Masarykova univerzita
Fakulta informatiky

White-box attack resistant cipher based on WBAES

Master thesis

Lenka Bačinská

Brno, spring 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Lenka Bačinská

Advisor: RNDr. Petr Švenda, Ph.D.

Acknowledgement

I would like to express my thanks to my supervisor, RNDr. Petr Švenda, Ph.D., for his guidance, ideas, patience and valuable feedback to this thesis.

I would also like to thank Mgr. Dušan Klinec for the time spent to explaining his work and ideas.

Abstract

This thesis is focused on secure symmetric cipher encryption in an untrusted environment. It studies the white-box version of Advanced Encryption Standard designed by Chow *et al.* in [1] and proposes several modifications of this algorithm to make it resistant against known white-box attacks on WBAES. The implementation of this modified cipher within the Java Cryptography Architecture is included.

Keywords

white-box attack resistant cryptography, AES, WBAES, white-box cipher implementation, Java Cryptography Architecture

Contents

Chapter 1

Introduction

For a long time, it was supposed, that the encryption and decryption operation is performed in the secure, trusted environment. We could imagine it as a machine, or a real black box, that takes some inputs (e.g. plaintext and the key) and gives us some results (ciphertext). And this is the only thing the attacker can see, the output of such a machine. Or, he can try several own plaintexts and see the corresponding ciphertexts, what leads to the chosen-plaintext attack. But he is not able to see inside the machine or change something there. This context is called the black-box context and for a long time it was enough to count with this.

However, nowadays more and more computations are being outsourced, performed on insecure devices like smartphones or the Internet (cloud), and we need to consider it. In the white-box context we expect the attacker is able to see the memory during computation, change it, change the application itself or pause the computation in any time and resume it later. For example, the AES algorithm consists of several rounds, so an attacker can run only one exact round and see the memory, where the encryption key and round keys are stored, because in some steps of computation the key needs to be loaded into the memory. The key data is more random than other data in the memory and there exist algorithms able to find such a randomness, and this way the attacker is able to obtain the encryption key.

When an attacker is so powerful, there is need to hide all the important stuff. Basic idea is to split the computation into two parts. One, that is critical and needs to be run in secure environment, but is small enough to run not very long and is performed only once, because the secure devices are usually slow. And the second part con-

sists of the very encryption, that is secure in white-box context.

In last years, several white-box ciphers were designed as white-box versions of known black-box ciphers. The most researched one is white-box AES [1, 2, 3, 4]. In some steps of the AES algorithm, the key is loaded into the memory. In key expansion step it is the encryption key and in all AES rounds the round key needed for AddRoundKey operation is stored in memory. In white-box context these need to be hidden. The most secure way is to create one look-up table, assigning one output to each possible input, and so behaving the same way as in black-box context. However, such table would be very large and thus not practical. That is why there is an effort to create a series of smaller tables. These would be created only once in the secure environment and the encryption operation can use them in untrusted environment.

However, white-box AES has some weaknesses.The key schedule is relatively easily reversible, what means, that an attacker needs only two consecutive round keys to obtain the encryption key. Some primitives, like S-box or MixColumn matrix are constant, publicly known. And MixColumn operation splits its input into four independent parts.

These vulnerabilities are used by the BGE attack against white-box AES [5]. There were several unsuccessful attempts to make WBAES resistant against this attack. The aim of this thesis is to suggest some improvements of white-box AES to make a cipher resistant against BGE attack, but in comparison to other trials, it is not required to stay compatible with AES or any other known cipher. Then to provide its implementation under the Java Cryptography Architecture.

Chapter two gives an overview of AES and white-box AES building blocks, along with the weaknesses of these algorithms. The third chapter outlines suggested improvements and describes basics needed for these changes. In the next chapter the scheme of our new cipher is explained in detail and in the next one, the details of our implementation are provided. The sixth chapter evaluates the designed cipher in both, the black-box and the white-box context, provides test vectors, memory usage and performance comparison. In the last chapter an alternative white-box scheme of this cipher, which requires smaller amount of memory, is introduced.

Chapter 2

AES, WBAES overview

Our new cipher, WBAES+, is based on WBAES, what is the most researched cipher for white-box context. This cipher is compatible with classical Advanced Encryption Standard, just several changes were made to make it secure in the white-box context.

This chapter describes operations in the AES algorithm and white-box AES building blocks.

2.1 Confusion, diffusion

Before we can proceed to explanation of the AES algorithm and its weaknesses, we first need the definitions of these two terms.

Claude Shannon in [7] defined confusion and diffusion as two properties, that are required for the design of a good cryptosystem in order to hinder the statistical analysis.

Confusion is a property, that refers to making the relation between ciphertext and the encryption key as complex and involved as possible. In other words, the key does not relate to the ciphertext in a simple way, each character of the ciphertext should depend on several parts of the key. Confusion is usually implemented via substitution.

Diffusion refers to a property that makes the relation between ciphertext and plaintext complex, dissipating the statistical structure of plaintext which leads to its redundancy into long range statistics of ciphertext. This means that the change of one bit in the plaintext results in change of several bits in the ciphertext and vice versa. As a mechanism for primarily diffusion, transposition techniques have been identified.

2.2 Advanced Encryption Standard

AES is the symmetric (same key for encryption and decryption operation) block cipher with input/output length 128 bits and 3 possible key lengths: 128, 192 and 256 bits. The authors of [8] have shown, that the key schedule is not secure enough for longer keys, and the complexity of their attack is not bigger for AES-256 than for AES-128. Also White-box AES counts only with 128bit keys, so only this case is explained in this thesis. For information about longer keys, please refer to [9].

Algorithm 1 AES-128

1: **function** AES($plaintext, key$)
2: $state \leftarrow plaintext$
3: AddRoundKey($state, key_0$)
4: **for** $r \leftarrow 1$ to 9 **do**
5: SubBytes($state$)
6: ShiftRows($state$)
7: MixColumns($state$)
8: AddRoundKey($state, key_r$)
9: **end for**
10: SubBytes($state$)
11: ShiftRows($state$)
12: AddRoundKey($state, key_{10}$)
13: **return** $state$
14: **end function**

As many other block ciphers, AES is based on States, where the State is a two-dimensional array and the input is written to it by columns.

The algorithm consists of 5 basic operations:

Key expansion

- the round keys are derived from the given encryption key. For this purpose the Rijndael key schedule [2] is used. In this algorithm the special function, $rcon$ is performed. It represents exponentiation of 2 to a user-specified value in Rijndael's finite field $GF(2^8)$.

Algorithm 2 Rijndael key schedule for AES-128

First 16 bytes are the encryption key
Set $rcon$ iteration value i to 1
Repeat 10 times:
    To create 4 bytes:
        Create variable *temp*, assign the previous 4 bytes to it
        Rotate *temp* 8 bits to the left
        Apply Rijndael's S-box on all bytes in *temp*
        Perform $rcon[i]$ and XOR result with the first byte of *temp*
        Increment i by one
        XOR *temp* with the block 16 bytes before the new key
    To create 12 bytes repeat 3 times:
        Assign the value of previous 4 bytes to *temp*
        XOR *temp* with the block 16 bytes before the new key

AddRoundKey

- bitwise XOR of the State and the round key

SubBytes

- application of the Rijndael S-box [C.1] on each byte of the state. This is the non-linear substitution step, that should make the cipher non-reversible.
- confusion element

ShiftRows

- rows of the state are cyclically shifted, the second of one byte, third of two bytes and fourth one of three bytes to the left.
- transposition step

MixColumns

- mixing operation on the columns of the State. Each column is multiplicated by the constant 4×4 matrix. This operation works with columns independently, each byte of the output then depends on 4 input bytes.
- diffusion element

2.3 White-box AES

In white-box context we expect, that the attacker is very powerful. He has the absolute access to the algorithm, application and the environment the application is running in. In case this application performs encryption, there is an easy way for him to obtain the encryption key if some actions to hide it are not taken.

To secure the AES cipher so that the key is not loaded into the memory during the encryption process, a set of look-up tables is created before the very encryption.

This section describes the table types and the procedures used to make the key extraction unfeasible.

2.3.1 AES algorithm suitable for white-box implementation

According to [1], some of the operations in AES algorithm are slightly regrouped to make the *SubByte* operation follow right after the *AddRoundKey* operation from previous round. We can say, that the rounds are moved up of one operation and the first one is the same as the next ones, starting with *AddRoundKey* and ending with *MixColumns*. Algorithm including these changes is described in 3.

Algorithm 3 AES-128 suitable for white-box implementation

1: **function** AES($plaintext, key$)
2: $state \leftarrow plaintext$
3: **for** $r \leftarrow 0$ to 8 **do**
4: AddRoundKey($state, key_r$)
5: SubBytes($state$)
6: ShiftRows($state$)
7: MixColumns($state$)
8: **end for**
9: AddRoundKey($state, key_9$)
10: SubBytes($state$)
11: ShiftRows($state$)
12: AddRoundKey($state, key_{10}$)
13: **return** $state$
14: **end function**

These two consecutive operations (*AddRoundKey*, *SubByte*) are then merged together into one look-up table, called T-box. The operations stay unchanged and are just precomputed into the look-up tables as shown in the following equation (2.1).

$$\begin{aligned} T^r_{i,j}(x) &= S(x \oplus k^r_{i,j}) && \text{if } r < 9 \\ T^9_{i,j}(x) &= S(x \oplus k^9_{i,j}) \oplus k^{10}_{i,j} \end{aligned} \tag{2.1}$$

where *i,j* are the coordinates in the AES state, *r* is the round number, $k^r_{i,j}$ is the corresponding byte of the round key and *S(x)* stands for *SubByte* operation.

There are no special look-up tables for *ShiftRows* in white-box implementation of AES. This operation is performed as the connection between the consecutive tables.

For a naive look-up table of *MixColumn* operation we would need about 16 GB of memory. If we want to have record for every 4-byte vector, for storing such a table we have 2^{32} possible inputs. The output length is 4 bytes, what gives us these 16GB tables.

However, we can use the linearity of the transformation matrix here and divide it into smaller tables connected by XOR tables. After this decomposition we have four tables with $2^8 = 256$ possible inputs and with one-byte outputs it gives us only one kilobyte of memory needed (plus XOR tables).

2.3.2 Tables protecting techniques

The look-up tables created are needed to be protected, otherwise an attacker can see the AES state in its plain form.

IO bijections Input/output bijection is a concatenation of several $4 \rightarrow 4$ bijections applied on the look-up tables in such a way, that the composition of two consecutive tables cancels the effect of IO bijections. IO bijections realize confusion to make an analysis of a single table harder.

Mixing bijections Mixing bijection is a linear transformation realizing the diffusion. It is represented as a multiplication by a non-singular mixing bijection matrix and used in combination with IO bijections in order to increase the security of concatenated

bijections. A small change in one sub-bijection results in many changes in the concatenated bijection.

External encodings External input/output encodings are designed to protect an input and an output of the algorithm. In case the AES encryption is part of a larger system, external encodings can be applied in a similar way than IO bijections.

2.3.3 Table types

There are four types of look-up tables created in the white-box version of AES algorithm.

Type I External input/output encodings are performed as multiplication with a 128×128 matrix. The matrix decomposition using its linearity is applied here and results in tables of type I ($8 \rightarrow 128$ mapping), which are applied in the first and the last round of the cipher.

Type II Type II tables, $8 \rightarrow 32$ mapping, are used to perform the very encryption, consisting of composition of *T-boxes*, *MixColumns* operation (excluded in the last round) and 32×32 mixing bijections.

Type III The effect of the mixing bijections added in type II tables need to be canceled. For this purpose, $8 \rightarrow 32$ tables of type III are generated.

Type IV The decomposition of matrix multiplication (external encodings, *MixColumns*) requires an addition operation. This is performed by a cascade of XOR tables, working with 4-bit inputs ($8 \rightarrow 4$ mapping).

2.4 WBAES weaknesses

The white-box AES scheme, as described in the previous section, has several weaknesses, that can be used by a white-box attack:

- Simple, reversible key schedule

- Usage of constant, publicly known primitives (Rijndael S-box, MixColumn matrix)
- Round function with simple algebraic description
- ShiftRows operation is easily removable in white-box context
- Usage of diffusion element with low diffusion power in one round (MixColumns operation), each byte of the output of one round depends on 4 input bytes

In 2005, an algebraic attack using these weaknesses was created. It is called BGE attack [5], and is able to recover the encryption key from the white-box AES implementation in 2^{30} computational steps.

There were several attempts to improve the WBAES scheme to make it resistant against this attack, but it turned out that AES is not particularly well suitable for white-box cryptography implementation due to its algebraic structure. That is why we decided to break the compatibility with this standard and change it to a new cipher with its white-box form in mind.

Chapter 3

WBAES+ foundations

In order to prevent the scheme against algebraic attacks, especially against the BGE attack, several improvements are suggested in [10]:

- New, non-reversible key schedule based on a hash chain of slow key derivation function
- Usage of key-dependent S-boxes instead of a public one
- Replacement of the MixColumns operation working independently on columns of the state by multiplication of whole state by a key-dependent MDS matrix

This chapter provides algebraic foundations needed for understanding these improvements and their modifications used in our new cipher.

3.1 Key derivation function

Password-based key derivation function (KDF) is a function designed to derive one or more secret keys from the given passphrase, usually using a hash function.

Typical usage of KDFs is creation of secret keys from a common secret value so that the attacker, who obtains one of the derived keys is not able to find out any useful information about the input secret value or other keys derived from it.

For this purpose several KDFs were designed to take a longer time to perform their computation, and so make a brute-force attack

or dictionary attack unfeasible. Many of these algorithms (for example Crypt(3) [11], FreeBSD MD5 crypt [12], PBKDF2 [13], Blowfish-based bcrypt [14]) base its slowdown on large (user-defined or constant) number of iterations of a sub-function combined with a non-secret salt.

As the speed of computer systems increases, the legitimate user can increase the number of iterations of KDF to make the time needed for computation constant. This way the brute-force attack should be still unfeasible, although the attacker's computing power increases. However, this assumption works only if attackers and legitimate users are limited to the same software implementations. Nowadays, the attackers are able to create their own hardware circuits which are faster (than software) and have a large computational power, e.g. via use of parallelism. To reduce the advantage of such attackers, *scrypt* memory-hard function was designed.

3.1.1 Scrypt

Scrypt [15] is a password-based key derivation function created by Colin Percival in 2009.

In comparison to its predecessors, this algorithm was designed with custom hardware attacks in mind. It is based on the concept of *sequential memory-hard functions* [15] defined below.

A sequential memory-hard function is a function which
(a) can be computed by a memory-hard algorithm on a Random Access Machine in $T(n)$ operations; and
(b) cannot be computed on a Parallel Random Access Machine with $S*(n)$ processors and $S*(n)$ space in expected time $T*(n)$ where $S*(n)T*(n) = O(T(n)^{2-x})$ for any $x > 0$.

That means, implementation of these attacks is artificially made unfeasilble by requiring a large amount of memory and precluding effective parallelization.

The *scrypt* algorithm is described in 4.

Algorithm 4 scrypt algorithm, taken from [15]

function SCRYPT(
$Pass$, ▷ Passphrase
$Salt$,
N, ▷ CPU/memory cost parameter
r, ▷ Latency-bandwidth parameter
p, ▷ Parallelization parameter
$dkLen$ ▷ Intended output length of the derived key
)
$(B_0 \dots B_{p-1}) \leftarrow \text{PBKDF2}_{\text{HMAC\_SHA256}}(Pass, Salt, 1, p \cdot MFLen)$
▷ $MFLen \dots$ length of block mixed by $SMix$
for $i \leftarrow 0$ to $p - 1$ **do**
$B_i \leftarrow SMix_r(B_i, N)$
end for
return $\text{PBKDF2}_{\text{HMAC\_SHA256}}(Pass, B_0 || B_1 || \dots || B_{p-1}, 1, dkLen)$
end function

Algorithm 5 SMix algorithm, taken from [15]

function SMIX(B, N)
$X \leftarrow B$
for $i \leftarrow 0$ to $N - 1$ **do**
$V_i \leftarrow X$
$X \leftarrow BlockMix(X)$
end for
for $i \leftarrow 0$ to $N - 1$ **do**
$j \leftarrow Integerify(X)\ mod\ N$ ▷ A bijective function from $\{0,1\}^k$ to $\{0, \dots, 2^k - 1\}$
$X \leftarrow BlockMix(X \oplus v_j)$
end for
return X
end function

Algorithm 6 BlockMix algorithm, taken from [15]

function BLOCKMIX(B)
    $X \leftarrow B_{2r-1}$
    for $i \leftarrow 0$ to $2r-1$ **do**
        $X \leftarrow Salsa20/8core(X \oplus B_i)$
        $Y_i \leftarrow X$
    end for
    return $(Y_0, Y_2, \ldots, Y_{2r-2}, Y_1, Y_3, \ldots, Y_{2r-1})$
end function

3.2 Twofish S-boxes

S-box (Substitution box) is a non-linear substitution operation used in many symmetric-key algorithms. It is a basic component providing confusion of the cipher, usually implemented as a look-up table. S-boxes can be both, constant (as in the case of AES) and generated dynamically from the key (Twofish).

Twofish cipher [16] uses four different, bijective 8-by-8-bit S-boxes created according to the following formula:

$$\begin{pmatrix} s_{i,0} \\ s_{i,1} \\ s_{i,2} \\ s_{i,3} \end{pmatrix} = \begin{pmatrix} . & \cdots & . \\ \vdots & RS & \vdots \\ . & \cdots & . \end{pmatrix} \cdot \begin{pmatrix} k_{8i+0} \\ k_{8i+1} \\ k_{8i+2} \\ k_{8i+3} \\ k_{8i+4} \\ k_{8i+5} \\ k_{8i+6} \\ k_{8i+7} \end{pmatrix} \tag{3.1}$$

$$\begin{aligned} s_0(x) &= q_1[q_0[q_0[x] \oplus s_{0,0}] \oplus s_{1,0}] \\ s_1(x) &= q_0[q_0[q_1[x] \oplus s_{0,1}] \oplus s_{1,1}] \\ s_2(x) &= q_1[q_1[q_0[x] \oplus s_{0,2}] \oplus s_{1,2}] \\ s_3(x) &= q_0[q_1[q_1[x] \oplus s_{0,3}] \oplus s_{1,3}] \end{aligned} \tag{3.2}$$

where k_j is the *j*-th byte of the key, RS is a 4×8 matrix derived from an RS code [priloha], q_0,q_1 are two fixed 8-by-8-bit permutations [priloha] and s_0,s_1,s_2,s_3 are the resulting S-boxes.

In case of a 128-bit key, each S-box depends on 16 bits of the key.

Authors of Twofish verified, that all 2^{16} possible permutations for each S-box are distinct. Usage of q_0,q_1 permutations provides different S-boxes also in case of equal key bytes.

3.3 MDS matrix

A linear $[n, k, d]$-code over a finite filed $GF(2^p)$ is a subspace with dimension k of the vector space $(GF(2^p))^n$, where the Hamming distance (the number of symbols in which the two codewords differ) between any two n-element vectors is at least d and d is the largest possible.

Such codes satisfy the Singleton bound [17]: $d \leq n-k+1$. A linear code, which achieves equality in this bound, is called a Maximum Distance Separable (MDS) code.

The generator matrix of an MDS code has the form

$$G = [I_{k\times k}|A], \tag{3.3}$$

where $I_{k\times k}$ is the $k \times k$ identity matrix and A is a matrix consisting of $k \times (n-k)$ elements, called an MDS matrix.

A necessary and sufficient condition for a matrix to be MDS is that all possible square submatrices are non-singular [18].

MDS matrices have become a fundamental component in many block ciphers to provide the diffusion property of the algorithm.

3.3.1 Cauchy matrix

A Cauchy matrix C is defined by two vectors $x = (x_1, x_2, \ldots, x_m)$ and $y = (y_1, y_2, \ldots, y_n)$ having pairwise distinct elements of a given field as an $m \times n$ matrix with elements c_{ij} in the form

$$c_{ij} = \frac{1}{x_i - y_j}; \qquad x_i - y_j \neq 0,\ 1 \leq i \leq m,\ 1 \leq j \leq n,. \tag{3.4}$$

The determinant of a square Cauchy matrix can be written as

$$det(C) = \frac{\Pi_{i<j}(x_j - x_i)(y_i - y_j)}{\Pi_{i,j}(x_i - y_j)} \tag{3.5}$$

That implies the *non-singular* property of square Cauchy matrices.

Square Cauchy matrices are then unitary ($C^{-1} = C^T$) and symmetric ($C = C^T$), what implies that they are *involutory* (the matrix is its own inverse).

Square Cauchy matrices can be used for generation of MDS matrices, as shown in [19]. This involutory property gives us the possibility of having the same matrix for encryption and decryption.

Chapter 4

WBAES+ scheme and implementation

WBAES+ is implemented in java language, using Bouncy Castle library. It is an extension of whitebox AES implementation proposed by D. Klinec [20].

This chapter talks about the changes needed to be done in this code/algorithm to make it resistant in white-box context and easy to integrate with existing applications.

4.1 Hash chain

As the Rijndael key schedule is relatively easily reversible (only two consecutive round keys are needed to obtain full encryption key), a more secure way for round keys creation needs to be found. A good idea is to use a one-way function (hash) as a new key schedule mechanism.

As proposed in [10], we define key schedule for our new cipher as a hash chain, where round keys are computed as follows:

$$k^r = \begin{cases} hash_{sc\_N,sc\_r,sc\_p,n\_sha}(key, salt) & \text{if } r = 0 \\ hash_{sc\_N,sc\_r,sc\_p,n\_sha}(k^{r-1}||key, salt) & \text{otherwise} \end{cases} \tag{4.1}$$

Where

key is a 128-bit encryption key

k^r is a round key for round r

$||$ is a symbol for concatenation of two binary arguments

$salt$ is a 128-bit salt (publicly known value) used in hashing algorithm to increase its resiliency against precomputation attacks. In our implementation string "*TheConstantSalt.*" is used

sc_N, sc_r, sc_p are the parameters for a hash algorithm chosen for our hash chain – *scrypt*. In our implementation we use values $sc\_N = 2^{14}, sc\_r = 8, sc\_p = 1$

n_sha is the number of nested applications of SHA256 algorithm on the input

The one-way property of a hash algorithm implies that the attacker is not able to derive the encryption key if he obtains several round keys and also that he cannot derive previous round key from obtained round key.

In order to ensure the inability of derivation of the following round key from an obtained one, the concatenation with the encryption key is used. Each round key then depends on the encryption key directly.

We decided to use *scrypt* as our hash algorithm. It is relatively young algorithm, but as long as there is no short-cut allowing significantly faster computation than with the reference algorithm found, it has advantages compared to older hash functions and KDFs [3.1].

In our implementation we use *scrypt* implemented in Bouncy Castle cryptography library, with the following parameters:

```
Scrypt.generate(P, S, N, r, p, dkLen)),
```

$P = \text{SHA256}^{16}(input)$

$S = "TheConstantSalt."$

$N = 2^{14} = 16384$

$r = 8$

$p = 1$

$dkLen = 16$

This hash chain is used not only as a key-schedule, but also for generation of other building blocks dependent on the key. The input of the hash function is slightly modified for each such primitive, so obtaining some of them would not leak any information about round keys and other primitives generated.

4.2 Key-dependent S-boxes

In our cipher, we do not use Twofish S-boxes exactly as described in 3.2. In the original form, key-dependent S-boxes increase complexity of an attack of 2^{16}, as each S-box depends on two bytes derived from the encryption key. However, this is an instance of problem not strong enough for an attacker with sufficient parallelization power. That is why we decided to increase the complexity much more – of $2^{13 \cdot 8}$, what makes it larger than currently best known attack on AES [21].

The idea is to extend each Twofish S-box by adding a dependency on 13 bytes derived from the key. The generation of such S-box can be described by the recursive function *sboxgen* defined in the following equation.

$$sboxgen(j, l, x) = \begin{cases} q'_{j,0}[x] & \text{if } l = 0 \\ q'_{j,l}[sboxgen(j, l-1, x) \oplus k_{4\cdot(l-1)+j}] & \text{otherwise} \end{cases} \tag{4.2}$$

Where

j is the S-box index

l is level of nesting

k is one byte derived from the key

$q'_{j,l}$ is one of the fixed Twofish 8-bit permutations

Our implementation uses modified nested permutations suggested in [10]:

$$\begin{aligned} q'_0 &= [q_0, q_0, q_1, q_0, q_1, q_0, q_1, q_0, q_1, q_0, q_1, q_0, q_1, q_1] \\ q'_1 &= [q_1, q_0, q_0, q_1, q_1, q_0, q_0, q_1, q_1, q_0, q_0, q_1, q_1, q_0] \\ q'_2 &= [q_0, q_1, q_1, q_0, q_0, q_1, q_1, q_0, q_0, q_1, q_1, q_0, q_0, q_1] \\ q'_3 &= [q_1, q_1, q_0, q_0, q_0, q_1, q_1, q_1, q_0, q_0, q_0, q_1, q_1, q_1] \end{aligned} \tag{4.3}$$

4.2.1 Key bytes for S-boxes

As mentioned before, we decided to use different round keys for encryption and building blocks generation in order to increase strength

of our cipher. For generation of key-dependent S-boxes we create special `roundKeys_for_S-boxes` using the following formula:

$$k^r_{Sbox} = \begin{cases} hash_{par}(key||"SBOXconstant", salt) & \text{if } r = 0 \\ hash_{par}(k^{r-1}||key||"SBOXconstant", salt) & \text{otherwise} \end{cases} \tag{4.4}$$

This equation differs from 4.1 only in concatenation with a constant string. Such a difference makes generated round keys for S-boxes non-transformable to round keys for encryption and vice-versa. So an attacker, succeeding in obtaining `roundKeys_for_S-boxes` will not be able to extract round keys from them.

As we are generating four S-boxes in order to use different S-box on each row of the State, we need $4 \cdot 13 = 52$ key bytes for one round S-boxes. These are derived from the round key for S-boxes using SHA-512 hash function.

4.3 Diffusion layer modification

In Advanced Encryption Standard and also in WBAES, each byte of the output in one round depends on four input bytes of that round. To increase the diffusion power of the cipher, we decided to make each byte dependent on all sixteen bytes of the input.

The diffusion element in AES/WBAES is *MixColumns* operation. As was said, it is performed as multiplication of each column of the State by the given 4×4 matrix. In our new cipher MixColumn operation is replaced by multiplication of the whole State by a 16×16 matrix, as illustrates the scheme 4.1.

Also the authors of [19] suggest AES diffusion layer modification from 4×4 to 16×16 matrix to increase security within one round. They use a precomputed involutory MDS matrix constructed using Cauchy matrices. Their procedure is a base for our implementation.

As this multiplication is performed only in the table generation part, we can sacrifice the performance in order to increase security, in comparison to the black-box version, where the computation must not consume much resources. Thanks to this, we can exclude the performance restrictions mentioned in [19].

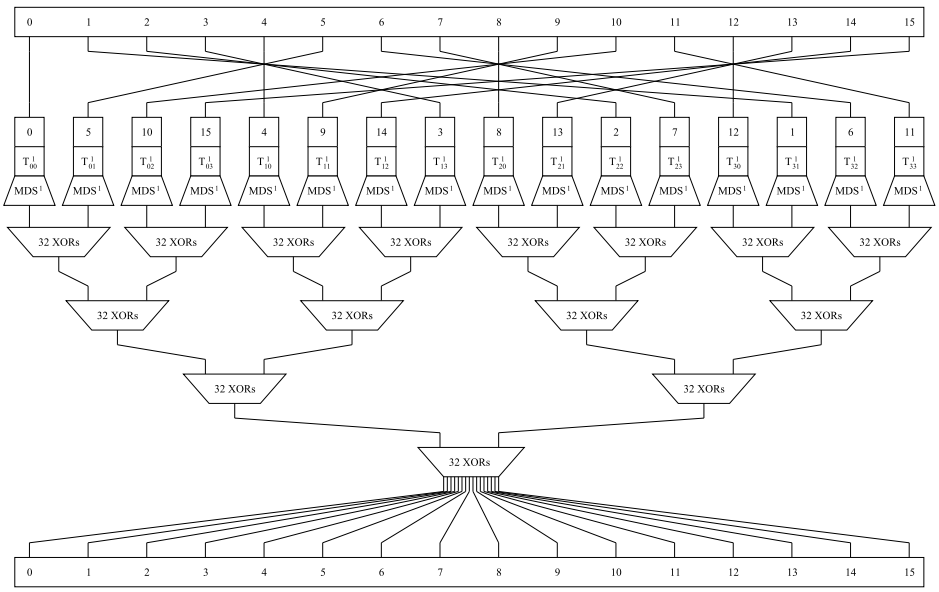

Figure 4.1: Black-box scheme of the first round in WBAES+.

We decided to generate matrices that are involutory to guarantee the same diffusion power in one round for both encryption and decryption.

In order to prevent against several white-box attacks (BGE attack [5], the Generic attack by Michiels [22]), our matrices are key-dependent.

4.3.1 Key bytes for MDS matrices

Similarly to S-boxes, the key bytes used for MDS matrices generation are not taken directly from the encryption key. Special `roundKeys_for_MDS` are derived from the encryption key using modified version of the hash-chain:

$$k^r_{MDS} = \begin{cases} hash_{par}(key||"MDSconstant", salt) & \text{if } r = 0 \\ hash_{par}(k^{r-1}||key||"MDSconstant", salt) & \text{otherwise} \end{cases} \tag{4.5}$$

4.3.2 MDS matrices generation

Cauchy matrices depend on the first row only, other rows are permutations of the bytes in the first one. These bytes are pairwise different and XOR of them is always 1.

Our MDS matrix for each round is thus generated from the corresponding `roundKey_for_MDS` as follows:

Algorithm 7 Generation of MDS matrix for one round.

1. All first 6-tuples and last 6-tuples from each byte of the `roundKey_for_MDS` are stored in a set.
2. First 14 bytes of first row of the matrix are first 14 nonzero bytes from the set.
3. The 15th byte is chosen from the other bytes in the set so that XOR of all 15 bytes and 1 results in a byte different from all these 15 bytes.
4. The 16th byte of first row is a result of XOR of previous 15 bytes and 1.
5. Other 15 rows are computed as predefined permutations of the first one.

The Set data structure in the first step ensures, that the bytes used in firstRow are pairwise different.

First option for the set is a java implementation of ordered set called TreeSet, which inherently sorts elements in increasing order. This makes more difficult to the attacker to obtain the corresponding `roundKey_for_MDS`, if he finds out the matrix. However, the entropy of MDS matrix is lower.

Second option is to use LinkedHashSet, which preserves the order of added elements. When the order of elements is taken into account, there is more possible combinations, and thus more possible MDS matrices (approximately $16!$ times more). However, if the attacker obtains one matrix, he knows exactly the corresponding round key.

Third option is a combination of previous two.

As long as *scrypt* and SHA-256 are uncracked, the matrices need to be secured more than the round keys they are generated from. That is why we decided to use LinkedHashSet as our set.

Usage of 6-tuples was chosen as a compromise between the entropy (number of possible bytes) and number of distinct bytes stored in the set. If there are too few different bytes (so the matrix cannot be created), the constant, publicly known matrix constructed in [19] is used in this round.

4.3.3 Non-suitable round keys

As was written above, not all round keys are suitable for MDS matrix generation as described in algorithm 7. If the round key is periodical, the six-tuples repeat and not enough distinct 6-tuples are derived to create the first row of the MDS matrix.

However, if an attacker finds out that the constant matrix was used for one round, he cannot derive the round key for MDS from the matrix. This information makes the round key space smaller for this round, but even if the attacker obtains the round key for MDS for this round, this will provide him no information about next or previous round keys for MDS, thanks to the concatenation with the encryption key in the key schedule. Moreover, the other round keys for this round (key for encryption, key for S-box) cannot be derived from this key thanks to the concatenation with *"MDSconstant"* string.

4.4 White-box implementation

With our modifications (namely MDS matrices multiplication) of the AES cipher, also the white-box scheme has changed, as illustrated in figure 4.2. This section describes differences between white-box versions of AES and our cipher.

4.4.1 Mixing bijections

MixColumns operation works with columns of the State independently, so the computation is divided into four parts. In each part all

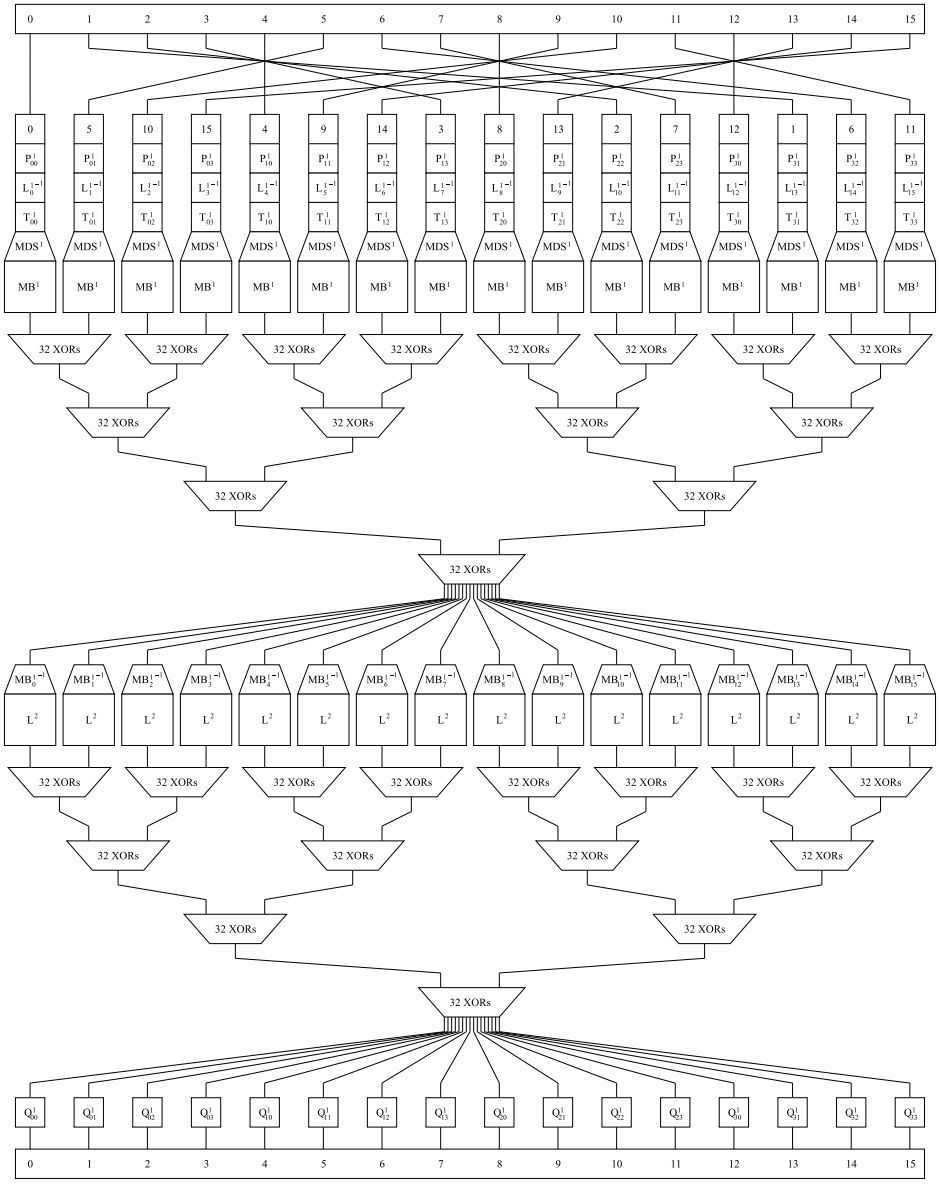

Figure 4.2: White-box scheme of the first round in WBAES+.

four bytes are multiplied by a 4×4 MC matrix, what creates four 32-bit strings. These are then protected by a 32×32 mixing bijection, implemented as multiplication by a 32×32 MB matrix.

In our cipher there is no such division. All sixteen bytes of the State are multiplied by a 16×16 MDS matrix and thus sixteen 128-bit strings are created. This implies the change of Mixing bijections size from 32×32 to 128×128.

4.4.2 Type II tables

By using a 16×16 MDS matrix multiplication the type II tables increased from 8×32 to 8×128, as illustrates figure 4.3.

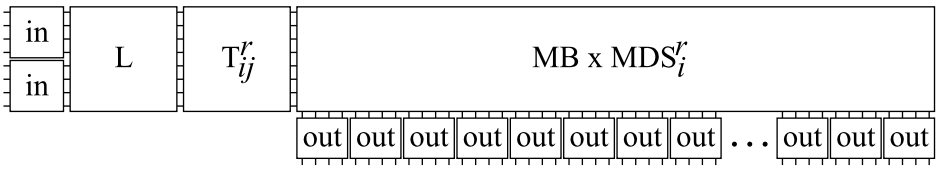

Figure 4.3: Table type II.

4.4.3 Type III tables

Tables of type III are used to cancel the effect of mixing bijections. As mixing bijections are increased to 128×128, also type III tables have changed to 8×128 4.4.

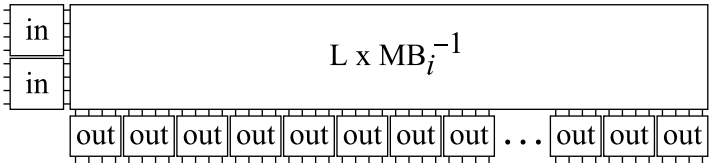

Figure 4.4: Table type III.

4.4.4 Type IV tables

Table type IV has the same structure and functionality as for WBAES, just the number of these tables has increased because of the size of type II tables and larger XOR cascade needed. Table 4.1 contains table numbers and sizes used in our cipher.

Type	Number of tables	Width	Total size in bits
I	$16 \cdot 2 = 32$	$8 \to 128$	$32 \cdot 2^8 \cdot 128 = 1048576$
II	$16 \cdot 9 = 144$	$8 \to 128$	$144 \cdot 2^8 \cdot 128 = 4718592$
III	$16 \cdot 9 = 144$	$8 \to 128$	$144 \cdot 2^8 \cdot 128 = 4718592$
IV (T1)	$32 \cdot 15 \cdot 2 = 960$	$8 \to 4$	$960 \cdot 2^8 \cdot 4 = 983040$
IV (MDS)	$32 \cdot 15 \cdot 9 = 4320$	$8 \to 4$	$4320 \cdot 2^8 \cdot 4 = 4423680$
IV (MB^{-1})	$32 \cdot 15 \cdot 9 = 4320$	$8 \to 4$	$4320 \cdot 2^8 \cdot 4 = 4423680$
Total size			20316160 b $= 2840$ kB

Table 4.1: White-box implementation size.

Chapter 5

JCA implementation

Java cryptography architecture is an API (application program interface) providing a framework for the implementation of security features in java. It uses a provider-based architecture and contains a set of APIs for several applications in digital security, such as symmetric and asymmetric ciphers, key generation and management, digital signatures, message authentication codes, etc. These APIs provide an easy way to add security into developer's applications.

Implementation of the JCE (Java Cryptography Extension) provider to our cipher implementation is also part of this thesis.

The provider we have implemented is common for both, WBAES+ and java implementation of WBAES by Dušan Klinec [20], which our WBAES+ implementation is based on. It supports four padding mechanisms (*NOPADDING*, *ISO9797M1PADDING*, *ISO9797M2PADDING* and *PKCS5PADDING*) and one mode of operation (*ECB*).

As was mentioned, in white-box context the encryption (decryption, respectively) operation is split into two parts.

Generation of the tables, which needs to be run in secure environment, because the key needs to be loaded into the main memory and

The very encryption, which uses these tables stored in the memory and can run in an untrusted environment

This will also reflect into JCE, namely the methods of `javax.crypto.CipherSpi` that we needed to extend.

5.1 Tables generation

For tables generation we implemented the method `engineInit` used by API method

```
Cipher.init(int opmode, Key key),
```

where `opmode` is one of `Cipher.ENCRYPT_MODE`, `Cipher.DECRYPT_MODE` and `key` is the encryption key. If the encryption key is not null, the tables are generated according to the scheme 4.2. These tables are then stored into the file *extb_tables_true.ser* when encrypting or *extb_tables_false.ser* when decrypting using java serialization (conversion of an object state to a byte stream that can be reverted back into a copy of this object).

Calling of this method must be performed in a trusted environment, serialized tables can then be stored in an insecure place for further usage.

5.2 White-box encryption

For deserialization of the tables, same `Cipher.init()` method is called, just this time with null value of the encryption key. After the tables are loaded, method `Cipher.update()` or `Cipher.doFinal()` can be called to encrypt or decrypt an input buffer.

Usage of these methods is safe in an insecure environment. Instead of loaded into the memory, the key is hidden in the look-up tables.

Chapter 6

Evaluation

6.1 Test vectors

In order to test the implementation of our new cipher, we have computed several test vectors, following the algorithm 8.

Algorithm 8 Test vectors generation

1: $state \leftarrow input$
2: $roundKeysForEncryption \leftarrow hashChain(key)$
3: $roundKeysForSboxes \leftarrow hashChain(key, "SBOXconstant")$
4: $roundKeysForMDS \leftarrow hashChain(key, "MDSconstant")$
5: **for** $r \leftarrow 0$ to 9 **do**
6: $\quad state \leftarrow state$ XOR $roundKeysForEncryption[r]$
7: $\quad roundKeyForSboxes \leftarrow$ SHA-512$(roundKeysForSboxes[r])$
8: $\quad$ **for** $i \leftarrow 0$ to 15 **do**
9: $\quad\quad sboxgen(i$ MOD $4, 13, state[i], roundKeyForSboxes)$
10: $\quad$ **end for**
11: $\quad shiftRowsPermutation(state)$
12: $\quad$ **if** $r \neq 9$ **then**
13: $\quad\quad matrix \leftarrow generateMDS(roundKeysForMDS[r])$
14: $\quad\quad transposePermutation(state)$
15: $\quad\quad state \leftarrow matrix \cdot state$
16: $\quad\quad transposePermutation(state)$
17: $\quad$ **end if**
18: **end for**
19: $state \leftarrow state$ XOR $roundKeysForEncryption[r]$
20: **return** $state$

These test vectors are based on test vectors for AES, and are com-

puted for both possibilities: the encryption key, which enforces usage of constant MDS matrix for some round and the key, which does not.

All test vectors are included in the appendix D.

6.2 Performance

The following table compares encryption speed of implementations of three ciphers: AES (Bouncy Castle implementation), white-box AES ([20]) and our new cipher.

The measurements were performed on the same platform:

Processor Intel Pentium Dual-Core CPU, T4400, 2.20 GHz

Memory 4.00 GB

System Linux - Ubuntu 12.04 LTS x86_64

Java version Java SE 1.7

with only the fundamental system processes running.

Time of encryption is the average time it takes the implementation to encrypt a 1MB file. In case of white-box implementations also time of tables loading is included, and thus time of cipher initialization is included also in case of AES implementation.

Cipher	Tables generation (ms)	Encryption (ms)
AES	n/a	145.8
WBAES	1589.6	18145.6
WBAES+	17206.8	37894.2

We can see, that white-box implementations are much slower than black-box AES, because there is need to read from many tables. For the same reason the encryption of our new cipher is slower than WBAES – larger number of look-up tables were generated.

The generation of look-up tables takes more time in our new cipher than in WBAES, especially because of usage of *scrypt* hash chain as a key schedule with parameters chosen so that it needs more time to derive all the round keys.

6.3 Storage requirements

The generated look-up tables are stored in binary form using java Serializable interface. In the following table are the amounts of memory computed (first column) and generated and stored using our JCA implementation (second column).

Cipher	Computed space (kB)	Implementation (kB)
WBAES	752	1682.7
WBAES+	2840	5677.2

The storage of the look-up tables of our new cipher requires more space, because of multiplication by MDS matrix instead of *MixColumns* operation, what implies bigger type II and type III tables and larger number of type IV tables. However, the size of tables of type III can be decreased, as described in 7.1.

The java implementation stores not only the tables, but also some metadata needed for reconstruction of the objects. That is why the memory taken by serialized tables is approximately twice as much as the size of the tables themselves.

6.4 Security analysis

The cipher was designed with its white-box resistance in mind, its security principles are explained in detail in chapter 4. This section summarizes these mechanisms and analyses results of ciphertext randomness testing.

6.4.1 Round keys

There are three different round keys used in each round of the cipher, not convertible to each other. These are the round keys for encryption, round keys for S-boxes generation and round keys for MDS matrices generation.

Each round key is constructed as a hash of previous round key concatenated with the encryption key (and S-box or MDS constant, where needed), so each round key is dependent directly on the en-

cryption key. This implies the infeasibility of derivation of previous or next round keys.

As the hash function for round keys derivation, *scrypt* key derivation function is used. This memory-hard function is intentionally slow to make the brute-force attack impractical and as long as no short-cut in its scheme is found, there is no faster attack than brute force.

6.4.2 Confusion layer

In order to ensure the confusion in the cipher, four different substitution boxes are applied in each round. Constant S-boxes can be generated randomly (e.g. CMEA), which make the cipher using them likely to be weak, or crafted carefully to be resistant against known attacks (e.g. DES). However, in white-box context, these are not secure enough.

That is why this cipher uses key-dependent S-boxes based on Twofish S-boxes, that are proven to be strong and resistant against many known attacks including differential cryptanalysis. Each of them depends on 13 bytes derived from the round key using SHA256 function, what increases the complexity of part of the BGE attack of 2^{104}, to 2^{128}.

The nested Twofish permutations have been chosen in such a way that as many as possible permutations for each S-box are distinct and the S-boxes resist linear cryptanalysis.

6.4.3 Diffusion layer

Diffusion of the cipher is accomplished by multiplication of the state by a key-dependent MDS matrix in each round.

The design of their construction implies the size of the MDS matrix space to be approximately $\frac{1}{64} \cdot 16! \cdot \binom{64}{16} = 1.6 \cdot 10^{26}$ (number of 16-tuples of distinct non-zero elements in the field $GF(2^6)$, which XOR results in 1).

6.4.4 Randomness testing

Typical cryptanalysis of a new cipher starts with application of several statistical testing tools, like STS NIST and Dieharder, that can tell us, if the output looks random enough or not.

We have put our cipher to the STS NIST tests, applied on several types of bit-strings generated as:

- Whole cipher with
  - the key that does not enforce constant MDS matrix and random plaintext
  - the key that does not enforce constant MDS matrix and non-random plaintext (zeros, ones, periodical)
  - the key that enforces constant MDS matrix and random plaintext
  - the key that enforces constant MDS matrix and non-random plaintext (zeros, ones, periodical)
- One round of the cipher with
  - derived round keys and random input
  - derived round keys and non-random input
  - zero round keys for encryption and MDS matrix generation, zero key bytes used to generate S-boxes and random input
  - zero round keys for encryption and MDS matrix generation, zero key bytes used to generate S-boxes and non-random input

The randomness tests declared almost all the generated strings to be perfectly random. Only one have failed a few tests – ciphertext from one round of the cipher with S-boxes generated from all zero bytes and non-random input.

This is caused by our design of the MDS matrices. Four distinct S-boxes are constructed and applied to a periodical string. After multiplication by our matrix, equal bytes of the state on these exact positions result in equal bytes in the output, and thus the result stays periodical.

Chapter 7

Future work

As we can see in the table 4.1, the overall size of the tables has increased in comparison to WBAES of 2088 kB, what is almost four times more. To decrease this size, another approach was considered.

As a future work, man can implement and analyse these modifications in white-box scheme of our cipher, from both, security and performance point of view.

7.1 Alternative white-box scheme

Because of usage of 16×16 MDS matrices, table type II must stay as described in section 4.4, what means 144 tables with the bit width $8 \rightarrow 128$. However, we can decrease the size of table type III and the number of type IV tables.

The base idea is usage of only $8 \rightarrow 32$ mixing bijections. The 128-bit string created after multiplication by an MDS matrix is divided into four parts, 32-bit wide. Each of these parts is then multiplied by another 32×32 MB matrix and concatenated with other parts.

The elements of the 4-tuples of such results representing the state columns are then XORed together. Notice that this approach is similar to the one in WBAES.

After application of type III tables on 32-bit parts of the results, all four 128-bit strings are XORed together.

The overall scheme of this approach is illustrated in 7.1, and the following table contains the numbers of tables and space needed in this approach.

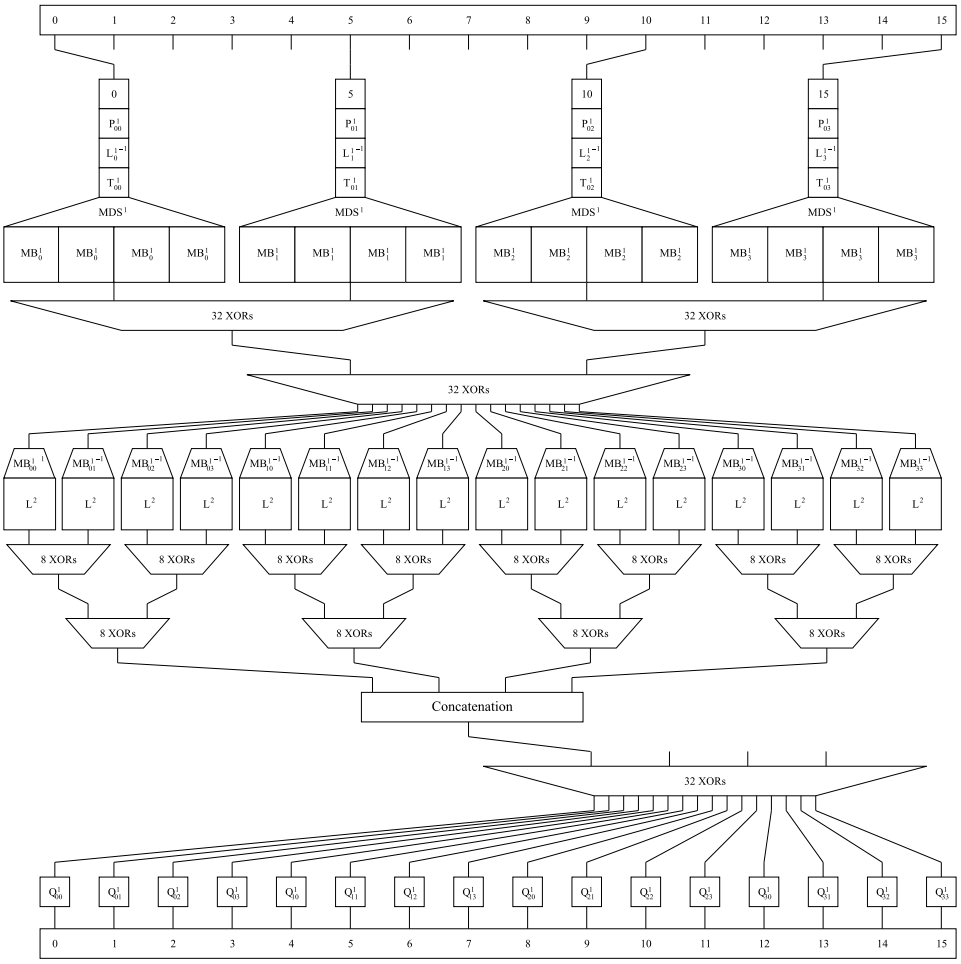

Figure 7.1: Alternative WBAES+ scheme.

Type	Number of tables	Width	Total size in bits
I	$16 \cdot 2 = 32$	$8 \to 128$	$32 \cdot 2^8 \cdot 128 = 1048576$
II	$16 \cdot 9 = 144$	$8 \to 128$	$144 \cdot 2^8 \cdot 128 = 4718592$
III	$16 \cdot 4 \cdot 9 = 576$	$8 \to 32$	$576 \cdot 2^8 \cdot 32 = 4718592$
IV (T1)	$32 \cdot 15 \cdot 2 = 960$	$8 \to 4$	$960 \cdot 2^8 \cdot 4 = 983040$
IV (MDS)	$4 \cdot 3 \cdot 32 \cdot 9 = 3456$	$8 \to 4$	$3456 \cdot 2^8 \cdot 4 = 3538944$
IV (MB^{-1})	$12 \cdot 8 \cdot 4 \cdot 9 = 3456$	$8 \to 4$	$3456 \cdot 2^8 \cdot 4 = 3538944$
IV (concat)	$32 \cdot 1 \cdot 9 = 288$	$8 \to 4$	$288 \cdot 2^8 \cdot 4 = 294912$
Total size			18841600 b = 2300 kB

Chapter 8

Conclusion

This thesis is focused on securing the symmetric encryption computation in an untrusted environment, the white-box context.

The most researched cipher for white-box cryptography, white-box AES was explained, with its building blocks, tables protections and weaknesses. The thesis then suggested three improvements of this cipher making it stronger in the white-box context and provides its implementation within Java Cryptography Architecture, along with the sample usage.

The analysis of security in chapter 6 has shown, that the improvements had no negative effects on the security of enciphered data and thus can be used as a substitute to WBAES, if backward compatibility can be sacrificed.

The encryption speed of the provided java implementation of this cipher is only a bit lower than WBAES implementation, but the memory requirements are higher. However, the thesis also suggests an improvement to the implementation of look-up tables protections in white-box version of this cipher to decrease the space needed for storage of these tables.

This thesis and its results can be useful in case someone needs to outsource his computation, but does not want anybody else to find out his sensitive information. As the implementation provided is written in java language, with the JCE Provider, it can be easily performed on java cards and its integration into larger systems is very simple.

Bibliography

[1] Stanley Chow, Phil Eisen, Harold Johnson, and Paul C. Van Oorschot. White-Box Cryptography and an AES Implementation. In *Proceedings of the Ninth Workshop on Selected Areas in Cryptography (SAC 2002)*, pages 250–270. Springer-Verlag, 2002.

[2] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In *Advances in Cryptology – CRYPTO 2012*, pages 850–867. Springer Berlin Heidelberg, 2012.

[3] Yaying Xiao and Xuejia Lai. A secure implementation of white-box AES. In *2nd International Conference on Computer Science and its Applications (CSA 2009)*, pages 1–6. 2009.

[4] Mohamed Karroumi. Protecting White-Box AES with Dual Ciphers. In *Proceedings of the 13th International Conference on Information Security and Cryptology, ICISC 2010*, volume 6829, pages 278–291. Springer Berlin Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24208-3.

[5] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a White Box AES Implementation. In *Selected Areas in Cryptography*, pages 227–240. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-24327-4.

[6] Brecht Wyseur. White-box cryptography: Hiding keys in software. [online], accessed 2015-05-14, 2012. URL `http://whiteboxcrypto.com/files/2012_misc.pdf`.

[7] Claude E. Shannon. Communication theory of secrecy systems*. *Bell system technical journal*, 28(4):656–715, 1949.

[8] Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, and Adi Shamir. Key Recovery Attacks of Practical Complexity on AES Variants With Up To 10 Rounds. Cryptology ePrint Archive, Report 2009/374, 2009. URL http://eprint.iacr.org/.

[9] Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001. URL http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[10] Dušan Klinec. White-box attack resistant cryptography. Master's thesis, Masaryk University, Faculty of Informatics, 2013.

[11] David Burren. crypt(3) – Trapdoor encryption. FreeBSD Library Functions Manual. [online], accessed 2015-05-14. URL https://www.freebsd.org/cgi/man.cgi?query=crypt(3).

[12] Pat Thoyts. md5crypt – MD5-based password encryption. FreeBSD Library Functions Manual. [online], accessed 2015-05-14. URL https://www.freebsd.org/cgi/man.cgi?query=md5crypt&manpath=FreeBSD9.0-RELEASEandPorts.

[13] Burton S. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0, RFC 2898, September 2000.

[14] Niels Provos and David Mazières. A future-adaptable password scheme. In *Proceedings of 1999 USENIX Annual Technical Conference*, pages 81–92. USENIX Association, 1999.

[15] Colin Percival. Stronger key derivation via sequential memory-hard functions. *Self-published*, pages 1–16, 2009.

[16] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Twofish: A 128-bit block cipher. In *First Advanced Encryption Standard (AES) Conference*, 1998.

[17] Richard C. Singleton. Maximum distance q-nary codes. volume 10, pages 116–118. IEEE Press, 2006.

[18] Florence J. MacWilliams and Neil J. A. Sloane. The theory of error-correcting codes. North Holland, Amsterdam, 1977.

[19] Jorge Nakahara Jr. and Élcio Abrahão. A New Involutory MDS Matrix for the AES. *Internal Journal of Network Security*, 9(2):109–116, 2009. URL `http://ijns.jalaxy.com.tw/contents/ijns-v9-n2/ijns-2009-v9-n2-p109-116.pdf`.

[20] Dušan Klinec. White-box AES (Java). [online], accessed 2014-03-15, 2013. URL `https://github.com/ph4r05/Whitebox-crypto-AES-java`.

[21] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique Cryptanalysis of the Full AES. In *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 344–371. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-25384-3.

[22] Wil Michiels and Paul Gorissen. Mechanism for Software Tamper Resistance: An Application of White-box Cryptography. In *Proceedings of the 2007 ACM Workshop on Digital Rights Management*, DRM '07, pages 82–89, New York, NY, USA, 2007. ACM.

[23] Yoni De Mulder, Peter Roelse, and Bart Preneel. Cryptanalysis of the Xiao – Lai White-Box AES Implementation. In *Selected Areas in Cryptography*, volume 7707 of *Lecture Notes in Computer Science*, pages 34–49. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-35998-9.

[24] Yoni De Mulder, Brecht Wyseur, and Bart Preneel. Cryptanalysis of a Perturbated White-Box AES Implementation. In *Progress in Cryptology – INDOCRYPT 2010*, volume 6498 of *Lecture Notes in Computer Science*, pages 292–310. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-17400-1.

[25] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2002.

[26] Brecht Wyseur. *White-Box Cryptography*. PhD thesis, Katholieke Universiteit Leuven, 2009.

[27] Donald Knuth. *The Art of Computer Programming: Volume 1: Fundamental Algorithms*. Addison-Wesley, 1968. ISBN 0-201-03801-3.

[28] Jonathan Knudsen. *Java Cryptography*. O'Reilly, 1998. ISBN 978-1-56592-402-4.

[29] Oracle and/or its affiliates. Java Cryptography Architecture (JCA) Reference Guide. [online], accessed 2015-05-14, 2014. URL `http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html`.

[30] Lawrence E. Bassham III, Andrew L. Rukhin, and Juan Soto. A statistical test suite for random and pseudorandom number generators for cryptographic applications, version STS-2.1. NIST Special Publication 800-22rev1a, 2010.

Appendix A

Contents of the attached CD

Part of the thesis is a CD with the following contents:

1. **thesis** – thesis in the pdf format
2. **latex** – source codes of the thesis
3. **programs** – java implementation of WBAES and WBAES+ with JCE Provider, available also on github (`https://github.com/xbacinsk/White-box_cipher_java`)

Appendix B

List of affected files

This is the list of files affected by the implementation done within this thesis.

The modifications of WBAES scheme are implemented in the following files:

```
AES.java
generator/AEShelper.java
generator/Generator.java
generator/GTBox8to128.java
generator/GXORCascade.java
generator/GXORCascadeState.java
```

The integration within Java Cryptography Architecture is implemented in the following files:

```
AES_Cipher.java
AES_CipherTest.java
AES_Provider.java
```

Implementation of `java.io.Serializable` interface was added to most of the other files.

Appendix C

S-box constants

C.1 Rijndael S-box

The following table represents a fixed substitution box used in AES algorithm, called Rijndael S-box.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1x	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2x	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3x	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4x	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5x	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6x	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7x	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8x	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9x	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
ax	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
bx	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
cx	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
dx	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
ex	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
fx	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

C.2 Fixed 8-by-8-bit permutations for Twofish S-boxes

q0

a9	67	b3	e8	04	fd	a3	76	9a	92	80	78	e4	dd	d1	38
0d	c6	35	98	18	f7	ec	6c	43	75	37	26	fa	13	94	48
f2	d0	8b	30	84	54	df	23	19	5b	3d	59	f3	ae	a2	82
63	01	83	2e	d9	51	9b	7c	a6	eb	a5	be	16	0c	e3	61
c0	8c	3a	f5	73	2c	25	0b	bb	4e	89	6b	53	6a	b4	f1
e1	e6	bd	45	e2	f4	b6	66	cc	95	03	56	d4	1c	1e	d7
fb	c3	8e	b5	e9	cf	bf	ba	ea	77	39	af	33	c9	62	71
81	79	09	ad	24	cd	f9	d8	e5	c5	b9	4d	44	08	86	e7
a1	1d	aa	ed	06	70	b2	d2	41	7b	a0	11	31	c2	27	90
20	f6	60	ff	96	5c	b1	ab	9e	9c	52	1b	5f	93	0a	ef
91	85	49	ee	2d	4f	8f	3b	47	87	6d	46	d6	3e	69	64
2a	ce	cb	2f	fc	97	05	7a	ac	7f	d5	1a	4b	0e	a7	5a
28	14	3f	29	88	3c	4c	02	b8	da	b0	17	55	1f	8a	7d
57	c7	8d	74	b7	c4	9f	72	7e	15	22	12	58	07	99	34
6e	50	de	68	65	bc	db	f8	c8	a8	2b	40	dc	fe	32	a4
ca	10	21	f0	d3	5d	0f	00	6f	9d	36	42	4a	5e	c1	e0

q1

75	f3	c6	f4	db	7b	fb	c8	4a	d3	e6	6b	45	7d	e8	4b
d6	32	d8	fd	37	71	f1	e1	30	0f	f8	1b	87	fa	06	3f
5e	ba	ae	5b	8a	00	bc	9d	6d	c1	b1	0e	80	5d	d2	d5
a0	84	07	14	b5	90	2c	a3	b2	73	4c	54	92	74	36	51
38	b0	bd	5a	fc	60	62	96	6c	42	f7	10	7c	28	27	8c
13	95	9c	c7	24	46	3b	70	ca	e3	85	cb	11	d0	93	b8
a6	83	20	ff	9f	77	c3	cc	03	6f	08	bf	40	e7	2b	e2
79	0c	aa	82	41	3a	ea	b9	e4	9a	a4	97	7e	da	7a	17
66	94	a1	1d	3d	f0	de	b3	0b	72	a7	1c	ef	d1	53	3e
8f	33	26	5f	ec	76	2a	49	81	88	ee	21	c4	1a	eb	d9
c5	39	99	cd	ad	31	8b	01	18	23	dd	1f	4e	2d	f9	48
4f	f2	65	8e	78	5c	58	19	8d	e5	98	57	67	7f	05	64
af	63	b6	fe	f5	b7	3c	a5	ce	e9	68	44	e0	4d	43	69
29	2e	ac	15	59	a8	0a	9e	6e	47	df	34	35	6a	cf	dc
22	c9	c0	9b	89	d4	ed	ab	12	a2	0d	52	bb	02	2f	a9
d7	61	1e	b4	50	04	f6	c2	16	25	86	56	55	09	be	91

C.3 RS code for Twofish S-boxes

Following matrix is the Reed-Solomon code matrix used for derivation of the key bytes needed for generation of Twofish S-boxes.

$$RS = \begin{pmatrix} 01 & a4 & 55 & 87 & 5a & 58 & db & 9e \\ a4 & 56 & 82 & f3 & 1e & c6 & 68 & e5 \\ 02 & a1 & fc & c1 & 47 & ae & 3d & 19 \\ a4 & 55 & 87 & 5a & 58 & db & 9e & 03 \end{pmatrix}$$

Appendix D

Test vectors for WBAES+

These are the testing vectors for WBAES+ implementation:

key = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

plaintext 32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34
ciphertext 8c 08 da 8e 43 fd 98 9e b3 a1 f2 85 e7 ae ef 39

plaintext 6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73 93 17 2a
ciphertext 18 b8 5e 45 1c 16 f8 d1 72 6d 93 d3 f1 f4 95 4f

plaintext ae 2d 8a 57 1e 03 ac 9c 9e b7 6f ac 45 af 8e 51
ciphertext 38 be 8b f9 b2 02 86 5f 6d 74 1f 40 61 a7 cc 36

plaintext 30 c8 1c 46 a3 5c e4 11 e5 fb c1 19 1a 0a 52 ef
ciphertext 22 5c 56 b4 e2 27 d5 58 f6 db f4 21 a6 55 d9 8d

plaintext f6 9f 24 45 df 4f 9b 17 ad 2b 41 7b e6 6c 37 10
ciphertext c8 f1 54 ed bc af ed e1 04 75 3e 06 68 ae e4 4c

key = 2b 7e 15 10 28 ae d2 a6 ab f7 18 88 09 cf 4f 3c

plaintext 32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34
ciphertext 1f ad 71 0c a1 1f 49 b2 0e 84 44 0c 62 80 bd cc

plaintext 6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73 93 17 2a
ciphertext 44 1d 12 c4 c3 89 00 b0 19 f5 8d 31 f2 06 ed 06

plaintext ae 2d 8a 57 1e 03 ac 9c 9e b7 6f ac 45 af 8e 51
ciphertext 2c 8d f3 1e 8f 5a 5f 02 2b 5b 3c 8d e6 d1 76 92

plaintext 30 c8 1c 46 a3 5c e4 11 e5 fb c1 19 1a 0a 52 ef
ciphertext 5b ca c9 c6 87 64 f9 46 69 7a fb 46 03 30 fa ef

plaintext f6 9f 24 45 df 4f 9b 17 ad 2b 41 7b e6 6c 37 10
ciphertext ee 9e d9 d1 ed 3c c3 2b e9 99 dc 56 9f ab 40 85

Appendix E

Sample usage of our implementation

Secure environment – look-up tables generation

```
SecureRandom rnd = new SecureRandom();

byte[] keyData = new byte[]{
    (byte)0x2b, (byte)0x7e, (byte)0x15, (byte)0x16,
    (byte)0x28, (byte)0xae, (byte)0xd2, (byte)0xa6,
    (byte)0xd2, (byte)0xa6, (byte)0x2b, (byte)0x7e,
    (byte)0x15, (byte)0x16, (byte)0x28, (byte)0xae,
};

Key key = new SecretKeySpec(keyData, "WBAES");

AES_Cipher encryptor = new AES_Cipher();
try {
    //generates look-up tables for encryption
    encryptor.engineInit(Cipher.ENCRYPT_MODE, key, rnd);
} catch (InvalidKeyException e) {}

AES_Cipher decryptor = new AES_Cipher();
try {
    //generates look-up tables for decryption
    decryptor.engineInit(Cipher.DECRYPT_MODE, key, rnd);
} catch (InvalidKeyException e) {}
```

Insecure environment – tables loading and encryption/decryption

```
int iolength = 16; //sample input/output length
SecureRandom rnd = new SecureRandom();

AES_Cipher encryptor = new AES_Cipher();
try {
    //loads look-up tables for encryption
    encryptor.engineInit(Cipher.ENCRYPT_MODE, null, rnd);
} catch (InvalidKeyException e) {}

AES_Cipher decryptor = new AES_Cipher();
try {
    //loads look-up tables for decryption
    decryptor.engineInit(Cipher.DECRYPT_MODE, null, rnd);
} catch (InvalidKeyException e) {}

byte[] outputEnc = new byte[iolength];
try {
    outputEnc = encryptor.engineDoFinal(input /*byte
        array with length iolength*/, 0, iolength);
} catch (Exception e) {}

byte[] outputDec = new byte[iolength];
try {
    outputDec = decryptor.engineDoFinal(outputEnc, 0,
        iolength);
} catch (Exception e) {}
```