementation, etc. However the key-expansion system is completely redesigned to get rid of vulnerabilities which are presented in AES.
3 A proposal for the new standard of block encryption

First of all it should be mentioned that the described algorithm of encryption was proposed to the public competition of block cipher selection to be a prototype during development of Ukrainian National Standard [4]. That is the reason of researching this cipher.

3.1 Cipher description
This section describes all parts of proposed cipher in details. As was mentioned before this cipher is based on the AES. The main difference compare to AES is the key expansion system which was considerably redesigned.

3.1.1 General parameters
This algorithm can be used with different types of input data. The block of input data can be 128, 256 or 512 bits. The length of the key can be also 128, 256 or 512 bits [4].

The state of the cipher can be represented as a matrix. Each element of the matrix is a byte. Matrix consists of $N_b$ columns. Each column consists of 8 bytes. So in total matrix has $8 \times N_b$ bytes. The state of the cipher for different versions is shown in the next figures.

![Figure 3.1 – Representation of bytes in 128-bits version of cipher](image)

![Figure 3.2 – Representation of bytes in 256-bits version of cipher](image)
The numbers represent input sequence of bytes (key, cipher state, etc.). Representation with columns is used for internal transformations. Each cell contains one byte of state.

In the next table (table 3.1) acceptable combinations of different blocks and keys are represented.

Table 3.1 – Acceptable combinations of blocks and keys

<table>
<thead>
<tr>
<th>Size of block, bits</th>
<th>Supported key size, bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 ((N_b = 2))</td>
<td>128, 256</td>
</tr>
<tr>
<td>256 ((N_b = 4))</td>
<td>256, 512</td>
</tr>
<tr>
<td>512 ((N_b = 8))</td>
<td>512</td>
</tr>
</tbody>
</table>

Amount of encryption rounds will change depending on the size of block and key. Amount of rounds for different versions of cipher is represented in the table 3.2.

Table 3.2 – Amount of encryption rounds for different versions of cipher

<table>
<thead>
<tr>
<th>Size of block, bits</th>
<th>Size of key 128 bits</th>
<th>Size of key 256 bits</th>
<th>Size of key 512 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 ((N_b = 2))</td>
<td>10</td>
<td>14</td>
<td>-</td>
</tr>
<tr>
<td>256 ((N_b = 4))</td>
<td>-</td>
<td>14</td>
<td>18</td>
</tr>
<tr>
<td>512 ((N_b = 8))</td>
<td>-</td>
<td>-</td>
<td>18</td>
</tr>
</tbody>
</table>

The research deals with the 128-bits version of the cipher, so in the next chapters will be presented information concerning only this version, even if it is not evidently mentioned.

3.1.2 Basic transformations

In the presented algorithm four basic transformations are used: key addition, bytes substitution, shift rows and mix columns. These transformations are the base of all high-level transformations [4]. Let us consider each transformation in detail.

3.1.2.1 Key addition

Two different types of adding the key is used in this cipher. First is the adding the key by modulo $2^{64}$, second is the classic XOR operation (adding by modulo 2) [4].

Adding by modulo $2^{64}$ can be performed in the next way:

$$b_i = a_i + k_i \mod 2^{64},$$

where $b_i = b_{0,i} \cdot 2^{08} + b_{1,i} \cdot 2^{18} + b_{2,i} \cdot 2^{28} + b_{3,i} \cdot 2^{38} + b_{4,i} \cdot 2^{48} + b_{5,i} \cdot 2^{58} + b_{6,i} \cdot 2^{68} + b_{7,i} \cdot 2^{78},$

$$a_i = a_{0,i} \cdot 2^{08} + a_{1,i} \cdot 2^{18} + a_{2,i} \cdot 2^{28} + a_{3,i} \cdot 2^{38} + a_{4,i} \cdot 2^{48} + a_{5,i} \cdot 2^{58} + a_{6,i} \cdot 2^{68} + a_{7,i} \cdot 2^{78},$$

$$k_i = k_{0,i} \cdot 2^{08} + k_{1,i} \cdot 2^{18} + k_{2,i} \cdot 2^{28} + k_{3,i} \cdot 2^{38} + k_{4,i} \cdot 2^{48} + k_{5,i} \cdot 2^{58} + k_{6,i} \cdot 2^{68} + k_{7,i} \cdot 2^{78},$$

$a_i, b_i, k_i$ are columns,

$0 \leq i < N_b$ is a number of column,

$a_{j,i}, b_{j,i}, k_{j,i}$ are bytes,

$0 \leq j < 8$ is a number of byte in column.
As it has presented in the above formula the adding by modulo \(2^{64}\) is performed by columns.

Adding by modulo 2 is performed by bytes. Each byte of state adds to the corresponding byte of summand state.

Adding by modulo \(2^{64}\) is used for initial transformation and also in the last round of encryption.

Adding by modulo 2 is used in adding key transformation in all rounds except the last round.

### 3.1.2.2 Bytes substitution

As it has represented in the figure 3.4 this transformation changes every byte of current state under the given table of substitutions. Four different «byte-to-byte» tables are used in this cipher (S0, S1, S2, S3). Bytes from 1 and 5 rows is substituted with S0 table, bytes from 2 and 6 rows with S1 table and so on [4].

![Bytes substitution](image)

Figure 3.4 – Bytes substitution

### 3.1.2.3 Shift rows

This transformation performs uniform distribution of bytes from every 8-bytes column among other columns. For this purpose every row shifts to the right on different amount of bytes. The amount of bytes for shifting depends on the block size. Table 3.3 represents shifts for different rows.

<table>
<thead>
<tr>
<th>Row number</th>
<th>Shift value, bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size of block — 128 bits</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

Shift of rows for different sizes of blocks is represented in the figures 3.5, 3.6, 3.7.
3.1.2.4 **Mix columns**

This transformation treats each column separately. Column is represented as the next polynomial:

\[ B(x) = b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x^1 + b_0 x^0 \]

where \( b_i \) is a byte in position \( i \).

This polynomial is multiplied on the fixed polynomial in the field \( GF(2^8) \):

\[ C(x) = \{01\} x^7 + \{05\} x^6 + \{01\} x^5 + \{08\} x^4 + \{06\} x^3 + \{07\} x^2 + \{04\} x^1 + \{01\} x^0 \]

where \( \{yz\} \) are byte constants in hexadecimal representation.

The main property of this transformation is a fact that amount of active (not null) bytes on the input and output of the transformation are connected with the next inequality:

\[ N_{in} + N_{out} \geq 9. \]

3.1.3 **Key expansion scheme**

Let \( K_M \) be the main key of encryption and \((K_1, K_2, ..., K_m)\) are the round keys which are generated by key expansion scheme.

The cipher has an SPN structure which is based on the AES cipher:

\[ Cipher_{K_M} = \prod_{i=1}^{N_r} \theta \circ \gamma \circ \sigma_{K_i} \quad (3.1) \]

Symbols:

\( \sigma_{K_i} \) - round key adding;
- nonlinear layer (bytes substitution);
\[ \gamma \]
- linear layer (mix columns, shift rows);
\[ \theta \]
- amount of rounds in block cipher.
\[ N_r \]

Proposed key-expansion scheme contains two steps:

1. Computing of intermediate value \( k_i \) which is based on the master key of encryption \( K_M \) and some constant.

2. Computing of round keys \( (K_1, K_2, \ldots, K_m) \) which are based on the master key \( K_M \), intermediate value \( K_t \) and some constant.

Intermediate value \( k_i \) is computed with the next algorithm (fig. 3.8):

\[
\text{Intermediate value } k_i \text{ is computed with the next algorithm (fig. 3.8):}
\]

\[
\begin{align*}
IM_{K_M} &= \prod_{i=1}^{3} \theta \circ \gamma \circ \sigma_{K_M} \\
K_t &= IM_{K_M}(iv)
\end{align*}
\]

where \( iv \) is some constant which reduces symmetry in the key.

Generation of round keys can be formalized with next expression:

\[
RK_{K_i}[K_M] = \sigma_{K_t+tmv_0} \circ \prod_{i=1}^{2} \theta \circ \gamma \circ \sigma_{K_t+tmv_i}
\]

where \( tmv_i \) are constants which are used for generation round keys. For each round this value should be unique but computation of these constants can be very simple (for example it is a simple shift on a few positions).

General algorithm for round keys generation is represented on the figure 3.9 [4].
The above scheme (figure 3.9) is similar to the scheme of computation \( K_i \) and also it is similar to the general scheme of the cipher. This issue reduces the complexity of implementation.

According to the [10] such scheme of key expansion has some properties, such as:
1. One-way mapping: having encryption key it is very easy to generate round keys, but having one or more round keys it is computationally very difficult to retrieve encryption key or another round key.
2. Non-linear dependence between each bit of encryption key.
3. Good statistical properties of this key schedule (verified by NIST STS statistical tests).
4. Simple implementation (based on cipher round transformations only), good key agility and possibility to generate round keys in direct and reverse order with the same computational complexity.

### 3.1.4 Encryption transformation

Encryption transformation is almost the same as in AES algorithm. General scheme for 128-bits version of cipher is represented in figure 3.10. Adding keys \( K_0 \) and \( K_{10} \) are performed by modulo \( 2^{64} \). In other cases simple XOR operation is used [4].

As it has seen this cipher is a classic SPN structure. The same structure is used in key-expansion algorithm so it simplifies implementation.
3.2 Formulation of the problem
As it has seen the proposed algorithm of encryption is almost similar to the AES algorithm. Only key expansion scheme was considerably redesigned. The main disadvantage of the key expansion scheme in AES is a vulnerability to the related-key attacks. Other ideas in AES are quite good. That is the reason why developers of new algorithm left all good ideas which present in AES and redesigned the vulnerable parts. The most problematic part of AES is key expansion algorithm which is the reason of redesigning.

But then another question arises: is the modified algorithm secure against the related-key attack? That is the main goal of this work: to prove security of algorithm against such type of attacks. In the following chapters the developed algorithm for proving security will be presented. And also practical results for developed algorithm will be presented.
4 The algorithm for estimation cipher security against related-key attacks

First of all this section describes the related-key attack which was found in AES cipher. Then the developed method for cipher estimation against such type of attacks will be presented.

4.1 Description of related-key attack

The idea of this attack is to inject a difference into the internal state, causing a disturbance, and then to correct it with the next infections. The resulting difference pattern is spread out due to the message schedule causing more disturbances in other rounds. The goal is to have as few disturbances as possible in order to reduce the complexity of the attack [5].

In the related-key scenario we are allowed to inject difference into the key, and not only into the plaintext as in the pure differential cryptanalysis. However the attacker cannot control the key itself and thus the attack should work for any key pair with a given difference.

Local collisions in AES-256 are best understood on a one-round example (fig. 4.1).

![Figure 4.1 – Local collisions in AES][5]

Here we need one active S-box (it is a short name for bytes Substitution operation) and five non-zero byte differences in the two subkeys. These five bytes split into two parts: one-byte disturbance and four-byte correction.

Due to the key schedule the differences spread to other rounds. The AES key schedule is mostly linear, so a sequence of several consecutive subkeys can be viewed as a codeword of a linear code. This is the case, particularly, when a trail does not have active S-boxes in the key schedule, which we try to achieve.

Let us figure out how to build an optimal trail for the key recovery attack. Typically, a trail is better if it has fewer active S-boxes. Disturbance differences form a codeword, which should have low weight. Simultaneously, correction differences also must form a codeword, and the key schedule codeword is the sum of the disturbance and the correction codewords. In further trails, the correction codeword is constructed from the former one by just shifting four columns to the right and applying the S-box and Mix Columns expansion. Synchronization is simple since the injection is made to the first
row, which is not rotated by Shift Rows. Otherwise, the task of synchronizing two codewords would have been much harder and would have lead to high-weight codewords [5].

An example of a good key-schedule pattern for AES-256 is depicted in figure 4.2 as a 4.5-round codeword.

![Disturbance and Correction Diagram](image)

In the first four key-schedule rounds the disturbance codeword has only 9 active bytes, which is the lower bound. We want to avoid active S-boxes in the key schedule as long as possible, so we start with a difference in byte b00 and go backwards. Due to a slow diffusion in the AES key schedule the difference affects only one more byte per key schedule round. The correction column should be positioned four columns to the right, and propagates backwards in the same way. The last column in the first subkey is active, so all S-boxes of the first round are active as well, which causes unknown difference in the first column. This «alien» difference should be canceled by the plaintext [5].

So such collisions can considerably reduce the complexity of key-recovery attack.

### 4.2 Proposed method for cipher security estimation

To prove the security of encryption algorithm against such type of attacks we will find the best differential characteristic. The best characteristic is such one that has as few active bytes as possible. So after we found the best differential characteristic we need to count the amount of active bytes which were used during its construction. Active byte is a non-null byte which was passed through the substitution table. As we know substitution of bytes is a non-linear transformation so the output is undefined and attacker should attack each active substitution transformation (which has complexity of $2^6$ operations). If the amount of active bytes exceeds some boundary value the cipher can be considered as safe against key-related attacks because the complexity becomes bigger than the complexity of brute force attack on the cipher. Boundary value depends on the size of cipher block and size of encryption key.

The best differential characteristic can be found by searching between all possible input differences. This can be done automatically with special software. As will be shown later this search can be done in reasonable time for 128-bits version of cipher. To do this a special technique should be used. It will be shown in the next chapters.
4.3 The algorithm for counting active bytes

As was mentioned before the best differential characteristic (with the fewest amounts of active bytes) can be found by searching between all possible input differences. For each characteristic we need to count the amount of active bytes after key-expansion scheme and after all rounds of encryption. Proposed algorithm is for 128-bits version of the cipher.

Let us examine the next example. It is a differential characteristic for $K_t$-computation. As was mentioned before $K_t$ is an intermediate value which is computed in the key-expansion scheme. General scheme for computation of $K_t$ is represented in figure 4.3.

![Figure 4.3 – Differential characteristic for $K_t$ computation](image)

Symbols:

- $iv$ – the difference in initial vectors (always equals zero);
- $K_M$ – master key of encryption (which is needed to be expanded);
- $SB$ – sub-bytes (bytes substitution with substitution tables);
- $SR$ – shift rows;
- $MC$ – mix columns.

In the example (Figure 4.3), three rounds (from left to right) of $K_t$ computation are shown. Input data is a difference of encryption key $K_M$. Then we check how this difference spreads after different transformations. We are interested in amount of active bytes (such bytes for which difference is not null) on the input of Sub-Bytes transformation. In the example this value equals to 4 for the first round, 1 for the second round and 12 for the third round. In general we have 17 active bytes.
It should be mentioned that in other transformations (which is linear) we make decision based on what is the best for cryptanalyst. For example the difference after Mix Columns has such value that when we add it to the key $K_M$ in the second round collision is occurred.

4.3.1 General description
Simpler algorithm for counting active bytes can be implemented. It this case we will control each column of the state instead of each byte. From section 3.1.1 it is known that the state of the cipher is represented as columns. Each column consists of 8 bytes. Thus 128-bits version has two columns. In the searching algorithm we can count amount of active bytes in columns without their exact position. Such algorithm will considerably reduce complexity. Accuracy of the results becomes worse because it is possible to find characteristic which does not really exist. Such characteristics will cut down the acceptable boundary of amount of active bytes.

Thus we can say that we have a minimal amount of active bytes which is possible to achieve with any differential characteristic. This is a very good result.

If the amount of active bytes for investigated algorithm does not exceed theoretical boundary we can say that such cipher is safe against the related-key attacks.

Figure 4.4 presents an example of differential characteristic for $K_t$-computation scheme. As shown in example we recalculated amount of active bytes after each transformation. The numbers in figure are amounts of active bytes in column. This example gives us 3 active bytes in each round so we have 9 active bytes in total.

![Figure 4.4 – Computation of active bytes](image)

In order to make it more understandable in the next sections each transformation will be described in detail.
4.3.2 Key addition
The operation of adding with key is performed with the next rule: we just substitute the value of one column from the value of corresponding column from opposite state and take an absolute value of this result. Some examples are represented in figure 4.5.

![Key addition example](image)

During adding we assume that non-zero bytes have such positions and values that collisions will occur. It means that we have the best case for the cryptanalyst. That is why this assumption can lower the resulting amount of active bytes but cannot increase it.

4.3.3 Bytes substitution
During bytes substitution transformation the amount of non-zero bytes does not change. That is why the values in the columns are not changed. But this transformation is very important because the security of the cipher depends on amount of non-zero bytes which are passed through the substitution tables. Such bytes become active bytes. The more active bytes the better cipher.

4.3.4 Shift rows
The operation of shifting bytes in 128-bits version of the cipher is performed as in figure 4.6:

![Shift rows example](image)

Figure 4.6 shows that last 4 bytes of left and right columns change places. In our algorithm we control just amount of non-zero bytes in column but do not control the actual places of this bytes. So immediately next question rises: how we should perform shift and what amount of bytes should be shifted from one column to another? This problem was solved in such way: we take into account all possible variants of shifting so the differential characteristic can get additional branches which are independent between each other. This part is the most hard part in algorithm. It takes a lot of
computation because on each round we have new branches and amount of this independent branches will grow exponentially depends on the amount of rounds.

Example of this transformation is represented in figure 4.7.

![Figure 4.7 – Shift rows](image)

Above example (Figure 4.7) shows that the state (0;2) has three independent branches for continuation and if we want to find best differential characteristic, we should compute all this branches.

Table 4.1 represents amount of different alternatives for shifting in the column depends on amount of non-zero bytes.

**Table 4.1 – Amount of branches depends on amount of non-zero bytes in column**

<table>
<thead>
<tr>
<th>Amount of non-zero bytes in column</th>
<th>Amount of branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

Total amount of branches after shift rows transformation will be equal to the product of possible shifts of the left and right columns. For example if in left column we have 3 non-zero bytes and in right column we have 4 non-zero bytes then amount of branches for this transformation is $4 \times 5 = 20$.

**4.3.5 Mix columns**

Mixing in columns is performed as the product of column and some fixed matrix. The main property of this transformation is a fact that the sum of non-zero bytes on the input and output cannot be less than $9$:

$$N_{in} + N_{out} \geq 9.$$  

Exception is the situation when we have all zero bytes in column, then on the output we will also have all zero bytes.
In our algorithm we assume that we have the best case for the cryptanalyst. It means that we will always have 9:

\[ N_{in} + N_{out} = 9. \]

Some examples of how Mix Columns works are represented in figure 4.8.

![Figure 4.8 – Mix columns](image)

This transformation does not create new branches.

### 4.4 Description of results

As a result of experiments the best differential characteristic for 128-bits version of cipher was found. This characteristic has 27 active bytes. The results for each round are represented in the table 4.2.

**Table 4.2 – Description of best differential characteristic**

<table>
<thead>
<tr>
<th>Part of cipher</th>
<th>Round</th>
<th>Amount of active bytes</th>
<th>Accumulated amount of active bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Kt ) computation</td>
<td>1</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>Key expansion scheme</td>
<td>1</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>Main encryption loop</td>
<td>1</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td></td>
<td></td>
<td><strong>27</strong></td>
</tr>
</tbody>
</table>

Table 4.2 shows that the differential characteristic in the main encryption loop is iterative and 1 active byte is added on each round. Detailed characteristic is represented in the next figures.

Symbols:
- \( K \) – round key. For different stages of cipher this value is different;
- \( SR \) – Shift Rows;
- \( MC \) – Mix Columns.

The stage of computing \( Kt \) is represented in figure 4.9. In this figure first 3 rounds are represented. As was mentioned before intermediate value \( Kt \) is needed for key expansion scheme.
Initially the state of transformation is equal to (0;0) and key is equal to (1;6). This key is the master key which we can choose. To be more precisely these values are the differences between keys or states.

![Diagram](image)

Figure 4.9 – \( K_t \) computation

In figure 4.10 two round of key-expansion scheme are represented. Initially the state of transformation is equal to (0;7). This value is output value of previous transformation. The difference of the key is equal to (1;6). This is the master key.

![Diagram](image)

Figure 4.10 – Round key computation

In figure 4.11 ten rounds of the main encryption loop are represented. On this stage we can choose input states so in this differential characteristic we took input state (0;6). Round key is equal to (0;7). This is the output value of previous stage. The difference (0;7) is the same for all round keys because the two rounds of key expansion scheme is the same for all 10 rounds. So the value on each round also will be equal.
As a result we got a differential characteristic which has 27 active bytes for 128-bits version of cipher. The theoretical threshold value for this version of cipher is equal to 26 active bytes. But as was mentioned before the value which we obtained is the minimum amount of active bytes which can be achieved. Practically the amount of active bytes in best differential characteristic can be higher but it cannot be lower. This is the main property of algorithm. This issue is connected with the implementation of algorithm. This implementation allows to find characteristics which are impossible in practice. But still this algorithm can show us the security of investigated cipher. For example if we would have amount of active bytes which is lower than the threshold we could say that we have possibility to find differential characteristic with such amount of active bytes which is potentially can lead to vulnerabilities.

4.5 Complexity of algorithm

The input values of algorithm are the different values of differences of the master key on the input of key-expansion scheme (it is a stage of $Kt$ computation). As we know the algorithm was build in such way that it controls 2 columns. Each column can have values between 0 to 9 (amount of active bytes). This means that each column have 9 different variants of input values. 128-bits version of cipher has 2 columns. The values
of each column can be picked independently so the total amount of input values is 81 (formula 4.1).

\[ N_{MK} = N_{Col}^c = 9^2 = 81 \]  

(4.1)

Symbols:

\[ N_{MK} \] - amount of differences for master key;
\[ N_{Col} \] - amount of states for one column;
\[ c \] - amount of columns in the cipher state.

Also we can choose different open texts on the input of main loop of encryption. Amount of different open texts are the same as amount of keys. It can be calculated with the next formula:

\[ N_{PT} = N_{Col}^c = 9^2 = 81 \]  

(4.2)

\[ N_{PT} \] - amount of possible differences for open texts.

Also as mentioned before the Shift Rows transformation can create new branches. It means that additional paths for searching differential characteristic appear. It is connected with different possibilities of shifts. Algorithm does not control each byte separately but it controls amount of active bytes in columns. That is why different variants of shifts appear because we do not know positions on which the active bytes are placed. Amount of such variants depends on amount of active bytes in column. Table 4.3 represents amount of branches which are created after Shift Rows for one column.

Table 4.3 – Amount of branches depends on the amount of active bytes in column

<table>
<thead>
<tr>
<th>Amount of active bytes in column</th>
<th>Amount of branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

To find total amount of branches after Shift Rows transformation we need to multiply amount of branches for each column.

\[ N_{DC} = \prod_{i=0}^{c} N_i \]  

(4.3)

\[ N_{DC} \] - amount of branches after Shift Rows transformation;
\[ c \] - amount of columns in the cipher state;
\[ N_i \] - amount of branches for each column (according to table 4.1).

For example if current state of cipher is (3;4) then after Shift Rows we will have 4*5 = 20 independent branches for searching differential characteristic.

If we will take into account that we searching among all possible input text differences then we can assume that for each column on the average we will have 3 different branches. It means that we will have 9 different branches for the whole transformation for 128-bits version of cipher.
This transformation is the most complex one because it is performed in each round so we have exponential growth of complexity with each round. Formula 4.4 demonstrates amount of branches depends on the amount of rounds.

\[ N_{DC} = \prod_{i=0}^{r} \prod_{j=0}^{c} N_{ij} \]  

(4.4)

\( N_{DC} \) - total amount of branches;
\( c \) - amount of columns in the cipher state;
\( r \) - amount of rounds;
\( N_{ij} \) - amount of branches for each column for one round (according to table 4.3).

We can reduce the complexity of algorithm if we will filter bad characteristics. Bad characteristic is a characteristic which exceeds some threshold value of active bytes. So when we noticed that differential characteristic reaches some threshold value we just stop to compute this characteristic. This considerably simplifies the computation.

General formula for counting is represented below:

\[ O = N_{PT} N_{MK} N_{DC} = N_{Col}^c N_{Col}^c \prod_{i=0}^{r} \prod_{j=0}^{c} N_{ij} = (N_{Col}^c)^2 \prod_{i=0}^{r} \prod_{j=0}^{c} N_{ij} \]  

(4.5)

Using formula 4.5 we can compute the total complexity for 128-bits version of cipher for all 10 rounds. It is expected that Shift Rows transformation creates 9 branches on each round.

In general we can say that we have 15 rounds (5 rounds of key-expansion scheme and 10 rounds of main loop of encryption).

The calculation for the 128-bits version of cipher is represented below:

\[ O_{128} = N_{PT} N_{MK} N_{DC} = 9^2 9^2 9^{15} = 3^{57} \approx 2^{90}. \]

The complexity is very big but this complexity can be considerably reduced if we will filter differential characteristic which exceeds some threshold. The minimal value for 128-bits version of cipher is 27 — this is minimal amount of active bytes which we can get. So if we have more than 27 active bytes in differential characteristic we can stop to compute it.

On the home computer this algorithm takes a time about 10 minutes.
5 Description of program implementation

This section represents an overview of program implementation. It does not describe in
details all code sources but gives general overview how the algorithm is implemented.
All source code can be seen in Appendix A.

Programming implementation of algorithm of searching best differential
characteristic with minimal amount of active bytes needs significant computing
resources. For implementation the C++ language was chosen because C++ programs
work faster than most of other. Object-oriented techniques were used to implement this
algorithm. In figure 5.1 general diagram of classes for developed program is
represented.

![Diagram of classes](image)

The set of classes can be divided into 3 logical parts:
1. Classes which define state (BytesState16).
2. Classes which implements primitive operations (IOperation16,
   MixColumns16Operation, ShiftRows16Operation,
   AddRoundKeyWithInitialKey16Operation, AddRound16Operation,
   CountActiveBytes16Operation).
3. Control classes (CipherController, KeyExpansionController).

Let us describe the classes which determine the state of the differential
characteristic. BytesState16 belongs to this type of classes. The main function of
BytesState16 is to hold intermediate values and result values of differences of bytes in
the key expansion scheme and main encryption loop. The interface of this class has next
structure:

class BytesState16
{
   public:
      typedef std::shared_ptr<BytesState16> Ptr;
      typedef std::list<Ptr> BytesStateList;

      BytesState16();
      BytesState16(unsigned short keyLeft, unsigned short keyRight);

      void getActiveBytes(unsigned short &left, unsigned short &right) const;
}
void getActiveBytesInitialKey(unsigned short &left, unsigned short &right) const;
void getActiveBytesInitialState(unsigned short &left, unsigned short &right) const;
void setActiveBytes(unsigned short left, unsigned short right);
void setActiveBytesInitialKey(unsigned short left, unsigned short right);
void setActiveBytesInitialState(unsigned short left, unsigned short right);
void setActiveBytesKey(unsigned short left, unsigned short right);

unsigned short getTotalAmountOfActiveBytes() const;
void increaseTotalAmountOfActiveBytesWithCurrentState();

bool checkActiveBytesAmountLimit();
bool checkForNullState();

void printState() const;
void printAll() const;

private:
unsigned short m_activeBytesInLeftColumn;
unsigned short m_activeBytesInRightColumn;

unsigned short m_activeBytesInLeftColumnInitialKey;
unsigned short m_activeBytesInRightColumnInitialKey;

unsigned short m_activeBytesInLeftColumnInitialState;
unsigned short m_activeBytesInRightColumnInitialState;

unsigned short m_activeBytesInLeftColumnKey;
unsigned short m_activeBytesInRightColumnKey;

unsigned short m_amountOfActiveBytes;
};

Objects of this class hold 4 states and total amount of active bytes. Also this class allows to control when amount of active bytes exceeds threshold value.

Next set of classes implements primitive operations:
2. ShiftRows16Operation – implements the algorithm of shifting rows.
3. AddRoundKeyWithInitialKey16Operation – implements adding operation in the key expansion scheme. This operation is implemented separately from general operation of adding key because in this operation another key is added.
4. AddRound16Operation – implements adding with key in encryption loop.
5. CountActiveBytes16Operation – implements counting of active bytes in the current state.

Each of these classes should implement next interface:

class IOperation16
{
public:
    typedef std::shared_ptr<IOperation16> Ptr;
    typedef std::list<Ptr> OperationList;

    virtual ~IOperation16() = 0 {};
};
/* Input: bytesState to which we apply operation. Output: list of
bytesState which is possible after this operation
(for example MixColumn can provide different outputs from the same input
state */
virtual BytesState16::BytesStateList performOperation(const
BytesState16::Ptr &bytesState) = 0;
}

All this classes should implement \textit{performOperation(...)} function. The return value
of function should be list of possible output states. As was mentioned above it is
possible to have several output branches. For example after Mix Column transformation
we can have independent branches. If operation does not create more than one branch
then in the returned list we will have just one value (for example adding key operation).

General interface for all classes of this type allows to work with them uniformly. It
makes the implementation easy to use and modify. For example if we want to add new
operation it is very easy to do with such architecture.

Next part of program is a set of control classes. \textit{CipherController} and
\textit{KeyExpansionController} belong to this set. \textit{KeyExpansionController} is responsible for
counting active bytes in key expansion scheme. Then control flow passes to the
\textit{CipherController} which counts active bytes in main encryption loop. Interfaces of these
classes are represented below.

class KeyExpansionController
{
public:
    typedef std::shared_ptr<KeyExpansionController> Ptr;

    KeyExpansionController(size_t amountOfRounds);
    void performComputation();
    BytesState16::BytesStateList getResult();
    BytesState16::BytesStateList getResultKeys();
private:
    void initOperations();
    void resetResult();
    void performComputation(const BytesState16::Ptr& state, int currentRound,
    IOperation16::OperationList::iterator currentRounditer);
size_t m_amountOfRounds;
    IOperation16::OperationList m_operations;
    BytesState16::BytesStateList m_result;
    size_t m_bestAmountOfActiveBytes;
};

class CipherController
{
public:
    typedef std::shared_ptr<CipherController> Ptr;

    CipherController(size_t amountOfRounds);
    void performKeyExpansionComputation();
    BytesState16::BytesStateList getResult();
}
private:
    void initOperations();
    void performKeyExpansionComputation(const BytesState16::Ptr& state, int currentRound, IOperation16::OperationList::iterator currentRoundIter);
    void performCipherComputation(const BytesState16::Ptr& state, int currentRound, IOperation16::OperationList::iterator currentRoundIter);

    size_t m_amountOfRounds;
    IOperation16::OperationList m_operationsKeyExpansion;
    IOperation16::OperationList m_operationsCipher;
    BytesState16::BytesStateList m_result;
    size_t m_bestAmountOfActiveBytes;

};

These classes implement all high-level logic of algorithm of searching best differential characteristic. The open interface is quite simple. It consist of two methods performKeyExpansionComputation() and getResult(). getResult() returns the state object which corresponds to the best differential characteristic. As was mentioned before the state holds difference on the output and another intermediate values such as difference in the input key, difference in the input data, difference in the round key.

Implemented program works in command line mode. After required operations are performed the information about minimal differential characteristic is shown on the screen. Example of how the program looks is represented in figure 5.2.

Figure 5.2 shows that we can get amount of active bytes on the output, difference on the output, difference on the input, difference of input key and difference of round key.

As it was shown before in section 4.5, theoretic complexity of the algorithm (total number of branches to be analyzed) is very big. Nevertheless, effective threshold-based optimization (dependent on number of active S-bytes) allowed significantly decreasing the complexity. Effective implementation in C++ gives the running time about 7 minutes on ordinary home computer.
6 Conclusions and future work

First of all in Section 2 the analysis of the role of block ciphers in the modern security systems was done. The actuality of developing the new standard for block symmetric encryption is substantiated.

In Section 3 the new perspective encryption algorithm was introduced. This algorithm is based on the AES but without vulnerabilities which is present in AES. It was proposed to the public competition of block cipher selection to be a prototype during development of Ukrainian National Standard [4]. That is the reason of researching this cipher. Unlike AES the new cipher has considerably revised key expansion scheme which is the weak point in AES. But the question concerning safety of new algorithm immediately arose.

During the research the new method for estimation security of cipher against related-key attacks was developed. It is represented in Section 4. This method is based on counting amount of active bytes in differential characteristics which can be calculated for encryption algorithm. Such method can show existence of such differential characteristic which can lead to the key recovery with complexity which is low then complexity of brute force attack.

Also Section 4 represents the calculation of the complexity for the developed method. This method with direct application has large computational complexity which is about $O = 2^{90}$ for 128-bits version of cipher. For modern computers such complexity is too big so some optimizations were performed. These optimizations are based on the dynamic elimination of differential characteristics which reach some threshold value of active bytes. Such optimization allows considerably reducing complexity. Optimized algorithm which is implemented in C++ language takes about 10 minutes for 128-bits version of cipher. The short overview of program implementation is in Section 5.

So proposed method of security estimation gives the analytic prove of security of perspective block cipher against related key attacks. During the experiments it was found that 128-bits version of cipher does not have such differential characteristic which can lead to the key recovery with complexity which is low then complexity of brute force attack.

In general, of course, this area of estimating block ciphers is wide for future researches. There are a lot of other experiments which should be done for more carefully analysis of cipher. As a result of these experiments current results (which we did) could be changed or corrected. But even now we can say that obtained result can be used for evaluating efficiency of block symmetric ciphers which can be very helpful for designing new ciphers and for assessment of existing ones.

In the future it is planned to apply this algorithm to other version of cipher (256-bits and 512-bits).
References

Appendix A – Source code in C++

BytesState16.h:

```cpp
#pragma once
#include <list>

class BytesState16
{
public:
    typedef std::shared_ptr<BytesState16> Ptr;
    typedef std::list<Ptr> BytesStateList;

    BytesState16();
    BytesState16(unsigned short keyLeft, unsigned short keyRight);

    void getActiveBytes(unsigned short &left, unsigned short &right) const;
    void getActiveBytesInitialKey(unsigned short &left, unsigned short &right) const;
    void getActiveBytesInitialState(unsigned short &left, unsigned short &right);
    void getActiveBytesKey(unsigned short &left, unsigned short &right) const;

    void setActiveBytes(unsigned short left, unsigned short right);
    void setActiveBytesInitialKey(unsigned short left, unsigned short right);
    void setActiveBytesInitialState(unsigned short left, unsigned short right);
    void setActiveBytesKey(unsigned short left, unsigned short right);

    unsigned short getTotalAmountOfActiveBytes() const;
    void increaseTotalAmountOfActiveBytesWithCurrentState();

    bool checkActiveBytesAmountLimit();
    bool checkForNullState();

    void printState() const;
    void printAll() const;

private:
    unsigned short m_activeBytesInLeftColumn;
    unsigned short m_activeBytesInRightColumn;

    unsigned short m_activeBytesInLeftColumnInitialKey;
    unsigned short m_activeBytesInRightColumnInitialKey;

    unsigned short m_activeBytesInLeftColumnInitialState;
    unsigned short m_activeBytesInRightColumnInitialState;

    unsigned short m_activeBytesInLeftColumnKey;
    unsigned short m_activeBytesInRightColumnKey;

    unsigned short m_amountOfActiveBytes;
};
```

BytesState16.cpp:

```cpp
#include "BytesState16.h"
#include <iostream>
```
BytesState16::BytesState16()
{
    m_activeBytesInLeftColumnKey = m_activeBytesInLeftColumnInitialState = m_activeBytesInLeftColumnInitialKey = m_activeBytesInLeftColumn = 0;
    m_activeBytesInRightColumnKey = m_activeBytesInRightColumnInitialState = m_activeBytesInRightColumnInitialKey = m_activeBytesInRightColumn = 0;
    m_amountOfActiveBytes = 0;
}

BytesState16::BytesState16(unsigned short initialKeyLeft, unsigned short initialKeyRight)
{
    m_activeBytesInLeftColumnInitialKey = initialKeyLeft;
    m_activeBytesInRightColumnInitialKey = initialKeyRight;

    m_activeBytesInLeftColumnKey = m_activeBytesInLeftColumnInitialState = m_activeBytesInLeftColumnInitialState = m_activeBytesInLeftColumn = 0;
    m_activeBytesInRightColumnKey = m_activeBytesInRightColumnInitialState = m_activeBytesInRightColumnInitialState = m_activeBytesInRightColumn = 0;
    m_amountOfActiveBytes = 0;
}

void BytesState16::getActiveBytes(unsigned short &left, unsigned short &right) const
{
    left = m_activeBytesInLeftColumn;
    right = m_activeBytesInRightColumn;
}

void BytesState16::getActiveBytesInitialKey(unsigned short &left, unsigned short &right) const
{
    left = m_activeBytesInLeftColumnInitialKey;
    right = m_activeBytesInRightColumnInitialKey;
}

void BytesState16::getActiveBytesInitialState(unsigned short &left, unsigned short &right) const
{
    left = m_activeBytesInLeftColumnInitialState;
    right = m_activeBytesInRightColumnInitialState;
}

void BytesState16::getActiveBytesKey(unsigned short &left, unsigned short &right) const
{
    left = m_activeBytesInLeftColumnKey;
    right = m_activeBytesInRightColumnKey;
}

void BytesState16::setActiveBytes(unsigned short left, unsigned short right)
{
    m_activeBytesInLeftColumn = left;
    m_activeBytesInRightColumn = right;
}

void BytesState16::setActiveBytesInitialKey(unsigned short left, unsigned short right)
{
    m_activeBytesInLeftColumnInitialKey = left;
    m_activeBytesInRightColumnInitialKey = right;
}
void BytesState16::setActiveBytesInitialState(unsigned short left, unsigned short right) {
    m_activeBytesInLeftColumnInitialState = m_activeBytesInLeftColumn = left;
    m_activeBytesInRightColumnInitialState = m_activeBytesInRightColumn = right;
}

void BytesState16::setActiveBytesKey(unsigned short left, unsigned short right) {
    m_activeBytesInLeftColumnKey = left;
    m_activeBytesInRightColumnKey = right;
}

unsigned short BytesState16::getTotalAmountOfActiveBytes() const {
    return m_amountOfActiveBytes;
}

void BytesState16::increaseTotalAmountOfActiveBytesWithCurrentState() {
    m_amountOfActiveBytes += m_activeBytesInLeftColumn;
    m_amountOfActiveBytes += m_activeBytesInRightColumn;
}

bool BytesState16::checkActiveBytesAmountLimit() {
    static const unsigned short activeBytesAmountLimit = 27;
    return m_amountOfActiveBytes > activeBytesAmountLimit ? true : false;
}

bool BytesState16::checkForNullState() {
    if (m_activeBytesInLeftColumn == 0 && m_activeBytesInRightColumn == 0)
        return true;
    else
        return false;
}

void BytesState16::printState() const {
    std::cout << "State: ": m_activeBytesInLeftColumn " \"\" m_activeBytesInRightColumn std::endl;
}

void BytesState16::printAll() const {
    std::cout << "Initial Key: " m_activeBytesInLeftColumnInitialKey " \"\" m_activeBytesInRightColumnInitialKey std::cout << "Key: " m_activeBytesInLeftColumnKey " \"\" m_activeBytesInRightColumnKey std::cout << "Initial State: " m_activeBytesInLeftColumnInitialState " \"\" m_activeBytesInRightColumnInitialState std::cout << "State: " m_activeBytesInLeftColumn " \"\" m_activeBytesInRightColumn std::endl;

IOperation16.h:

#pragma once
```cpp
#include <list>
#include "BytesState16.h"

class IOperation16
{
public:
    typedef std::shared_ptr<IOperation16> Ptr;
    typedef std::list<Ptr> OperationList;

    virtual ~IOperation16() = 0 {};

    /* Input: bytesState to which we apply operation. Output: list of
    bytesState which is possible after this operation
    (for example MixColumn can provide different outputs from the same input state */
    virtual BytesState16::BytesStateList performOperation(const BytesState16::Ptr &bytesState) = 0;
};

AddRoundKey16Operation.h:

#pragma once
#include "IOperation16.h"
#include "BytesState16.h"
#include <vector>

class AddRoundKey16Operation : public IOperation16
{
public:
    AddRoundKey16Operation();
    virtual ~AddRoundKey16Operation();

    virtual BytesState16::BytesStateList performOperation(const BytesState16::Ptr &state);
    static std::vector<int> countPossibleStatesAfterAdding(int activeBytes, int activeBytesKey);
};

AddRoundKey16Operation.cpp:

#include "AddRoundKey16Operation.h"
#include "BytesState16.h"

AddRoundKey16Operation::AddRoundKey16Operation()
{
}

AddRoundKey16Operation::~AddRoundKey16Operation()
{
}

BytesState16::BytesStateList AddRoundKey16Operation::performOperation(const BytesState16::Ptr &state)
{
    BytesState16::BytesStateList resultList;
    BytesState16 *newState = new BytesState16(*state.get());
```
unsigned short activeBytesLeftKey, activeBytesRightKey, activeBytesLeftState, activeBytesRightState;
state->getActiveBytesKey(activeBytesLeftKey, activeBytesRightKey);
state->getActiveBytes(activeBytesLeftState, activeBytesRightState);

unsigned short activeBytesLeft = abs(activeBytesLeftState - activeBytesLeftKey);
unsigned short activeBytesRight = abs(activeBytesRightState - activeBytesRightKey);
newState->setActiveBytes(activeBytesLeft, activeBytesRight);
resultList.push_back(BytesState16::Ptr(newState));

std::vector<int> activeBytesLeft = countPossibleStatesAfterAdding(activeBytesLeftState, activeBytesLeftKey);
std::vector<int> activeBytesRight = countPossibleStatesAfterAdding(activeBytesRightState, activeBytesRightKey);

for (std::vector<int>::iterator iterL = activeBytesLeft.begin(), eL = activeBytesLeft.end(); iterL != eL; iterL++)
{
    for (std::vector<int>::iterator iterR = activeBytesRight.begin(), eR = activeBytesRight.end(); iterR != eR; iterR++)
    {
        BytesState16 *newState = new BytesState16(*state.get());
        newState->setActiveBytes(*iterL, *iterR);
        resultList.push_back(BytesState16::Ptr(newState));
    }
}
return resultList;

AddRoundKeyWithInitialKey16Operation.h:

#pragma once
#include "IOperation16.h"
#include "BytesState16.h"
#include <vector>

class AddRoundKeyWithInitialKey16Operation : public IOperation16
{
public:
    AddRoundKeyWithInitialKey16Operation();
    virtual ~AddRoundKeyWithInitialKey16Operation();

    virtual BytesState16::BytesStateList performOperation(const BytesState16::Ptr &state);
    static std::vector<int> countPossibleStatesAfterAdding(int activeBytes, int activeBytesKey);
};

AddRoundKeyWithInitialKey16Operation.cpp:

#include "AddRoundKeyWithInitialKey16Operation.h"
#include "BytesState16.h"

AddRoundKeyWithInitialKey16Operation::AddRoundKeyWithInitialKey16Operation()
{
}

AddRoundKeyWithInitialKey16Operation::~AddRoundKeyWithInitialKey16Operation()
{
}

BytesState16::BytesStateList AddRoundKeyWithInitialKey16Operation::performOperation(const BytesState16::Ptr &state)
{
    BytesState16::BytesStateList resultList;
    BytesState16 *newState = new BytesState16(*state.get());

    unsigned short activeBytesLeftInitialKey, activeBytesRightInitialKey,
    activeBytesLeftState, activeBytesRightState;
    state->getActiveBytesInitialKey(activeBytesLeftInitialKey,
    activeBytesRightInitialKey);
    state->getActiveBytes(activeBytesLeftState, activeBytesRightState);

    unsigned short activeBytesLeft = abs(activeBytesLeftState -
    activeBytesLeftInitialKey);
    unsigned short activeBytesRight = abs(activeBytesRightState -
    activeBytesRightInitialKey);
    newState->setActiveBytes(activeBytesLeft, activeBytesRight);
    resultList.push_back(BytesState16::Ptr(newState));
    /*
     * std::vector<int> activeBytesLeft =
     * countPossibleStatesAfterAdding(activeBytesLeftState, activeBytesLeftInitialKey);
     * std::vector<int> activeBytesRight =
     * countPossibleStatesAfterAdding(activeBytesRightState, activeBytesRightInitialKey);
     */
    for (std::vector<int>::iterator iterL = activeBytesLeft.begin(), eL =
        activeBytesLeft.end(); iterL != eL; iterL++)
    {
        for (std::vector<int>::iterator iterR = activeBytesRight.begin(),
            eR = activeBytesRight.end(); iterR != eR; iterR++)
        {
            BytesState16 *newState = new BytesState16(*state.get());
            newState->setActiveBytes(*iterL, *iterR);
            resultList.push_back(BytesState16::Ptr(newState));
        }
    }
}
std::vector<int>
AddRoundKeyWithInitialKey16Operation::countPossibleStatesAfterAdding(int activeBytes, int activeBytesKey)
{
    int low = activeBytes > activeBytesKey ? activeBytesKey : activeBytes;
    int high = activeBytes < activeBytesKey ? activeBytesKey : activeBytes;

    std::vector<int> result;
    for (int i = -low; i <= low; i++)
    {
        int value = high + i;
        if (value >= 0 && value <= 8)
            result.push_back(value);
    }

    return result;
}

CountActiveBytes16Operation.h:
#pragma once
#include "IOperation16.h"
#include "BytesState16.h"

class CountActiveBytes16Operation : public IOperation16
{
public:
    CountActiveBytes16Operation();
    virtual ~CountActiveBytes16Operation();

    virtual BytesState16::BytesStateList performOperation(const BytesState16::Ptr &state);
};

CountActiveBytes16Operation.cpp:
#include "CountActiveBytes16Operation.h"
#include "BytesState16.h"

CountActiveBytes16Operation::CountActiveBytes16Operation()
{
}

CountActiveBytes16Operation::~CountActiveBytes16Operation()
{
}

BytesState16::BytesStateList CountActiveBytes16Operation::performOperation(const BytesState16::Ptr &state)
{
    BytesState16::BytesStateList resultList;
    BytesState16 *newState = new BytesState16(*state.get());
newState->increaseTotalAmountOfActiveBytesWithCurrentState();
resultList.push_back(BytesState16::Ptr(newState));
return resultList;
}

MixColumns16Operation.h:

#pragma once
#include "IOperation16.h"
#include "BytesState16.h"

class MixColumns16Operation : public IOperation16
{
public:
    MixColumns16Operation();
    virtual ~MixColumns16Operation();

    virtual BytesState16::BytesStateList performOperation(const BytesState16::Ptr &state);
};

MixColumns16Operation.cpp:

#include "MixColumns16Operation.h"
#include "BytesState16.h"

MixColumns16Operation::MixColumns16Operation()
{
}

MixColumns16Operation::~MixColumns16Operation()
{
}

BytesState16::BytesStateList MixColumns16Operation::performOperation(const BytesState16::Ptr &state)
{
    BytesState16::BytesStateList resultList;
    BytesState16 *newState = new BytesState16(*state.get());

    unsigned short activeBytesLeft, activeBytesRight;
    state->getActiveBytes(activeBytesLeft, activeBytesRight);
    newState->setActiveBytes(activeBytesLeft == 0 ? 0 : 9 - activeBytesLeft, activeBytesRight == 0 ? 0 : 9 - activeBytesRight);

    resultList.push_back(BytesState16::Ptr(newState));
    return resultList;
}

ShiftRows16Operation.h:

#pragma once
```cpp
#include "IOperation16.h"
#include "BytesState16.h"

class Shi:
    public IOperation16
    {
    public:
        ShiftRows16Operation();
        virtual ~ShiftRows16Operation();

        virtual BytesState16::BytesStateList performOperation(const BytesState16::Ptr &state);
    
    private:
        typedef std::list<unsigned short> Container;
        Container getStateAfterShift(int amountOfActiveBytesInColumn);
    };

ShiftRows16Operation.cpp:

#include "ShiftRows16Operation.h"
#include "BytesState16.h"

ShiftRows16Operation::ShiftRows16Operation()
{
}

ShiftRows16Operation::~ShiftRows16Operation()
{
}

BytesState16::BytesStateList ShiftRows16Operation::performOperation(const BytesState16::Ptr &state)
{
    BytesState16::BytesStateList resultList;

    unsigned short activeBytesLeft, activeBytesRight;
    state->getActiveBytes(activeBytesLeft, activeBytesRight);

    Container possibleStatesOfLeftColumnAfterShift =
        getStateAfterShift(activeBytesLeft);
    Container possibleStatesOfRightColumnAfterShift =
        getStateAfterShift(activeBytesRight);

    for (Container::iterator iterL =
         possibleStatesOfLeftColumnAfterShift.begin(), eL =
         possibleStatesOfLeftColumnAfterShift.end(); iterL != eL; iterL++)
    {
        for (Container::iterator iterR =
             possibleStatesOfRightColumnAfterShift.begin(), eR =
             possibleStatesOfRightColumnAfterShift.end(); iterR != eR; iterR++)
        {
            BytesState16 *newState = new BytesState16(*state.get());

            int activeBytesLeftAfterShift = *iterL;
            int activeBytesRightAfterShift = *iterR;

            newState->setActiveBytes(activeBytesLeftAfterShift +
                                     activeBytesRight - activeBytesRightAfterShift,
                                     activeBytesRightAfterShift + activeBytesLeft - activeBytesLeftAfterShift);
```
resultList.push_back(BytesState16::Ptr(newState));
}

return resultList;
}

ShiftRows16Operation::Container ShiftRows16Operation::getStateAfterShift(int amountOfActiveBytesInColumn)
{
    Container result;

    switch (amountOfActiveBytesInColumn)
    {
    case 0:
        result.push_back(0);
        break;
    case 1:
        result.push_back(0);
        result.push_back(1);
        break;
    case 2:
        result.push_back(0);
        result.push_back(1);
        result.push_back(2);
        break;
    case 3:
        result.push_back(0);
        result.push_back(1);
        result.push_back(2);
        result.push_back(3);
        break;
    case 4:
        result.push_back(0);
        result.push_back(1);
        result.push_back(2);
        result.push_back(3);
        result.push_back(4);
        break;
    case 5:
        result.push_back(1);
        result.push_back(2);
        result.push_back(3);
        break;
    case 6:
        result.push_back(2);
        result.push_back(3);
        break;
    case 7:
        result.push_back(3);
        result.push_back(4);
        break;
    case 8:
        result.push_back(4);
        break;
    default:
        throw new std::exception();
    }

    return result;
}
ActiveBytesCounter.cpp:

```cpp
#include <iostream>
#include <ctime>
#include "BytesState16.h"
#include "CipherController.h"
#include "KeyExpansionController.h"
#include "Cipher32Controller.h"

int main()
{
    CipherController cipherController(10);
    cipherController.performComputation();

    //KeyExpansionController keyExpansionController(4);
    //keyExpansionController.performComputation();

    std::cout << "Time: " << clock() / CLOCKS_PER_SEC / (double)60 << std::endl;

    return 0;
}
```